A View-Based System for Eliciting Software Process Models

Josée Turgeon

School of Computer Science
McGill University, Montreal

Submitted in September 1999
A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Abstract

We propose an approach, together with specific underlying techniques and a system to support these, for eliciting models of software development processes. Software process elicitation involves: gathering process information from the agents involved in a development process, from documents, and through observation; modeling this information; and verifying that the model built is consistent and complete.

The elicited models can be used in process assessment and for identifying improvement opportunities, which could lead to improvement in product quality, delivery and costs. Process models can also be used for other purposes, such as measurement of software products and processes; insertion of software tools in development processes; training new developers on the overall development tasks; project planning; risk assessment; software process guidance and automation; etc.

In our approach, process information is gathered from different sources, and the separate descriptions (called *views*) are merged -- after ensuring their consistency and completeness -- to form the entire model. Our hypothesis is that models built from information gathered from multiple views are of higher quality than those built without consideration for such views. The techniques underlying our approach include: (1) planning for elicitation; (2) eliciting the different views; (3) checking for intra-view consistency; (4) identifying common components across views; (5) merging the views; (6) checking for the overall model quality; and (7) modifying the model if necessary. The thesis demonstrates these features through usage scenarios of the system, called *V-elicit*. The underlying techniques, together with the supporting V-elicit system, constitute a novel contribution in the software process field.

The validation of our approach is demonstrated through five case studies and a comparison with tools described in the literature. These studies show that V-elicit: (i) helps develop process models of superior quality (in terms of coverage); (ii) is more time

consuming for developing models, but this time can be reduced by concurrent view development (involving multiple elicitors); and (iii) supports more elicitation tasks than do competitive tools.

Résumé

Dans cette thèse, nous proposons une approche pour l'extraction de processus de développement de logiciels, ainsi que des techniques spécifiques et un système supportant cette approche. L'extraction de processus logiciels inclut la collection d'information sur le processus, la modélisation de cette information, et la vérification que ce modèle est consistant et complet. L'information requise est obtenue des agents impliqués dans le processus, dans la documentation existantes, et par des observations.

Les modèles extraits peuvent être utilisés dans l'évaluation des processus et de leurs améliorations possibles, pouvant conduire à une amélioration au niveau de la qualité des logiciels, de leurs coûts, et de leur temps de développement. Les modèles peuvent aussi être utilisés à d'autres fins, comme par exemple la mesure des processus et des logiciels; l'insertion d'outils dans le processus de développement; la formation de nouveaux employés sur les différentes tâches de développement; la planification de projets; l'évaluation de risque; l'encadrement et l'automatisation du processus; etc.

Dans notre approche, l'information est obtenue de sources différentes, et les différentes descriptions (appelées vues), après vérification qu'elles sont consistantes et complètes, sont fusionnées pour former le modèle complet. Notre hypothèse est que les modèles construits à partir d'information provenant de plusieurs vues sont de plus haute qualité que ceux construits sans considération de ces différentes vues. Les techniques supportant notre approche incluent: (1) la planification du processus d'extraction; (2) l'obtention des différentes vues; (3) la vérification que l'information dans chaque vue est consistante; (4) l'identification des composantes communes entre les vues; (5) la fusion des vues; (6) la vérification de la qualité globale du modèle; et (7) la modification du modèle si nécessaire. Cette thèse démontre ces techniques à travers des scénarios d'utilisation de notre système, appelé V-elicit. Ce système et les techniques supportées sont une nouvelle contribution dans le domaine des processus de développement de logiciels.

La validation de notre approche est démontrée par cinq études de cas et la comparaison avec les outils décrits dans la littérature. Ces études montrent que V-elicit: (i) aide à développer des modèles de qualité supérieure (en terme de couverture); (ii) demande plus de temps pour développer des modèles, mais que ce temps peut être réduit en développant les vues en parallèle (plusieurs personnes travaillant à cette extraction); et (iii) supporte un plus grand éventail de tâches reliées à l'extraction de modèles que les autres systèmes comparables.

Acknowledgments

First of all, I would like to thank my supervisor, Professor Nazim Madhavji, for his continuous support and help throughout my Ph.D. studies, especially for the fruitful discussions, insights, and comments on my work. He also helped me in finding the human resources necessary for developing the V-elicit system, and validating it through case studies. But most of all, he helped me develop the skills necessary to do research.

Special thanks should also go to McGill University and the School of Computer Science, for providing the resources necessary to complete my Ph.D.

For all other students and research assistants who helped me in the development and validation of my system, thank you very much for your participation and effort. Such large development effort (and validation) would not have been possible without you.

My colleagues at University of New Brunswick (Saint John campus) also helped me and supported me during the past two years. In particular, Dr. Gupta, a professor in statistics, has helped me a lot in choosing appropriate data analysis techniques for my case studies. Many thanks. Also, I would like to thank the University of New Brunswick, for letting me work there while finishing my Ph.D.

Finally, I would like to thank my family, and especially my boyfriend Michel Tassé, for their constant moral support throughout my studies. Such support, although not as visible as direct help, has been very important for me.

This research work has been partly supported by NSERC, through a 4-year postgraduate scholarship.

Table of content

List of figures

List of tables

Chapter One - Introduction

It is widely recognized that the development of software is often plagued with quality problems, late delivery and cost over-runs [Gib94]. Kitson and Masters [KiM93] report that, from the SEI process assessments carried out between 1987 and 1991, 81% of the organizations assessed were at level 1 on their 5-level Capability Maturity Model[1], and 12% were assessed at level 2.

In order to improve software development capability, one should improve software development processes [KeH89]. By fixing the defects only in the software systems leads to short term product improvements. By fixing the problems in the processes, it can lead to defect prevention, and thus to long term product quality, timely delivery of systems, and development within budgets. One of the first steps in process improvement, however, is to have visibility into the existing processes [KeH89, Mad91, Nej91, OiB92, KTL92, Hum93, McB93, PSV94, TSK95], so as to simplify problem detection and analysis of changes an organization needs to make for improvement.

One way to obtain such visibility is to build a *descriptive* (or *as-is*) model of the process concerned, by eliciting appropriate process information. Such a model forms a blueprint of the process concerned and is a concrete basis for making process improvements.

A process model is represented explicitly, using formal descriptions of the tasks performed, when they are performed, entry/exit criteria, the artifacts produced and consumed (e.g., requirements, design, code, etc.), the technical procedures and tools used

---

[1] CMM [PCC93]: It is a model with 5 levels of maturity, and is used to assess an organization's capability to develop software, and to guide it in improving its processes. At level 1, the productivity and quality of software are low, and the schedules are missed. As the organization matures to upper levels, productivity and quality increase and schedules are more on target. At level 2, some techniques are introduced to help achieve these goals, for example requirements management, software quality assurance, configuration management, and project planning and tracking. A key criterion to be at level 3 is to have the software development processes defined. At level 4, projects are managed quantitatively, and at level 5, processes are improved continuously.

to produce the artifacts, the structure of the artifacts, the agents responsible for each task, etc. Overall, the elicitation process, through which a process model is built, encompasses the following key activities: planning, gathering data, modeling, analysis, and validation [MHH94].

Once a process model has been built, it can be used for a number of purposes: process assessment with respect to software quality, development costs and schedules [McB93, Nej95]; measurement of software products and processes [MBB92, Pfl93, Vis94, LHR95, BDT96]; insertion of software tools in development processes [BMH96]; training new developers on the overall development tasks [HuK89]; project planning [Kel91]; risk assessment [Bro95]; software process guidance and automation [Fer93, BEM94, BGR94, ScW95, BNF96, BHM97, NWC97, SuO97, BeK98, DEA98]; etc. It is therefore important that the process model is of high quality (consistent, complete, and accurate).

Note that this *process-oriented* approach, for improving software quality, costs and deliverability, is complementary to that of building new methods/techniques/tools for software development, such as object-oriented design methods and CASE tools. Typically, a development process uses these methods and tools, and the process model should show where and how these are used in the process.

1.1 Problem definition and research hypothesis

Many researchers have proposed general approaches for eliciting process models. In some cases ([RHM85, MHH94, BFL95]), the approach identifies and describes the general steps to be performed (e.g., planning the elicitation process, gathering information, analyzing model, etc.). In other cases ([KeH89, Rom93, McB93]), the dynamics of the core elicitation steps is provided, typically as an iterative approach: in each cycle, inconsistencies from the previous cycle are resolved, and further details are added to the model. Such iterations are performed until a model is satisfactory. Other elicitation approaches focus on the model to be produced, by providing guidance on the types of

information to be gathered, and the order in which each type should be elicited (e.g., first elicit product information, then activity information, and finally resource information) [Kaw92, Gal92, ADH94, BDT96]. While these approaches are a good starting point, they are intuitive and rely on elicitors to build quality models. Additional technical support is required to ensure that the models will be of high quality. In the approaches above, such support is rather limited.

An important issue for process elicitation is that, often there isn't a single person who knows the complete process [DeG93, Rom93, Ver96]. A recent case study carried out by Siddiqui [Sid97] indicates that agents performing a process may not have a common understanding of a process and, consequently, they lack visibility into the artifacts produced by the process and the activities carried out, by as much as 50-75%. Other intuitive observations also support the results of this case study [DNR90, SaW94, SKV95, EsB95].

In order to elicit a complete process model, it is thus important to obtain information from multiple sources (such as different agents, project documentation, observations, etc.). The partial descriptions of a process obtained from different sources are called *views*.

While utilizing multiple sources for process information can lead to quality models, this approach is not without impediments. In particular, different agents may give inconsistent or conflicting information about the same process [KeH89, Rom93, Fra93, SKV95, Ver96]. For example, the terminology used can be different; some process components might be missing in some views; development tasks might not be grouped in the same way into activities at higher levels of abstraction; level of details given might be different; etc. Process documentation might also give inconsistent and incomplete information [HMB94, Rom93, Vot93]. Such inconsistencies can affect the quality of the elicited process model.

Until now, the elicitation approaches proposed in the literature have generally not dealt with the problem of multiple views and inconsistencies across them, leaving it to the elicitor to resolve them. One exception to this is the work by Verlage [Ver96], who has proposed techniques to help detect similarities and inconsistencies between two views, but the inconsistency resolution and view merging processes are not supported. Unfortunately, the elicitor may not detect all the conflicts and inconsistencies (especially when the same model component is involved in multiple inconsistencies), or may not analyze appropriately the alternatives for resolving the inconsistencies. Besides, the manual approach can be prohibitively time consuming and costly.

The goal of this thesis research is thus to develop a coherent set of techniques and technical support for systematically eliciting software process models using multiple sources of information. Our research hypothesis here is that such a view-based approach (and its technical support) to eliciting software process models would result in high quality models, especially in terms of their completeness. Such models would form a stronger baseline for process analysis and improvement.

1.2 Technical approach and assumptions

The general elicitation approach taken is to first plan the elicitation process and identify the sources of information necessary for a proper coverage of the process being elicited. Then, models are built from each source of information independently, and are represented internally as entity-relationship diagrams [Pen89, Gal92, ADH94]. Such partial models (or *views*) are all checked separately for internal consistency, using constraints specified in first-order logic [BeT93].

The full process model is a combination of all the views. The process of combining them starts with an identification of similar components across the views (a technique adapted from the requirement engineering area [LeF91]). Then, discrepancies among them (i.e., what is not similar) are found and resolved. Finally, views are merged into one process

model. Such a final model is checked for consistency, validated by different people involved in the process being elicited, and checked against development policies for feedback purposes.

We have made our approach flexible by letting the elicitor decide on the type of information to be modeled, and to specify what constitutes an inconsistency in a model or a view (through the specification of constraints in first-order logic).

This approach and the set of techniques meet our goal of providing support for systematically eliciting software process models using multiple sources of information. Tool support has also been developed for each of these techniques, in a system called "V-elicit".

Throughout the development of these techniques, we have built upon existing concepts and tools where possible. Table 1 shows which foreign components have been utilized at both the conceptual and implementation levels (with appropriate adaptations in the case of the concepts), for each of the key features in our approach. An empty cell indicates an entirely new feature.

The technical choices were driven by the following assumptions:

A1. A process model can be specified using an entity-relationship diagram.

A2. Using entity-relationship diagrams allows the elicitor to define the types of information a model should contain.

A3. By using a language based on first-order logic, one can define what an inconsistency is (inside a single view or model), and the inconsistency verification can then be automated.

A4. The identification of similar components across views can be partly automated through the computation of a similarity score across the components.

A5. By a careful identification of types of inconsistencies across views, and their possible solutions, the view merging process can be automated using the solutions provided by the elicitor.

A6. By using a language based on first-order logic, one can formally describe development policies, and their verification on a given model can be automated.

These assumptions are validated later in this thesis (see Sections 7.2.3 and 7.2.5).

Features of our elicitation approach	Utilized at conceptual level	Utilized at implementation level
View / model notation and storage	• Model representation using entity-relationship diagrams [Pen89, Gal92, ADH94]	• OBST [CRS92] – an object-oriented database
View / model visualization and editing, and overall user interface	• Structured text-based editing of models [MHH94] • generating graphical representation in Dotty [CDP95]	• X-elicit [MHH94] • Dotty [KoN96] • Motif [HeF94]
Constraint checking	• First-order logic constraints for capturing business rules [BeT93]	
Identification of similar components	• Heuristics for matching rules in different software requirement descriptions [LeF91]	
Discrepancy detection / resolution		
View merging		

Table 1 – Concepts and tools utilized in the development of V-elicit

1.3 Research method

Our general research approach has been one of theory and tool building followed by experimental research for validating the tool and techniques developed. Throughout this research process, the ideas underlying V-elicit have also been discussed with other researchers and practitioners in the field.

We first listed the requirements and separated them into the different elicitation phases. We then developed a user-definable modeling schema (i.e., notation to be used in the models and views). Techniques for each elicitation phase have been developed separately. For each phase, we first looked at existing (but small) processes, for understanding the

problems and developing solutions. We also looked for existing concepts or tools that could be utilized (fully, partly, or with adaptation). The techniques developed this way were then implemented and tested individually.

Finally, the overall approach has been validated and compared with other approaches and tools through case studies and literature comparison. To this end, we have used well known techniques for experimental design [FeP97] (including the use of GQM [BaW84] for identifying the metrics to be used) and for data analysis (two-way ANOVA followed by Student-Newman-Keuls range test [Hic93], Friedman test [Dan90]).

1.4 Key results, and originality statement

The key results of our research are both the set of techniques covering all phases of the multi-view elicitation process, and the tool supporting such techniques. To our knowledge, none of the existing elicitation approaches and tools provide such a comprehensive set of techniques for dealing with view-based elicitation. More specifically, the techniques for identifying discrepancies across views, for helping in their resolution, for merging the views, and for checking a model against development policies, are new.

1.5 Organization of thesis

The rest of this thesis is organized as follows. Chapter Two discusses related work. Chapter Three presents the requirements for a view-based elicitation system. Chapter Four describes a user-definable modeling schema which dictates the content and structure of process models. Chapter Five describes the elicitation process through an example. Chapter Six gives algorithmic details for the techniques used to support elicitation. Chapter Seven describes the validation of our approach through case studies and literature comparisons. Finally, Chapter Eight concludes this thesis.

Chapter Two - Related work

In this chapter, we first describe some background on software process models. The second subsection describes several efforts on software process elicitation, and the approaches taken. We then discuss the research advance in the area of views and view-based modeling. The last section summarizes related work, and puts our work into context.

2.1 Background on process models

A *process* is "a set of partially ordered steps intended to reach a goal" [FeH93]. In the case of a *software development process* (or simply *software process*), the goal is the development or enhancement of the software products or systems. This process can be described formally in a *software process model*.

The following types of items are usually represented in a process model: [FeH93, CKO92]

- *process step*: an atomic action of a process that has no externally visible substructure.
- *process element* (or *activity*): any component of a process (can be a single process step or a very large part of the process containing multiple steps).
- *artifact*: a product created or modified by the enactment of a process element.
- *agent*: an actor (human or machine) who performs a process element
- *role*: a coherent set of process elements to be assigned to an agent as a unit of functional responsibility.

Sometimes, such information is modeled from different *perspectives*. For example, the following three perspectives are provided in the Statemate modeling language [KeH89]: *functional*, *behavioral*, and *organizational*.

The *functional* perspective represents *what* is done, i.e., the set of activities performed, their decomposition into sub-activities or process steps, and the artifacts produced or used by these activities. Figure 1 shows the functional perspective of a review process[2], using the dataflow diagram notation (with multiple levels of abstraction on the same diagram). The review activity contains three steps: preparation, meeting, and writing report. It interacts with the document production activity. The arrows labeled document, notes, feedback and report are the artifacts produced at various times by the activities. Information and checklist are provided by the external entities "user" and "SQA" respectively. The output of the entire process is the validated document.

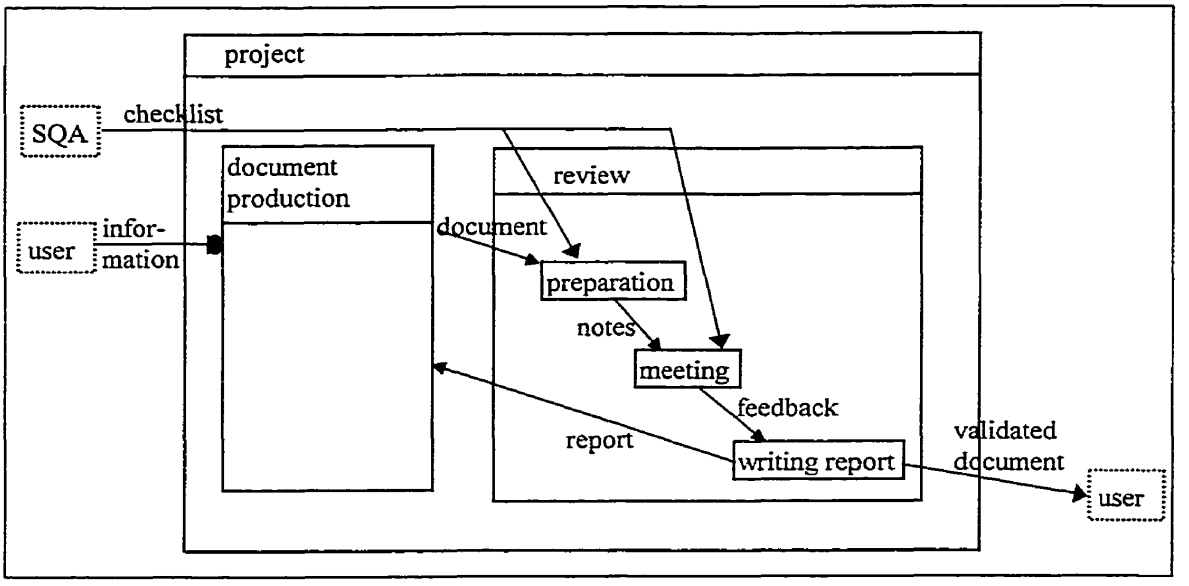


Figure 1 - Functional perspective of a review process

The information related to *when* and *how* the activities are performed is represented in the *behavioral* perspective. For example, the ordering of the activities defined in Figure 1 is shown in Figure 2. The arrows represent the precedence relationship among activities. Such arrows may have conditions specified on it (e.g., "need modifications"), indicating when such path should be taken during process execution.

[2] This review process is described in [Pre97].

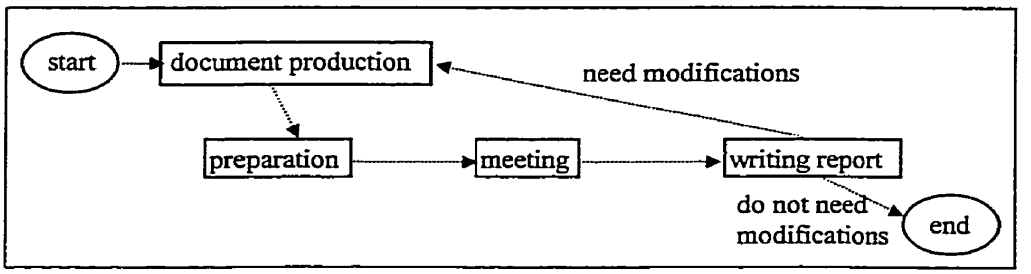


Figure 2 - Behavioral perspective of a review process

The way *agents* are grouped into *teams* and the *communication channels* between the agents are represented in the *organizational* perspective. An example of this perspective is given in Figure 3, showing the roles and teams involved in the process, the verbal communication among them, and how the artifacts (dashed boxes) are handled: the initial information to the analyst is provided verbally; the document is saved in a file ("IO") and a hard copy is passed to the review team ("hand carried"); internal review artifacts (checklist, notes, and feedback) are on paper ("hand carried"); and the final report is written to a file ("IO") and sent back to the analyst by e-mail. Notice that such a graph does not present any information regarding the sequence of the tasks and communication: this is provided in the behavioral perspective above (Figure 2).

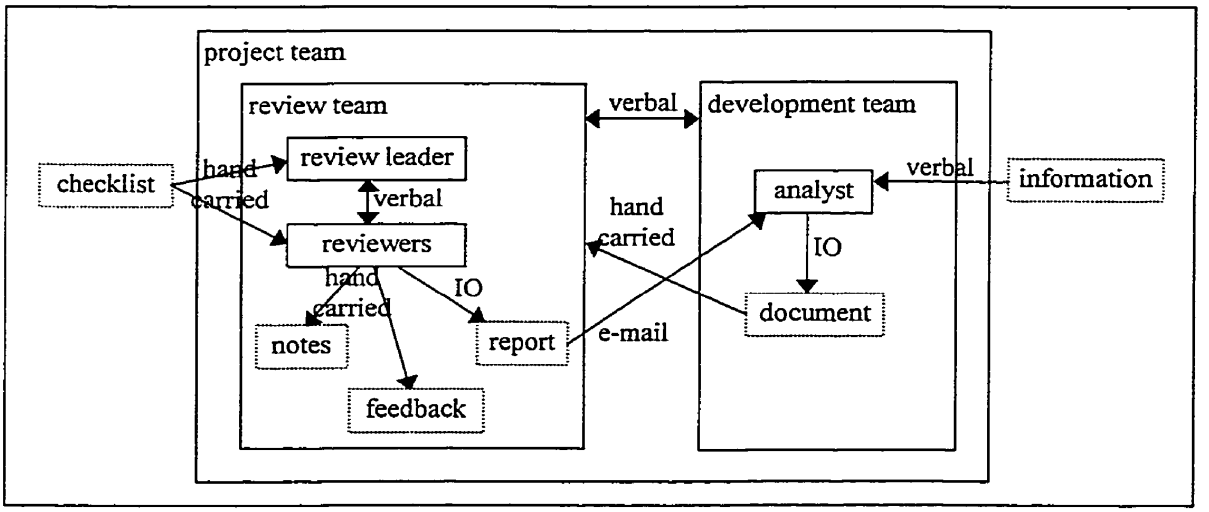


Figure 3 - Organizational perspective of a review process

A software process model is the union of all these perspectives. Yet, other perspectives could include *actor dependency* [YuM94], *informational* and *quantitative* [CKO92].

There exists many notational paradigms for encoding and representing software process models. Examples are : algorithmic programming languages (as in APPL/A), state-transition diagrams (as in Statemate), Petri Nets (as in FUNSOFT Nets), rule-based languages (as in Marvel), etc. [CKO92].

The choice of the modeling language to be used is driven by the intended use of the model. Curtis et.al. have identified the following five primary objectives (or uses) of a process model [CKO92]:

- *facilitate human understanding and communication*: The software process can be large and complex, and managers and participants may have difficulties having a grasp on its entire performance. Having a model of the process helps in understanding the process, and communicating it to the people involved. It can also help in training new participants.
- *support process improvement*: A model can be used for analyzing a software process, and identifying improvement opportunities. Once potential changes are identified, the model can also be used for assessing the impact of the change. Simulation of the model can be used for this purpose.
- *support process management*: The plan of the software project can be based on a process model. The manager can reuse parts of the models from previous projects, for the new project. This plan can then be used to control the software project. Check-points and measurement points can be identified in the process model.
- *automate process guidance*: A model can provide guidance to the process performers about the tasks to be done next, the tools to be used, the available documents, etc.
- *automate execution support:* In an environment, a model can be used to control the behavior in the development process. The ordering of the different steps can be enforced by verifying which steps can be performed next, from a given state of the

process. Some of the tasks can also be automated, such as collecting measurement data.

As one can see, the goal of process improvement is only one of the many possible uses of a process model. Thus, our research goal of developing techniques and technical support for eliciting software process models could have an impact on a number of different areas in the software process field.

2.2 Related work on software process elicitation

Several efforts have been made in industry on modeling software processes, especially in those organizations seeking to achieve level 3 (defined) on the SEI Capability Maturity Model. Table 2 gives some examples of such modeling efforts. The identified authors describe the benefits of modeling processes, as well as some lessons learned.

Author(s)	Organization	Description	Reference
Frailey	Texas Instruments	building a corporate-wide model	[Fra91]
Favaro	European Space Agency	establishing an European Space Software Development Environment (ESSDE)	[Fav92]
Drew	Paramax	process definition	[Dre93]
Carr, Dandekar, Perry	AT&T	process architecture and interfaces	[CDP95]
Tanaka, Sakamoto, et.al	OMRON Corporation	software process improvement	[TSK95]

Table 2- Example efforts in industry on modeling software processes

Many researchers have also attempted to elicit small-scale process models, often as a way to validate their modeling languages or approaches. Others have presented elicitation approaches. These works are summarized in Table 3, and the ones providing an elicitation approach are discussed below.

Researcher	Organization (if tried in industry)	Elicitation approach	Process modeling language / tool used	Notational paradigm (*)	Refer-ence
Radice et.al.	IBM	general steps	ETVX	- systems analysis and design	[RHM85]
Kellner and Hansen	Ogden Air Logistics Center	by iterations on levels of abstraction	Statemate	- systems analysis and design - events and triggers - state transitions and petri-nets - data modeling	[KeH89]
Gruhn and Jegelka	Lion	not described	FUNSOFT Nets	- state transitions and petri-nets	[GrJ92]
Kawalek	British Telecommunications	by type of information	Process Modeling Cookbook	- systems analysis and design - state transitions and petri-nets	[Kaw92]
Galle	European Space Agency	by type of information	ERD / dataflow / PERT charts / text	- systems analysis and design - data modeling - precedence networks	[Gal92]
Rombach	NASA's Software Engineering Laboratory and TRW	by iterations on views	MVP-L	- AI languages and approaches - control flow - formal languages - object modeling	[Rom93]
Scacchi and Mi	many (over 30)	not described	Articulator	- AI languages and approaches - object modeling	[ScM93]
McGowan and Bohner	Contel Corporation	by iterations on levels of abstraction	SADT	- systems analysis and design	[McB93]
Madhavji et.al.	IBM, Transport Canada	general steps	Elicit and Statemate	- data modeling	[MHH94, Sid97]
Phalp and Shepperd	Schlumberger Technologies	general steps	dataflow diagrams	- systems analysis and design	[PhS94]
Aumaitre, Dowson, and Harjani	European Space Agency	by type of information	Process Weaver	- systems analysis and design - AI languages and approaches - state transitions and petri-nets - data modeling	[ADH94]
Barghouti et.al.	AT&T	not described	Marvel	- AI languages and approaches - object modeling	[BRB95]

Researcher	Organization (if tried in industry)	Elicitation approach	Process modeling language / tool used	Notational paradigm (*)	Refer-ence
Bandinelli et.al.	Italtel and British Airways	by iterations on levels of abstraction	finite state machines and SLANG	- events and triggers - state transitions and petri-nets - data modeling	[BFL95, EBL96]
Yu and Mylopoulos	Flight Dynamics Division of NASA Goddard Space Flight Center (done by Briand et.al. [BMS95])	not described	Actor-Dependency Model	- AI languages and approaches - data modeling	[YuM94]
Broecker et.al.	Robert Bosch GmbH	by type of information	MVP-L	- AI languages and approaches - control flow - formal languages - object modeling	[BDT96]
Nguyen, Wang, and Conradi	Norwegian banking software house	not described	EPOS	- object modeling - precedence networks	[NWC97]
Sa and Warboys	none	by iterations on levels of abstraction	OBM	- AI languages and approaches - object modeling	[SaW94]
Cook and Wolf	none	automatic from event data	none specified	---	[CoW95]
Sommerville et.al.	none	by views	none specified	---	[SKV95]
Verlage	none	by gathering and analysis of views	MVP-L	- AI languages and approaches - control flow - formal languages - object modeling	[Ver96]
Turgeon and Madhavji	none	by gathering, analyzing, and merging views, and validation against given constraints	V-elicit	- systems analysis and design - data modeling	[TuM96]

Table 3 - Research efforts in eliciting process models

* Notational paradigm: basis of the notation used, as described in [CKO92] (Table 2)

Some researchers just present the types of information that can be elicited, and their ordering [Kaw92, Gal92, ADH94, BDT96]. This kind of information can be useful when gathering process data, especially at the front-end of the elicitation process.

In an early IBM study on process modeling, Radice et.al. [RHM85] described six phases for process model elicitation and process improvement. The phases related to elicitation are: planning, on-site study (interviews), and analysis. After each day of interview, the elicitation team should analyze the interviews and evaluate the process. Guidelines for interviewing the process participants were provided.

More recently, Kellner and Hansen [KeH89] have used rounds of interviews, with each iteration yielding more details in the descriptions. Each iteration formed a basis for validating the model from the previous iteration and manually reconciling conflicts.

In the context of NASA processes, Rombach [Rom93] first captured the different views and modeled them, and then reviewed and modified the models until all conflicts were resolved. The consistency analysis across views was carried out manually.

In [McB93], McGowan and Bohner present some elicitation steps, including preparation, conducting interviews and constructing models. They first build a model, and then iterate between reviewing and refining the model, in order to obtain a model that reflects the actual process. They also present steps to improve the process using that model. Their description of the steps to be performed is abstract.

In a joint process improvement effort between CEFRIEL and Italtel, Bandinelli et.al.[BFL95] have defined three elicitation phases: knowledge elicitation, formalization, and model review and assessment. The main idea is to perform those phases in sequence, backtracking to previous phases when more information in the model is needed.

The above researchers have used specific notations and tools to represent process models, but the general paradigm for elicitation is that of iterating until the elicited model is satisfactory. The base criteria for completeness is often ensuring that the various process attributes (such as inputs, outputs, resources, entry/exit criteria, etc.) are filled with appropriate values from the process being modeled.

A detailed elicitation method has been described by Madhavji et.al. [MHH94]. This method, called Elicit, has the following steps: understand the organizational environment, define objectives, plan the elicitation strategy, develop process models, validate process models, analyze process models, post-analysis, and packaging. They have also developed a tool to capture and organize textual process information in a hierarchical format. This organized information is then translated into a graphical model using tools such as Statemate. This approach helps in the early (more intuitive) phases. The Elicit approach has been applied to industrial-scale models, although it does not have automated supported for dealing with multiple sources of process information.

There have also been some other proposals for elicitation methods, but as yet they have not been used (tested) extensively. For example, Sa and Warboys [SaW94] propose a method where a collection of abstract objects is defined first, and then each object is refined to the next level of abstraction. After the refinement, they check the consistency of the information with the previous level. Each newly defined object can then in turn be refined. This stepwise refinement is terminated when one reaches a satisfactory level of detail.

Also, Cook and Wolf [CoW95] have presented an approach and specific techniques for automatic generation of descriptive process models from event data collected from a process. They describe and compare three inference methods: RNet, Ktail, and Markov. This can be used for giving a specific execution thread of the process, but not the overall model.

Some view-based methods have also been proposed in the literature. For example, Verlage [Ver96] has proposed the following steps for eliciting a model from different views: independent modeling of views, detecting similarities between views, detecting inconsistencies between views, and merging views. A similarity analysis function, based on the semantics in the MVP modeling language, helps identifying the common elements across views. A tentative set of consistency rules can be applied to detect inconsistencies between two views. The choice of the rules to be applied depends on the relationships between the two views (i.e., how much they overlap). The differences in the abstraction hierarchies are not resolved: each hierarchy is kept separately. This research is at an early stage, and full implementation and validation of the approach is still pending.

A second view-based approach is proposed by Sommerville et.al. [SKV95]. After identifying and defining the viewpoints, appropriate questions are generated for eliciting process information from different viewpoints. Separate models are then built, but they are not merged. They just propose to manage the interfaces between the viewpoints. In this approach, the information gathering step is well defined, but there is no technique for subsequent steps.

In [TuM96], Turgeon and Madhavji describe their early work on view-based process elicitation. This thesis, in fact, builds on the ideas and concepts in the paper, and represents an operational body of work which has undergone significant test cases and comparative analysis.

Most of the elicitation methods proposed above, although providing good advice on how to elicit process models, do not use a systematic approach and rigorous techniques for such a task. Also, only few of them have dealt with the problem of views. They generally let the elicitor resolve the inconsistencies across views, or propose to keep the views separate and manage the interface between them only. We believe that for the purpose of a common understanding of the development process, such inconsistencies need to be resolved, and that automated support should be provided for such a task.

Notice that other process-based software development environments and executable process modeling languages could also be used for software process elicitation, even though such a possibility was not indicated in the literature. They permit a user to specify a model in a given language, which is a goal of the elicitation process. Such tools include: Adele-Tempo [BEM94], APEL [DEA98], JIL and Little-JIL [SuO97,WLM98], Merlin [ScW95], Oz [BeK98], ProcessWise / ProcessWeb [BGR94, GrW96], and others.

2.3 Related work on view modeling

Many researchers have expressed the need for people-related views when visualizing a process model [Pen89, Dei92, JaM94, EsB95, LHR95]. Suggestions include that only the relevant or localized information should be shown, at the level of details that are needed. For example, a designer may prefer to see the details of his/her work only, but a manager may need a broader range of information (covering many phases) at a higher level of abstraction.

Other researchers have pointed out the need to use different views when eliciting a process model [Rom93, SKV95, Ver96, TuM96]. These are described in the previous subsection, with their specific elicitation approaches and techniques.

Finkelstein et.al. have proposed a structure for describing views called "Viewpoint" [FKN92]. Their Viewpoint allows one to describe the work of different people, using different notations. The core part (specification) is related to the product itself, but the process is also described in a "work plan". Inconsistencies within and across viewpoints are dealt with, but these inconsistencies are only in the product that is built, not in the process description. In [FGH93], they have described how to find the inconsistencies, and handle them. Their algorithm is to first map different representations into a common schema, and then use first-order logic notation to specify what an inconsistency is, and what to do when one is found.

In [ACF96], Avrilionis, Cunin, and Fernström have described a view-based approach to the model evolution, that permits one to analyze and modify a part of a model only (view). Because views are less complex and smaller in scope than models, this analysis and modification process becomes easier. A model is first decomposed into its constituent views, and the interfaces among them are clearly identified. The views can then be modified, and recombined into the entire model using transformation steps specified by the user. Their mechanism, called "OPSIS", supports process modeling languages based on Petri-net notation.

2.4 Summary and analysis

To recap, in Section 2.1 we started by describing the expected output of the software process elicitation task: the process model itself, and the information it contains. We have also shown the multiple purposes of a software process model. The other sections described the related work on elicitation and on view modeling.

As we have seen in section 2.2, there have been many process model elicitation efforts. The methodological approaches taken are often deficient in the way they deal with multiple sources of information. Specifically, none of the elicitation approaches, to our knowledge, provide a comprehensive set of techniques for dealing with views and inconsistencies across the views. These inconsistencies are resolved manually and intuitively. Also, in some cases [RHM85, McB93, MHH94, BFL95], while elicitation steps are described (e.g., planning, information gathering, and modeling), there is not much technological support available during these steps. Similarly, general steps have also been presented for view-based elicitation [Rom93, Ver96], but with limited concrete technological support. Thus, in our research, our goal was to explore view-based methods further and provide technological support for them.

Where techniques and tools were available for information gathering steps [MHH94, SKV95, CoW95], we reused these as appropriate, permitting us to focus on the technically challenging parts of synthesizing a model based on multiple views.

In summary then, our research focuses on the problem of eliciting process views from multiple sources, finding and resolving inconsistencies across a set of views, merging them into a final model, and checking the quality of that model. Specific technological support is described to solve this problem.

Chapter Three - System requirements and their rationale

The technological support for view-based elicitation entails identification of a core set of requirements that the supporting system should satisfy. In this chapter, we describe such core requirements (R1 – R10), together with their rationale. These requirements have been separated in two categories: the first one contains requirements related to the modeling notation and structure, and the second one describes the elicitation tasks the system should support or automate.

Modeling notation and structure

R1: We should be able to enter process information, obtained from different sources, separately into the system, and maintain it as separate entities.

rationale: The information from each view should be kept separately because we will need to know "who said what" when finding and resolving the inconsistencies across views. It may be useful to keep the separate pieces of information as references even after the whole model has been elicited so that they can be revisited in future revisions to the model.

R2: The type of process information to be gathered should be user-definable.

rationale: Different organizations (or even different projects within an organization) may not use process models in the same way. For example, one organization may want to use a process model for guiding a new project, and thus needs information related to the activities to be performed, their inputs and outputs, and their ordering. On the other hand, another organization may want to use a process model to assess the throughput in the process, and thus will need to capture specific product and process metrics at particular points on the model.

R3: The scope of information (in terms of their types) that can be provided by each source of information or view should be user-definable.

rationale: In the same project, all the sources may not be able to provide the same kind of information. For example, a programmer may not be aware of the cost associated to his/her activities. Thus each view needs to be tailored to suit the specific context.

Elicitation tasks supported or automated

R4: The tool should be able to verify the individual views separately, for intra-view consistency.

rationale: Examples of inconsistencies within a view are: inputs to an activity are missing; an activity depends on the result of another one in the same view, but starts before the termination of the latter activity; etc. If the information in a given view is not consistent, the final model is also likely to be inconsistent. While it is possible to check the final model at the end, it is less efficient to do so on large models. Also, trying to merge inconsistent views can result in making bad decisions related to the resolution of inconsistencies across views.

R5: The tool should help identify process elements that are common amongst the different views.

rationale: Due to the fact that some process activities may involve multiple people, such activities would be modeled in all the related views. Also, communication does occur during the development process, and the interfaces among people are typically represented in all the corresponding views. If we do not know which process elements in one view correspond with which ones in the other views, we would not be able to merge the views.

R6: The tool should be able to detect inconsistencies in the different views.

rationale: Views, even though self-consistent, may not be consistent with one another. Thus, in building a common model, it may be problematic to merge such inconsistent views. In order to resolve such inconsistencies, we first need to pinpoint them.

R7: The tool should assist the user in selecting appropriate solutions in order to resolve the inconsistencies. (Each view presents one alternative solution to the inconsistency.)

rationale: Having some statistics such as the proportion of views having one solution can help in choosing the right alternative. The tool can definitely provide this information.

The solution selection process is based on the meaning of the information entered, so the user should be involved in this process. Resolving these inconsistencies is necessary in order to build one coherent model.

R8: The tool should be able to merge the views into a global model, based on the information in the views and the solutions to the inconsistencies.

rationale: The set of views contains complementary information as well as overlapping information. This information, when all merged, will form the whole model (assuming that the set of views was selected appropriately such that it covers the entire process). The tool would have enough information at this stage for merging the views.

R9: The tool should be able to check for the quality (consistency, completeness, etc.) of the global model.

rationale: We should ensure that the global model represents a connected process, that it is not just a set of unconnected views linked together.

R10: The tool should be able to verify a model against given development policies (e.g., that all documents should be reviewed independently). This implies the following two aspects of the requirement:

R10a - The elicitor should be able to formally specify development policies.

R10b - The tool should be able to verify a model against a user-defined policy.

rationale: The use of an elicited process model is limited unless some analysis is also performed on it, for feedback purposes. It is important to do at least an initial analysis immediately after developing a model, so as to determine the status of the current process. For example, the process may have changed over time, and verifying it against development policies would make explicit where the process differs from the policy.

These requirements have been discussed extensively, with several researchers from the software engineering lab at McGill, with several practitioners from industry, and with numerous visiting researchers, through system demos and presentations. Literature also mentions some of these requirements [Ver96, Rom93, MHH94], although they are not specified in the specific terms described above. Thus, as a starting point, these requirements can be considered valid for developing an elicitation system.

These requirements are associated to the six assumptions (see Section 1.2) as follows:

assumption A1	requirement R1
assumption A2	requirements R2 and R3
assumption A3	requirement R4
assumption A4	requirement R5
assumption A5	requirements R6 to R9
assumption A6	requirement R10

We recognize that these core requirements alone are not sufficient for building a complete system. Other requirements, such as those dealing with user interface, database, quality issues, portability issues, etc., are also needed. These are not described in this thesis, although we deal with these requirements explicitly in system implementation, including system documentation and validation.

Chapter Four - Modeling schema

In this chapter, we describe the notation and structure used for describing and representing process models in our system. Since such notation and structure is used throughout the elicitation process, especially in the major inputs and outputs of our system, their understanding is necessary before the presentation of the actual elicitation steps in the following chapters.

In order to accommodate the specific needs of each organization using the proposed elicitation approach, we require a flexible process model *schema* in which different types of process information are user-definable (requirement R2). Indeed, the information needed in a process model depends on the purpose for which the organization is going to use the elicited model. For example, for general information on a process, it may suffice that the process model describes activities, their ordering, and the artifacts produced in the process. On the other hand, for project planning and management purposes, additional information such as cost, quality controls, deadlines, duration of activities, and roles will be important.

In some cases, such as when the model is going to be used for specific process improvements, we may need to model the product (software) also, in conjunction with the process. Such information can be useful in analyzing product-process relationships and in making specific improvements. For example, we can model defect profiles during development, and then examine the related processes to determine the causes of the problems found. In order to model the product as well as the process, we need a "flexible" model schema where the product information needed (including metrics) can also be added.

The process and product information should be structured in such a way that we can visualize (or work with) a subset of the information. The reason is that the information in a model can be voluminous, and one needs to be able to focus on a specific part of the

model, or a specific type of information at a time, in order to comprehend the issues of concern. Two concepts are used for this purpose: (a) the *view*, that contains the information related to one agent (or one source of information) (e.g., the sub-process related to a designer), and (b) the *aspect* (e.g., *functional perspective*, as described in section 2.1), which is a subset of the types of information a model contains.

Some process information might be redundant within one model. For example, if a high-level design process yields a software architecture, then obviously, the entire design process also yields this architecture. In order to avoid eliciting and keeping redundant information, we can use algorithms to generate this information. Elicitation time can then be reduced, and model analysis can still be performed on the entire set of information (including the information that can be generated).

The next three sections describes the model schema used in our system (called *V-elicit*), the concepts of aspects and views, and the information generators, respectively. In Section 4.4, we show how a user can define his/her own modeling schema in V-elicit. In Section 4.5, we discuss the alternatives we had considered for this modeling schema, and the final choices. Finally, we summarize our modeling schema and compare it to current process modeling languages in Section 4.6.

4.1 Schema for process and product models

As explained in the previous section, an important requirement for this model schema is that it should be flexible and user-definable. We have chosen to use an entity-relationship (ER) structure, where the types of entities, relationships and attributes of the entities can be defined dynamically.

An example of such an entity-relationship schema is given in Figure 4. It is based on the literature describing the desirable entities and relationships that should be modeled

[BeD92, DeO92, ArK94]. The boxes represent the entity types (e.g., activity), and the edges represent the relationship types (e.g., activity produces artifact).

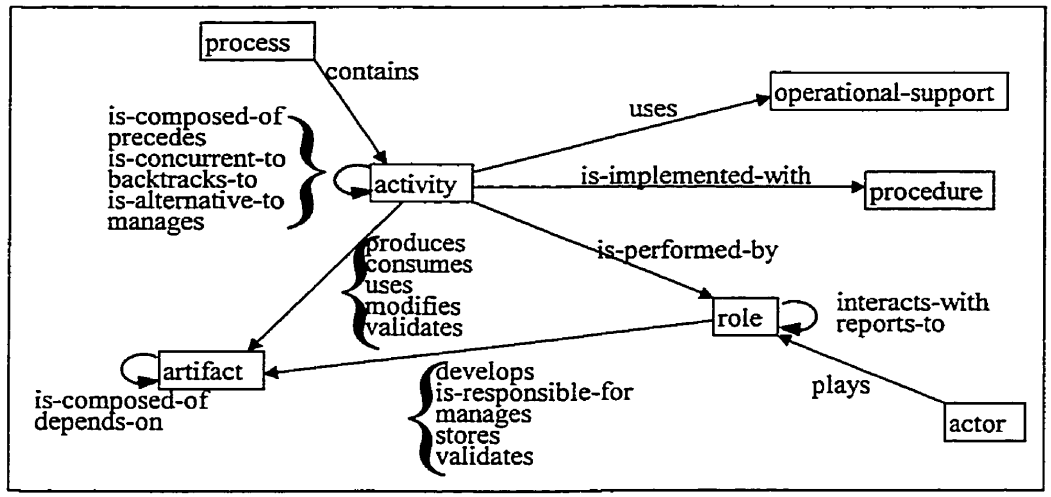


Figure 4 - An instance of the entity-relationship schema for process modeling

An entity of a given type can contain attributes, such as: cost, effort, and timing of an activity; the location of a resource; the identifier (or number) of a document in the configuration management tool; etc. Many types of attributes are allowed: *integer*, *float*, *character*, *boolean*, *string*, and *time*. These types have been chosen because they represent typical types necessary in process models. The attribute time is actually a 5-tuple (<year,month,day,hour,min>), where the value 0 for the rightmost elements means that we do not need this degree of precision. For example, the specification of a day should indicate values for the year, month, and day only (e.g., <1997,5,25,0,0>). Seconds are not represented in the type because they are generally not used in software processes.

One of the attributes of an entity can be a user-definable subtype. For example, we can differentiate between development activities, management activities, and quality assurance activities. By making this categorization, we will be able to perform better analyses, focusing on one subtype at a time if necessary. For example, one could analyze the time spent on quality assurance activities, one of the major cost driver of quality.

Having specified the types of information the model should contain, we can then create models such as the one in Figure 5, which represents a review process[3]. In this graph, the nodes represent the entities (boxes for activities, and ellipses for artifacts). The tree structure represents the activity decomposition (relationships of type "activity is-composed-of activity"). For example, the review activity contains three sub-tasks: preparation, meeting, and writing report. The roles shown besides the activities are also entities, and the relationship of type "activity is-performed-by role" is shown by the fact that the role is besides the activity it is performing. Duration of the activities (attribute) is also shown besides them. The inputs and outputs to the activities are illustrated via the arrows between the boxes and the ellipses. This information is shown at different levels of abstraction: for example, the relationship between "document" and "preparation" is represented again at higher level of abstraction in the relationship between "document" and "review". So a person looking at this model without looking at the details of the review activity will still see this interface with the other activities.

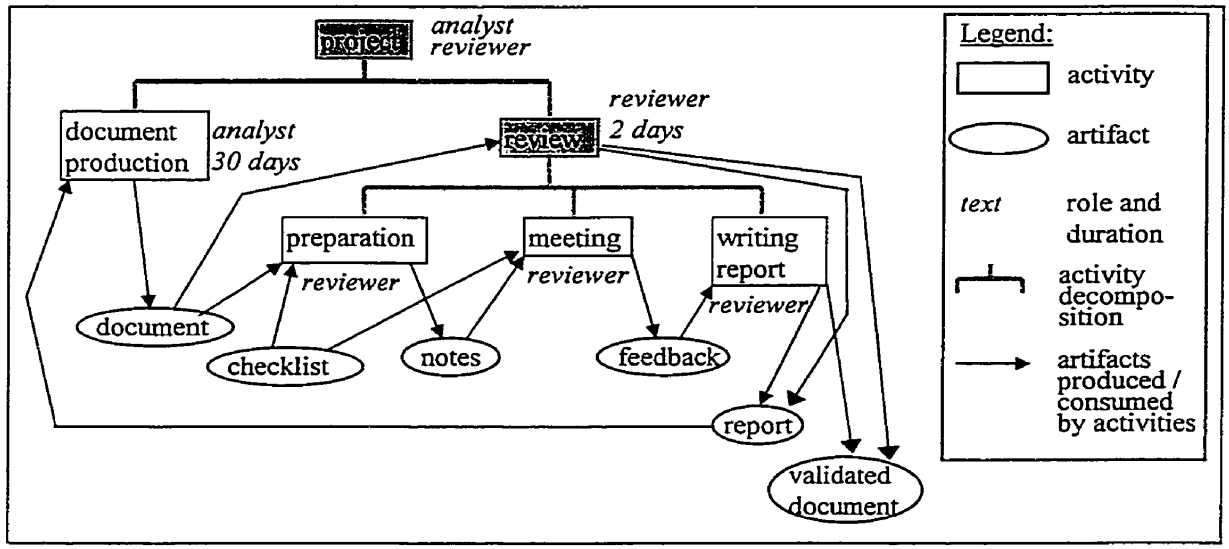


Figure 5 – Example of an entire model

[3] This review process is the same as the one described in Section 2.1. As a reminder, it starts when a document is submitted. The reviewers read it individually, and then discuss the problems in the document during the meeting. At the end of the meeting, feedback is provided in a report, which is sent back to the document production activity for making the requested modification (if necessary). The output is a validated document.

In this example, the behavioral information can be easily derived from the input/output flows. However, in many cases, such information need to be clearly stated. As an example, assuming that additional information describing the "meeting" activity in Figure 5 needs to be modeled, the behavioral information could be modeled such as that in Figure 6[4]. In this graph, the ordering of activities is shown through relationships of type "activity precedes activity", "activity is-concurrent-to activity", and "activity backtracks-to activity".

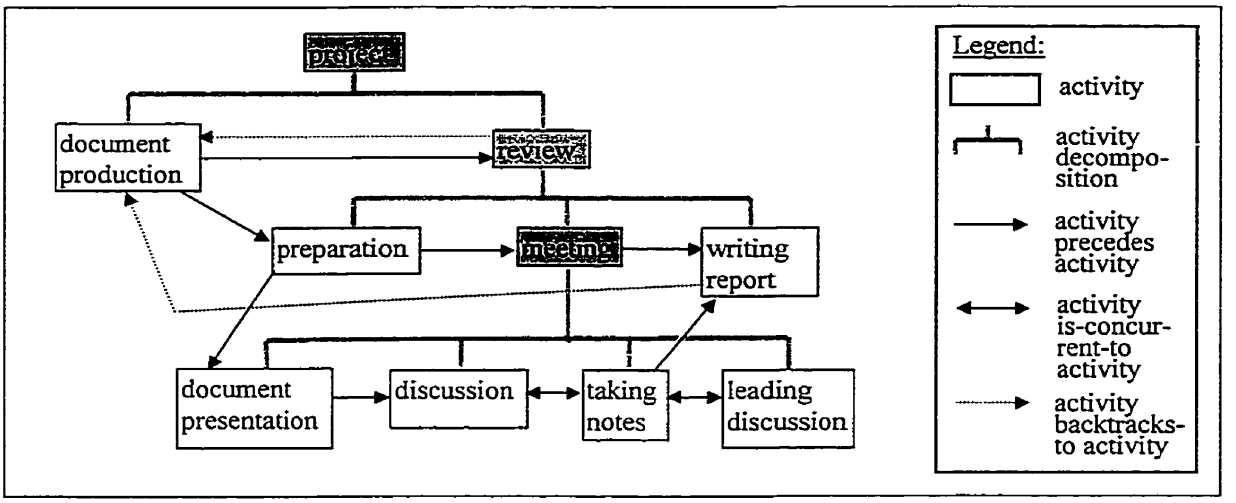


Figure 6 – Modeling behavioral process information

The modeling schema can be modified as needed by the user of the system. For example, one can add entities and relationships for modeling a product (illustrated in Figure 7) to the schema illustrated in Figure 4. The entity type "activity" is the same in both Figure 4 and Figure 7, and it is used to make the connection between the process information and the product information (through the relationship type "activity produces module").

[4] For graph simplification purposes, we did not redraw all the relationships from Figure 5, but such information is still assumed as part of the entire model.

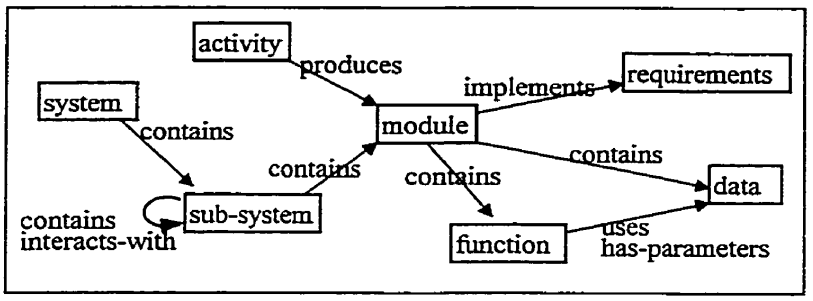


Figure 7 - An instance of the entity-relationship schema for product modeling

The different types of entities, attributes, and relationships are formally defined by the user as follows:

```
entity type =
      (
        name,
        {list of possible subtypes}
      )
e.g., ("activity", {"production", "management", "quality assurance"} )

attribute name =
      (
        name,
        type of value (integer/float/character/boolean/string/time),
        related entity type
      )
e.g., ("cost", float, "activity")

relationship type =
      (
        name as 3 words: <entity type> <relationship keyword> <entity type>,
        complementary type
      )
e.g., ("activity produces module", "is-produced-by")
```

Each type name is defined by strings. This permits one to make modifications to the modeling schema, and to have generic algorithms operating on process models containing any type of information.

In the case of relationship types, the string contains three words: the first one and the last one are the entity types involved in the relationship, and the middle word describes the meaning of the relationship. For example, the relationship type "activity produces module"

(see Figure 7) describes a relationship between an entity of type "activity" and another of type "module". The direction of the relationship is not important here: a complementary type is added to each relationship type definition, describing the relationship in the opposite direction. In the case here, the complementary relationship type of "activity produces module" is the keyword "is-produced-by", meaning that in the opposite direction, the relationship is of type "module is-produced-by activity".

The additional information specified in the entity type (list of subtypes) and the attribute name (type of value and related entity type) are used for type checking only.

4.2 Aspects and views

A large process model can have hundreds or thousands of nodes and relationships. In order to be able to visualize all this information, we need mechanisms to help focus on the desired parts of the model at a given time. The two concepts used for this purpose are: *aspects* and *views*.

An *aspect* comprises a subset of the types of information (entity/relationship/attribute) contained in a model. This subset is defined in an *aspect type*, using three lists:

```
aspect type =
      (
        {list of entity types},
        {list of relationship types},
        {list of attribute names}
      )
```

The user can define his/her own aspects, based on the ER schema s/he has defined. For example, an *information flow* aspect, containing only activities and the artifacts produced and consumed by these activities (with no attributes), can be defined as follows:

```
information flow =
      (
        {activity, artifact},
        {activity produces artifact, artifact is-consumed-by activity},
        {}
      )
```

Figure 8 shows the *information flow* aspect of the model in Figure 5. Notice here that the information flow aspect does not show the conditions upon which to terminate the review iterations: this would be shown (as an attribute) in an aspect showing the behavioral perspective of the model, such as a *control flow* aspect. This does not mean that we have an infinite loop of reviews.

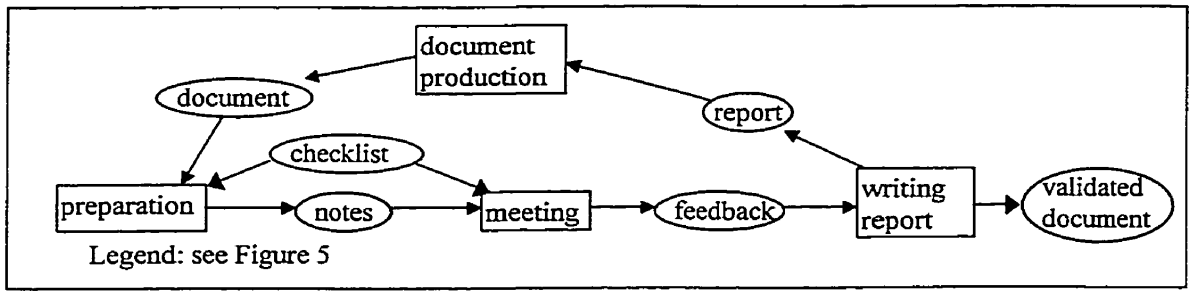


Figure 8 - Example of an aspect (information-flow aspect)

Other commonly used aspects are described as follows:

```
activity decomposition =
      (
        {activity},
        {activity is-composed-of activity},
        {}
      )
activity cost =
      (
        {activity },
        {},
        {cost}
      )
control flow =
      (
        {activity},
        {activity precedes activity, activity is-concurrent-to activity,
                    activity backtracks-to activity, activity manages activity},
        {pre-condition, post-condition}
      )
```

```
role assignment =
        (
          {activity, role},
          {activity is-performed-by role},
          {}
        )
tool usage =
        (
          {activity, tool},
          {activity uses tool},
          {tool_use_category[5]}
        )
```

This notion of aspect is also used in other process modeling languages. For example, Kellner [KeH89] has used different perspectives for visualizing process models (e.g., *functional perspective*[6], which is similar to our information flow aspect). This way, the process information that we see is less complex than if everything were shown at the same time. The difference in our approach is that the aspects are user-definable.

The second concept used for visualizing a part of the model only is the *view*. As briefed earlier, a *view* represents process information related to one agent, a subset of the roles/responsibilities of an agent, or a source of information such as a document. It has the same structure as a process model, with the same entity and relationship types (or a subset of them), but it has a reduced scope. For example, it may cover only some parts of a review process, and may not contain all the details. An example of a view for the model in Figure 5 is illustrated in Figure 9. In this analyst's view, we do not have the details of the review activity. We can assume that the analyst is aware of the fact that s/he should submit his/her documents for review, but is not aware of how exactly the review is performed. Also, notice that some types of information might not be provided by one view: in our case, the role assignment and activity duration information is not provided in this view.

---

[5] This metric is based on the categories of tool usage ("very low" to "very high") used in COCOMO 2.0 [BCH95]. Such categories are based for example on how much is the tool integrated with others or with the process. In COCOMO, the tool usage factor is used as a cost driver to help estimate the effort in a project.
[6] see Section 2.1.

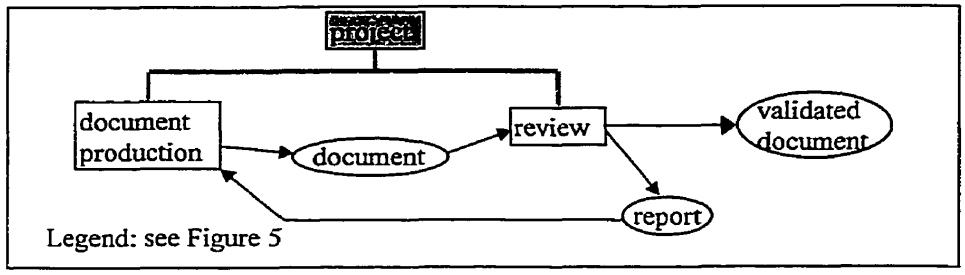


Figure 9 - Example of a view: the analyst's view

Other agents in the process might have different views. For example, the reviewer could provide additional details on the specific tasks performed during the review (see Figure 10), because s/he is involved in that process. On the other hand, a manager would probably know about the general activities, but not the technical details such as the information flow. However, such a person would typically have information on costs and schedules, as well as people assigned to each task. Figure 11 shows an example of the manager's view.

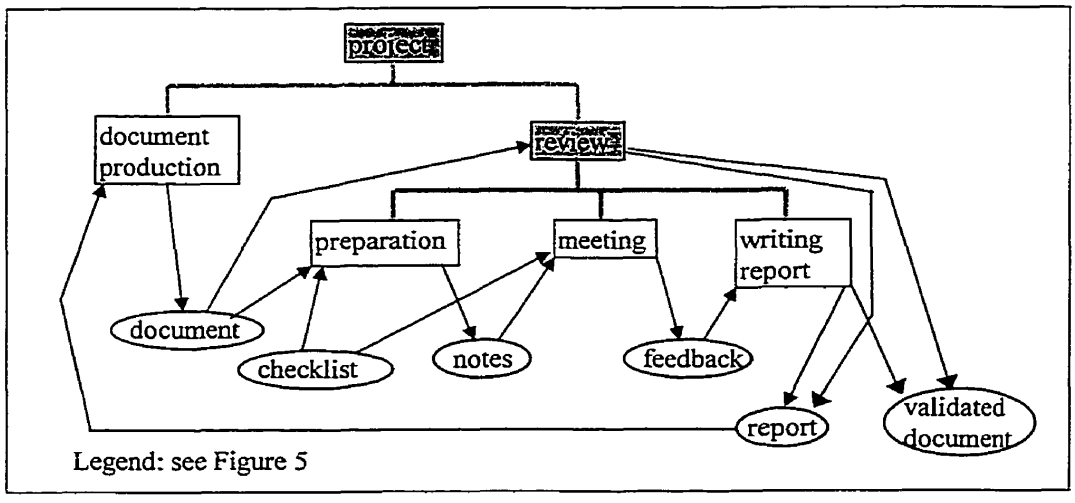


Figure 10 - Example of a view: the reviewer's view

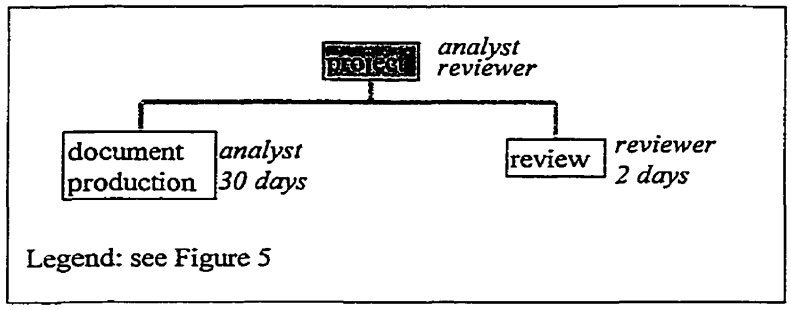


Figure 11 - Example of a view: the manager's view

The type of information a view may contain is user-definable (requirement R3, Chapter Three), through the specification of the view type. We define a *view type* as a subset of the aspect types it may contain.

view type = {list of aspect types}

The view definitions for our views above are as follows:

technical view type = { activity decomposition, information flow }

(view type used for the analyst's view and the reviewer's view)

managerial view type = { activity decomposition, role assignment, activity cost }

We can also extract different *aspects* of a view, just like for models. As an example, Figure 12 and Figure 13 show the *activity decomposition* aspect and the *information flow* aspect of the analyst's view in Figure 9 respectively.

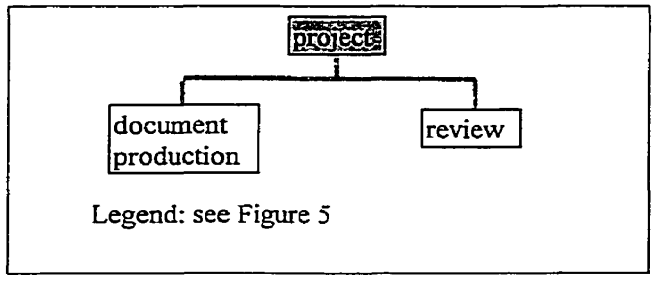


Figure 12 - Activity decomposition aspect of the analyst's view

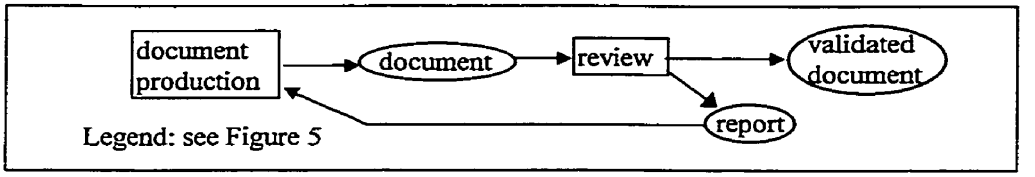


Figure 13 - Information-flow aspect of the analyst's view

Notice that these two concepts (aspect and view) are orthogonal, and that one can take a view of a given aspect (from an complete model), or an aspect of a given view. For example, the information flow aspect of the analyst's view in Figure 13 can be seen as an aspect of the analyst's view shown in Figure 9, or a view of the information flow aspect shown in Figure 8.

This combination of aspects and views permits one to visualize parts of a large process model at any given time, from a specific agent's point of view, and to focus on the desired kind of information.

4.3 Attribute and relationship generators

Often, desired information can be generated from other information contained in a model (or a view). For example, if the coding activity (represented by a specific node in a process model) produces source code, then the entire software process (represented by the root node of the model) also produces this source code. Yet, another example is that the cost of an activity is the sum of the cost of the sub-activities.

The above examples are related to *entity decomposition* in a process model, but we can also generate information from other kinds of relationships. For example, we can generate *artifact-to-artifact* dependency relationships by examining the activities that use some artifacts to produce others. For example, if the coding activity requires a design document in order to produce the source code, then a dependency relationship between the design document and the source code can be generated. Similarly, *role-to-artifact* or *role-to-role*

dependency relationships can be generated based on the information flow aspect (production and use of artifacts in activities) and the specification of the roles performing such activities.

It is important to note that the purpose of generating such relationships is to derive information from a process model to support technical and managerial decisions. The approach of generating information rather than explicitly representing it in a process model would save space (at the cost of computation), and reduce considerably human time and effort spent on eliciting redundant information.

We have identified two kinds of generators: *hierarchical* generators (based on the entity decomposition), and *linear* generators (for example the dependency relationships generated), which are described in the following sections.

4.3.1 Hierarchical generators

For the hierarchical generators, both relationships and attributes can be generated.

Generating relationships

Depending on the type of relationship, the way of generating the information at upper levels in the entity decomposition is different.

For example, if a designer is involved in the high-level design activity, then s/he is also involved in the entire design activity, and even in the entire project (see Figure 14). This type of generation is called "aggregation", because the relationships in one entity are the union of the relationships of the children entities. For completeness with the operations on sets, we have also defined the "intersection" type, but up to now we have not found any case where this type could be applicable.

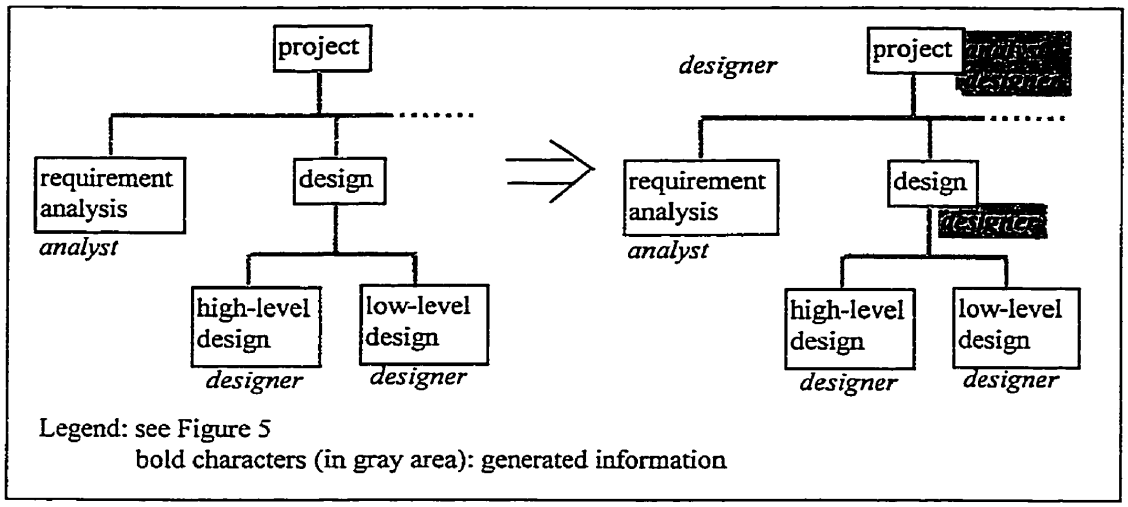


Figure 14 - Generating the relationship "activity is-performed-by role"

In some cases, the entities at upper level should have visibility of only those relationships to objects that are visible to other entities outside its subtree. For example, in the case of the ordering of activities, the precedence relationships among sub-activities are not important at upper levels (see Figure 15). The same happens with the flow of artifacts among activities. The temporary artifacts created by sub-activities, and used only by other sub-activities, should not be visible to the parent activity (see Figure 16). This type of generator is called "external", because it generates only those relationships to entities that are visible outside of the subtree.

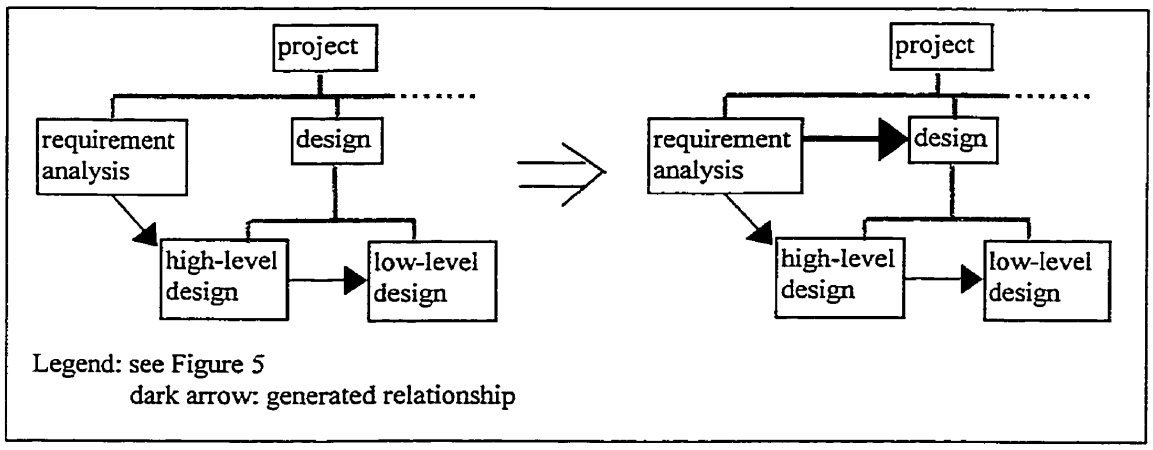


Figure 15 - Generating the relationship "activity precedes activity"

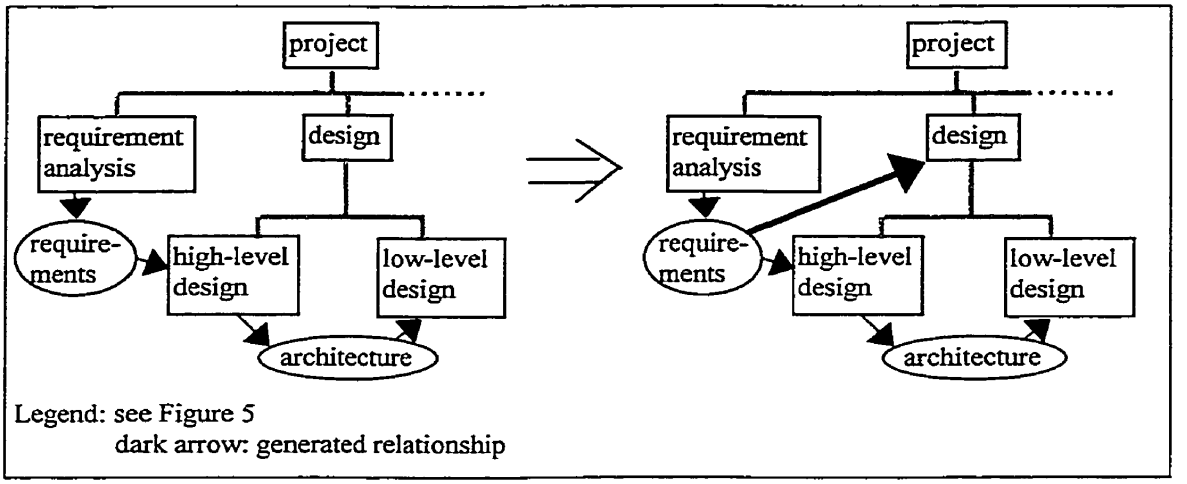


Figure 16 - Generating information flow relationships

It might happen that for a specific type of relationship, we do not want to generate it. For example, there are different levels of management, and the person managing a sub-activity (e.g., the design phase) is usually not the same as the one responsible for the whole project. So we cannot generate the relationship "leader manages project" from the relationship "leader manages design". In such a case, we use the dummy type of generator "none".

These different kinds of generators have been identified based on our experience with our modeling schema and peer reviews. They might still not be complete, but this does not affect the result of the elicitation process: we always have the choice of capturing all the relationships instead of generating them.

Generating attributes

As for the relationships, there are many ways of generating attribute values at upper levels in the entity decomposition. These depend on the type of attribute (number, character, boolean, string, or time), and also on the meaning of the attribute. Examples are provided below, with the different types of generators.

For identifying the different types of attribute generators, we need to look at the possible operations on these attributes, applicable to a set of values (one value per child entity). These operations should be commutative in order to be applicable to a set of values that are not ordered (the ordering of children for a parent entity is not meaningful).

In the case of numbers, the standard commutative operations are the "add" and "multiply". We can also use the operations on sets of numbers ("max" and "min") related to the comparison operations. So this gives us four types of generators: "sum", "product", "max", and "min".

An example of an attribute that can be generated using the type "sum" is the cost of activities (see Figure 17). Assuming that the design activity is composed of the high-level design and the low-level design (and nothing more), and that it costs 4000$ for producing the high-level design, and 12000$ for the low-level design, then the design activity composed of these two activities costs 16000$.

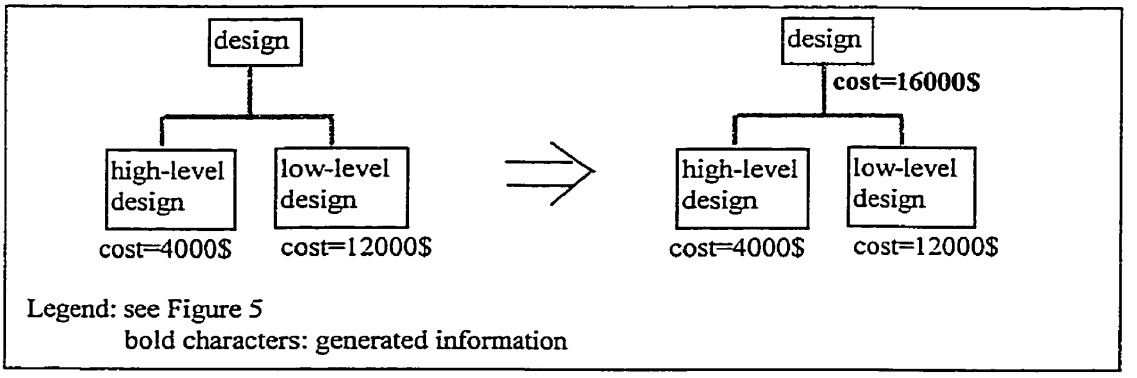


Figure 17 - Generating the cost attribute

The comparison operators ("max" and "min") could be used when the number represents a scale (usually for qualitative measures). For example, lets assume that we have an attribute expressing the level of stability of different parts of a software on a scale [1..5] (1 meaning "very unstable", and 5 meaning "very stable"). If all parts of a software are stable (4 or 5 on the scale) but only one part is really unstable (1 on the scale), then the overall software might be considered as very unstable as well. So in this case, the attribute

for the overall software would be generated as the minimum of the values related to each of its parts.

Up to now, we have not identified cases where the "product" type of generator could be used.

Logical operators should be used for boolean attributes. The only two such operators that can be applied to a set of values at the same time are the "and" and "or" operators. An example for the "and" operator is the attribute "validated" for artifacts: an artifact is considered as validated only if all of its subparts have been validated as well. For the "or" type of generator, an example can be the attribute "risky" for activities: an activity is risky if any of its sub-activity is.

For the other types of attributes (i.e., *character*, *string*, and *time*), the only types of generators allowed are the "min" and "max", except for the string attributes, which cannot be generated. For example, in the case of time attributes, a "min" generator should be used for a start time (an activity starts when the earliest sub-activity starts), and "max" generator can be used for end time (an activity finishes when all the sub-activities are finished).

As for relationship generators, we also allow the use of a "none" type of generator, when an attribute can not be generated. For example, the duration of an activity is not necessarily the sum of the duration of the sub-activities, because these could be performed concurrently. Of course, this information could be generated by some more complex techniques such as PERT/CPM, but the goal here was to identify generic types of generators that could be applicable to many attributes. It is always possible to avoid generating the values, and instead capture them all in the model.

4.3.2 Linear generator for relationships

Some kinds of information can also be generated from relationships other than the entity decomposition as in the previous section. For example, artifact dependency relationships can be generated from the information on the artifacts used by activities for producing other artifacts (i.e., the relationships "activity produces artifact" and "artifact is-consumed-by activity"). Figure 18 illustrates such generated relationships. This kind of generator is used for creating new types of relationships based on the existing ones, avoiding the elicitation and storage of such relationships.

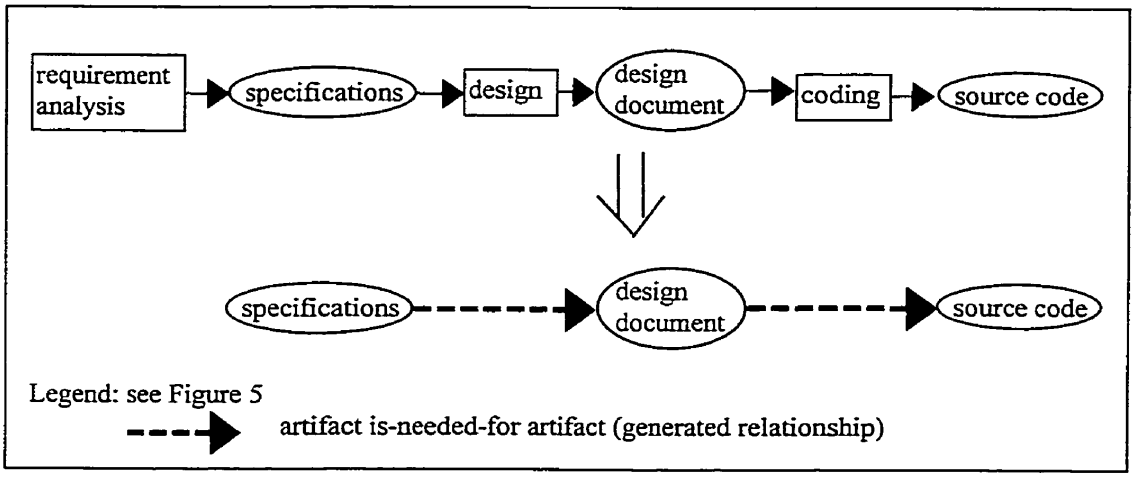


Figure 18 - Generating dependencies from information flow aspect

Notice that in our example above, the specifications could be further decomposed into separate requirements, and the design could be decomposed into specific design modules. In such case, the artifact dependency relationships generated would actually reflect the trace between each module and its requirements. Similarly, such traceability could be obtained for code modules and test cases.

For this kind of generators, we cannot just use types of generators, such as for hierarchical generators, for specifying how the relationships should be generated. The specification should include the relationship types involved, and the relationship type created. In our example above (Figure 18), this specification would be:

existing types: artifact is-consumed-by activity
activity produces artifact
new type: artifact is-needed-for artifact

This means that if we have a relationship of type "artifact is-consumed-by activity" from an artifact D1 to an activity A, and a relationship of type "activity produces artifact" from this activity A to another artifact D2, then we can generate the relationship of type "artifact is-needed-for artifact" from artifact D1 to artifact D2. As an example, the relationship "specifications is-needed-for design document" can be generated from the relationships "specifications is-consumed-by design" and "design produces design document".

The ordering of the relationship types specified is very important. The following structure must be used in the specification of the generator:

existing types: A <relationship_type> B
B <relationship_type> C
new type: A <relationship_type> C

The common entity in both existing relationships (B above) should always be the second one in the first type, and the first one in the second type. If this causes a problem (i.e., the relationship types are defined from B to A, from C to B, or from C to A), we can use the complementary type of a relationship type. For example, the following generator definition does not have the proper structure:

existing types: activity is-performed-by role
activity produces artifact
new type: role develops artifact

but we can use the complementary type of "activity is-performed-by role" ("role performs activity") to fix the problem[7]. So the correct generator definition would be:

existing types: role performs activity
activity produces artifact
new type: role develops artifact

[7] See Section 4.1 for more details on the relationship types and their complementary type.

This concept of linear generators using two existing relationship types can be generalized to any number of existing types. For example, one may want to generate dependency relationships among roles: a person requiring an artifact to do his/her task depends on the person developing this artifact. Here is the specification of the generator that would be needed:

existing types: role performs activity
activity uses artifact
artifact is-produced-by activity
activity is-performed-by role
new type: role depends-on role

Such multi-types generators can be specified using the regular generators (containing only two existing types), by specifying intermediate types that are used in the following generator. For example, the above multi-types generator is specified using the following three regular generators:

1. existing types: role performs activity
activity uses artifact
new type: role uses artifact
2. existing types: role uses artifact
artifact is-produced-by activity
new type: role depends-on activity
3. existing types: role depends-on activity
activity is-performed-by role
new type: role depends-on role

Generating the final type is then performed by generating the intermediate types in the sequence above. The system can find such sequence of generators based on the available types and the final type to be generated.

4.3.3 Summary

As we have seen in the previous sections, some information can be generated from a process model. We can use the entity decomposition structure for generating new relationships and attributes at upper levels of abstraction (hierarchical generators). Relationships of new types (e.g., "artifact is-needed-for artifact") can also be generated using other existing relationships (e.g., "artifact is-consumed-by activity" and "activity produces artifact"). These generators save significant human time by not having to elicit and keep redundant information in a process model.

The use of these generators in our elicitation system, V-elicit, is shown in Chapter Six.

4.4 Defining types in the V-elicit system

Each type of information (entity types / relationship types / attributes / aspect types / view types) to be used in the different views and models should be defined prior to their use in the elicitation process. It is not possible to specify a model without an underlying modeling schema. Typically, the schema would be defined prior to the elicitation process, but modifications can be made dynamically during such process when new needs are discovered.

Since these types are usually similar from one process elicitation effort to another, their definition is stored in a library of types. During a specific elicitation effort, the elicitor can select the types required in his/her situation from the list available in the library.

For each concept defined in sections 4.1 (entity, relationship, and attribute) and 4.2 (aspect and view), there is a list of user-defined types in the library, with the possibility to add more types, or to view/modify the characteristics of these types.

For example, in Figure 19, we can see the list of entity types and the subtypes related to the entity type highlighted. Buttons are provided below for adding new types and subtypes. Deleting types is not allowed for security purposes (i.e., if a model is still using that type). By double-clicking on one type or subtype, a window such as in Figure 20 appears, showing the information related to that type. In our case, the information related to an entity type is its name and a textual description of this type (if necessary). All information can be modified (and saved using the left button), except the name. A similar window is used for defining a new type.

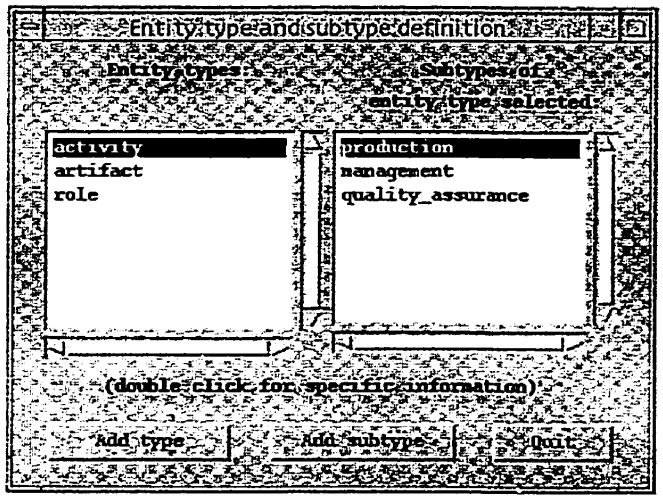


Figure 19 - List of entity types defined

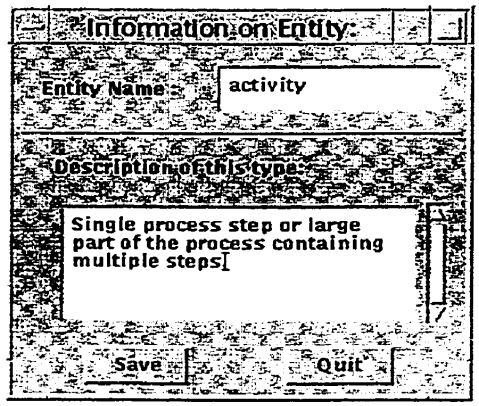


Figure 20 - Specification of an entity type

For relationship types and attribute names, similar windows are used (one with the list of types, and one showing the information on that type). In the case of relationship types, the following information should be defined through the specification window: the type itself and its complementary type, and the type of generator to be used (for hierarchical generation of relationships). For attribute names, the following information is needed: the name, the type of the attribute (integer, float, boolean, character, string, or time), the type of hierarchical generator to be used, and the entity type that can have such attribute (e.g., the related entity type of the "cost" attribute can be "activity").

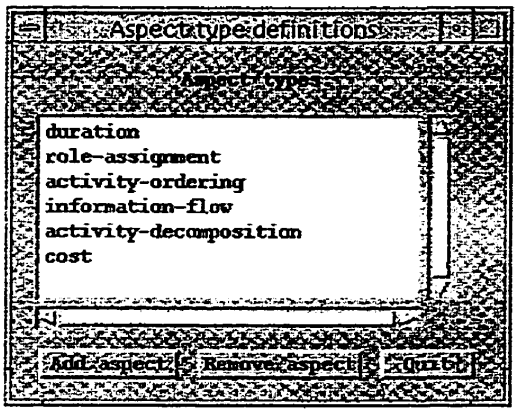


Figure 21 - List of aspect types defined

Having defined the basic entity, relationship, and attribute types, we can then use them to define the set of *aspects*[8] to be used. The first window shown just lists the aspect types (Figure 21), permitting one to add more (or delete some) through the buttons at the bottom of the window, or visualize/modify aspect types by double-clicking them.

The definition of the aspect type in term of entity types, relationship types, and attribute names is performed through a window such as that in Figure 22 (which defines the aspect type *activity decomposition*). The left part contains the available entity/relationship/ attribute types from the library. These can be selected for the aspect type being defined, by double-clicking them (which moves them to the right part). Double-clicking on the types in the right part removes a type from the definition of the aspect type. These selection operations can also be performed through the buttons in the middle part.

[8] As a reminder, an aspect (see Section 4.2) is a subset of entities, relationships, and attributes types found in the entire model.

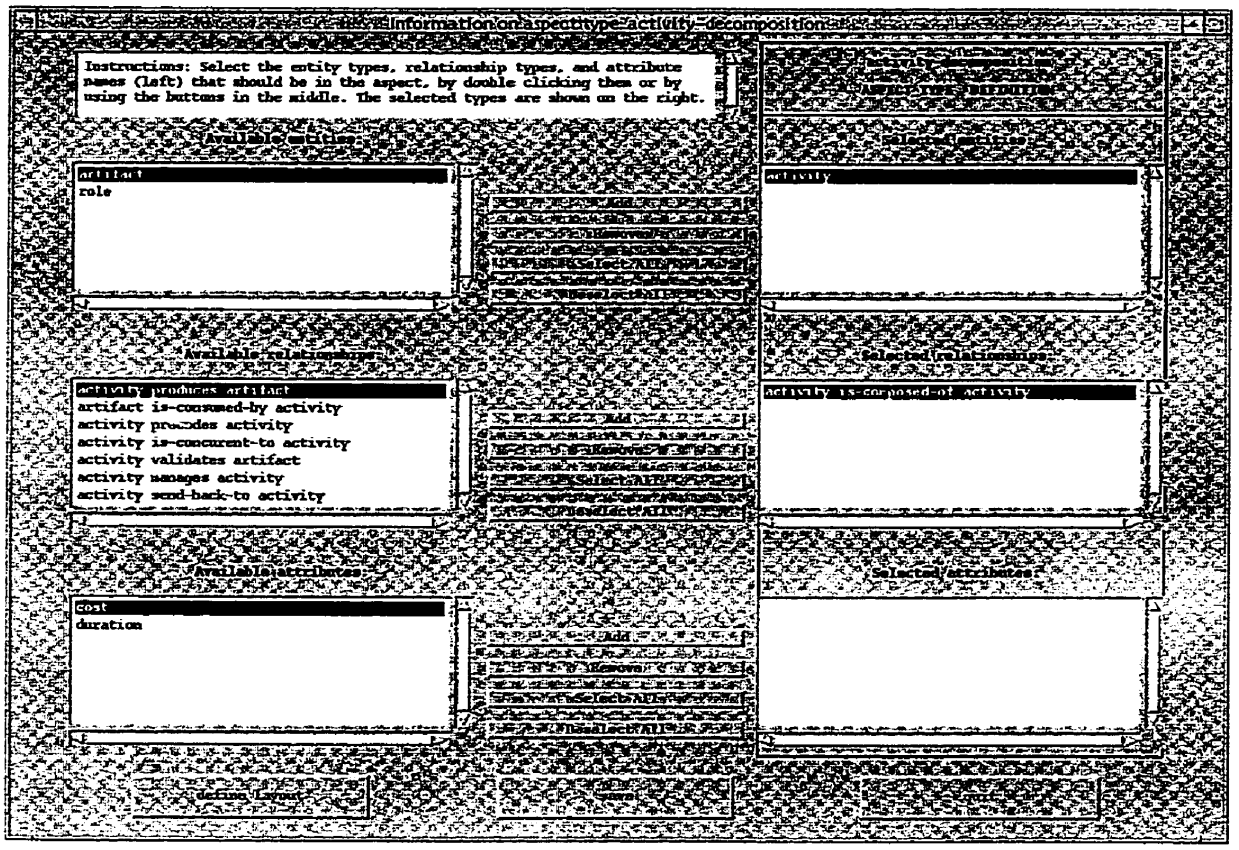


Figure 22 - Definition of an aspect type (activity decomposition)

The button "define layout" at the bottom of the window permits one to define the way an aspect will be presented (as a graph) to the elicitor or the users. This is the specification of the graphical notation to be used for each aspect. It is defined within the aspect type, in order to use a consistent graphical notation across views. For example, in Figure 23, the graphical notation to be used for the "activity decomposition" aspect is the following: the activity is represented as a black rectangle, and the relationships of type "activity is-composed-of activity" is represented as a black solid line. This can be changed by selecting a new shape for an entity type or relationship type, and then clicking on the specific button "change". When the aspect contains more than one entity type or relationship type, the style selected on the right is the one related to the type selected on the left.

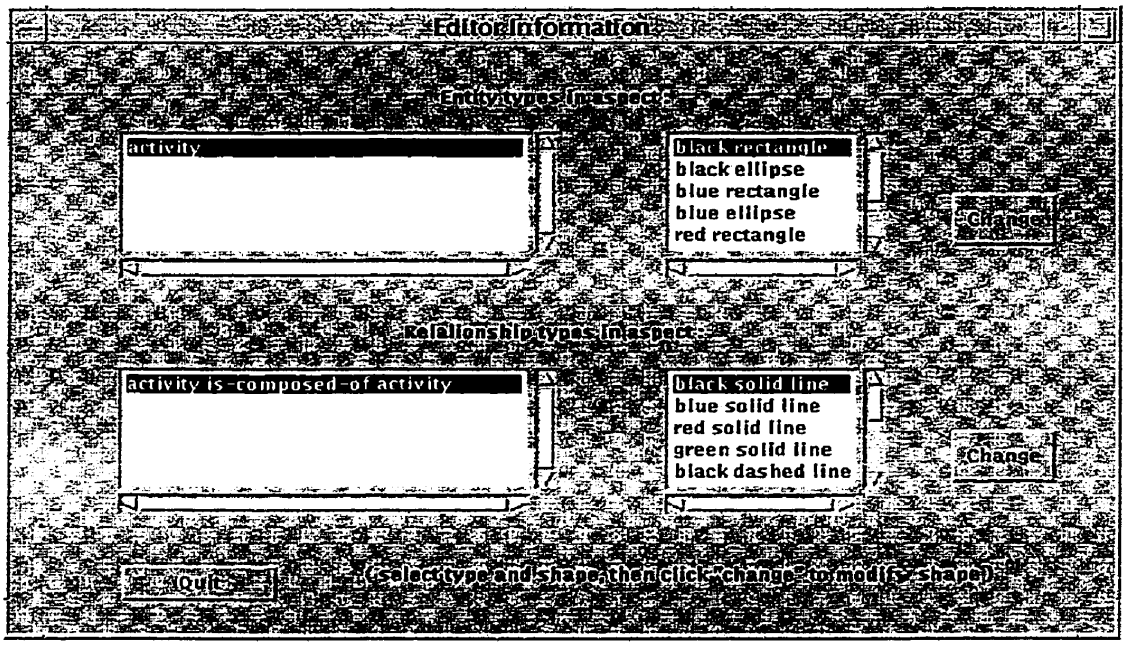


Figure 23 - Definition of the aspect layout for visualization

For defining view types, similar windows as for the aspect type are used: one containing the list of view types, and one for selecting the aspect types contained in the view type being defined. No layout information is needed because it is part of the aspect type information.

4.5 Alternative data structures rejected for the schema

As we have seen in the previous sections, the modeling schema used in our system is based on entity-relationship diagrams. This permits one to have a user-definable schema, that can suit the specific needs of different organizations or even different processes within an organization.

Most of the current process modeling languages do not allow such user specification of the schema to be used: the user is constrained to use the notation provided with the language, even if it does not provide exactly what they need. For example, one might need

both actor dependency relationships (as in [YuM94]) and control flow information (as in Statemate [KeH89]) in the same process model. However, this is usually not possible in other modeling languages. In the few languages where this is possible, the cost of such flexibility is a lack of automated analysis of the model. For example, the X-elicit tool built previously at McGill University [MHH94] permits one to define the modeling schema, but no analysis is possible because of its attribute (textual) structure. Also, few languages allow comprehensive modeling of both process and product information. For this reason, we rejected the idea of using existing languages.

However, other data structures could have been used instead of the entity-relationship diagram (ERD) structure. Because of the generally hierarchical structure of the process information, we could have structured the entities in a tree structure. The problem with this structure is that the elicitor is constrained to model a process or a view in a top-down fashion, and is not free to model the information in the manner in which it is gathered. This is not always a natural way of specifying the process for the people providing such information.

Another solution would be to use an object-oriented structure instead. This structure seems to be easier for generating relationships using the "is-a" type of relationship (through inheritance). However, having some special types of relationships ("is-a" in this case) increases the complexity of the most time-consuming algorithms by having to deal with different implementations of the different relationship types. For this reason we have rejected such a structure.

Entity-relationship diagrams have already been used in software process modeling [Pen89, Gal92, ADH94], but often for products only, or for functional descriptions only. But Feldman and Fitzgerald have shown that we can model behavioral information (facts and rules) using ERD [FeF85]. Thus, this structure is suitable for behavioral modeling of a process.

4.6 Summary and analysis of the modeling schema used

In this chapter, we have presented the modeling schema that is used in our software process elicitation tool, as well as how the information can be structured (by aspects and views), and how to generate information from the existing one. Our modeling schema is based on the entity-relationship diagram structure, and is user-definable, meeting our requirements R2 and R3 in Chapter Three.

The property of being user-definable cannot be found in most of the current process modeling languages. In the cases where it is user-definable, the modeling tool does not perform automatic analyses on the information modeled.

It is to be noted that for model *presentation* and *editing* purposes, the ERD is probably not the best tool. It lacks formal notation (e.g., to specify a precondition in a mathematical way), and presentation conciseness. Other higher level existing tools such as the ones described in Appendix D (e.g., APEL, Statemate, etc.) could overcome these problems. However, as an *internal representation* of a model, ERD is flexible, and can be linked to other presentation tools. For this purpose, a suitable translator would be necessary. Since this issue does not affect the elicitation techniques developed, it does not fall within the scope of the work for this thesis.

A limited portion of the ERD features have been used so far into our modelling schema. In particular, n-ary relationships, cardinality specification, and attributes of relationships have not been implemented yet. Such features have not been required for the types of information modelled during our research. Since these were not necessary for the validation of our research hypothesis, we decided to postpone their implementation.

The models and views built using the described modeling schema are used as inputs and outputs of our elicitation process, as described in the next chapter.

Chapter Five - Elicitation approach and scenario

In this chapter, we describe our proposed elicitation approach that meets our requirements stated in Chapter Three, through a demonstration of our prototype system (called V-elicit). Subsection 5.1 presents the overall approach, the different steps, and the links between them. In subsection 5.2, each step is described using an example. Finally, the last subsection summarizes the approach and techniques used. The details and algorithms of the techniques developed are presented in Chapter Six.

5.1 Overall approach

The key elicitation steps of our approach are depicted in Figure 24[9]. In Table 4, each elicitation step is mapped to a technique developed in our system or to an external tool used in our approach.

The purpose of step 1 (plan for elicitation) is to understand what should be elicited. We should know the boundaries of the process to be elicited, the kind of information to be elicited, the level of details needed by the users of the elicited process model, and who/what can give us the process information required (e.g., agents, existing process documentation, etc.) (requirements R2 and R3).

[9] Notice that the notation used here for *presenting* this elicitation process is not an entity-relationship diagram, although ERD could have been used, showing an information-flow aspect (the steps are activities, and the text on the arrows are artifacts). As explained in Section 4.6, other notations such as the one used here can be more concise and easier to understand.

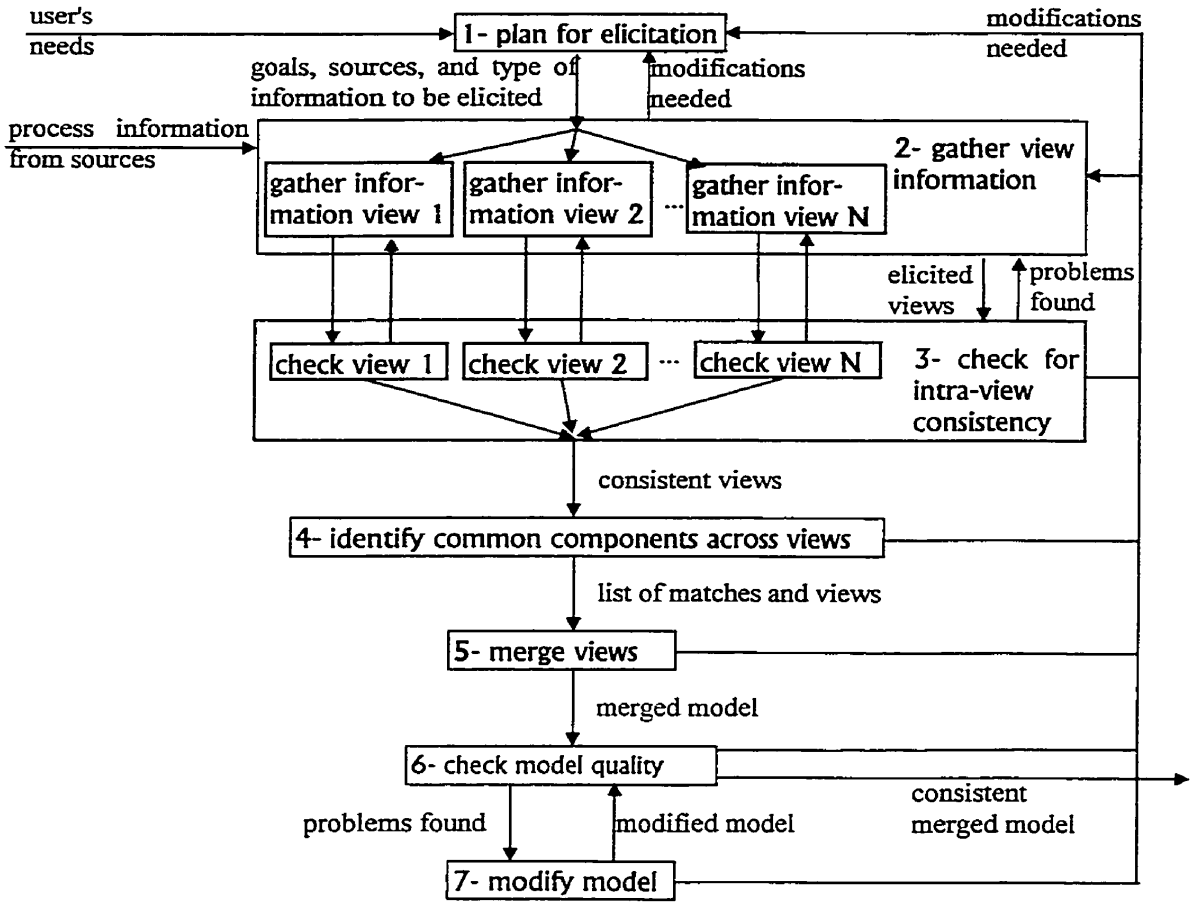


Figure 24 - Elicitation steps (dataflow)

elicitation step	technique / tool
1- plan for elicitation	elicitation planning
2- gather view information	process information editor (tool: X-elicit and Dotty)
3- check for intra-view consistency	constraint verification
4- identify common components across views	component matching
5- merge views	view merging
6- check model quality	constraint verification
7- modify model	process information editor (tool: X-elicit and Dotty)

Table 4 - Techniques / tools associated with each elicitation step

Using this information, in step 2 (gather view information), we can then gather the information from different sources, using a process information editor (requirement R1). There is one instance of such an editor for each view. We are using the existing *X-elicit* system developed at McGill, and the graph visualization tool *Dotty*, for this purpose. In step 3, we should check for intra-view consistency for each view developed (check for intra-view consistency), iterating with step 2 as necessary (requirement R4). Again, there is one instance of the consistency check step for each view. Notice that each view can be treated separately, and that we do not need to wait until we have all the views elicited to start checking them.

Once all the views have been modeled and checked, we then need to merge them into a final model. However, we should first find the common elements in the different views, in step 4 (identify common components across views) (requirement R5), so that in step 5 (merge view), we can detect and resolve the inconsistencies across the views, and merge the views incrementally into a final model (requirements R6, R7, and R8).

In step 6 (check model quality), we check the quality of the final model, and iteratively make modifications if necessary in step 7 (modify model) (requirement R9). The model is then checked against development policies (requirement R10), for providing feedback.

At any time after the "gather view information" step, the elicitor can return to this step for modifying the views. Views can also be added or removed at any time, or the type of information to be elicited can be modified, by backtracking to the "plan for elicitation" step. Whenever the views are modified, they should go again through the process of checking intra-view consistency (step 3), view merging (steps 4 and 5), and model verification (steps 6 and 7). The techniques used in such cases are not different than the ones used when these steps are performed for the first time.

The system developed (*V-elicit*) is aimed to show all these elicitation steps and their ordering. The techniques and tools used in all these steps are described in subsection 5.2,

through an example. It should be noted that each of these steps uses the modeling schema presented in Chapter Four.

5.2 Scenario for each step

This section describes the techniques and tool support listed in Table 4, used in the steps identified in Figure 24. The example used throughout this section to demonstrate the different techniques presented is a "system analysis process", containing some documents to be produced and different levels of reviews. Three views are used:

- Bob: an analyst, who knows which documents are to be produced because he is involved in the production of those documents, but he has a weak knowledge of the review process.
- Peter: the manager, who has a broad knowledge of the process, but does not know the details.
- William: a reviewer in the IT team, who can provide details of the review process (especially the part where he is involved), but cannot tell about the document production part or the other levels of review.

Figure 25, Figure 26, and Figure 27 show the information that Bob, Peter, and William provided respectively during their interview[10]. The figures shown here do not contain the entire information related to each view (for simplicity of the graphs). The reader should refer to the Appendix A for the complete information.

[10] This is the information each of the three persons knows, even before the elicitation process begins. That is why this information is not presented in the form the elicitor would see it in V-elicit. The step of putting this information into V-elicit is described later in this section.

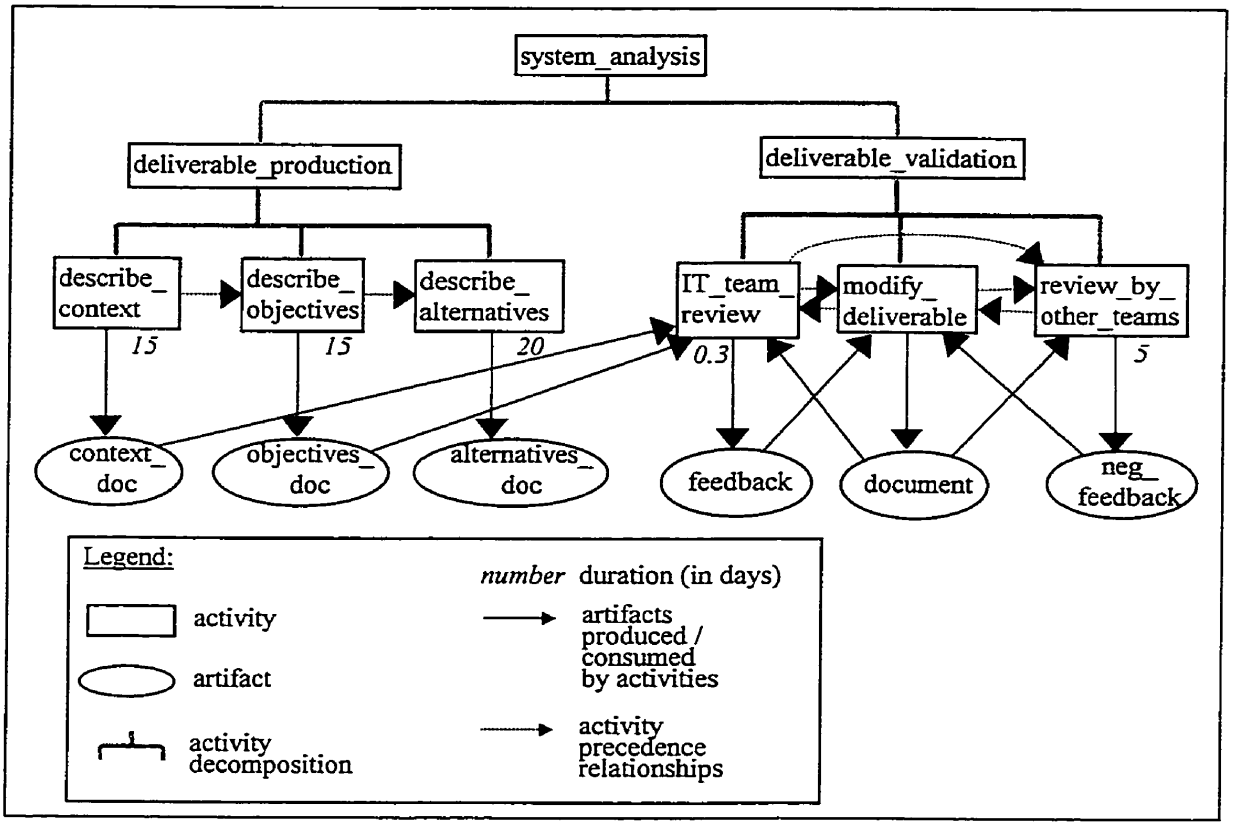


Figure 25 - Bob's partial view

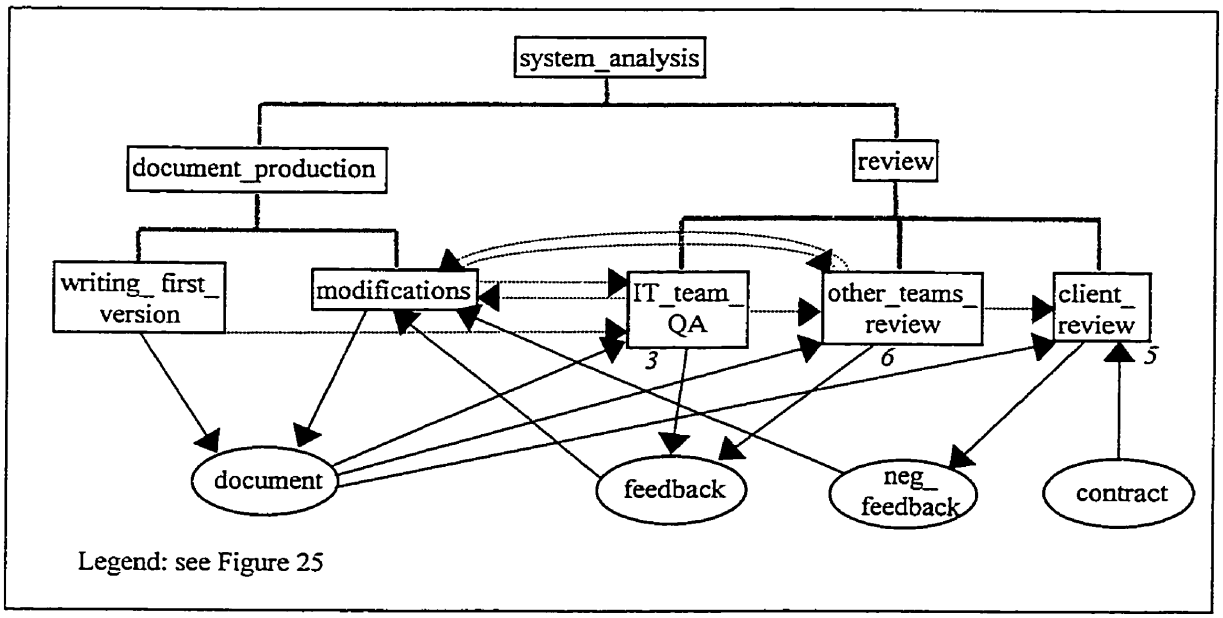


Figure 26 - Peter's partial view

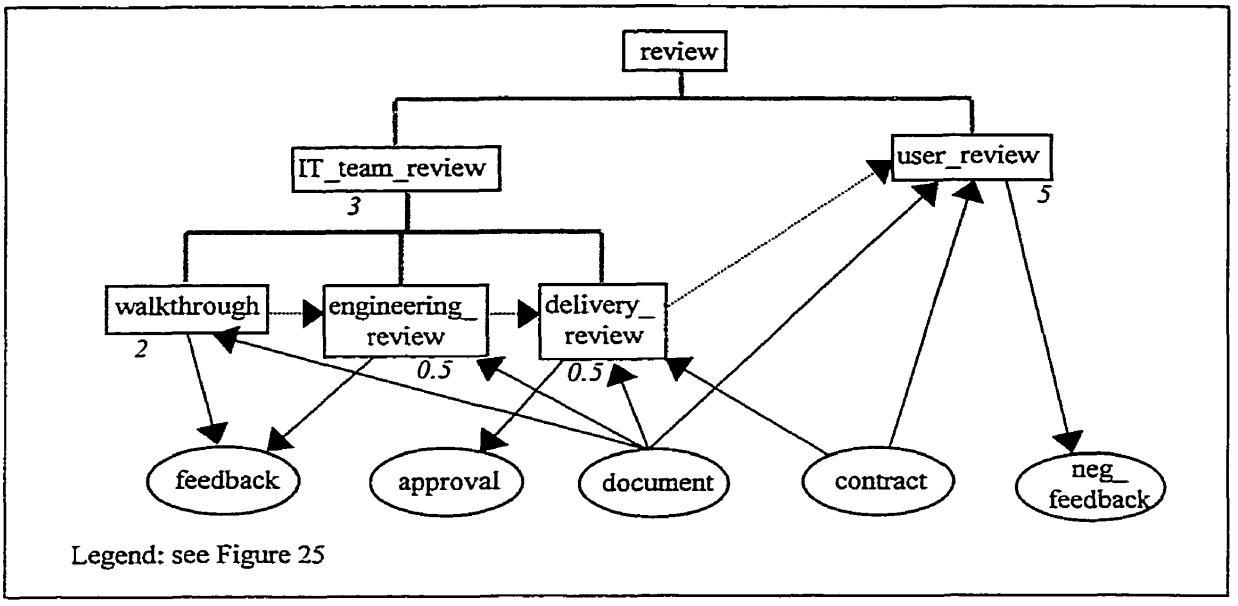


Figure 27 - William's partial view

By examining the three views presented above, it becomes readily clear that the process for merging these three views into one model is not trivial.

In V-elicit, we should first create a new project to elicit, or select a project being elicited. This is performed in the left part of the window shown in Figure 28. On the right part, we can specify some global characteristics of this project if necessary. An initial set of characteristics is provided, but one can add or delete some using the buttons at the bottom of the window. These characteristics do not affect the elicitation process. They can be used for project categorization and analysis.

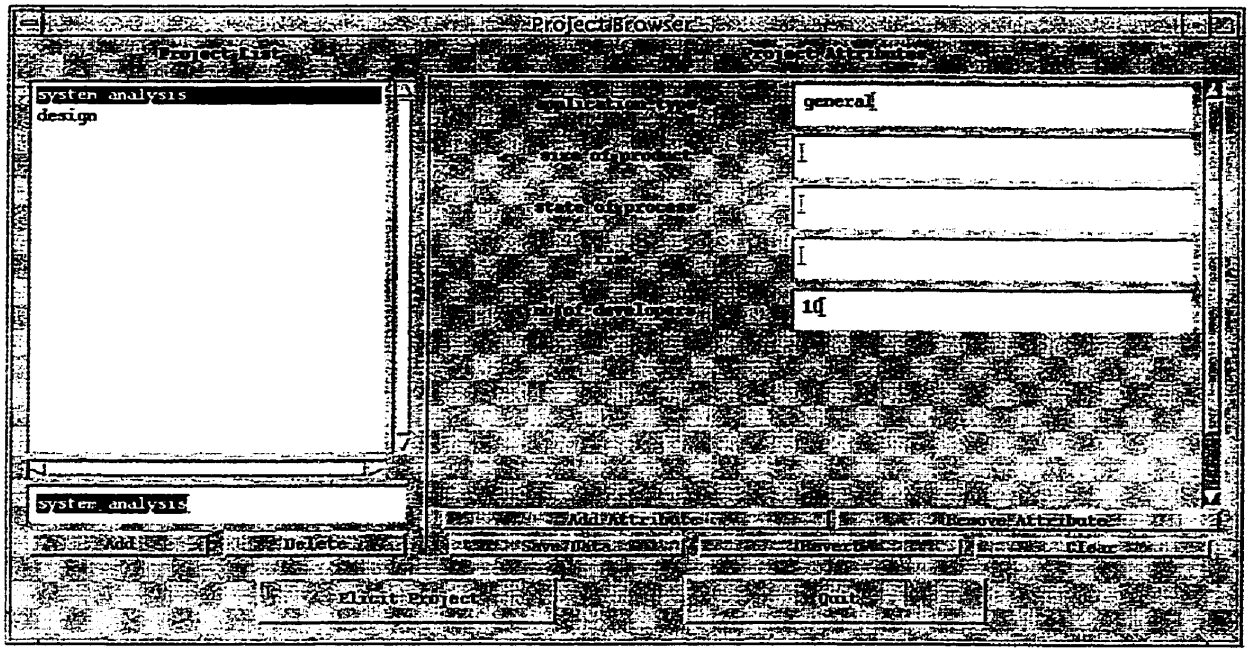


Figure 28 - Creating or choosing a project for elicitation

When clicking on the "elicit project" button, the steps to be performed are presented in a window such as that in Figure 29. Each of the steps shown are described in the following sub-sections. Section 5.3 summarizes our elicitation approach. Notice that the system presents the ideal ordering of the steps using arrows, but this ordering is not enforced. Backtracking, as described in Figure 24, can be done by selecting the appropriate step. Also, for an expert elicitor, this is more convenient for going back and forth in the process, or for skipping some steps and trying others informally.

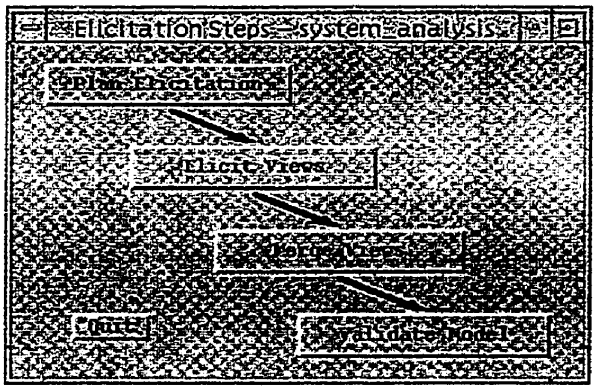


Figure 29 - Steps for the elicitation process

5.2.1 Step 1: Plan for elicitation

There are three sub-steps in elicitation planning: defining elicitation goals, list the potential sources of information, and choose the sources of information to use. These are shown in a window such as in Figure 30, opened when clicking on the "Plan Elicitation" button in Figure 29.

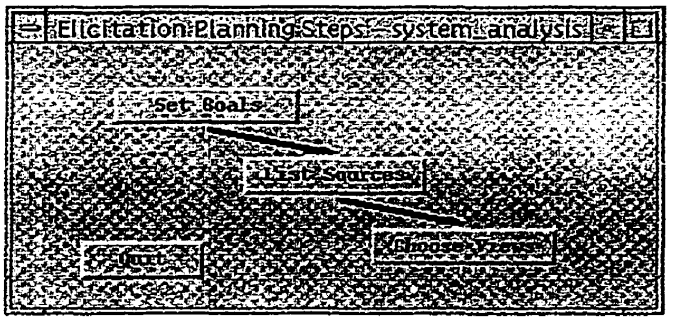


Figure 30 - Steps for planning the elicitation process

When defining the elicitation goals, we should select the type of information (*aspects*) that we need in the final model. This is performed in the window shown in Figure 31, opened from the first button in Figure 30. In the top-left corner of the window, there is a list of aspects that are available for selection. By selecting one or more desired aspects from this list, they are moved to the "selected aspects" list in the top-right corner. This information will be used in later steps, to ensure that the view information covers all the aspects and that these have been considered in merging the various views. The system also guides the elicitor in collecting some other useful information such as the *scope* of the process to be elicited, the *level of details* needed, who is going to be the *user* of the elicited model, etc. (see lower part of Figure 31). The template for entering this kind of information can be modified to fit the elicitor's needs, through the "add goal" and "remove goal" buttons at the bottom. Buttons are also provided for saving information, coming back to the last information saved, and clearing all data entered.

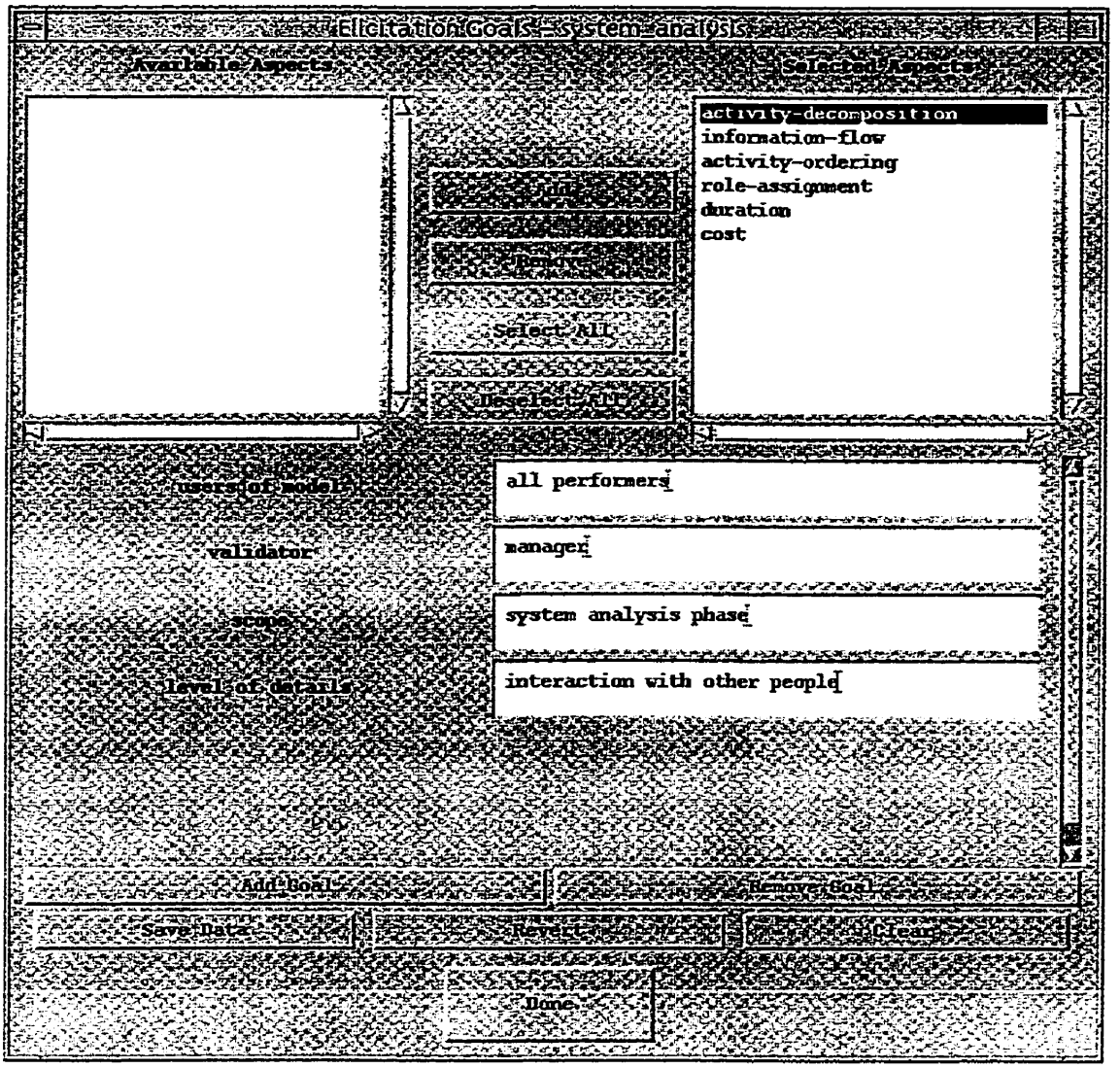


Figure 31 - Specifying elicitation goals

In the second elicitation planning sub-step (see the "list sources" button of Figure 30), the elicitor should list the potential sources of information, and the roles and responsibilities in the case of agents (Figure 32). Three lists are used, and the connection between the lists is done through the highlighted elements: when an agent is selected (highlighted), his/her list of roles appears in the middle part of the screen, and when a role is selected, the associated responsibilities are shown on the right scrolled list. Information is entered in these lists through the type in boxes and the add/delete buttons below each scrolled list. Notice that the elicitor may decide not to enter responsibility information if this is not

going to affect the elicitation process (usually when the view related to that agent should cover all his/her responsibilities). Buttons for saving the information entered or for coming back to the last information saved are also provided.

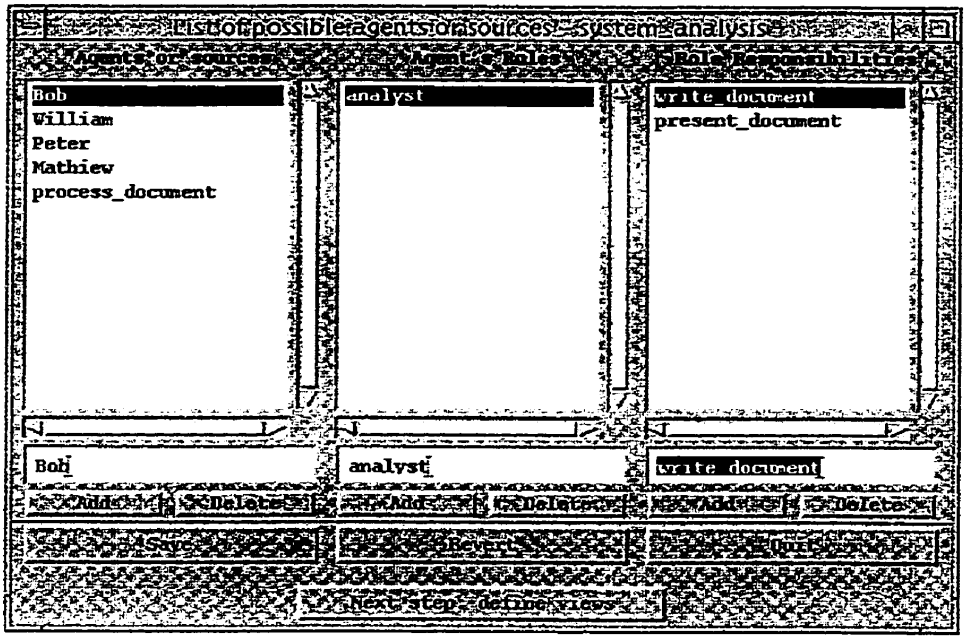


Figure 32 - Listing the potential sources of information

In the case the source of information is not an agent (e.g., "process_document" in Figure 32), it should be entered also in the list of sources of information, but no role or responsibility information is entered in the second and third list.

The "next step" button in Figure 32 opens the window in Figure 33. This window is used for specifying the type of information each source can provide. The information from the previous window is displayed, and the elicitor can enter a *view type* for the role specified (first box under the label "viewtype"), or even for each responsibility associated to the role (one box per responsibility identified in Figure 32, if any). This *view type* defines the list of aspects available from this source, as described in Section 4.2. When the *view type* is not entered for one of the responsibilities, it is assumed to be the same as the last box above it containing a view type. In the case that no responsibility (or even no role) is

specified, one box only is shown under the label "viewtype" with no associated responsibility.

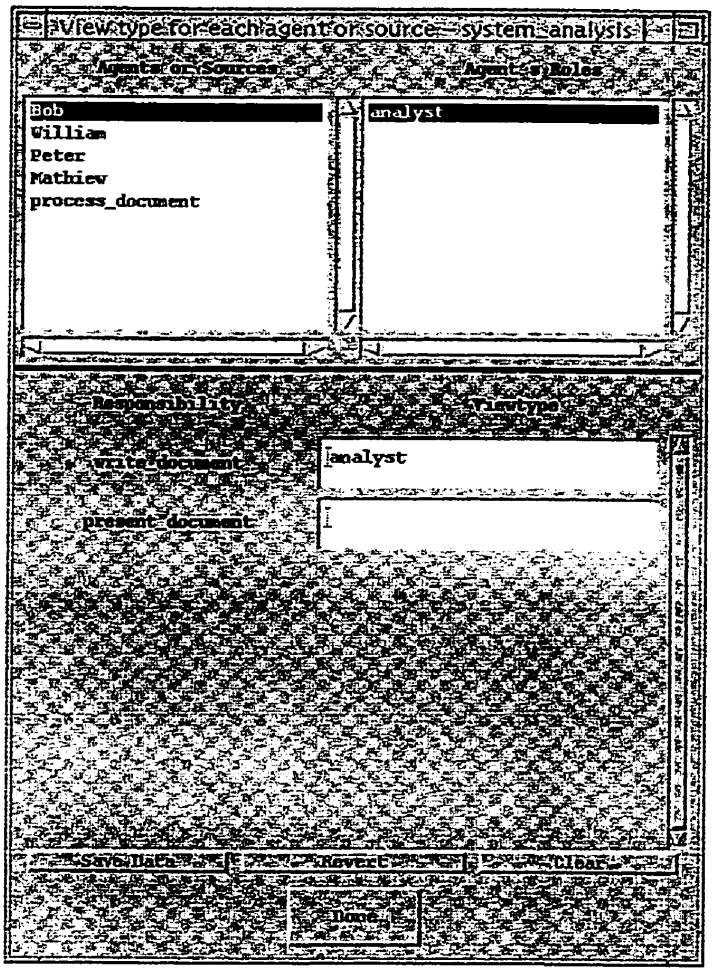


Figure 33 - Specifying the types of information for each source (view type)

The potential list of sources entered in Figure 32 is then used, in the last elicitation planning sub-step (see the third button in Figure 30), to choose the definite sources from which to gather the process information (Figure 34). The selection is performed in the same way, in the top part of the window, as for aspect type selection in Figure 31. In order to help in choosing the views, the information entered in Figure 32 and in Figure 33 is displayed on the bottom of the window in Figure 34, for the view highlighted in the list. When selecting a role in the list on the left, the related responsibilities are shown in the list on the right.

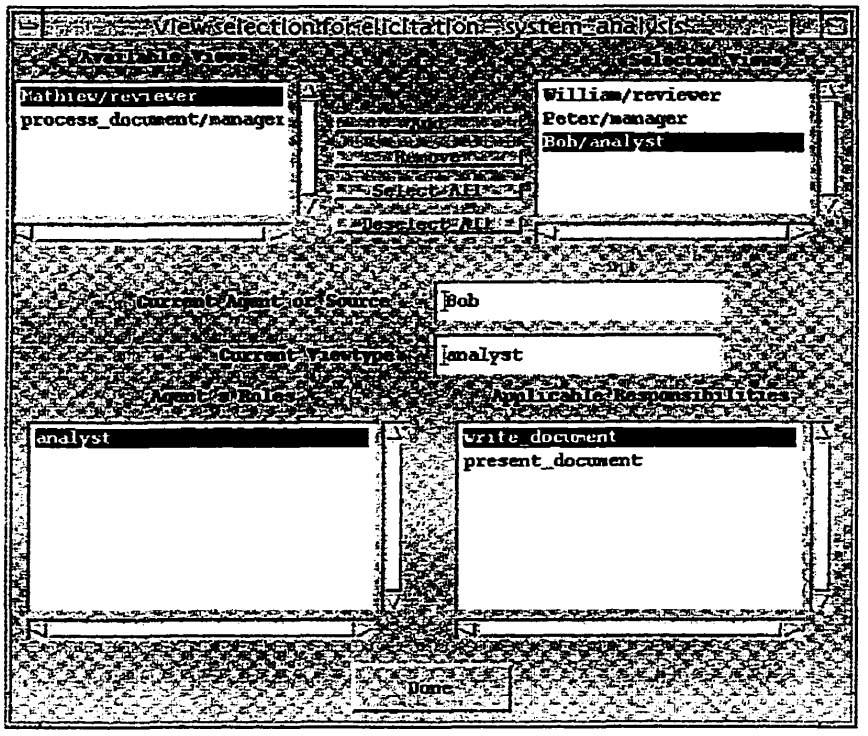


Figure 34 - Choosing sources from which to elicit

Ideally, we would like to obtain information from all the agents, but this is rarely possible. Some agents may not be available, and the labor costs are generally high for those available. So we should select a representative subset that will cover the process, at different levels of details. However, some redundancy is helpful for validating the information gathered. This subset of sources is used by the system for managing the elicitation process, by making sure that each view is elicited and checked for consistency before we merge all the views.

5.2.2 Steps 2 and 3 : Eliciting views

Having defined the list of views, and the type of information we can obtain from them, we can now elicit each view. In this second step, presented in Figure 29, we are defining the consistent views independently. When opening this step, we first have to choose the view

on which we want to work (see Figure 35), and then the steps are presented for eliciting this view (see Figure 36).

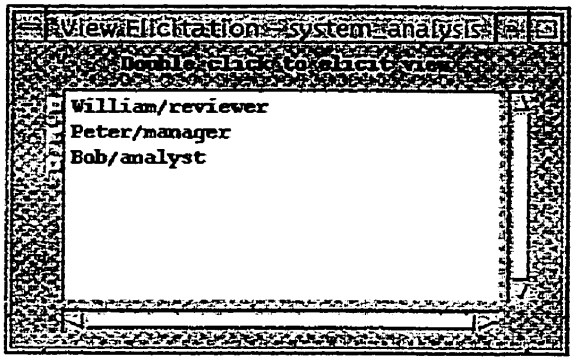


Figure 35 - Selecting a view to be elicited

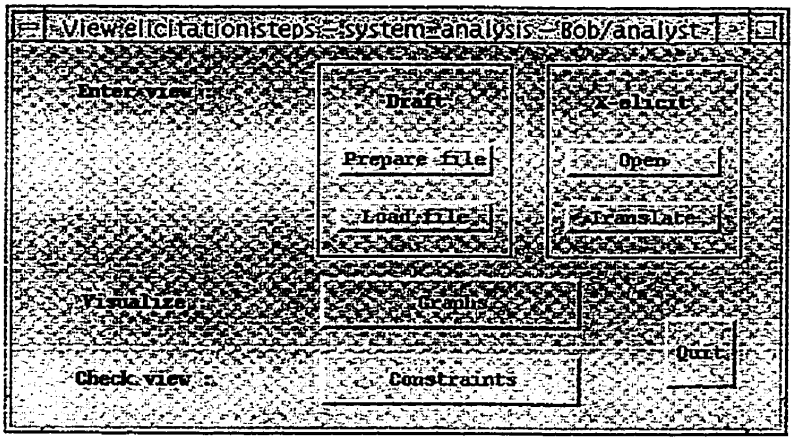


Figure 36 - Steps for eliciting a view

The first two steps ("enter view" and "visualize") are related to Step 2 - Gather view information (see Figure 24), and they are used for entering or modifying the information related to this view in the system. This is described in Section 5.2.2.1. We should also check that the views built are consistent (Step 3 - check for intra-view consistency, in Figure 24). This is performed in the last step ("check view"), and is discussed in Section 5.2.2.2 below.

5.2.2.1 Step 2: Gather view information

In this step, a view (e.g., Figure 25) is entered into the system. There are three features that can be used here: "draft", "X-elicit", and "graph". The "draft" part is used as a starting point when the information gathered is not structured at all; it permits the elicitor to enter the information in a completely unorganized way. In the case that the information is already structured, or after structuring the information gathered using the "draft" part, the "X-elicit" part can be used. This part is actually a link to the X-elicit tool, built at McGill [MHH94], which is a textual process-model editor. It helps in structuring the information by entity decomposition, and in showing the relationships (and attributes) as attributes to the entities. In both cases ("draft" and "X-elicit"), the information can be visualized graphically as it is edited using the "graph" part. These three parts are detailed in the rest of this section.

The "draft" part is used to map unstructured information (mainly from interviews) onto the V-elicit modeling schema. In general, the existing documents used as sources of information are already structured in a way that shows the decomposition of activities. But this is not the case when dealing with people: they often start talking about one part of the process and then jump onto another part, or start by giving details and then provide the general structure of the process. In this case, a structured editor needing the decomposition information first cannot be used until all the information is gathered and analyzed by the elicitor. A more efficient way would be to enter the information as it is gathered, and let the system organize the information at a later point in time. That is precisely what this "draft" part does.

This unstructured editor is simply a text file grouping the relationships of the view by relationship types. The first button ("Prepare file" in Figure 36) creates the file and lists the relationship types to be elicited (from the elicitation planning step). Figure 37 shows such file for Bob's view. The elicitor can then edit this text file, and enter the relationships by specifying the two entities involved, under the section related to the appropriate

relationship type (see Figure 38). The file is translated into the V-elicit modeling schema through the "Load file" button in Figure 36.

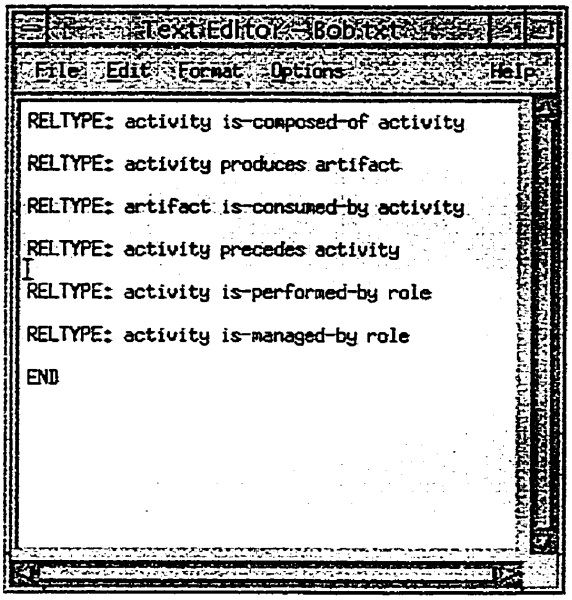


Figure 37 - File generated in the "draft" part

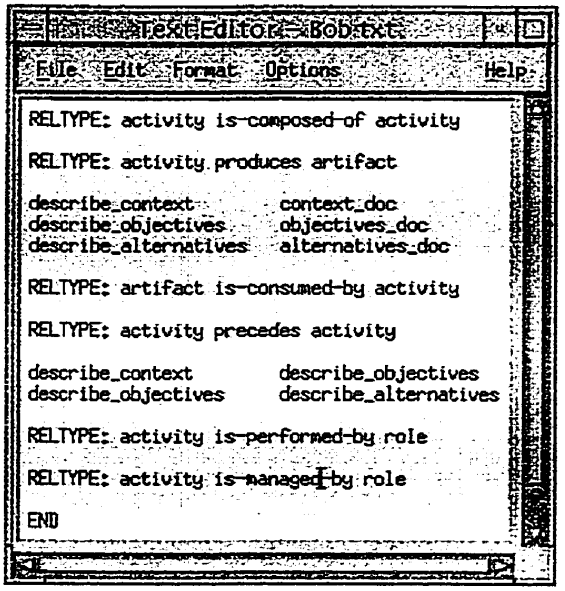


Figure 38 - Example information entered (unstructured) using the "draft" part

Once the view is translated to the V-elicit structure, it can be visualized using the "graph" part. Figure 39 shows an example of such graph. The graph represent only one aspect, selected by the user (Bob's information flow aspect in our case), for reducing the amount of information to visualize at once. The graphs are actually displayed in a tool called "Dotty" [KoN96].

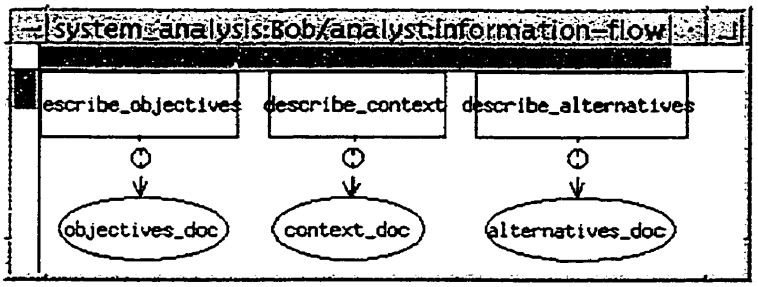


Figure 39 - Bob's information flow aspect (incomplete) as specified in Figure 38

Such graphs can help in understanding the information currently entered, and in gathering more information. For example, the elicitor can ask the agent whether his/her view being elicited is correct, and inquire additional details. The appropriate changes can be made to the text file, and then the V-elicit views are updated by clicking the "Load file" button again.

Notice that the "draft" part can be used with any user-defined relationship types, satisfying our requirement R2 and R3. Also, different text files are used for each view, permitting one to enter separately the information from different views (requirement R1).

These text files are useful in getting started with one view, but they become more and more difficult to understand and modify as the size of the files increase. A tool that can show the information in a structured way is needed for this purpose. However, this tool should also be tailorable such that user-defined types can be used. For these reasons, we have chosen to use the X-elicit tool, adapting its attributes to suit our specific needs. This link is represented by the "X-elicit" part in Figure 36.

The button "Open" in Figure 36 opens the X-elicit tool (see Figure 40) for the specified views, and the button "Translate" (in Figure 36) is for parsing the view information into the V-elicit system. As for the "draft" part described above, an iterative process of aspect visualization and view information changes can be used to help in understanding the information currently entered and getting additional information.

In X-elicit, the information is entered in templates (see Figure 40) containing a list of *attributes* to be filled (the attributes define the types of information). There is one such template for each type of entity (the type is shown on the title bar of the window, with the name of the project being elicited). Each entity is described in such a template, with specific values for its attributes. For example, the template for an activity has attributes "Goal", "Purpose", "Artifact-Input", "Artifact-Output", etc.

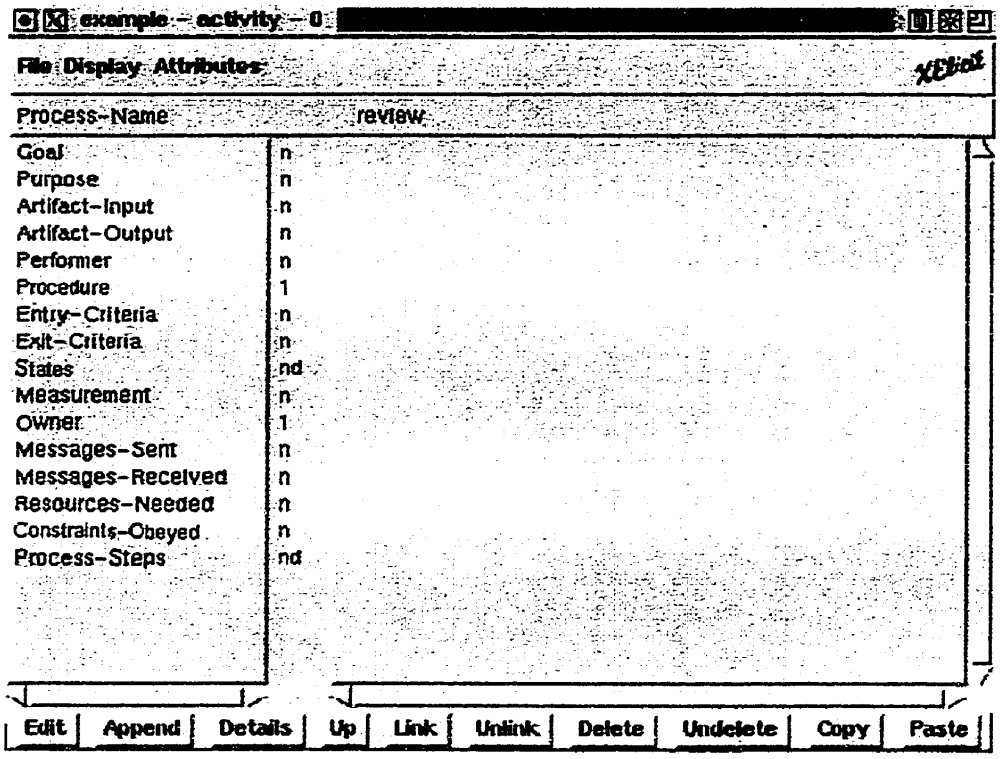


Figure 40 – X-elicit tool

Each attribute has a cardinality: "1" if only one value is allowed, and "n" if we can have a list of values. A "d" indicates that we can get more information on this attribute in another template. For example, the "Process-Steps" attribute represents the decomposition of the current activity into sub-activities: for each sub-activity, the related set of information (values of the attributes) can be shown in another template by selecting it and clicking the "details" button.

Such a template can easily be modified because it is all defined in text files. So for each view, we can specify templates containing only the type of information the associated agent (or source) can provide, as defined in the elicitation planning step.

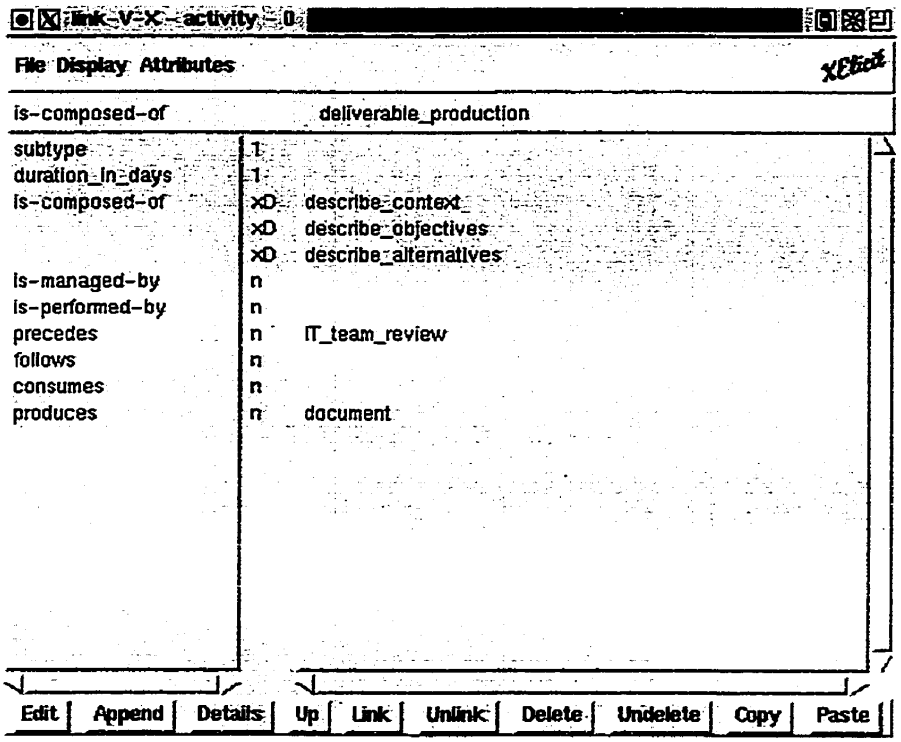


Figure 41 - X-elicit adapted by our system: an example based on Bob's view

Figure 41 shows an example of a template for activities, used for Bob's view (see Figure 25, or Figure 77 to Figure 81 for complete details). It represents the activity "deliverable production"[11], and shows its attributes (subtype and duration in days), and its relationships with other entities (e.g., "produces" is the relationship of type "activity produces artifact"). Each relationship is reproduced twice: once for each entity type involved. For example, the relationship "activity produces artifact" is in the activity template as X-elicit attribute "produces", and in the artifact template as X-elicit attribute "is-produced-by" (meaning "artifact is produced-by activity"). When the information is translated into V-elicit, the redundant information is checked for consistency. In the case when the relationship is not specified in one of the redundant attributes (for example, if we have a value in the "produce" attribute, but not in the "is-produced-by" attribute), the system asks the user

[11] The title bar indicates that the entity is an activity. The keyword "is-composed-of" besides the name of the activity ("deliverable_production") indicates that this template has been accessed from another template where the entity was listed under the "is-composed-of" attribute.

whether the redundant relationship should be generated or not, so the elicitor can avoid entering twice the information.

In addition, more checks are performed when translating the information entered in X-elicit into the V-elicit database. We check that the values entered fit the V-elicit attribute value type expected (e.g., an integer for the attribute duration in days). We also check that the values entered as relationships are existing entities (e.g., the value the X-elicit attribute "produces" should be an artifact that is already specified).

The X-elicit tool, separately, has already been used successfully for capturing and organizing process information in several industrial-scale projects [MHH94]. The limitation with this tool, however, is that it does not have any notion of views. The information from different sources is entered in the same model, and when there is a conflict, we are compelled to solve it immediately and modify the information given by other people. This is not a problem when we have only one source of information, but is not practical in many circumstances. The V-elicit system creates one model per view in the X-elicit tool.

As discussed in Section 4.6, our choice of model visualisation and editing tools is not necessarily the best one. A link to advanced graphical tools, for example, could make the presentation of the models more concise than when using an ERD format (although ERD is the format used for internal representation). The tools used here (X-elicit and Dotty) have been chosen for their availability, and because they meet our basic requirements.

Figure 42 shows the graph of the activity decomposition aspect elicited for Bob's view (the result of this step). The other aspects elicited are shown in Appendix A, Figure 77 to Figure 81.

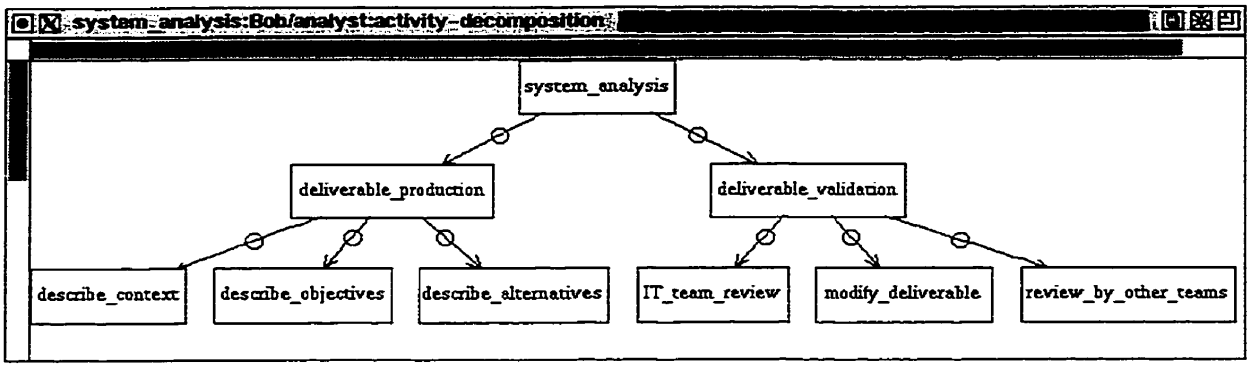


Figure 42 - Activity decomposition aspect elicited for Bob's view

5.2.2.2 Step 3: Check for intra-view consistency

Once a view is elicited, the elicitor needs to check for inconsistencies within the view. For example, s/he can check that *each activity within a view produces some artifact*. Because the type of information represented in a view is user-definable, we need to provide a mechanism to the user for defining what an inconsistency is in a given view. These inconsistency definitions are called *constraints*. Based on such definitions, the V-elicit system can then check an elicited view against these constraints.

For example, the constraint *each activity produces some artifact* could be specified in V-elicit as shown in Figure 43[12] (this window is opened by the "constraints" button in Figure 36). Using a mathematical notation for the sets and quantifiers, this constraint is the same as:

$$\forall e \in \{\text{activity}\} \bullet \exists d \in \{\text{artifact}\} \bullet$$
$$\text{ThereIsRel}(e, d, \{\text{"activity produces artifact"}\})$$

[12] In this window, a new constraint can be typed in the text area, or an existing constraint can be selected through the "constraints list" button. The buttons on top of the window help in writing the constraint: the "editor" button helps in structuring the constraint based on the constraint-specification language (see Appendix C), and the other buttons give lists of keywords to type (to avoid misspelling). The constraint is checked by clicking on the "execute" button.

The meaning of this constraint is that, for all activity e, there is an entity d of type "artifact" such that there is a relationship from e to d of type "activity produces artifact" (i.e., the activity e has at least one artifact as output).

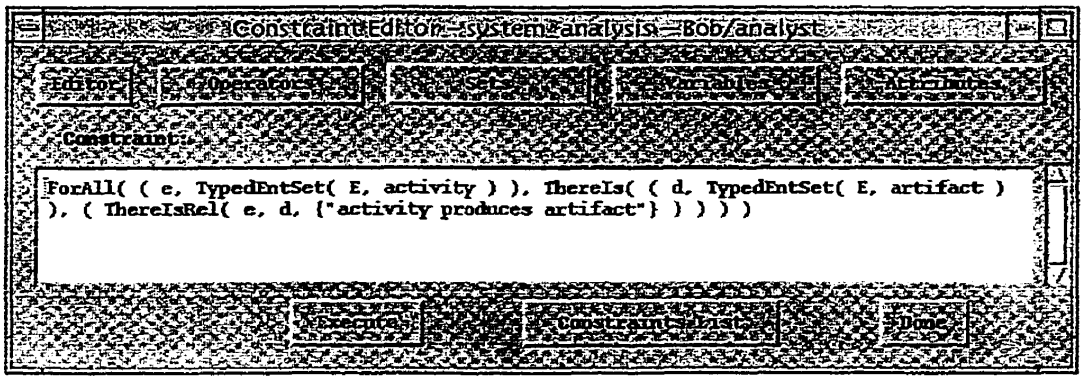


Figure 43 - Example of constraint specification

When verifying the elicited view against the specified constraint, the system evaluates the latter from left to right. On a ForAll clause, it builds the set of possible values, and loops on each value. For a ThereIs clause, a similar approach is taken, except that the loop on each value of the set is terminated as soon as one value making the rest of the constraint true is found.

The "execute" button in the bottom of Figure 43 starts the verification of the constraint. Before its evaluation, the elicitor can choose whether to evaluate the constraint on one aspect only, or on the entire model (Figure 44). The time taken for the evaluation is obviously better when the verification is performed on one aspect only, but if none of the aspects contain the entire set of information required to evaluate the constraint, then it should be verified on the entire model[13].

[13] In typical situations, the elicitor selects a constraint from a list developed by an expert. Each constraint in such a list contains a textual description of the constraint, that may also indicate on which aspect such a constraint can be evaluated. The details of such a list are described later in this section.

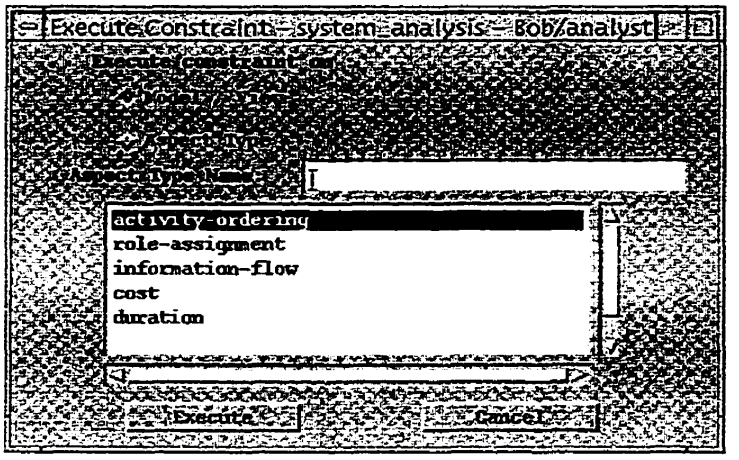


Figure 44 - Choosing the aspect on which the constraint is evaluated

The result of constraint evaluation is "True" or "False", together with the set of values (and associated variable) in the ForAll clause(s) that violates the constraint (i.e., making the rest of the constraint evaluate to false) in the case of a "False" result. Figure 45 shows such a result window, for the constraint specified above. The upper part specifies which constraint has been evaluated, and the lower part gives the result of the constraint verification.

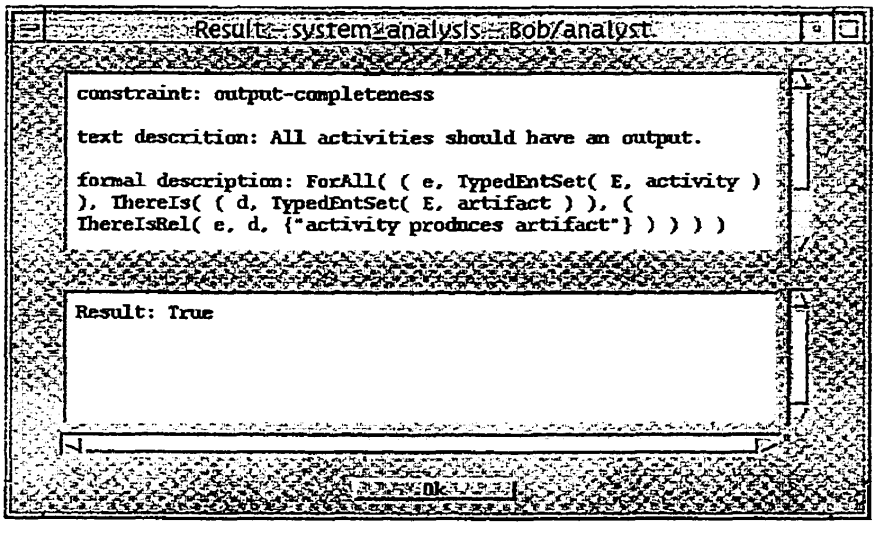


Figure 45 - Result of the evaluation of a constraint that is satisfied

Our constraint was satisfied in this case. But if we check that each activity is managed by someone (a role) on Peter's view, then the result is negative (the "activity is-managed-by role" relationships are represented using dashed lines in Figure 46). Figure 47 shows this result, by listing (at the bottom part of the window) the values of the variable "e" (from the ForAll clause) for which the constraint is not satisfied[14]: "other_teams_review" and "client_review". The elicitor should then go back to the "elicit view" step ("draft" or "X-elicit" parts), for making the appropriate modifications (for example, adding a "other_team_manager" role, and a relationship of type "activity is-managed-by role" with the activity "other_teams_review").

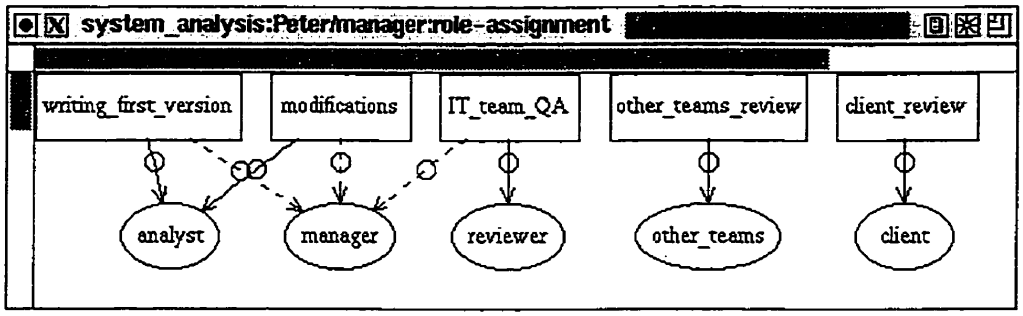


Figure 46 – Role assignment aspect of Peter's view

Up to now, the only constraints that we have checked were related to the *structure* of the views (i.e., to the modeling schema used). However, the quality of the model can also be related to the conformance to rules defined in the organization. To this end, we can check constraints related to the *meaning* of the entities and relationships modeled, for example that *every document is indeed reviewed.* This is not a *structural* issue but a *semantic* issue. Such a check in software development, for example, can be a central part of a Quality Assurance plan to meet product quality requirements. Figure 48 shows the result of verifying such a constraint on Bob's view. The technique is the same; it is just that the

[14] Notice that a constraint may not be satisfied by a *combination* of values for two or more variables (in multiple "for all" clauses). In such case, values would be printed within two horizontal lines (one per line). Because of such a case, it is necessary to indicate the variable name together with the value causing the non-conformance problem.

constraint language is powerful enough to allow such verification. Section 6.1.2 discusses in more detail these two kinds of constraints.

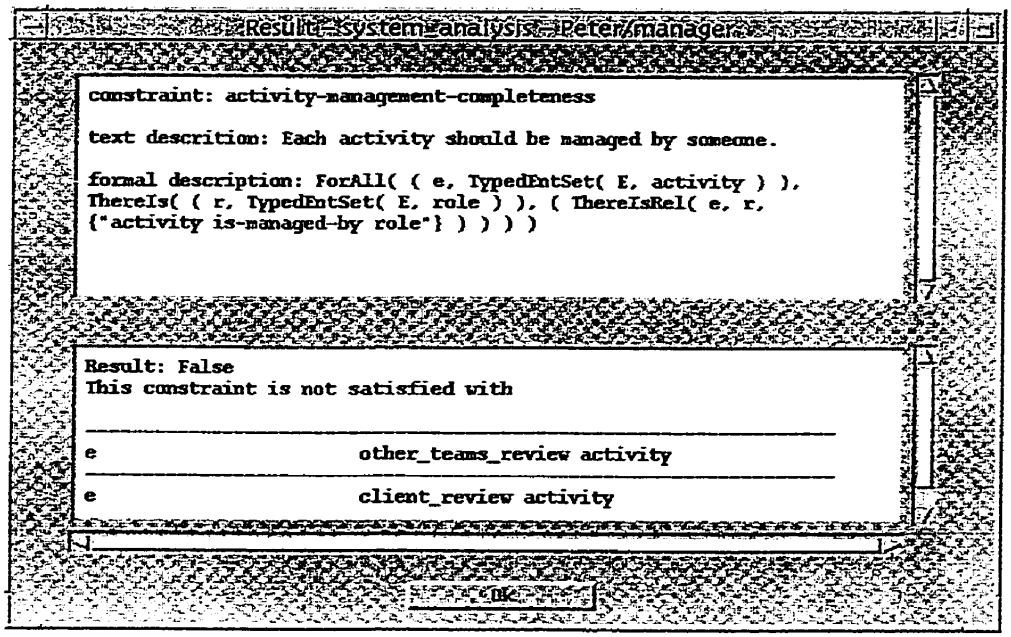


Figure 47 - Result of the evaluation of a constraint that is not satisfied

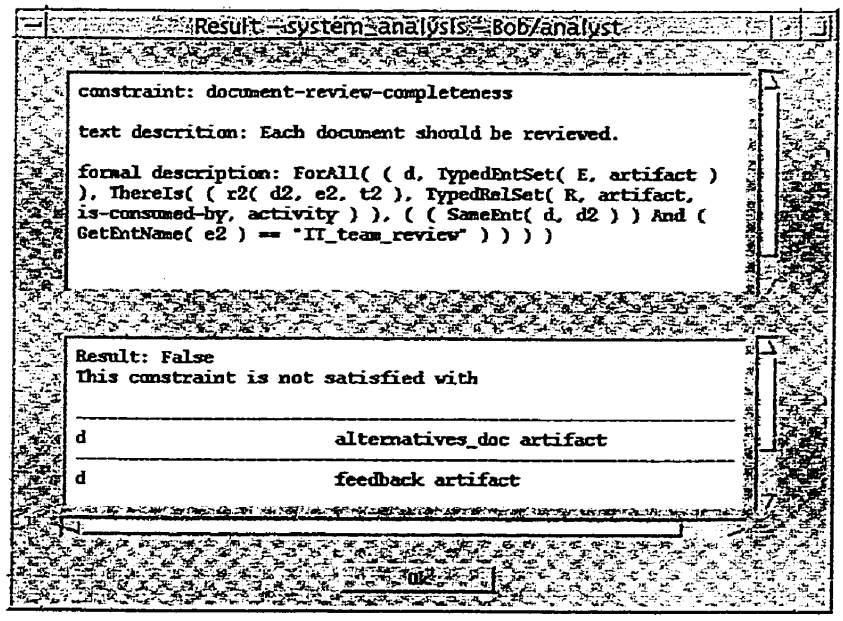


Figure 48 - Result of a constraint related to the meaning of the information

Notice that the variable for a relationship (r2(d2,e2,t2) in Figure 48) contains four variables: one for the relationship itself (r2 here), two for the entities involved in the relationship (d2 and e2 here), and one for the type of relationship (t2 here).

An important point that the elicitor should keep in mind is that the result of a constraint verification may not be as expected, for the reason that the constraint itself may not have been well specified or may not be valid in practice. In our example in Figure 48, the result shows a problem with the "feedback" artifact, but this artifact should not necessarily be reviewed. Here, the constraint was not specific enough on the kind of artifact (artifact subtype) to be reviewed. In this case, either the constraint should be refined further or the interpretation of the result should be appropriate.

These constraints can be saved in the database, for later use. The elicitor can then select the desired constraints from the constraint list. This is done through a window such as that in Figure 49 (opened by clicking the "constraint list" button in Figure 43). The constraints are categorized in the following three sets: *project-specific*, *structural*, and *organizational* constraints (see the upper part of Figure 49).

Structural constraints are those related to the *structure* of an aspect. For example, in an aspect showing the dependencies among the artifacts, the graph should be a DAG (directed acyclic graph). Thus one constraint could be that no cycles among the dependencies are allowed. Another constraint could force the graph to have a tree structure. These kinds of constraints can be applied to many different aspects, and in general do not contain any information related to the entity types or relationship types used.

The other types of constraint are related to the types of information modeled and their meaning. For example, the constraint that each activity should be performed by someone (shown in Figure 49). These constraints can be either *project-specific* or *organizational* constraints, depending on whether they are applicable to a single (or few) project, or to all

projects in the organization. The purpose of organizational constraints is to avoid the duplication of a constraint in every project.

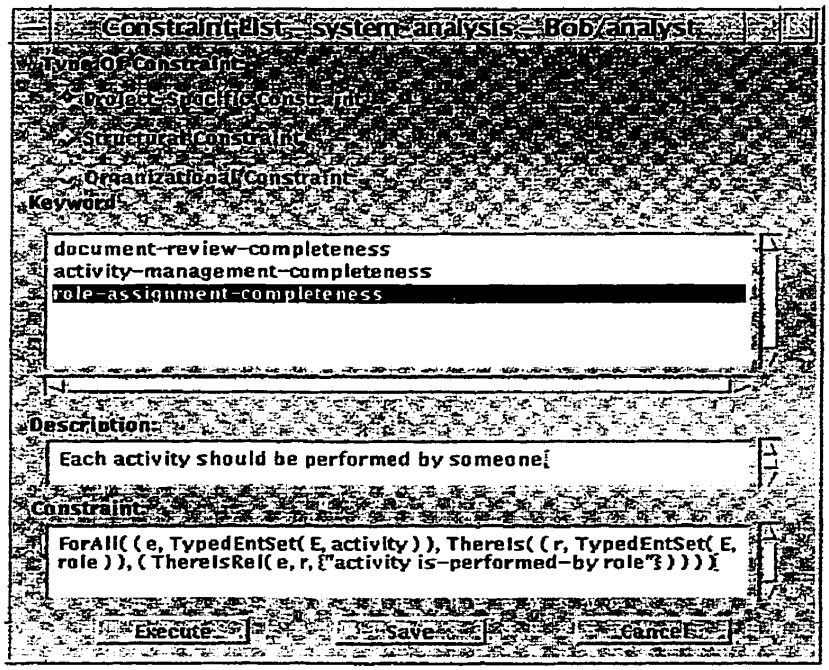


Figure 49 – Selecting a constraint from a list

When the type of constraints is selected (from the radio buttons on the top of the window in Figure 49), the list of these constraints is shown in the first scrolled list (below "keyword"). By selecting one of them in the list, its details are shown in the lower part of the window: a textual description reminding what this constraint is supposed to check, and the formal specification of the constraint. Both textual description and formal specification can be modified and saved using the "save" button. The "execute" button is used to launch the verification of the selected constraint on the current view (shown on the title bar). It has the same effect as the "execute" button in Figure 43.

In this section, we have provided a general understanding of the constraints and their use. A detailed description of the constraint language, as well as the different kinds of inconsistency it can handle, is provided in Section 6.1.

5.2.3 Steps 4 and 5 : Getting a merged model from the views

In the previous step, we considered view elicitation and intra-view analysis. Assuming that such analysis is carried out within each of the views elicited, in this step we need to do cross-view analysis. Our goal is to identify the overlapping information across the views (step 4), and to make sure that there are no inconsistencies in these overlaps (step 5). For such analysis, we need to take into account that there may be terminology differences across the views.

The sub-steps to be used for such cross-view analysis are presented in a window such as that in Figure 50. The first step ("match entities") refers to our Step 4 here, described in Section 5.2.3.1 , and the other steps are parts of the Step 5 described in Section 5.2.3.2 below.

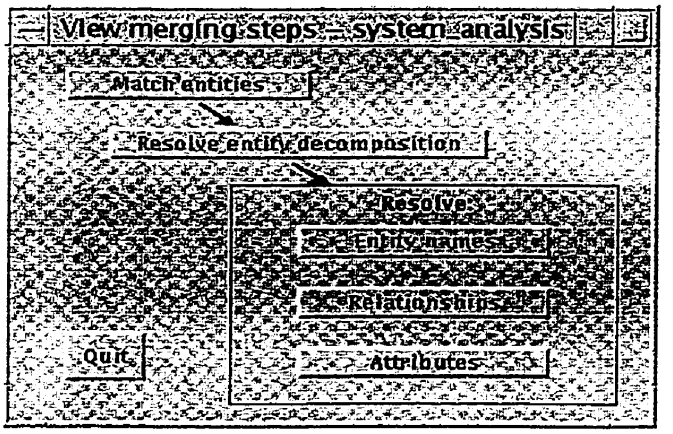


Figure 50 – Steps in analyzing and merging views

5.2.3.1 Step 4: Identify common components across views

Component matching is a mechanism for detecting common process elements (overlaps) across different views. For example, we need to know that "modify deliverable" in Bob's view (Figure 25), and "modifications" in Peter's view (Figure 26) refer to the same activity, and that William's view (Figure 27) does not contain this activity. This task is not so

obvious due to terminology differences. Thus, we need to examine the descriptions of the entities (i.e., their relationships and attributes) in order to match them.

Our technique is to compute a similarity score ([0..1]) for each pair of entities in each pair of views, and find the most probable matches based on the highest scores. The general idea of this similarity score is to compute the percentage of similar items (entity name, relationships, and attributes) between two entities. The precise formula used is presented in Section 6.2.

The first step is to ask the elicitor which entities (types) should be matched, and in which order. The elicitor should also specify the relationship types and attributes that should be used in verifying the similarity of the entities: this selection permits him/her to avoid using information with a high probability of being inconsistent, in the entity matching process. A window such as that in Figure 51 is used: the left part contains the entity/relationship/attribute types available in this project, and the elicitor can move them to the list on the right for selection.

For each attribute, the system then asks the degree of similarity expected. For example, Bob's "review_by_other_teams" activity has a duration of 5 days (Figure 25), but Peter's "other_teams_review" lasts 6 days (Figure 26). This difference could make the similarity score between the two activities to decrease, but the elicitor may think that this difference is not significant enough, and that this similarity in the attribute should increase the similarity score between the two activities. The elicitor can enter the percentage of difference allowed for each attribute to be used (number and time attributes only). Figure 52 shows how this information is specified.

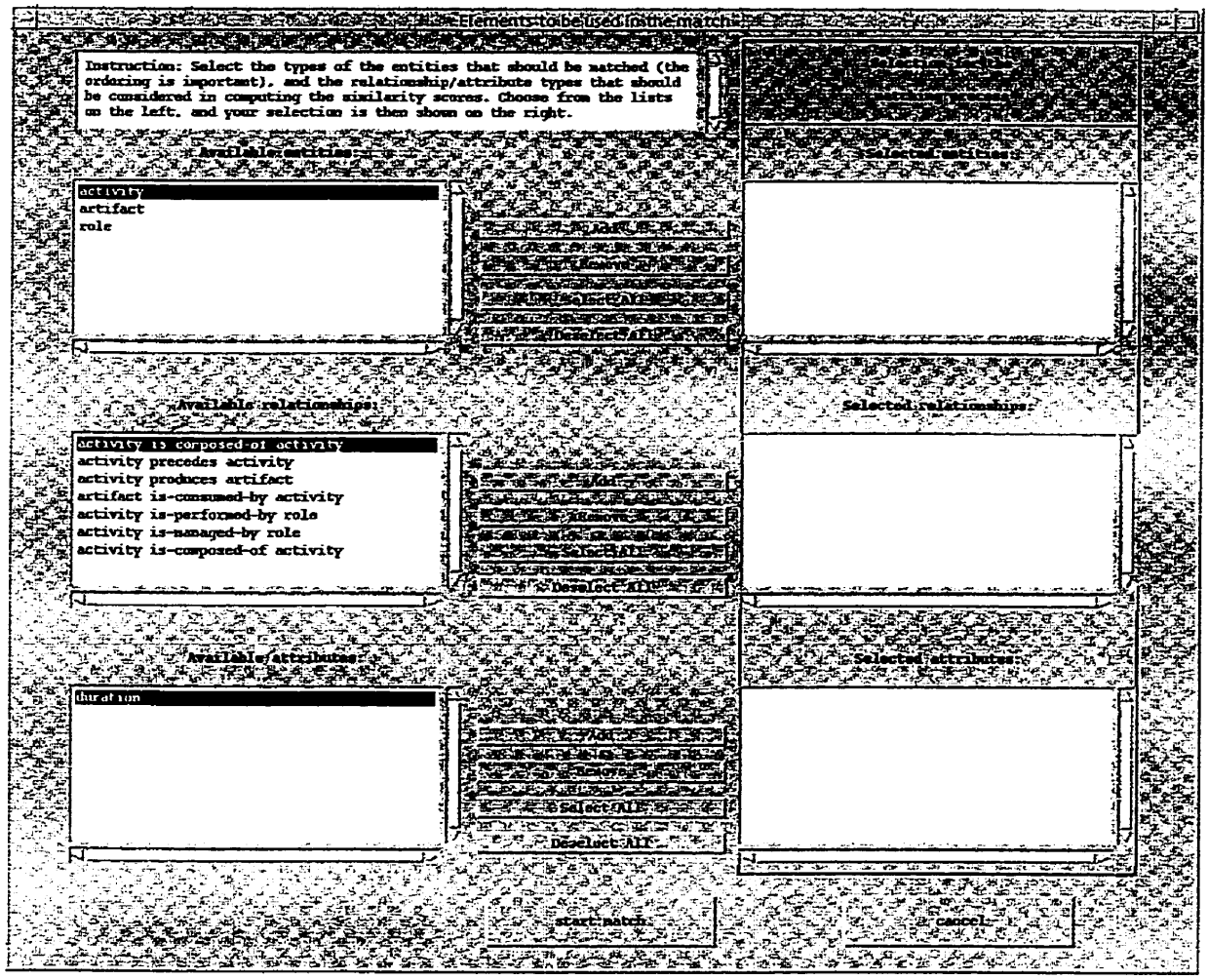


Figure 51 - Selecting the types of the entities to be matched,
and the relationships/attributes to be used

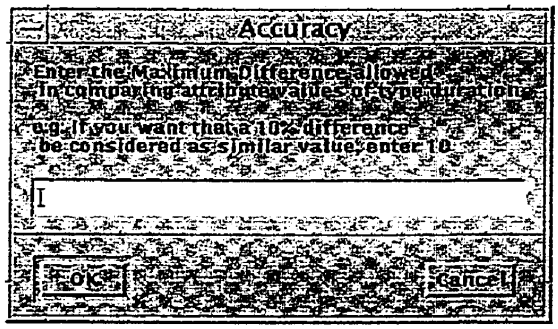


Figure 52 – Specifying the level of similarity allowed for an attribute

In order to prevent the identification of matches between entities that are not similar enough, the elicitor then has to specify the minimal value for the similarity scores. This is performed in a window such as that in Figure 53. Section 6.2.1 describes how such minimal value is used in the matching algorithm. For our example, we have used a minimal value of 0.2[15].

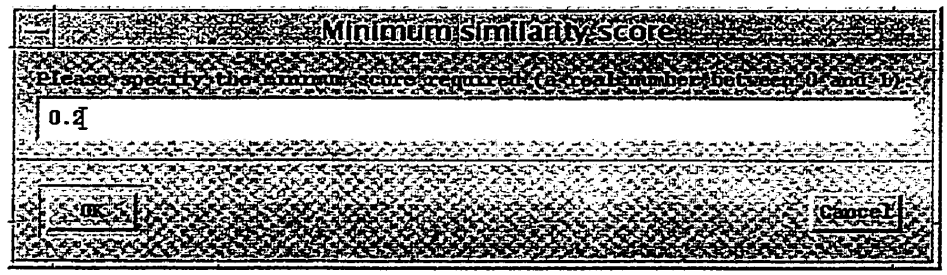


Figure 53 – Specifying the minimal value for the similarity score

The V-elicit system then computes the similarity scores, one entity type at a time, finding the matches corresponding to the highest scores. The results are shown in a window such as that in Figure 54, for each pair of views (showing the roles identified as matched between Peter's and William's views).

The results of matching roles and artifacts for each pair of views are not shown here because these are trivial in our example: the entities with the same name are matched[16]. However, the activities are not as trivial to match because of the use of different names to denote similar activities. Figure 55 shows the matches identified by V-elicit between Peter's view and William's view.

As one can see, one of the matches seems incorrect: Peter's "IT_team_QA" activity is probably related to William's "IT_team_review" activity. This can be modified by the

---

[15] The threshold value of 0.2 has been used based on our experience in modeling processes using our tool. Additional research and experiments are required for identifying the best value to be used (which would probably be different depending on the type of the process elicited or the elicitation settings).
[16] Note that our system also works when roles and artifacts have different names across views.

buttons at the bottom of the window. Deleting a match is done by selecting the match and pressing the "delete" button. When adding a match, a window such as that in Figure 56 appears, permitting the elicitor to specify which entities should be matched. In order to help in choosing the matches, the elicitor can have a look at the similarity scores stored in the file indicated in the window ("system_analysis_activity_scores" in our case; see Figure 57 for an example of such a file). Note that the system checks for conflicts with other matches before making the requested changes.

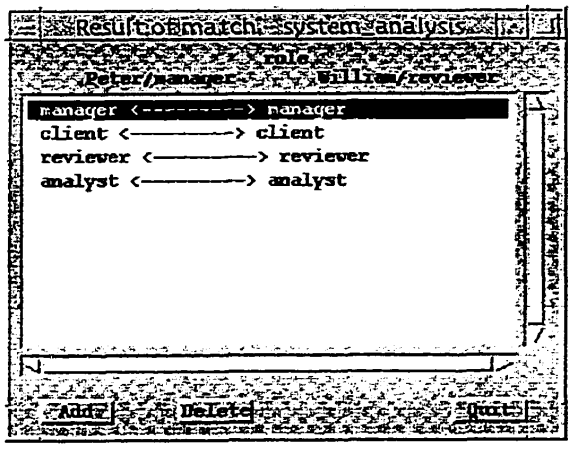


Figure 54 - Result of matching the roles between Peter's and William's views

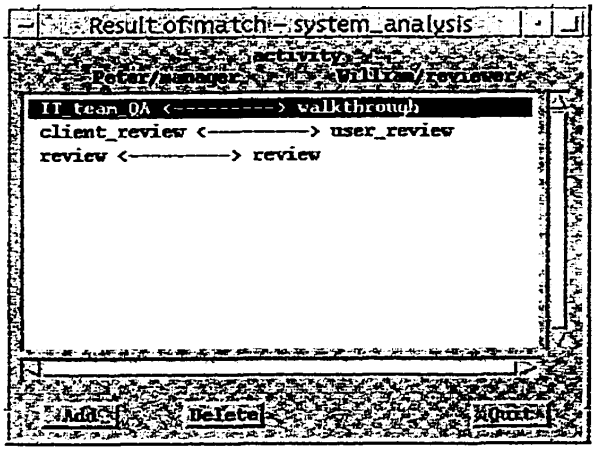


Figure 55 - Result of matching the activities between Peter's and William's views

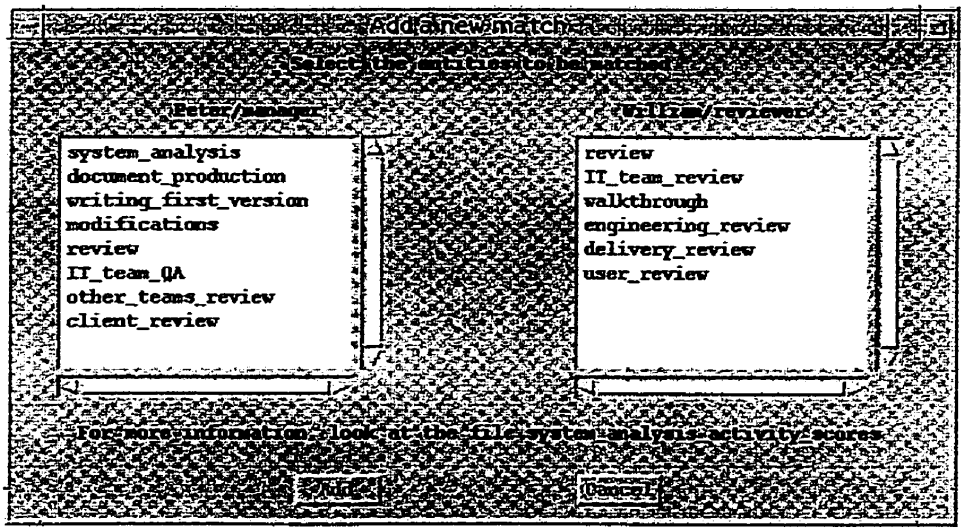


Figure 56 - Adding a new match

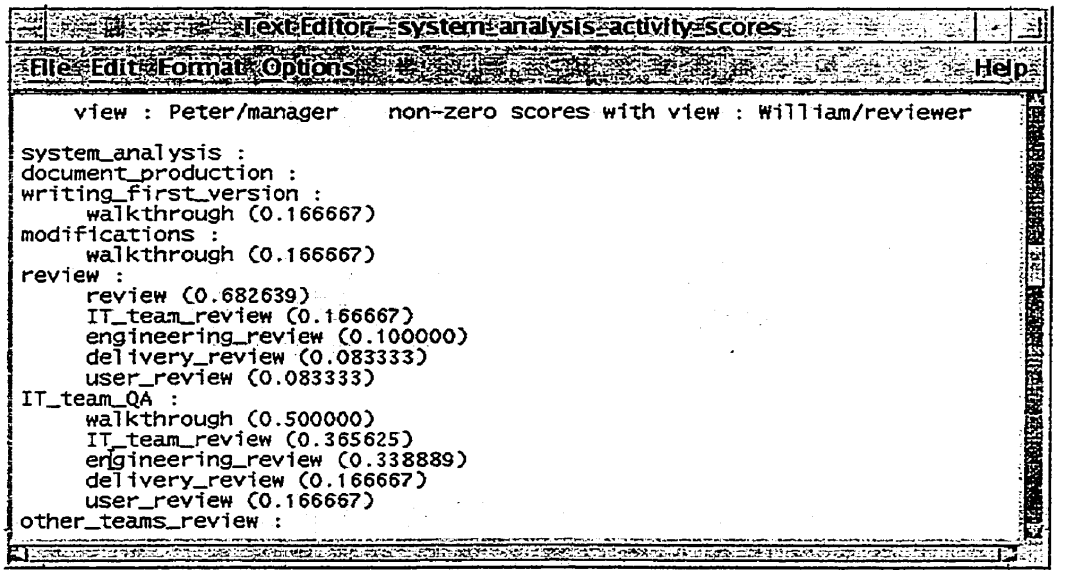


Figure 57 - Report generated by the matching algorithm showing similarity scores[17]

It is necessary to allow changes by the user because we cannot be absolutely sure that the system is giving the right matches. For example, in the cases where we do not have much

[17] This file is organized by pair of views (e.g., Peter's and William's views in our example). The entities from the first view (Peter) are listed, and for each entity, the entities from the second are listed (indented) with the similarity score between the two entities. Entities from the second view that have a similarity score of zero are not listed.

information about the entities, the system lacks information needed for comparing entities, and it may not be able to find the matches[18]. In our example (Figure 55), Peter's "IT_team_QA" activity has not been matched properly because the descriptions of William's "walkthrough" and "IT_team_review" activities are very close, when considering only the related elements that are found in both views.

From our experience, we have found that in actual cases from the software industry, the matches are not always found, but the similarity scores of the expected matches are high enough to identify them easily by looking at only a few scores. For example, in Figure 57, Peter's "IT_team_QA" activity is best matched with William's "walkthrough" activity (similarity score of 0.5), but the activities "IT_team_review" and "engineering_review" could also be a reasonable match with similarity scores of 0.366 and 0.339 respectively. The activities "delivery_review" and "user_review" however have a too low score (both 0.167), compared to the best one found, to be considered for a match with Peter's "IT_team_QA" activity.

Figure 58, Figure 59, and Figure 60 show our final matches for activities (after modifications) for each pair of views. These are used in the next step, when checking for inconsistencies across views.

[18] Our assumption is that for each entity type, there is at least one related element that can be compared and matched. For example, we can start by matching the agents using the person's name, which is probably the same across the views. Title of documents or file names can also be used as starting point. We can then use relationships to these entities for matching the other types of entities. We also assume that the relationships do not link most of the entities of one type to most of the entities of the second type, otherwise the scores cannot identify the similar components, because the comparison elements are the same for most of the entities.

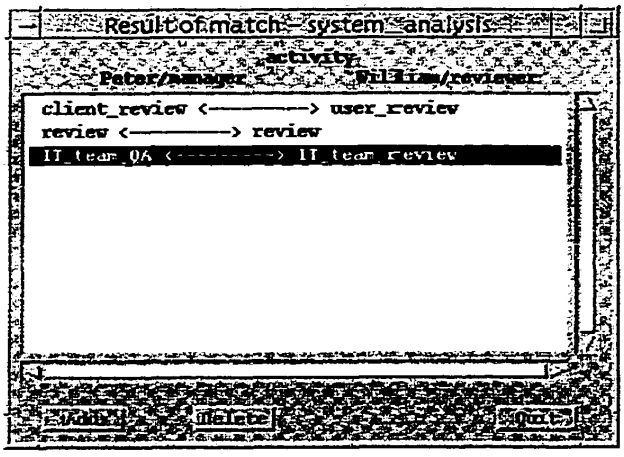


Figure 58 - Final matches for the activities between Peter's and William's views

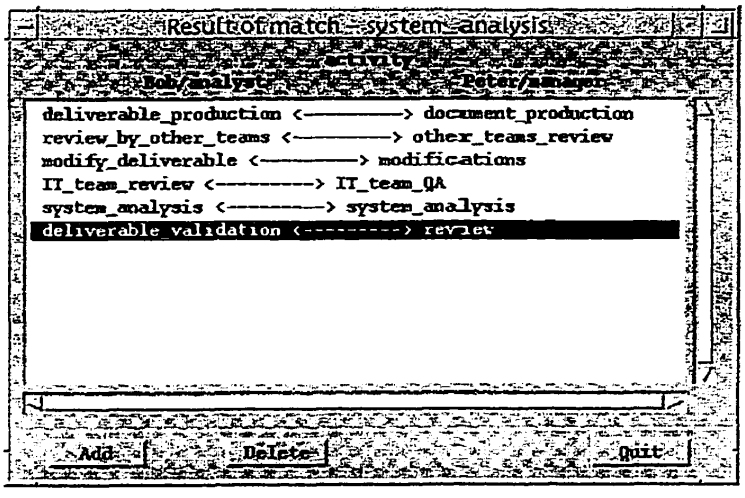


Figure 59 - Final matches for the activities between Bob's and Peter's views

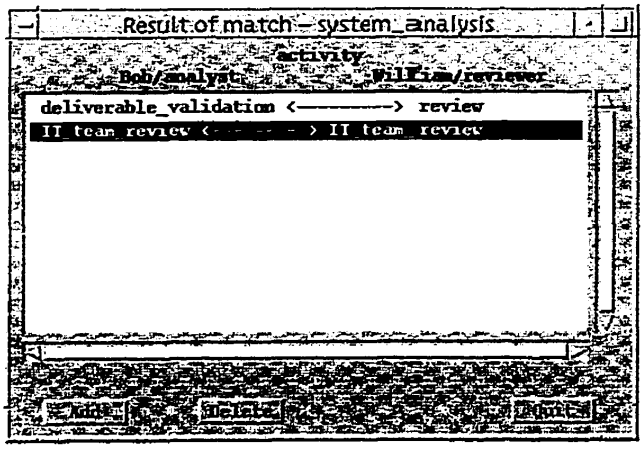


Figure 60 - Final matches for the activities between Bob's and William's views

5.2.3.2 Step 5: Merge views

Now that we know where the overlap is in the views (i.e., which entities are similar, as shown in Figure 58, Figure 59, and Figure 60), we can detect the inconsistencies across views, and resolve them. View merging is a mechanism for solving this problem, and for building a merged model concurrently as the inconsistencies are resolved.

In our approach, inconsistencies are categorized into four types:

(i) those related to the decomposition of entities such as different grouping of entities (e.g., "modify_deliverable" activity in Bob's view as part of the review process, but part of the document production process in Peter's view), or details missing (e.g., "describe_context" activity in Bob's view but not in Peter's view and William's view);

(ii) those related to the name of common entities (e.g., Peter's "client_review" activity, and William's "user_review" activity);

(iii) those related to the relationships between process entities such as missing input/output in one or more views (e.g., the "contract" artifact used in one of the substeps of the "IT_team_review" activity in William's view, but not used in Bob's view and Peter's view for such activity); and

(iv) those related to the attributes of the common entities (e.g., different duration for the "IT_team_review" activity across the views: 0.3, 3, and 3 days for Bob's view, Peter's view, and William's view respectively).

There is one button for each of these types of inconsistencies in Figure 50.

This categorization is based on the elements of an ERD structure: entities, entity names, relationships, and attributes. Entity types are not considered here because we are not going to compare entities of different types.

The first type of inconsistency to resolve is the one related to the entity decomposition ((i) above), because it can affect the resolution of the other types of inconsistencies, through

the choice of the entities to be kept in the final model and their meaning. It is described in Section 5.2.3.2.1 below. The three other types ((ii), (iii), and (iv) above) are independent of each other, and they can thus be resolved in any sequence. Section 5.2.3.2.2 gives an overview of these types of inconsistency and their resolution.

Notice that the intent here is to illustrate how the merging process is performed, and the kind of interaction between the system and the elicitor. For complete information about the different types of inconsistencies handled, as well as the algorithm used for detecting and resolving them, the reader should refer to Section 6.3.

5.2.3.2.1 Resolving inconsistencies related to entity decomposition

The kinds of inconsistencies we are looking at in the entity decomposition are whether or not some entities are missing, or whether or not the entities are grouped in different ways in the decomposition hierarchy. In our system analysis example (see Figure 25, Figure 26, Figure 27, and Appendix A for complete information), we see many of these consistency problems. In the case of the roles and the artifacts, the only kind of inconsistency one can find in this example is the missing entities. For example the "contract" artifact is missing in Bob's view. In the activity decomposition, one can see a variety of kinds of inconsistencies:

(a) "system_analysis" missing in William's view (the root activity is not the same),
(b) "modifications" not under the same subtree in Bob's view and in Peter's view,
(c) "IT_team_review" decomposed in William's view but not in the other views,
(d) "deliverable_production" and "modifications" missing in William's view,
(e) "describe_context", "describe_objectives", and "describe_alternatives" specified in Bob's view only,
(f) "writing_first_version" specified in Peter's view only
(g) "other_teams_review" missing in William's view
(h) "client_review" missing in Bob's view

The V-elicit system takes each of these problems in turn for resolution, one entity type at a time. The elicitor is actually controlling the ordering of the entity types resolved, by selecting which type is next through a window such as that in Figure 61.

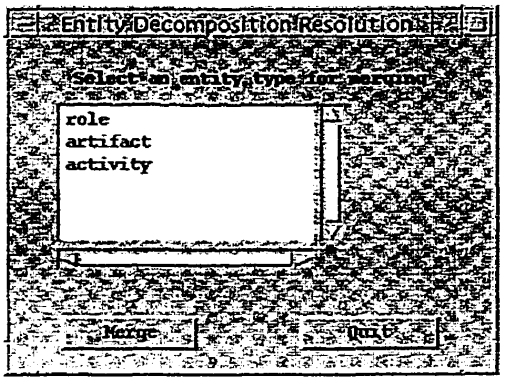


Figure 61 - Selecting the next entity type (decomposition) to be merged

Within one entity type, the ordering of the problems resolved is top-down, from the root to the leaves, in a recursive way. As the inconsistencies are resolved, the final model is built concurrently.

The recursion to be performed should start at the root level, but sometimes there is no single root entity (e.g., the roles and artifacts in our example), or the root entities are not the same (e.g., the activities in our example). In order to start the recursive merging algorithm with the same conditions each time, we are adding a temporary root to all views, and to the final model being built. This temporary root entity is removed when the views are all merged into the final model.

Once this temporary root is added, we then recursively solve the problems related to each node, starting at the root. For each inconsistency, the user should decide on the solution to be adopted, and then the views and the final model are modified accordingly. Before recursively checking the children of a node, we have to make sure that in each view, we have the same children for this node, and that the subtrees under these children contain the same matched entities.

As an example, let us work on the activity decomposition of our system analysis example. Starting at the temporary root level, V-elicit checks that the children (the actual roots of each view) are the same in each view: they are not (inconsistency (a) above), so the system asks the elicitor whether to keep the "system_analysis" root or not. This is performed in a window such as that in Figure 62. It is actually shown as a grouping problem, where the elicitor should decide whether to have only one entity ("system_analysis") grouping all activities in the project or not .

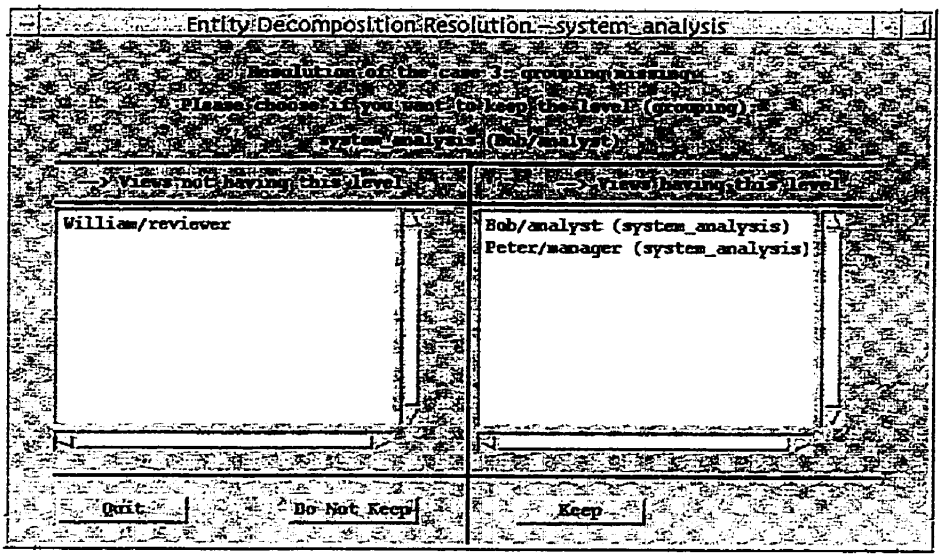


Figure 62 - Resolving the problem with the inconsistent root activity

The layout of each window for resolving an inconsistency is the same: the top part explains the type of inconsistency and which entity is affected, then the middle part shows which views have which solution (showing who and how many views gave one solution), and the lower part has buttons for taking the decision. The "quit" button stops the merging process. When double-clicking on one of the views, two graphs are opened: one showing the view before starting the merging process (containing the real information gathered from the source associated to this view), and one showing the view with the modifications made since the merging process began.

The purpose of having such decision window is to help focus on a simple inconsistency at a time (with two possible solutions only, involving only few entities). The access to the

view information through the graphs is very useful in understanding the inconsistency, and deciding which solution should be adopted. It is also very useful to see at a glance if one solution was adopted in many views or not (if more views have one solutions, then the chances of being the right solution are higher).

For our example, we decide to keep this global activity. It is then added to William's view and to the final model. Then no other nodes can be found at that level, so we are going to the next level.

The first problem found at the second level is the "modify_deliverable" activity not grouped with the same activities: in Bob's case, it is under the "deliverable_validation" activity, but in Peter's case it is under the "document_production" activity (inconsistency (b) above). This problem is presented in Figure 63.

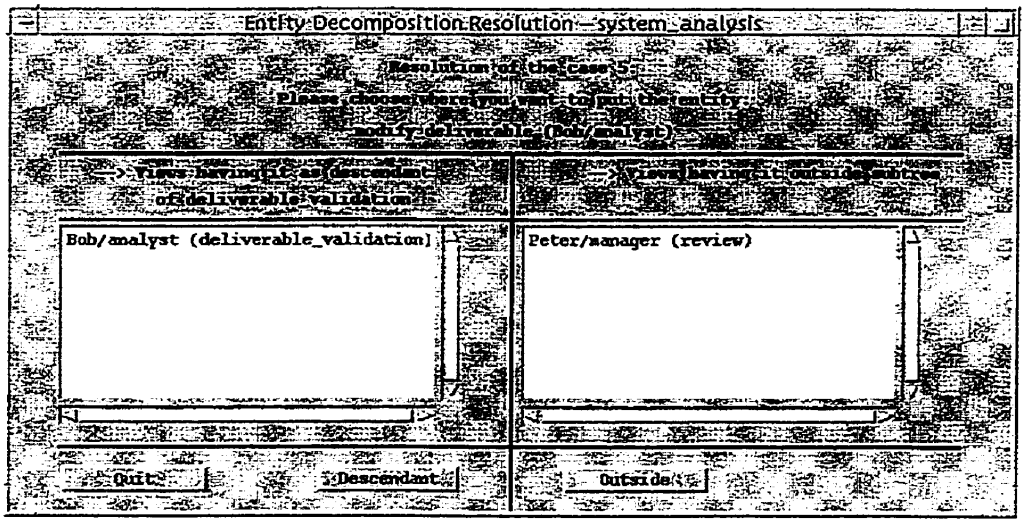


Figure 63 - Resolving when an entity is under different subtrees

Again, the elicitor can choose whether to group the entities as specified or not. Notice that William's view does not appear in the lists: since it does not contain the entities involved in this grouping problem, it does not contain any of the solutions presented.

The merging process goes on this way, resolving one inconsistency at a time. In our example, two other kinds of inconsistencies were found: when an entity is further decomposed in one of the views (inconsistency (c) above), and when an entity is missing in at least one of the views (inconsistencies (d) to (g) above). Figure 64 and Figure 65 show the windows used for these two kinds of inconsistencies respectively.

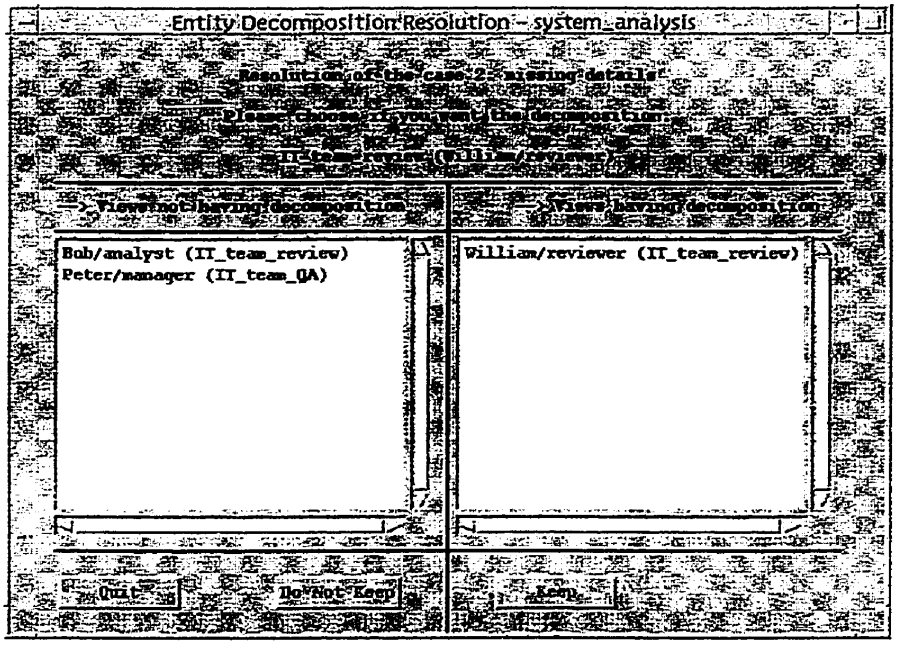


Figure 64 - Resolving when more details are provided in some views

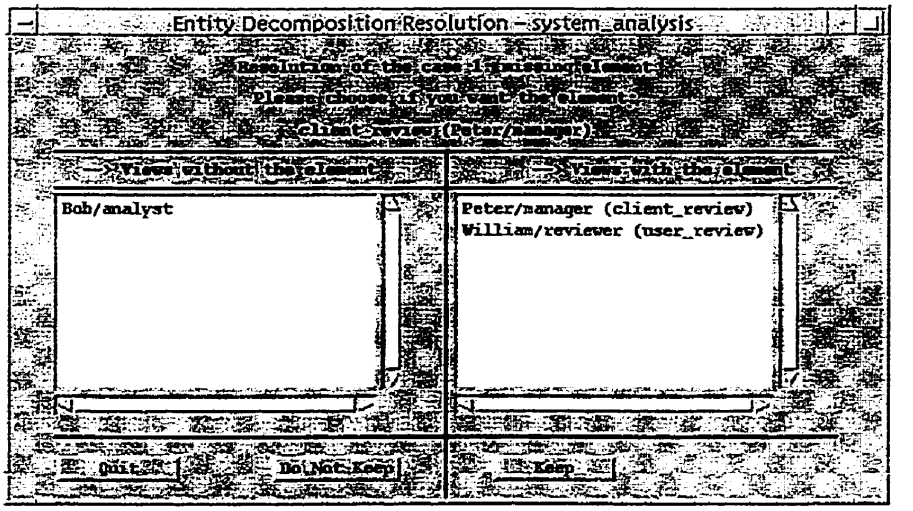


Figure 65 - Resolving when an entity is missing in some views

The final result is an entire activity decomposition for the final (merged) model, satisfying the decisions taken by the elicitor through the merging process. Figure 66 shows one final (merged) activity decomposition aspect we could get with our example.

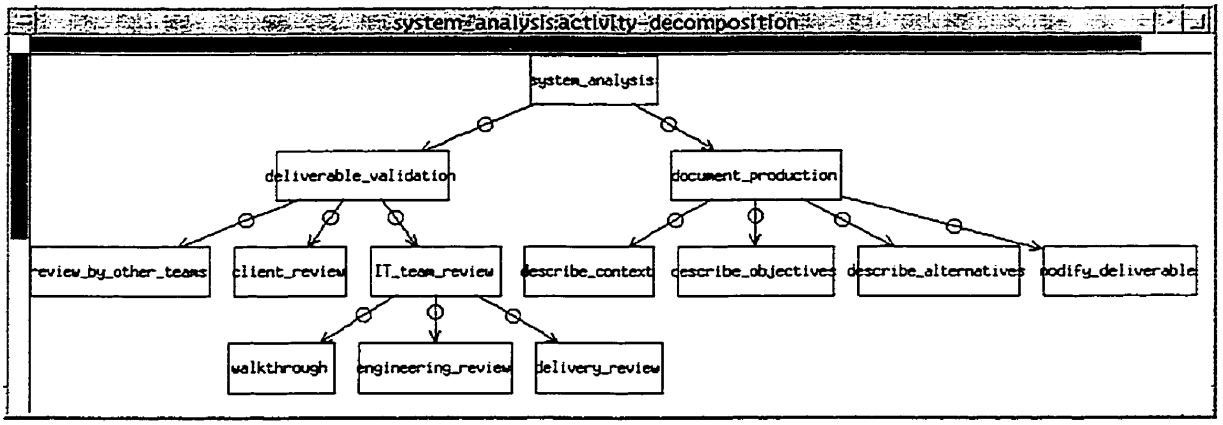


Figure 66 - Final model after resolving the inconsistencies (activity decomposition only)

Inconsistencies are categorized into basic inconsistency types, such as the "missing details" and "missing element" cases above. A set of boolean characteristics is used for characterizing these categories. They are evaluated on one entity, with respect to another view. For example, in the case of a missing element, the characteristics are:

- the element is not matched (e.g., "client_review" activity in Peter's view has no similar entity in Bob's view)
- none of its descendants are matched
- at least one of the siblings or descendants of siblings is matched

The detection of inconsistencies is performed by evaluating the set of characteristics on each children in each view, with respect to each other view. When a problem is found, the layout of the resolution window presented is dependent on the basic type of inconsistency found.

The description of each type, with their characteristics, is given in Section 6.3.1.

5.2.3.2.2 Resolving other types of inconsistencies

For the three other types of inconsistency ((ii) related to entity names, (iii) relationships, and (iv) attributes, introduced earlier), a similar approach is taken. The difference is in the complexity of the basic types. In the case of the entity names, there are only two cases: the names are either the same or they are different. Similarly, we have only two cases for inconsistencies related to relationships: the relationship is either in all the views or not in some of the views. The same apply for inconsistencies related to attributes.

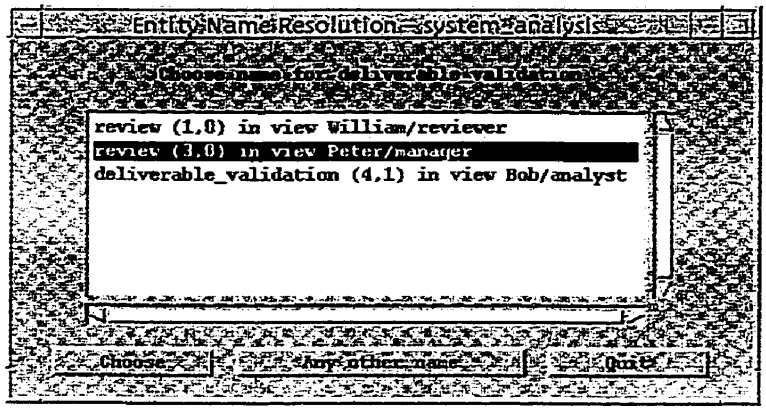


Figure 67 - Resolving an inconsistency related to entity names

The window used for resolving an inconsistency related to the name is shown in Figure 67. An important information to take into account in this resolution is the number of modifications made to the initial view during the entity decomposition resolution. For example, Bob's "deliverable_validation" activity is matched to Peter's "review" activity, and they do not have the same name. But they were not containing the same set of activities originally: the "modify_deliverable" activity was under the "review" part in Bob's view but not in Peter's view. This could be a reason for the name difference. Since we have decided to put the "modify_deliverable" activity under the "document_production" part (see Figure 66), the name used to express the review activity without the modification (i.e., the name used in Peter's view) has higher chances of being the right one. This information on how much the entity has been modified is given as the tuple "(<number of added

subentities/subtrees>, <number of deleted subentities/subtrees>)" after each name in Figure 67.

Resolution of attributes is similar to the name resolution, showing the different values from which to choose instead of entity names in the resolution window. Figure 68 shows such a window for resolving the different duration for the "other_teams_review" activity. Notice that the name used for the activity in each view is specified, such that when looking at the graph information, the entity having the problem can be identified. Notice also that when a view does not even contain the entity, it is not listed in the resolution window (this is the case with William's view here).

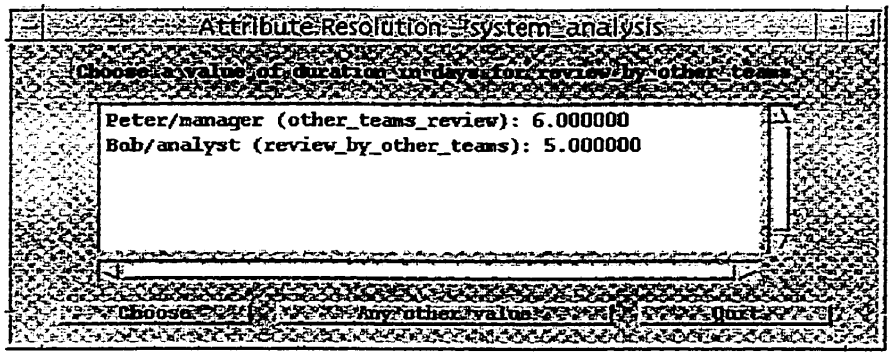


Figure 68 - Resolving an inconsistency in the attributes

The resolution window of inconsistencies related to relationships just needs to ask whether the elicitor wants this relationship or not (Figure 69). In order to help him/her choosing whether or not to keep one relationship, the system shows which views contain the relationship and which views do not, in a way similar to that for entity decomposition inconsistencies. In the case that one view does not have the relationship, but that its original view (before the modifications in the entity decomposition resolution step) does not contain one of the entity involved in the relationship (i.e., this source couldn't give this relationship), the view is not shown in the resolution window. In our example, William's view is not shown because it does not contain the "review_by_other_teams" activity.

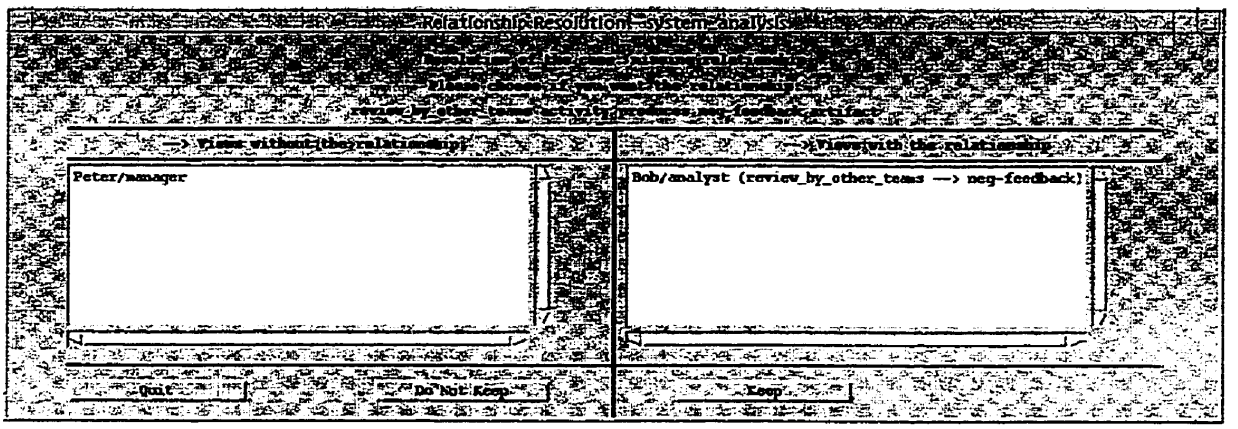


Figure 69 - Resolving a missing relationship

The result of merging the three views is shown in Appendix B, where each aspect of the final model is presented, as one can visualize it in the V-elicit system. Notice that the final result could be different if the elicitor decided to resolve the inconsistencies in a different way.

5.2.3.2.3 Summary

In this section, we have shown the general idea of how the inconsistencies across views are detected and resolved, through an example. We have seen how the V-elicit system can help the elicitor in performing his/her knowledge-intensive task, and keeping track of the rationale for resolving the inconsistencies in one way or another. However, we did not discuss all the possible types of inconsistencies, for example when we get two completely different (unrelated) decomposition of an entity from the different views. These are discussed in detail in Section 6.3.

5.2.4 Steps 6 and 7: Check model quality and modify model

Once the views are all merged into one model, we have to check again the quality of the model (step 6), as we did for each view in step 3. We are using the same technique (*constraint verification*) as in step 3. Typically, this step would also involve validating the merged model with the persons who provided the information in the initial views.

The problem here is that an entity in the final model may contain more information than any of the similar (*matched*) entities in the different views. This combination of information may bring new inconsistencies, that couldn't be detected in the separate views.

New constraints can also be checked now, using the new superset of entity/relationship/attribute types. For example, if no view had the information flow aspect together with the artifact dependency aspect, we couldn't check earlier for consistency between these two types of information. In the merged model, we can add this verification.

When a problem is found, we have to make the appropriate modifications to the merged process model (step 7). The X-elicit tool is used again in this step, as for step 2. The difference is that only one X-elicit model is created, not one per view, but the mapping between the V-elicit data structure and the X-elicit templates is done in the same way. The set of entity/relationship/attribute types presented in X-elicit is the union of the types in each view.

The final model can then be checked against development policies, using external validity constraints. For example, the elicitor can check that each document has been reviewed independently. Again, the same technique as in step 3 is used (constraint verification). Such validation, although it does not detect problems in the model quality, does constitute an important feedback for process analysis and improvement.

5.3 Summary of the elicitation approach

In this chapter, we have described our approach for eliciting software process models using different views. The following steps were described: planning for elicitation, eliciting each view, checking each view for intra-view consistency, identifying common components across views, merging views (including detecting and resolving inconsistencies across views), checking the quality of the merged model, and modifying this model if necessary.

In the planning step, the scope of the model to be elicited is defined, with the kind of information to be elicited. The different sources of information are then analyzed, and a subset of these sources (views) are chosen for elicitation purposes.

Each of the views are then elicited. The information is entered using the X-elicit tool or unstructured (draft) text files, and the information can be visualized in graphs. Constraints are used for checking the consistency and quality of the views.

Before merging all the views, the common components across the views are identified using the matching algorithm. A similarity score is computed for each pair of entities of the same type, based on the related information such as the entity name, relationships with other entities, and attributes. The highest scores define the matches between the entities, which can then be modified by the elicitor.

Having identified the common components, the inconsistencies across views are then detected, and subsequently resolved with the help of the elicitor. The final model is built accordingly, and checked against development policies using external validity constraints.

Overall, these steps define a systematic approach, with tool support, for view-based elicitation. They satisfy the set of requirements specified in Chapter Three. Our approach,

together with the set of techniques presented, helps in eliciting process models of high quality (as shown later on in Chapter Seven, on validating our approach and system).

Chapter Six - Techniques for consistency checking and view merging

In working on the view-based elicitation problem, we have focused on developing new techniques for checking inconsistencies within and across the views, and merging these views. Such a complete set of techniques does not exist in other process elicitation tools.

The next sections describe the details of the specific techniques developed: constraint verification, component matching, and view merging. These techniques were used in steps 3, 4, and 5 of the elicitation method, respectively. Constraint verification was also used in step 6.

6.1 Constraint verification

As we have seen in Sections 5.2.2.2 and 5.2.4, constraints are used for checking inconsistencies within a view or a model. These constraints can be defined by the elicitor, using the user-defined types of information and a language similar to first-order logic.

In this section, we describe the constraint language in detail (6.1.1), we discuss the different types of constraints (6.1.2), we explain how the attribute and relationship generators can be used to simplify the specification of the constraint (6.1.3), and we give implementation details of our algorithm (6.1.4). The last section summarizes our constraint verification technique.

6.1.1 Constraint language

This section describes the language developed for specifying constraints. The formal notation in Extended Backus-Naur Form (EBNF) is given in Appendix C.

Constraints are composed of *ForAll* and *ThereIs* clauses that define the variables to be used, followed by a condition to be verified using the defined variables. The meaning of these clauses is that for all values (*ForAll*) or at least one value (*ThereIs*) of the variable defined, the rest of the constraint (including following *ForAll* and *ThereIs* clauses) should be true.

The variables in the clauses are defined in the format "variable is element of a specified set". Variables can be either entities or relationships. A relationship variable is of the form r(e1,e2,t) where r is the variable name, e1 and e2 are variables to be used for accessing the entities involved in the relationship, and t is the variable containing the type of the relationship. The set on which the variable is defined can be either a predefined set (E for the set of all entities, and R for the set of all relationships), a set derived from other sets using the union and intersection operations on sets, or a user defined set where the elements are a subset of another set based on some condition on the elements (i.e., a set defined using the construct $\{a \in A|...\}$). In the case of user defined sets where the condition is the type of entity or relationship, a shorthand can be used (TypedEntSet or TypedRelSet respectively).

For the condition in the constraint (or in the specification of a user defined set), we can use the attributes of the entities, and operate on them through usual mathematical notation. In some cases, we have defined specific functions for such operations, such as GetInterval between two time attributes or values. We have also added functions for testing that entities are connected through relationships of specified types (e.g., ThereIsRel, ThereIsPath, etc.). The condition can be an aggregated one, using the standard boolean operators: "not", "and", "or", and "implies".

This constraint notation developed is based on standard mathematical notations, and should thus be complete. The functions we have added are either based on the specifics of the ERD modeling notation (e.g., functions to verify if a relationship or an attribute exists), or they are shorthand for more complex functions used very often (e.g., the

function "SameEnt" that verifies if two variables point to the same entity, which is just a shorthand of verifying that the two entities have the same name and the same type).

6.1.2 Type of constraints

In Section 5.2.2.2, we have seen examples of two kinds of constraints: one related to the structure of the views (Figure 47), and one related to the meaning of the entities and relationships modeled (Figure 48). In general terms, a view or a model can be inconsistent due to two reasons: (a) *internal invalidity*, that is related to the *structure* of the view or *incompleteness* of the information elicited; and (b) *external invalidity*, that is related to the *fitness* of the view with respect to *organizational or project policy*. This is analogous to the syntactic and static-semantic errors (internal invalidity), and deep-semantic errors (external invalidity) in computer programs.

Below, we give examples of the two types of constraints.

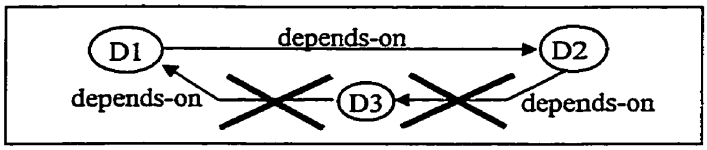
6.1.2.1 Constraints to check internal validity

We describe four example constraints: cycles in dependency graphs, production of artifacts across the levels of abstraction, consistency between activity dependency and inputs/outputs, and consistency of cost between levels of abstraction. Graphs are provided to help the reader understand the constraints written, but such graphs are not provided in our system.

(a) the artifact dependency graph should not contain cycles: for all relationship r of type "artifact depends-on artifact" between artifact d1 and artifact d2, there is no path of (one or more) relationships of type "artifact depends-on artifact" from d2 to d1. In the example graph below, we have such path passing through d3, but more entities could be involved, or we could have a direct link from d2 to d1.

$\forall$(r(d1,d2,type)$\in$TypedRelSet(artifact depends-on artifact))•

$\neg$ThereIsPath(d2,d1,artifact depends-on artifact)



(b) consistency of the relationships of type "activity produces artifact" across different levels of abstraction: for all relationship r1 of type "activity produces artifact" between activity a1 and artifact d, and for all relationship r2 of type "artifact is-consumed-by activity" between artifact d and activity a2, if the parent of activity a1 is not the parent of activity a2, then there should be a relationship of type "activity produces artifact" between the parent of a1 to d.

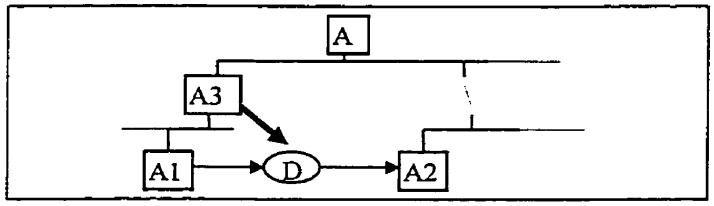
$\forall$(r1(a1,d,type)$\in$TypedRelSet(activity produces artifact))•

$\forall$(r2(d,a2,type2)$\in$TypedRelSet(artifact is-consumed-by activity))•

(ThereIsRel(a3,a1, activity is-composed-of activity) $\wedge$

$\neg$ThereIsPath(a3,a2,activity is-composed-of activity))

$\Rightarrow$ ThereIsRel(a3,d,activity produces artifact)

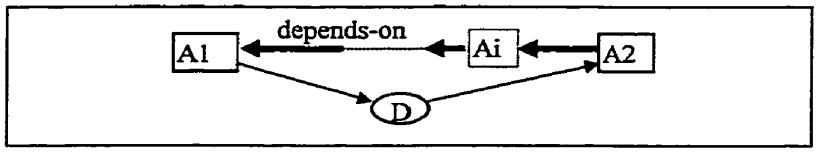


(c) consistency between the input/output relationships and dependency relationships: for all relationship r1 of type "activity produces artifact" between activity a1 and artifact d, and for all relationship r2 of type "artifact is-consumed-by activity" between artifact d and activity a2, there is a path of (one or more) relationships of type "activity depends-on activity" from a2 to a1.

$\forall$(r1(a1,d,type)$\in$TypedRelSet(activity produces artifact))•

$\forall$(r2(d,a2,type2)$\in$TypedRelSet(artifact is-consumed-by activity))•

ThereIsDirectedPath(a2,a1,activity depends-on activity)

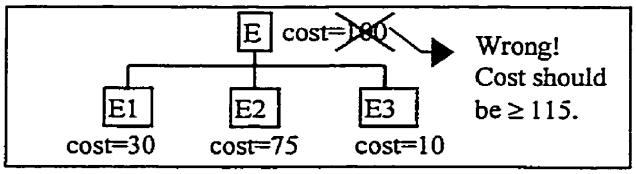


(d) consistency of cost across levels of abstraction: for all activity e, the cost of the activity is greater than or equal to the sum of its children's costs.

$\forall$(e $\in$ TypedEntSet(activity))•

e.cost $\geq$ Sum(cost,{ei$\in$TypedEntSet(activity) |

ThereIsRel(e,ei,activity is-composed-of activity) })



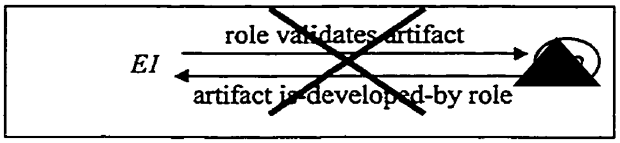
6.1.2.2 Constraints to check external validity

In this section, we show four examples of known software development principles (two process-related and two product-related), written in the constraint format, such that they can be checked by V-elicit: independent validation, development-phase ordering, side-effects, and interface complexity.

(a) independent validation (all software artifacts should be validated by people (reviewers) other than those who have developed them) (see [Pre97], section 17.1.2): for all relationships of type "role validates artifact" from e1 to e2, there is no relationship of type "artifact is-developed-by role" from e2 to e1.

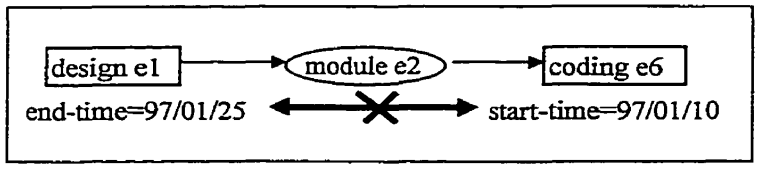
$\forall$(r(e1,e2,type)$\in$TypedRelSet(role validates artifact))•

$\neg$ThereIsRel(e2,e1,artifact is-developed-by role)



(b) a module should be fully designed before coding it (see [Pre97], sections 2.1.2 and 13.1): for each module artifact produced in the design phase and consumed by a coding activity, the design activity should be completed before the coding activity begins.

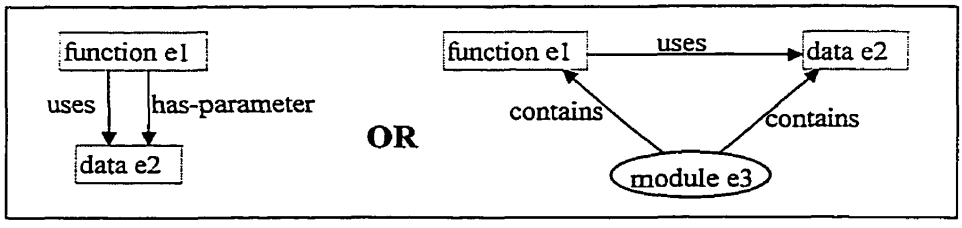
$\forall$(r(e1,e2,type)$\in${r2(e3,e4,t1)$\in$TypedRelSet(activity produces artifact)|
(subtype(e4)=module) $\wedge$ (e3.phase=design) })•
$\forall$(r3(e5,e6,type2)$\in${r4(e7,e8,t2)$\in$
TypedRelSet(artifact is-consumed-by activity) |
(e7=e2) $\wedge$ (e8.phase=coding) })•
e1.end-time $\leq$ e6.start-time



remark: "phase", "start-time" and "end-time" are attributes defined for activities.

(c) side effects (data coupling) (see [Pre97], section 13.5.4): if a function reads some data, then it should be passed through the parameters or the function and data should be defined in the same module (or class).

$\forall$(r(e1,e2,t)$\in$TypedRelSet(function uses data))•
$\exists$(e3 $\in$ {e4 $\in$ TypedEntSet(module)|
ThereIsRel(e4,e1,module contains function) })•
ThereIsRel(e1,e2,function has-parameter data) $\vee$
ThereIsRel(e3,e2,module contains data)



(d) interface complexity (coupling) (see [Pre97], section 13.6): each function should have at most 7 parameters (note: this standard can be specified in an organization, as a way to avoid interface complexity, but the number can vary from one organization to another).

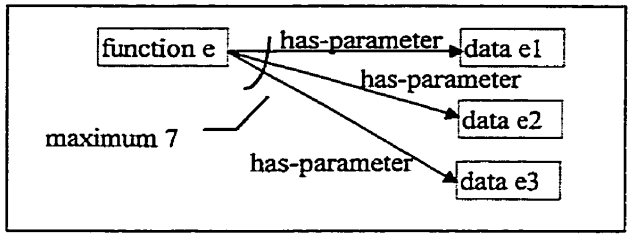
$\forall$(e $\in$ TypedEntSet(function))•

{ei $\in$ TypedEntSet(data) |

ThereIsRel(e,ei,function has-parameter data) }

≤ 7

where '#' means cardinality of the set



6.1.2.3 Summary and analysis

Both *internal validity* constraints and *external validity* constraints can be checked, as shown in Section 5.2.2.2. The kinds of inconsistencies found using *internal validity* constraints are similar to the kinds of checking done in other modeling languages such as Statemate [KeH89], FUNSOFT Nets [GrS92], and OBM [SaW94]. The advantage in V-elicit is that it can be done on any user-defined type of information. *External validity* constraints, to our knowledge, are not checked in any other process modeling tool. Some are described in [DNR90, NeR91], but they are specified informally.

It is important to note that if an elicited view (or model) violates an *external validity* constraint, it could be for three reasons, which (ideally) should be investigated and an appropriate course of action should be taken: (i) the process view is incorrectly elicited; (ii) the process itself is defective; and (iii) the constraint itself is not valid.

The number of constraints to be specified for a comprehensive coverage of the organizational policies can be huge, and in such an environment these constraints would have to managed appropriately. Also, the issue of designing a comprehensive set of policies, and ensuring that the constraints specified do not have conflicts, is not trivial.

Such issues are out of the scope of this thesis, and they will probably be dealt with in future work.

Finally, simulation is often used for *dynamic analysis* of a process model [KeH89, GrS92, ADH94]. The problems detected in this manner include deadlocks, reachability problems, race conditions, etc. This kind of analysis is complementary to our constraint verification. In fact, one approach to quality process modeling and analysis is that tools such as V-elicit could be used to build rigorous descriptive models prior to spending resources to carry out dynamic analysis.

6.1.3 Use of generators

In order to simplify the specification of a constraint using multiple relationships, we can use generated types. For example, if we want to check for independent validation (as described in part (a) of Section 6.1.2.2), but that we have only the following types available:

activity is-performed-by role
activity produces artifact
activity validates artifact

the constraint would have to be specified as:

∀(r1(a1,d1,t1)∈TypedRelSet(activity produces artifact))•
  ∀(r2(a2,d2,t2)∈{r21(e1,e2,t21)∈TypedRelSet(activity validates artifact)
                                                | SameEnt(d1,e2) })•
  ∀(r3(a3,r1,t3)∈{r31(e3,e4,t31)∈TypedRelSet(activity is-performed-by role)
                                                | SameEnt(a2,e3) })•
  ¬ThereIsRel(a1,r1,activity is-performed-by role)

This complex specification can be reduced to the following constraint:

∀(r(r1,d1,type)∈TypedRelSet(role validates artifact))•
  ¬ThereIsRel(r1,d1,role develops artifact)

with the following two specifications for the linear generator for relationships[19]:

1. existing types: role performs activity
   activity validates artifact
   new type: role validates artifact

2. existing types: role performs activity
   activity produces artifact
   new type: role develops artifact

By using generated types in the constraint specification, the understanding of the constraint is greatly improved, and so is the efficiency of the constraint checking algorithm.

6.1.4 Implementation details

The constraints are built using one object per element of the language defined in Appendix C. In the case of non-terminating symbols, the object contains pointers to the possible elements of the derivation used, and this derivation is kept in a type variable.

A set of functions (one per language element, including punctuation symbols) is used for reading a constraint in a string format and creating a constraint object. The syntax of the constraint definition is verified at this point.

Each of the objects in the constraint has two major functions: "verify" for checking the correctness of the constraint defined (e.g., no use of undefined variables, no type mismatch between a variable and the set from which it is defined, etc.), and "evaluate" for verifying if the constraint is satisfied in the model or view specified. In the case of the "ForAll" and "ThereIs" parts, there are also functions for building the set of values for the variable specified. The set is built during the evaluation of a constraint on a model or a

[19] Refer to Section 4.3.2 for information on how to describe such relationship generator.

view. Notice that in the case where the set is not related to the first quantifier of the constraint, it will have to be rebuilt for each value of the variable related to the first quantifier.

As the constraint is evaluated on a model or a view, the values of the variables making the constraint evaluate to false are kept in a list, and they are printed only when the constraint evaluation is finished.

6.1.5 Summary of the constraint verification feature

In this section, we have presented the details about the use and implementation of the constraints. We have seen that two types of constraints can be defined: internal validation constraints and external validation constraints. We have also discussed how the generators can be used to help in the constraint definition. Details on the language and its implementation have also been provided.

Constraint verification is a mechanism that permits the verification of models and views modeled using a user-defined language. A constraint defines what an inconsistency is, based on the type of information used in the model. The traditional ways of checking for inconsistencies do not permit one to work on arbitrary type of information, and they do not allow one to define inconsistencies, so the semantics of the model cannot be verified (as with the external validity constraints).

Our work on constraint definition has been influenced by the work of Behm and Teorey [BeT93], who have used relative constraints as a way of capturing business rules in first-order logic (e.g., a project's budget cannot exceed its department's budget). These constraints were not formally defined and no tool was available for verifying the constraints, but they had the idea of using first-order logic to let the user define his/her own constraints.

6.2 Component matching

The second technique developed for V-elicit is component matching. It is the first step in merging the elicited views into one model. Its goal is to find the entities in different views that represent the same process element (e.g., same activity, artifact, role, etc.).

In Section 5.2.3.1, we have given an example on how this was used by the elicitor, focusing on the user interaction and the results presented to the user. In this section, we describe the internal algorithm used for computing the similarity score and choosing the matched entities (6.2.1), and we explain how the relationship generators can be used to help compare entities (6.2.2). The last section summarizes our component matching technique.

6.2.1 Algorithm and formula for computing similarity scores

This section describes how the system computes the similarity score used for matching the entities across views (i.e., identify which ones are the same). The algorithm is actually performed for each pair of views, independently of the other views. Throughout this section, we uses the two views in Figure 70 and Figure 71. These views contain few entities and relationships, making it easier for the reader to understand the algorithm. The same algorithm can be applied to the views in Section 5.2 as well.

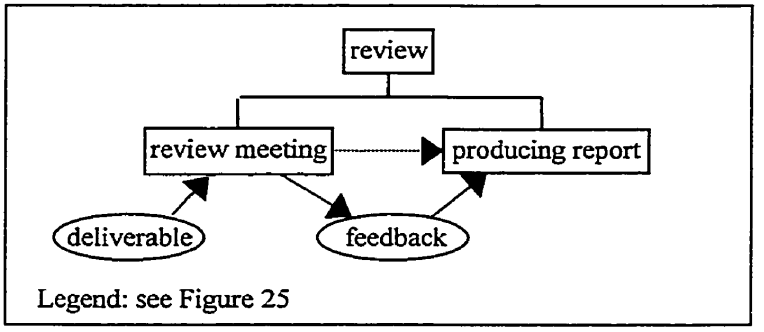


Figure 70 - Sam's view

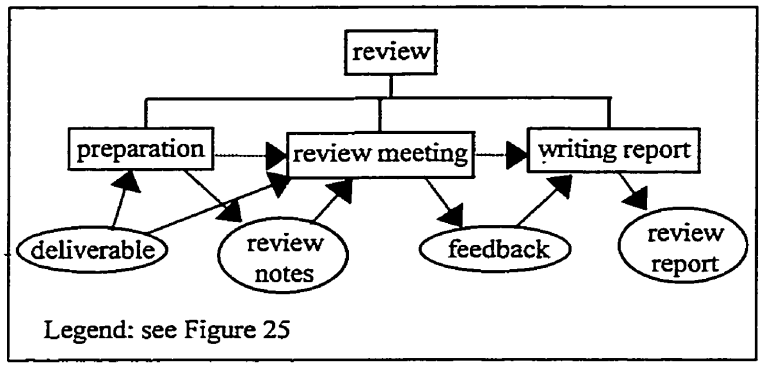


Figure 71 - Sally's view

The general idea in computing the similarity score between two entities is to compare the elements related to these entities (name/relationship/attribute). For example, if we look at Sam's "review meeting" activity, there are five related elements: the name of the entity, and the relationships "deliverable is-consumed-by this entity", "this entity produces feedback", "this entity precedes producing_report" and "review is-composed-of this entity". When comparing this activity to Sally's "review meeting" activity for example, we check to see if the latter activity also contains these five related elements.

It is easy to compare entity names and related attributes, but not relationships because they are related to other entities not necessarily matched yet. For example, how can we know that Sam's "review_meeting precedes producing_report" relationship is the same as Sally's "review_meeting precedes writing_report" relationship, if we have not compared yet the "producing_report" activity and the "writing_report" activity?

The approach taken, as described by the algorithm in Figure 72, is to match the entities one type at a time (line #3), and to use the results of the previous iterations for comparing the related components of the entities (managed in lines #2 and #19). In our example, we can decide to match the artifacts first, and then the activities. This choice is made by the elicitor (line #1, performed in a window such as that in Figure 51).

```
Algorithm

 1. determine ordering of entity types to match
 2. types_matched ← empty set
 3. for each entity type et
 4.     rels_first_pass ← set of relationship types between et and
                                                    et2 ∈ types_matched
 5.     rels_second_pass ← set of relationship types from et to et
 6.     for each pair of views
 7.         for each pair of entities (one in first view and one in second view)
 8.             compute first_pass_score
 9.         for each pair of entities (one in first view and one in second view)
10.             compute second_pass_score
11.         for each entity in first view
12.             determine its best match (entity with highest similarity score)
13.         for each entity e1 in second view
14.           determine its best match (entity e2, with highest similarity score)
15.             if e2's best match is e1
16.               if the score of the best match is higher than a minimum score
17.                 then e1 and e2 are matched
18.     show result to user and let him make changes to matches
19.     add et to the set types_matched
```

Figure 72 - Component matching algorithm

When matching the artifacts, we compare only the names and attributes of the artifacts. We cannot use the relationships with the activities because at this point, we do not know which activities are the same across the views. We could use these relationships only if the activities were matched first. The result of this iteration, for our example, is that each pair of artifacts having the same name are matched (there is no attribute to consider in this example).

With this result, we can then compare the activities, using the names, attributes, and relationships with the artifacts (rel_first_pass in line #4 of Figure 72). The score computed using this information is called the first pass score (see Figure 73)[20]. For example, in Sam's "review meeting" activity, we have three related elements that can be compared: the entity name, and the relationships "deliverable is-consumed-by this entity" and "this entity

[20] This is computed in lines #7-8 in Figure 72.

produces feedback". If we compare this activity with Sally's "preparation" activity, we see that there is only one similar related element (the relationship "deliverable is-consumed-by this entity"), so the first pass score is 0.33:

$$FirstPassScore = \frac{NameSim + NbAttMatch + NbRelMatch}{1 + NbAtt + NbRel} = \frac{0+0+1}{1+0+2} = \frac{1}{3} = 0.33$$

First pass score from entity A in first view to entity B in second view

$$FirstPassScore = \frac{NameSim + NbAttMatch + NRelMatch}{1 + NbAtt + NbRel}$$

NameSim:
- split A's name and B's name into words, keeping only their stem and removing unwanted words such as "the" (the list of unwanted words in user-definable)
- NameSim = (# words of A's name that is also in B's name) / (# words of A's name)

NbAttMatch:
- For each attribute of A (that should be considered),
  add 1 in NbAttMatch if attribute value is matched to B's attribute value.

NbAtt:
- NbAtt = Number of attributes of A
  (that should be considered in matching algorithm)

NbRelMatch and NbRel:
- For each relationship of A (of type that should be considered)
  Add 1 to NbRel
  Let A' be the second entity of the relationship considered (rel. of type t)
  If there is a relationship of type t between B and B' in view 2,
  and that B' has been matched to A', then add 1 to NbRelMatch

Second pass score from entity A in first view to entity B in second view

$$Score = \frac{(NameSim + NbAttMatch + NbRelMatch) + IntRelScore}{(1 + NbAtt + NbRel) + NbIntRel}$$

Let MS (Mean Score) between entity A and B be the mean of FirstPassScore from A to B and FirstPassScore from B to A.

For each relationship with entity A (of type that should be considered)
- Let A' be the second entity of the relationship considered (rel. of type t)
- Find B' such that we have a relationship of type t between B and B', and MS between A' and B' is the maximum one.
- If the MS found is higher than MinScore, then add MS to IntRelScore.
- Add 1 to NbIntRel if MS has been added to IntRelScore

Figure 73 - Formula for computing similarity scores

Now, if we compare Sam's "review_meeting" activity to Sally's "review_meeting" activity, all the related elements are found, and the score is 1.00:

$$FirstPassScore = \frac{NameSim + NbAttMatch + NbRelMatch}{1 + NbAtt + NbRel} = \frac{1+0+2}{1+0+2} = \frac{3}{3} = 1.00$$

Remark: the same formula (first pass score) was also used when matching the artifacts, with the following variables set to 0: NbAttMatch, NbAtt, NbRelMatch, and NbRel.

Notice that if the starting point of the comparison is Sally's view, the score is different. For example, Sally's "review meeting" activity contains four related elements (not three as in Sam's view): the entity name, and the relationships "deliverable is-consumed-by this entity", "review_notes is-consumed-by this entity" and "this entity produces feedback". If we compare this activity to Sam's "review meeting" activity, we can see that there are only three similar related elements, for a first pass score of 0.75:

$$FirstPassScore = \frac{NameSim + NbAttMatch + NbRelMatch}{1 + NbAtt + NbRel} = \frac{1+0+2}{1+0+3} = \frac{3}{4} = 0.75$$

Both scores should be computed, the mean being used in the next step when computing the similarity score.

The NameSim part in our example returned a 1 in case the two names were the same, and 0 if not. The computation of such value is actually more complex than that, and can return any value between 0 and 1. For example, having to compare the names "documentation review" and "formal review of documents", the system would first separate the words, keeping only the stems and removing unwanted words such as the "of" in the second name. This gives the following words for each name:

"documentation review": document, review

"formal review of documents": formal, review, document

The computation of the name similarity then gives 1.0 (2/2) for the first name, and 0.67 (2/3) for the second one. These values are obtained by dividing the number of words in the name that are also in the other name, by the total number of words in the name. For simplicity of our example, we have not included such case in Sam's and Sally's views.

All first pass scores between Sam's view and Sally's view are shown in Table 5. Each cell contains two numbers (not in parenthesis): the first one is the score computed from Sam's view to Sally's view (i.e., the variables NbAtt and NbRel are set to the number of related elements in Sam's view), and the second one is from Sally's view to Sam's view. The number in parenthesis is the related fractional number, that will be used (numerator and denominator separately) in the second pass.

	Sally							
Sam	review		preparation		review meeting		writing report	
review	1.00 (1/1)	1.00 (1/1)	0.00 (0/1)	0.00 (0/3)	1.00 (1/1)	0.13 (0.5/4)	0.00 (0/1)	0.00 (0/3)
review meeting	0.17 (0.5/3)	0.50 (0.5/1)	0.33 (1/3)	0.33 (1/3)	1.00 (3/3)	0.75 (3/4)	0.00 (0/3)	0.00 (0/3)
producing report	0.00 (0/2)	0.00 (0/1)	0.00 (0/2)	0.00 (0/3)	0.00 (0/2)	0.00 (0/4)	0.75 (1.5/2)	0.50 (1.5/3)

Table 5 - First pass scores between Sam's view and Sally's view

The relationships between the activities (i.e., relationships of type "activity is-composed-of activity" and "activity precedes activity" in our example) can also be used for determining the matches between activities. After computing the first pass score for each pair of activities between two views, we can use this information to assess the similarity between two activities. This means we can now have an idea about the similarity between the relationship "review_meeting precedes producing_report" in Sam's view and the relationship "review_meeting precedes writing_report" in Sally's view, by checking the first pass score between "producing_report" and "writing_report".

The final score (or "Second pass score" in Figure 73) is the first pass score, improved using the relationships between entities of the same type (rels_second_pass in line #5 of Figure 72)[21]. As an example, using the mean first pass score between Sam's "producing report" activity and Sally's "writing report" activity (0.625)[22], and the mean first pass score between Sam's "review" activity and Sally's "review" activity (1.00), we can compute the final score this way:

$$Score = \frac{(NameSim + NbAttMatch + NbRelMatch) + IntRelScore}{(1 + NbAtt + NbRel) + NbIntRel}$$

$$= \frac{(3) + IntRelScore}{(3) + NbIntRel} = \frac{(3) + (0.625 + 1)}{(3) + (2)} = \frac{4.625}{5} = 0.925$$

The final scores for each pair of entities between Sam's view and Sally's view are presented in Table 6. The highest scores for each entity are highlighted[23].

	Sally							
Sam	review		preparation		review meeting		writing report	
review	**0.833**	**0.708**	0.000	0.000	0.333	0.071	0.000	0.000
review meeting	0.100	0.125	0.400	**0.400**	**0.925**	**0.661**	0.200	0.200
producing report	0.000	0.000	0.250	0.200	0.333	0.190	**0.844**	**0.675**

Table 6 - Final scores between Sam's view and Sally's view

When all the scores are computed, the system can determine the best matches, using the highest scores[24]. For example, for Sam's "review meeting" activity, the scores with Sally' s activities are 0.1, 0.4, 0.925, and 0.2, the 0.925 being with Sally's "review meeting" activity. For Sally's "review meeting" activity, the scores with Sam's activities are 0.071,

[21] This is computed in lines #9-10 in Figure 72.
[22] Using the information in Table 5 (row 3, column 4), we compute the mean between the two numbers: (0.75 + 0.50) / 2 = 0.625.
[23] This is checked in lines #11 to 14 in Figure 72.
[24] This is done in lines #15 to 17 in Figure 72.

0.661, and 0.19, the 0.661 being with Sam's "review meeting" activity. The highest scores in both cases are between the same entities, so the two activities are said to be matched.

We can see that an activity in one view has no match at all in the other view when these highest scores do not coincide. For example, the scores for Sally's "preparation" activity are 0, 0.4, and 0.2, the highest being with Sam's "review meeting" activity, but this activity has its highest score with Sally's "review meeting" activity.

The match can also be rejected in the case where the similarity score is low[25]. This lower bound on similarity score is given by the elicitor or the agent.

As discussed in Section 5.2.3.1, the elicitor should also check the matches found after the matches for one entity type are detected, and make the appropriate modifications if necessary. These modifications should be made before continuing the algorithm with the next entity type, in order to use the appropriate matches in the other iterations.

6.2.2 Use of generators

Hierarchical generators are obviously needed for this algorithm. Relationships and attributes are often specified for entities at lower levels only in the decomposition hierarchy. Such information is necessary for entities at upper levels in order to match them.

In the case of the linear relationship generators, they permit the elicitor to add other relationships that can be used in computing the similarity scores. These additional relationships are typically generated from relationships that cannot be used yet because the entities involved are not matched. For example, the relationship "artifact is-needed-for artifact" (generated from the produces/consumes relationships with activities) can be used

[25] This is checked in line #16 in Figure 72.

to replace the relationships "activity produces artifact" and "artifact is-consumed-by activity", that cannot be used when the activities are not matched.

In the example presented in the previous section (Sam and Sally), such mechanism is not necessary because the artifacts can all be matched just by their name. However, in real situations, artifact names are often different across views, making it impossible to match them without using additional information.

In order to illustrate such situation, lets modify Sam's view from previous section (the modified view is shown in Figure 74), keeping Sally's view the same (Figure 75). Generated relationships across artifacts are also shown in these two figures.

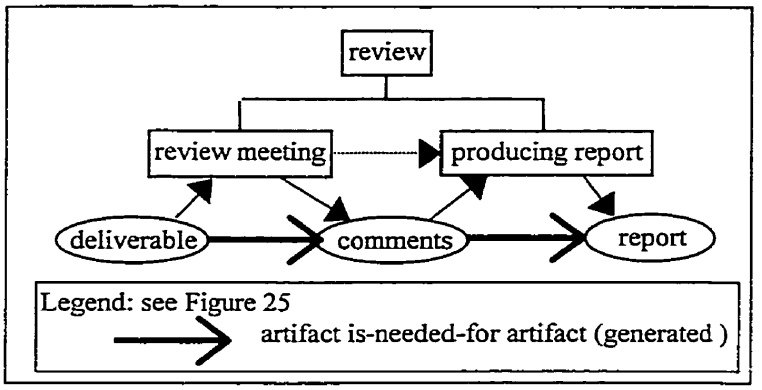


Figure 74 - Sam's view modified (including generated relationships)

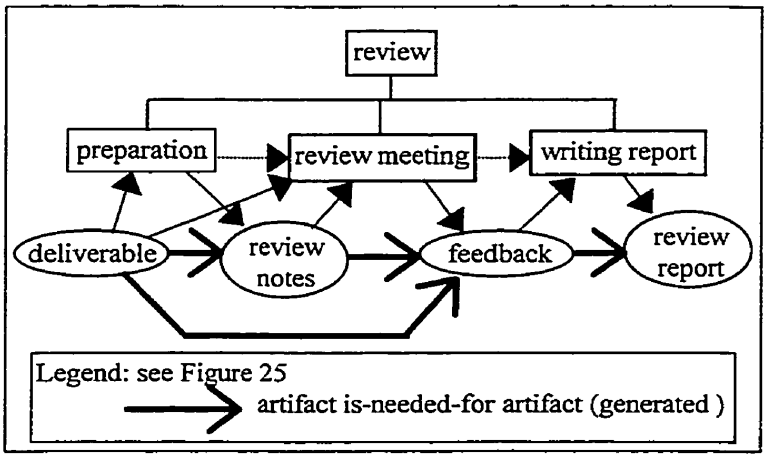


Figure 75 - Sally's view (with generated relationships)

As one can see, the artifacts "feedback" and "comments" are actually the same artifact, and they should be matched. However, because they do not have the same name, they are not matched (the similarity score is 0).

By using the generated relationships "artifact is-needed-for artifact" in the second pass of the matching heuristic, the similarity score between "feedback" and "comments" becomes 0.58, and these two entities are then matched (the similarity score between "comments" and "review notes" is only 0.5).

Linear relationship generators are very useful with the entity types that are matched first, because there is not much information we can use, and the generators increase the chances of finding the right matches.

6.2.3 Summary and analysis of the component matching feature

In this section, we have provided details on how the component matching algorithm finds similar entities across views. We have presented the formula used for assessing the similarity between two entities from two different views ("similarity score"). From this information, the entities that are similar are detected, or at least easily identified using the similarity score. We have also discussed the parameters that the user can modify to help obtain a better matching result: the ordering of the entity types and the use of generators. Once we know which entities are similar, we can then see the differences (or inconsistencies) across the views. This is the topic of the next section.

Our algorithm is based on Leite and Freeman's heuristics [LeF91] for matching rules in different software requirement descriptions (or idea behind the heuristics). They have applied their heuristics to facts and rules: similarity scores for facts are derived by comparing each word of the facts (as in our "NameSim"), and a combined score for rules is then computed using scores on facts and weights. Our similarity scores are computed in a similar way, but they are applied to ERD instead of rules and facts. Also, we have

developed the two-pass formula in order to be able to use additional information (i.e., relationships between entities of the same type), which is critical when there is not much information that can be used.

The time complexity of our matching algorithm is as follows[26]:

$$t(nbviews, nbents, nbrels) \in O(nbviews^2 * (nbents^2 + nbrels^2))$$

where: nbviews = number of views

nbents = number of entities

nbrels = number of relationships

t = the computing time of the algorithm, as a function of the number of views, entities, and relationships

Indeed, the algorithm has to run on each pair of views ($nbviews^2$), and for each pair, the entities and relationships of one view are compared to each entity and relationship of the other view ($nbents^2 + nbrels^2$).

As one can see, this algorithm is polynomial, so it is considered as "efficient" (i.e., it can handle quite large models) [BrB96]. However, it might still take a lot of time on large-scale models. In such cases, it would probably be best to separate the process into more manageable pieces, for example by modeling each software development phases (e.g., requirement analysis, design, coding, etc.) separately. This would reduce the number of entities to be dealt with at the same time when merging views, reducing the time required to perform such a task. However, one should be careful here in the re-composition of the entire process model from the separated development phases. Further work is necessary here, for investigating into approaches for handling large models, containing thousands of entities.

---

[26] The time complexity, specified using an asymptotic notation ("Big-Oh"), means that the actual running time of the algorithm is bounded by the function indicated, multiplied by a constant. The constant is related to the actual number of instructions performed, and the speed of the computer used. [BrB96]

6.3 View merging

The third technique developed for V-elicit is view merging.

In this section, we describe the different types of inconsistencies across views, how each are detected and resolved, and how the final model is built (Section 6.3.1 to Section 6.3.3). We then present how other researchers deal with the problem of inconsistencies across views, even in other domains such as requirement elicitation and knowledge engineering (Section 6.3.4). The last section summarizes our view merging technique.

6.3.1 Detecting and resolving inconsistencies related to entity decomposition

When resolving the entity-decomposition kind of inconsistency, we are interested in entities that are missing, as well as in the different grouping of entities in different views. Examples of such inconsistencies are the "system_analysis" activity missing in William's view (the root activity is not the same), and the "IT_team_review" activity decomposed in William's view but not in the other views (see Figure 25 to Figure 27).

As we have seen in Section 5.2.3.2.1, the inconsistencies found across views are categorized into basic inconsistency types. They are identified using a set of boolean characteristics, evaluated on one entity in one view with respect to another view. We have identified eight such basic inconsistency types, and two cases where there is no inconsistency (see Table 7). Each of these cases is described later in Sections 6.3.1.1 to 6.3.1.10.

	Inconsistency type	Description
Case #1	Missing element	An entity is in one view but not in the other.
Case #2	Detail missing	An entity is further decomposed in one view, but not in the other.
Case #3	Finer decomposition	In one view, a set of entities is shown under a single parent entity, whereas in the second view, more sub-groupings are used.
Case #4	Different grouping	Some entities (matched in the two views) are not grouped in the same way under their parent entity (which are not matched).
Case #5	Different decomposition	Some entities are not under the same parent entity in the two views (the parent entities are matched)
Case #6	Details taken from outside (leaf)	All the matched entities under one parent entity (in one view) are not under the same entity (a leaf) in the other view.
Case #7	Details taken from outside (non-leaf)	All the matched entities under one parent entity (in one view) are not under the same entity (not a leaf) in the other view.
Case #8	Different details	Two matched entities (in the two views) are both further decomposed, but the entities involved in both decomposition are completely different.
Case #9	No inconsistency (leaf)	Two matched entities (in the two views) are both leaves (not further decomposed).
Case #10	No inconsistency (non-leaf)	Two matched entities (in the two views) are both further decomposed, and they both have the same set of matched entities under them.

Table 7 - Basic types of inconsistency, and cases with no inconsistency

In order to identify these basic types of inconsistencies, we use the following set of boolean characteristics: (please refer to Figure 25 and Figure 26 for the examples provided)

C1 - element is matched:

The element has been matched to one of the entities in the second view. This entity in the second view is referred to as "matched element".

E.g.: if we analyze Bob's "system analysis" activity with respect to Peter's view, C1 is true, and the matched element is Peter's "system

analysis" activity. On the other hand, if we analyze the same activity with respect to William's view, C1 is false.

C2 - descendant matched:

At least one of the descendants of the element has been matched to one entity in the second view.

E.g.: This characteristic is true for Bob's "deliverable validation" activity with respect to Peter's view (the descendant "IT_team review" is matched). This is not the case with Bob's "deliverable production" activity (none of the descendants are matched in Peter's view).

C3 - outside subtree entity matched:

At least one of the siblings of the element, or descendant of siblings, has been matched to one entity in the second view.

E.g.: In the case of Bob's "deliverable production" activity, there is at least one sibling ("deliverable validation") that is matched in Peter's view, so this characteristic is true. But for any of the descendants of this "deliverable production" activity, C3 is false, because none of them is matched in Peter's view.

C4 - element is leaf:

The element does not have any descendant.

E.g.: Bob's "describe context" activity is a leaf, but not the "deliverable production". Notice that this characteristic is independent of the view with respect to which we analyze the entity.

C5 - matched element is leaf:

The entity in second view that is matched to the element we are looking at does not have any descendants.

E.g.: This is the case for Bob's "IT team review" activity (matched to Peter's "IT team QA" activity), but not for "deliverable validation" activity (matched to Peter's "review" activity)

C6 - group = union

Groups are sets of entities under a subtree with a match to an entity in the second view. For a given level of decomposition, there is one group per entity on that level. For example, in Bob's view (with respect to Peter's view), if we look at the second level of decomposition, we have the following two groups:

1. ["deliverable production"] (one element only because the descendants are not matched)
2. ["deliverable validation", "IT team review", "modify deliverable", "review by other teams"] (all four elements because they are all matched)

In the same way, we can build the groups in Peter's view, with respect to Bob's view:

1. ["document production", "modifications"] (notice that "writing first version" is not matched in Bob's view)
2. ["review", "IT team QA", "other teams review"] ("client review" is not matched in Bob's view)

For this characteristic to be true, the group for the element (e.g. the first group for Bob's "deliverable production" activity) should be a union of zero or more groups in the other view (groups in Peter's view here). This is not the case in our example above. This characteristic would be true for Bob's "system analysis" activity because its group contains all the matches in that view (with respect to Peter's view), and that the group for Peter's "system analysis" activity is the same (with respect to Bob's view).

C7 - descendant of matched element is matched:

At least one of the descendants of the matched element has been matched to one entity in the first view.

E.g.: This characteristic is true for Bob's "deliverable validation" activity, because its matched entity in Peter's view ("review") has some

descendants that are matched in Bob's view ("IT_team QA" and "other teams review"). This is not the case with Bob's "modify deliverable" activity because its matched entity in Peter's view ("modifications") has no descendants.

C8 - group of element = group of matched element:

For this characteristic, the groups are formed like in C6, but only for a specific element and its matched element (not for all subtrees), even if the matched element is at a different level in the hierarchy. For example, the group for Bob's "IT team review" activity and its matched activity in Peter's view ("IT team QA") are these activities themselves (matched together) because they are leaves, so this characteristic is true for Bob's "IT team review" activity with respect to Peter's view. This is not the case for Bob's "deliverable validation" activity, because its group contains the "modify deliverable" activity, but the group of Peter's "review" activity does not contain it.

Each characteristic above can be true or false. The 8-tuple built out of that will be used for determining if there is an inconsistency, and if so the basic type of this inconsistency. An example of such an 8-tuple is (T,F,T,F,F,F,T,F)[27] for Bob's "deliverable production" activity with respect to Peter's view.

For an 8-tuple of boolean values, there can be 256 (2^8) possibilities. But in our case, some combinations are impossible. Here are the possible reasons to reject a combination (these are summarized using a formal notation in Table 8) :

R1 - If the element is not matched (C1=F), there is no matched element, so the characteristics related to the matched element should be false (C5=F & C7=F & C8=F).

[27] There is one boolean value for each of the characteristics defined above.

Reason to reject	Related constraint on combination (tuple)
R1	$(C1 = False) \Rightarrow (C5 = False) \wedge (C7 = False) \wedge (C8 = False)$
R2	$(C2 = True) \Rightarrow (C4 = False)$ $(C7 = True) \Rightarrow (C5 = False)$
R3	$(C1 = False) \wedge (C2 = False) \Rightarrow (C6 = True)$
R4	$(C3 = False) \Rightarrow (C6 = True)$
R5	$(C2 = False) \wedge (C5 = True) \Rightarrow (C8 = True)$ $(C2 = True) \wedge (C5 = True) \Rightarrow (C8 = False)$
R6	$(C1 = True) \wedge (C4 = True) \wedge (C7 = False) \Rightarrow (C8 = True)$ $(C1 = True) \wedge (C4 = True) \wedge (C7 = True) \Rightarrow (C8 = False)$
R7	$(C1 = True) \wedge (C2 = False) \wedge (C7 = False) \Rightarrow (C8 = True)$ $(C1 = True) \wedge (C2 = True) \wedge (C7 = False) \Rightarrow (C8 = False)$ $(C1 = True) \wedge (C2 = False) \wedge (C7 = True) \Rightarrow (C8 = False)$
R8	$(C4 = True) \wedge (C5 = True) \Rightarrow (C8 = True)$
R9	$(C2 = False) \wedge (C3 = False) \Rightarrow (C6 = True) \wedge (C7 = True) \wedge (C8 = True)$
R10	$(C2 = False) \wedge (C7 = True) \Rightarrow (C6 = False) \wedge (C8 = False)$

Table 8 - Summary of the reasons to reject some combinations of characteristics

R2 - If the element has descendants matched (C2=T), it cannot be a leaf (C4=F). The same applies to the matched element if there is one. If the matched element has descendants matched (C7=T), it cannot be a leaf (C5=F).

R3 - If the element is not matched (C1=F) and none of its descendants are matched (C2=F), then its group is empty, so it is a union of zero or more groups of the other view (C6=T).

R4 - If none of the siblings or descendants of siblings are matched (C3=F), the group for the element contains all remaining matches, so the group should be the union of the groups in the other view (C6=T).

R5 - In the case the matched element is a leaf (C5=T), the group for the matched element is the element itself only. So if the element has descendants matched (C2=T), its group will contain more than the element itself, so it will not be the same as the matched element's group (C8=F). But if the element does not have descendants matched (C2=F), its group will contain only the element itself, like the matched element's group (C8=T).

R6 - The same reasoning as in R5 apply in the case the element is matched (C1=T), and that it is a leaf (C4=T). If the matched element has descendants matched (C7=T), its group will be different than the element's group (C8=F). But if the matched element does not have descendants matched (C7=F), its group is the same as the element's group (C8=T).

R7 - In the case the element is matched (C1=T), if neither the element nor the matched element has descendants matched (C2=F & C7=F), then both groups are the element itself only, so the groups are the same (C8=T). If the element has descendants matched (C2=T) but not the matched element (C7=F), or vice versa (i.e. C2=F & C7=T), the groups cannot be the same (C8=F).

R8 - In the case both element and matched element are leaves (C4=T & C5=T), their groups are the same (C8=T) because the groups are the element only.

R9 - If the element have neither descendants matched (C2=F) nor siblings or descendants of siblings matched (C3=F), than the only possible non-empty group is the element itself...if it is matched. If it is not matched, all groups are empty. In this situation, the group (empty or containing only one element) should be a union of the groups in the other view (C6=T) because this other view has only empty groups or one group with one element only. There are no other matches. In the case the element is matched (so there is only one element in all groups), this element should be the matched element, it cannot be its descendants (C7=F). The group of the matched element also contains only itself because there are no other matched elements, so the groups are the same (C8=T).

R10 - If the element does not have descendants matched (C2=F), but the matched element has some descendants matched (C7=T), then their respective groups is not the same (C8=F). Also, the group of the matched element (in the second view) contains the initial element with others, so it is not possible to isolate the element in groups, and the element's group cannot be a union of the other view's groups.

This set of reasons for rejecting combinations shrinks the number possible tuples to 33. For each of these 33 possible combinations, it is possible to come up with an example having these characteristics, so our set of reasons for rejecting a combination (R1 to R10 above) is complete. Table 9 shows the conditions necessary for each basic inconsistency type (refer to Sections 6.3.1.1 to 6.3.1.10 for additional information). A "—" indicates that the value for such characteristic is irrelevant in the identification of the type of inconsistency.

	C1	C2	C3	C4	C5	C6	C7	C8
Case #1 - missing element	F	F	T	--	--	--	--	--
Case #2 - details missing	T	--	--	T	F	--	F	--
or	T	F	--	F	T	--	--	--
Case #3 - finer decomposition	F	T	--	--	--	T	--	--
Case #4 - different grouping	F	T	--	--	--	F	--	--
Case #5 - different decomposition	T	T	--	--	--	--	T	F
Case #6 - details taken from	T	T	--	--	T	--	--	--
outside (leaf) or	T	--	--	T	--	--	T	--
Case #7 - details taken from	T	T	--	--	F	--	F	--
outside (non-leaf) or	T	F	--	F	--	--	T	--
Case #8 - different details	F	F	F	--	--	--	--	--
Case #9 - no inconsistency (leaf)	T	--	--	T	T	--	--	--
Case #10 - no inconsistecy (non-leaf)	T	--	--	F	F	--	--	T

Table 9 - Characteristics for each basic inconsistency type

When figuring out which case applies to the current element, it is not necessary to evaluate each characteristic. Some of them can be avoided in some cases. This is particularly useful for those that are complex (and time consuming) to evaluate, such as characteristics C6 and C8. The following is an algorithm showing the ordering of characteristic evaluation: (notice that it has been verified for each of the 33 possible combinations)

```
if C1
    if C4
        if C5 --> case 9
        else
            if C7 --> case 6b
            else --> case 2a
    else
        if C5
            if C2 --> case 6a
            else --> case 2b
        else
            if C8 --> case 10
            else
                if C2
                    if C7 --> case 5
                    else --> case 7a
                else --> case 7b
else
    if C2
        if C6 --> case 3
        else --> case 4
    else
        if C3 --> case 1
        else --> case 8
```

In Section 6.3.1.1 to Section 6.3.1.8 below, we describe each of the basic inconsistency types (using a generic entity decomposition as an example), how they are detected using the characteristics above, how they are resolved, and how the final model is modified after these cases are resolved. Sections 6.3.1.9 and 6.3.1.10 describe the cases where there is no inconsistency, and show what is done on the final model in these cases. A realistic example is used to illustrate each of these cases; it is shown in the three views in Figure 76. Section 6.3.1.11 provides more details on the algorithms used and on the ordering of the resolution of the different types of inconsistencies. Finally, Section 6.3.1.12 summarizes this section and shows the completeness of the set of basic inconsistency types.

Activity decomposition aspect for three views:

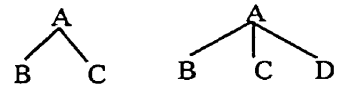
View #1:	View #2:	View #3:
1. design	1. design	1. design
1.1 production	1.1 production	1.1 understand requirements
1.1.1 HLD	1.1.1 architectural design	1.2 production
1.1.1.1 map DFD to architecture	1.1.1.1 map DFD to architecture	1.2.1 map DFD to architecture
1.1.1.2 design main structure	1.1.1.2 add other modules	1.2.2 add other modules
1.1.2 LLD	1.1.2 data design	1.2.3 design main structures
1.1.2.1 add other modules	1.1.2.1 design main structures	1.2.4 design local structures
1.1.2.2 design local structures	1.1.2.2 design local structures	1.2.5 do algorithms
1.1.2.3 do algorithms	1.1.3 procedural design	1.2.6 plan control
1.1.2.4 plan control	1.1.3.1 do algorithms	1.3 validation
1.1.3 modifications	1.1.3.2 plan control	
1.2 validation	1.2 validation	
1.2.1 preparation	1.2.1 team validation	
1.2.2 meeting	1.2.2 global validation	
1.2.3 produce report	1.2.3 interface validation	
	1.3 modifications	

Remark: indentation and numbering has been used here to show the decomposition of activities

Figure 76 - Example views used to illustrate the different types of inconsistencies

6.3.1.1 Case #1: Missing element

Generic example:



Entity from which the discrepancy is found: D

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not (but at least one is matched, or one descendant is matched). These children can all be further decomposed. The element D and its descendants (if any) are not matched.

Characteristics:

- element (D) is not matched (C1=F)
- none of its descendants are matched (C2=F)
- at least one of the siblings or descendants of siblings is matched (C3=T)

*Examples from our three views (*Figure 76*):*

The "understand requirements" activity is in view #3 but not in view #1 and view #2, and the "modifications" activity is in view #1 and view #2, but not in view #3.

Possible situations leading to such a case:

This case can happen when someone is not aware of details. This omission can also occur if there is a step that was not performed during that particular instance of the process, but is performed in other circumstances. For example, some type of validation might not be necessary, but the person sending the document for validation may think that all types of validation will be performed.

This case can also occur with other types of entities. For example, a user's guide (an artifact) might not contain some parts like introduction or conclusion; a team (role composition) might not contain one of its roles; etc.

Possible solutions presented in the resolution window[28]*:*

- missing element is added
- missing element is not added

The resolution window can show the list of views containing the element, and the list of views not containing it. The elicitor can then choose if the entity should be kept or not. If the decision is to keep it, it is added with all its decomposition to the global model being built. If the decision is not to keep it, it is removed from the global model (if it was there) and from the views containing it, including the whole decomposition.

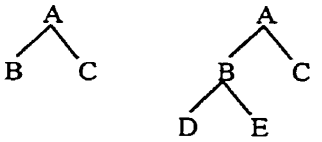
---

[28] For an example of such resolution window, see Figure 65.

Remark: when the system finds a missing element, the whole subtree is treated at once, and it does not recursively find discrepancies under that element, except if it was there in a third view.

6.3.1.2 Case #2: Detail missing

Generic example:



Entity from which the discrepancy is found: B

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which may or may not be matched. These children can all be further decomposed (except B in first view). Element B should be matched, and one of the views should decompose it, but not the other. The information given in the decomposition should not overlap with other information under the other children (i.e., not matched).

Characteristics:

The characteristics are different depending on which element (i.e., B from which view) has been used for evaluating the type of inconsistency.

a) Characteristics for leaf element (B in the first view):

- element (B) is matched (C1=T)
- element is leaf (C4=T)
- matched element is not a leaf (C5=F)
- none of the descendants of the matched element are matched (C7=F)

b) Characteristics for the non-leaf element (B in the second view):

- element (B) is matched (C1=T)
- none of the descendants are matched (C2=F)
- element is not a leaf (C4=F)
- matched element is a leaf (C5=T)

*Example from our three views (*Figure 76*):*

The "validation" activity is decomposed in view #1 and view #2, but not in view #3.

Possible situation leading to such a case:

This case can happen when someone knows that the element exists, but does not have any details about it. This is usually the case when many people are working on one project: no one may know exactly what the others in the project are doing, but they may have a general idea of the others' tasks.

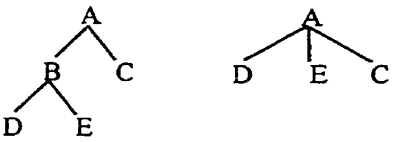
Possible solutions presented in the resolution window:

- this decomposition is kept (usual case)
- this decomposition is not kept (if those details are not necessary for the users of the final model)

For resolving this case, the system presents to the elicitor the list of views that further decompose the entity (even if the details are different), and the list of views that do not decompose it. The elicitor can then decide whether or not to keep the decomposition. If the decision is to keep it, the first level of the decomposition is added in the global model. Otherwise, the whole decomposition is deleted in the global model (if necessary).

6.3.1.3 Case #3: Finer decomposition

Generic example:



Entity from which the discrepancy is found: B

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed. Element B (not matched in the second view) is made of children in the second view, and may have additional elements, which are missing in the second view. Those missing elements will be analyzed only when children of B will be checked (when B's level will all be resolved).

Characteristics:

- element (B) is not matched (C1=F)
- at least one descendant is matched (C2=T)
- group=union (C6=T)

*Examples from our three views (*Figure 76*):*

The "HLD" activity in view #1 is composed of few activities under the same "production" activity in view #3. The same happens with the following activities: "LLD" in view #1, and "architectural design", "data design", and "procedural design" in view #2.

Possible situation leading to such a case:

This case can happen when someone has a hierarchical structure in mind, and another person has a flat structure in mind.

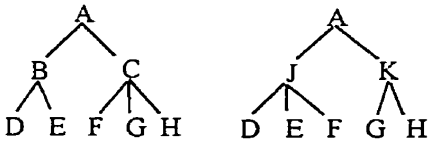
Possible solutions presented in the resolution window:

- add this level of decomposition
- do not add this level of decomposition

For resolving this case, the system can present to the elicitor the list of views containing the entity (with the same grouping), and the list of views not containing it, but containing some of the descendants. The elicitor can then choose if the entity should be kept or not. If the decision is to keep it, it is added to the global model, and the children are moved under the new entity. Notice that the child may not be matched, but one or more of its descendants may be matched and should go under the new entity. If the decision is not to keep the entity, it is removed from the global model, and the children are moved directly under the entity's parent.

6.3.1.4 Case #4: Different grouping (with unmatched elements as roots)

Generic example:



Entities from which the discrepancy is found: B, C, J, and K

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed. For the entities from which the discrepancy is found, they are not matched, but they have descendants matched. The discrepancy is that the entities under these subtrees are not grouped in the same way, and this different grouping is not just a finer decomposition.

Characteristics:

- element (B, C, J, or K) is not matched (C1=F)
- element has descendants which are matched (C2=T)
- group =/= union (C6=F)

*Example from our three views (*Figure 76*):*

The following activities provide different grouping of the subtasks between view #1 and view #2: "HLD", "LLD", "architectural design", "data design", and "procedural design".

Possible situations leading to such a case:

This case happens when the criteria for grouping the entities is different from one view to another. It can happen also in the case of overlapping entities (that were not matched), for example in the case that one person says that modification of a document falls in the production task, and another person says it falls in the review task (and that the production tasks and/or review tasks have not been matched together because their descriptions were too different).

Possible solutions presented in the resolution window:

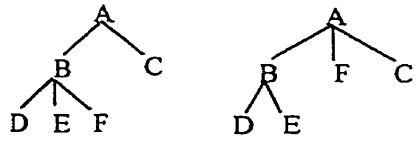
- one of the decomposition presented in the views
- any other way of grouping the entities

The resolution of this case overlaps with the techniques used in other cases. The elicitor should first select which entities should be used for the grouping of elements (in our generic example above, the elicitor would have to select a subset from {B, C, J, K}. This selection is performed like the one in case #1. The elicitor should then decide where the sub-entities should go (i.e., under which parent). This is the same as resolving case #5.

Remark: this case is more general than the case #5 below, where the roots are matched. So in the case that the roots are matched for some subtrees across some views, but that they are not matched in other cases, then case #4 is applied first.

6.3.1.5 Case #5: Different decomposition (with matched elements as roots)

Generic example:



Entity from which the discrepancy is found: B

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed. The entity from which the discrepancy is found is matched, as well as some of its descendants, but the set of matched descendants is different in each view.

Characteristics:

- element (B) is matched (C1=T)
- element has descendants which are matched (C2=T)
- matched element has descendants which are matched (C7=T)
- group =/= matched element's group (C8=F)

*Example from our three views (*Figure 76*):*

The "production" activity in view #1 contains the "modifications" activity, but this is not the case in view #2.

Possible situation leading to such a case:

This case can happen when someone does not consider one subtask to be part of one task. A situation like the one described in the basic type #4 above can also bring this type of inconsistency. The difference here is that in this case, the descriptions of the main task (B) are very similar, so they have been considered the same (i.e., matched).

Possible solutions presented in the resolution window:

For each matched entity that is under B in one view, but not in the other view:

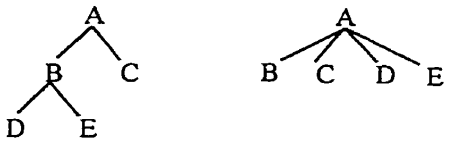
- keep this entity under B
- do not keep this entity under B

For resolving this case, the elicitor has to decide, for each entity under B in every view, if it is kept under B or not. Of course, the entities always under B in every view are not going through that process. For this decision, the system can present to the elicitor the list of views having the entity under the subtree of B, and the list of views having the entity elsewhere. If a view does not contain the entity, it does not appear in the lists.

Each time the elicitor decides if an entity should go under B or not, the model is modified. If the decision is to keep the entity under B, the entity is moved as a child of B in the model. If the decision is not to put the entity under B, the entity is moved as a sibling of B in the model.

6.3.1.6 Case #6: Details taken from outside (leaf)

Generic example:



Entity from which the discrepancy is found: B

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed (except B in the second view). Element B should be matched, and it should have matched descendants in the view where it is not a leaf.

Remark: this case is improbable because it means that, for example, an activity is performed completely (through B) and then it is performed again through its sub-activities (D and E).

Characteristics:

The characteristics are different depending on which element (i.e., B from which view) has been used for evaluating the type of inconsistency.

a) Characteristics for the non-leaf element (B in the first view):

- element (B) is matched (C1=T)
- element has descendants which are matched (C2=T)
- matched element is a leaf (C5=T)

b) Characteristics for the leaf element (B in the second view):

- element (B) is matched (C1=T)
- element is a leaf (C4=T)
- matched element has descendants which are matched (C7=T)

Example:

Examples are not provided here because this case is improbable. It is theoretically possible to have such a case, so we are dealing with it, but we have not identified situations where this can happen.

Possible solutions presented in the resolution window:

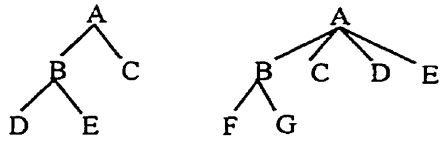
For each matched entity that is under B in the view where B is not a leaf

- keep the entity under B
- do not keep the entity under B

Remark: the resolution of this basic type is the same as for the basic type #5

6.3.1.7 Case #7: Details taken from outside (non-leaf)

Generic example:



Entity from which the discrepancy is found: B

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed. Element B should be matched, and it should be decomposed in both views, but it should have matched descendants in only one of the views.

Characteristics:

The characteristics are different depending on which element (i.e., B from which view) has been used for evaluating the type of inconsistency.

a) Characteristics of the element containing matched descendants (B in the first view):

- element (B) is matched (C1=T)
- element has descendants which are matched (C2=T)
- matched element is not a leaf (C5=F)
- none of the descendants of the matched element are matched (C7=F)

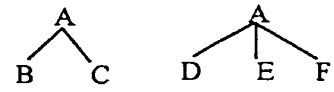
b) Characteristics of the element not containing matched descendants (B in the second view):

- element (B) is matched (C1=T)
- none of its descendants are matched (C2=F)
- element is not a leaf (C4=F)
- matched element has descendants which are matched (C7=T)

Remark: this case is very similar to the basic type #6 above, except that other information were specified in the second view, instead of leaving the element B as a leaf. This case is also improbable, for the same reason. Its possible solutions and resolution strategy are exactly the same as for case #6 above (please refer to that section for the complete information).

6.3.1.8 Case #8: Different details

Generic example:



Entities from which the discrepancy is found: B, C, D, E, and F

In this type of discrepancy, the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which are not matched at all. These children can all be further decomposed, but none of the descendants are matched.

Characteristics:

- element (B, C, D, E, or F) is not matched (C1=F)
- none of its descendants are matched (C2=F)
- none of the siblings or descendants of siblings are matched (C3=F)

*Examples from our three views (*Figure 76*):*

The following activities in view #1 and view #2 (with respect to the other view, not with view #3) represents alternatives in decomposing the "validation" activity: "preparation", "meeting", and "produce report" in view #1; "team validation", "global validation", and "interface validation" in view #2.

Possible situation leading to such a case:

This case can happen at lower levels of details, where two people do not describe their specific subtasks in the same way. Typically, documents flowing between them are temporary, and the same roles are involved in all of the subtasks.

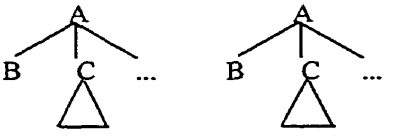
Possible solutions presented in the resolution windows:

- keep one of the decomposition
- keep any subset of the entities (including empty subset, if the level of details is too low)

For resolving this case, the system can first ask the elicitor which view to keep as a solution, or if s/he wants to select a subset of entities. In the case that one view is kept, the model is modified such that it contains the new set of selected entities. If the elicitor wants to select his/her own subset of entities, the system can then ask him/her, for each entity, if s/he wants to keep it or not. This is performed in the same way as for resolving inconsistencies of basic type #1.

6.3.1.9 Case #9: No inconsistency (leaf)

Generic example:



element = B

In this case (which is not a discrepancy), the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not (B should be matched). These children can all be further decomposed (except B). Element B does not have any discrepancy in this case because it is a leaf in both views, and it has been matched.

Characteristics:

- element (B) is matched (C1=T)
- element is a leaf (C4=T)
- matched element is a leaf (C5=T)

*Examples from our three views (*Figure 76*):*

All the leaf activities (matched) that are not further decomposed in any of the views are in such case (e.g., the "map DFD to architecture" and "modifications" activities).

Possible situation leading to such a case:

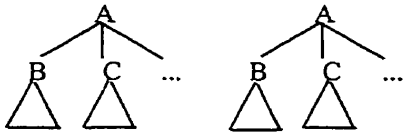
This case is the one when, for example, two people say that one activity (or artifact, role, etc.) exists, and both do not further decompose it.

Resolution:

There is no resolution to be made, but the system should add the entity to the global model if it is not already there.

6.3.1.10 Case #10: No inconsistency (non-leaf)

Generic example:



element = B

In this case (which is not a discrepancy), the parent element (A) is matched, and it does not overlap with other parts. Under A, there are some children which can be matched or not. These children can all be further decomposed. Element B is matched, it is further decomposed in both views, and the descendants matched are in both subtrees of B. Notice

that there might be other discrepancies at lower levels under B, but these are resolved later.

Characteristics:

- element (B) is matched (C1=T)
- element is not a leaf (C4=F)
- matched element is not a leaf (C5=F)
- group = matched element's group (C8=T)

*Examples from our three views (*Figure 76*):*

There are three such cases in our example:

- the "design" activity in all views
- the "production" activity between view #3 and view #1, or between view #3 and view #2
- the "validation" activity between view #1 and view #2

Possible situation leading to such a case:

This is the case where both persons agree on the existence of the entity (activity, artifact, role, etc.), and also agree on the sub-entities that should go under this entity.

Notice that if there are no matched descendants, we get into the discrepancy described in case #8 above when going further down (next step of recursion). This is the case in our third example above (the "validation" activity between view #1 and view #2).

Resolution:

There is no resolution to be made, but the system should add the entity to the global model if it is not already there.

6.3.1.11 Algorithmic details

When detecting the inconsistencies across views, we need to use previous decisions in order to avoid detecting inconsistencies that are not there anymore. In order to keep track of the previous decisions and how they affect the different views, we are actually making the same changes to the views as for the final model[29], when resolving each inconsistency. For example, if the elicitor decides to keep the "modifications" task under the "production" activity as in view #1 (see Figure 76), then the "modifications" activity is moved accordingly in view #2.

However, when showing to the elicitor the list of views having one solution, and the list of views having another solution (in the resolution windows), we should use the original information provided by these views. For example, if an activity has been added to one of the views (arbitrarily under one of the other activities), we should not use this view when analyzing how this activity is grouped with other activities (i.e., the view with the added activity should not appear in the lists of views having one grouping solution or another).

In order to meet those two needs, we need to keep two versions of each view: the original one, and the one that is modified after each resolution of an inconsistency.

Keeping two copies of each view can take a lot of memory. In order to reduce the memory used, we can use tags for the entities ("original", "added", or "deleted") instead of copying them. For the relationships, we can use two different relationship types to keep track of the original relationships and the modified ones ("entity was-composed-of entity" and "entity is-composed-of entity" respectively).

Another use of the modified views is that they permit the elicitor to stop the resolution process at any time, and return later for continuing the resolution process. This is very

---

[29] See the sections above on how the final model is built as each inconsistency is found.

useful in the case where the elicitor has no idea of the solution, and has to go back to some sources to find out the right solution.

The ordering of resolution (of the inconsistencies) is top-down, from the roots to the leaves in the entity decomposition tree. All views are used at the same time. We first list all the inconsistencies (with their type) within one level on decomposition, across all views, and then we resolve these inconsistencies one at a time.

We start by resolving the inconsistencies that may affect the other types of inconsistencies (i.e., the ones related to the grouping of entities in the decomposition). The case #3 is checked first because in the case the elicitor decides not to keep the level of decomposition, then the inconsistencies related to the next level of decomposition appear at the current level when moving up the entities. We then check the other cases that may modify the decomposition structure: case #4 and case #5 (cases #6 and #7 are considered within case #5 because their resolution is performed in the same way).

After resolving these cases, we have to redo the list of inconsistencies within the current level in the tree, because the kind of modifications made to the views when resolving these cases may affect other types of inconsistencies, and even show new ones that were not apparent at first. Two situations can occur here: either entities involved in some inconsistencies have been moved down in the hierarchy, or that entities have been moved up in the hierarchy to the level currently worked for inconsistency resolution. In the first case, the inconsistencies related to these entities are no longer visible at that level, and their resolution is postponed to a later iteration. In the second case, the entities moved are now visible, and they can then be involved in some inconsistencies (either existing ones now apparent, or new ones). Since the inconsistencies "created" after the resolution of the cases #4 to #7 are only related to the entities moved up, we are sure that the cycles of resolution of these cases always terminate (there is a limited number of entities in the model, that can be moved up). Notice that the entities moved down in the hierarchy when resolving an inconsistency cannot be moved up again.

We then resolve the remaining cases that do not have much effect on the other types of inconsistencies, because they are just adding or removing entities (cases #1, #2, and #8).

Once these remaining cases are resolved, we can go to the next level of details in the hierarchy, by recursively calling the resolution function. The process stops at the leaf level, or when an entire subtree representing the details of an entity is added (from case #1 or #2).

6.3.1.12 Summary and analysis

In this section, we have presented the different types of inconsistencies handled (related to the entity decomposition), how the system detects them, how the elicitor can resolve them, and how the system is then building the final model reflecting the choices of the elicitor for the solution of the inconsistencies.

We have seen that the inconsistencies were detected through a set of 8 characteristics. This set is complete because it maps each of the combinations of characteristic values (the 33 possible 8-tuples) to only one of the ten cases identified (types of inconsistencies or cases with no inconsistency). In the case that two combinations are mapped to the same type, the characteristic that is different in the two combinations has no effect on the resolution of such type of inconsistency. For example, in an inconsistency of case #1, the fact that the entity is a leaf or not does not change the way of resolving this type of inconsistency. The elicitor has to choose whether to keep the entity or not, and this is independent of the fact that the entity is further decomposed or not.

Having such a system that detects the inconsistencies, helps in choosing the solution, and builds the merged model from the information gathered, is of great help in merging the views. By experience, we know that these inconsistencies related to the entity decomposition are the most difficult ones to resolve. The complexity comes mainly from

the fact that many entities may be involved at the same time when the problem is the grouping of the entities. The automatic identification of inconsistencies with their type, and the automatic merging of the views into one model, helps the elicitor in focusing only on the decisions to be made for resolving the inconsistencies, leaving the tedious details to the system.

6.3.2 Detecting and resolving inconsistencies related to names and attributes

In these kinds of inconsistency, the detection and the resolution is much simpler than for inconsistencies related to the entity decomposition, because the number of cases is reduced to two: the names or attribute values are either the same or not. For example, in the views used in Section 5.2 (see Appendix A for entire information on them), we had cases where the names used were the same across the views (e.g., "system analysis" activity), and others where the names were different (e.g., "modify deliverable" in Bob's view but "modifications" in Peter's view). Similarly with the attributes, we had cases where the values were the same across views (e.g., the duration of the "client review" in Peter's view and the "user review" in William's view), and others where differences were identified (e.g., the duration of the "IT team review" in Bob's and William's views, and the "IT team QA" in Peter's view).

The algorithm for detecting and resolving these inconsistencies is very simple: we just go through all the entities, comparing the name or attributes of the matched entities in each view. When an inconsistency is found, the user is asked for the right value (for the name or attribute). The final model is then modified accordingly.

In order to help the elicitor in choosing the right value, the resolution window also presents the degree of difference between the entity in the final model and the entities in the views (as described in Section 5.2.3.2.2). For example, in Figure 67, we see the number of items added and deleted in the subtree related to the entities having inconsistent names. This information helps the elicitor in choosing the right solution.

Such feature gives a major advantage over the manual approaches for resolving inconsistencies of such types.

6.3.3 Detecting and resolving inconsistencies related to relationships

Detecting inconsistencies related to the relationships is also quite simple, because we just have to check, for each relationship, if it is in all the views, and add it to the final model. In the case the relationship is missing in some views, the elicitor should just tell whether the relationship should be there or not.

The definition of a missing relationship is actually not as trivial as one may think. For example, the relationship "modifications produces document" in Peter's view (Figure 26) is not considered as missing in William's view (Figure 27) because one of the entities ("modifications") is not in the view. In such case, how could William talk about such relationship if he was not even aware of one of the entities involved? The same apply when one view describe one of the task in more details than in the other views. These other views cannot specify the dynamics of the subtasks if the detailed subtasks are not specified.

In other cases, one relationship may seem to be missing, but it might just be specified at higher levels of abstraction. For example, in Bob's view (Figure 25), we can see the relationship "IT_team_review produces feedback", which is not in William's view (Figure 27), even if both entities were defined. Actually, the relationship was specified in William's view at a lower level of details, through the relationships "walkthrough produces feedback" and "engineering_review produces feedback". For dealing with this problem, we can use the hierarchical relationship generator first, and then look at the missing relationships.

Linear relationship generators can also be useful in the case the type of information is not the same across the views. For example, if one view has no indication of the activity

ordering, but that it has information on the input/output of the activities, we can use the linear relationship generator to find the dependencies across activities, and then compare them with the activity ordering information in the other views. This way, we can identify early the inconsistencies related to the mix of types of information across views. If such opportunity is not taken at this point, then we can still find such problems when analyzing the final model using constraints as described in Section 5.2.4, but in this case it is more difficult to see the reason(s) for an inconsistency. The additional information provided in the resolution windows (to help making decision) is not provided during constraint verification. So, whenever it is possible, we should use the linear relationship generators to provide similar information to be compared across all views.

6.3.4 Related work

Other researchers have worked on the problem of merging information from different sources (or views). In Section 2.2, we have identified some notable efforts in the context of software process modeling and elicitation [KeH89, Rom93, Ver96]. In all cases, a manual approach was used for resolving inconsistencies across views.

We have also examined other fields in which similar problems could be found, for example, Requirements Engineering and Knowledge Engineering. Our idea was to use their approach if one was appropriate, or at least utilize some concepts if they were applicable to our elicitation problem.

The *Requirements Engineering* area also faces the problem of gathering information (requirements) from multiple sources. A software system to be built is rarely for a single user, and different users may not have the same requirements. They also have to model the end-user processes where the new system would be integrated, in order to see how it would fit in these processes. We can thus envisage using some of the requirement merging techniques for software process elicitation.

Easterbrook has presented an approach for resolving conflicts in specifications given by different persons in [Eas91]. For each conflict, issues are elicited and criteria is established by which to judge possible resolution. A list of options is then generated, where each option is related to an issue. The participants can give their level of satisfaction with the criteria attached to the issues, and a global satisfaction score is computed, helping in the choice of the solution.

In Easterbrook's approach, specific techniques and tool support are almost nonexistent. Most of the work is done manually. Support is given for entering information about conflicts in templates (in all phases), and for calculating satisfaction scores for each option once individual scores have been entered. The overall idea of categorizing conflicts, providing a list of options, and using some criteria for evaluating a solution, is the only aspect that could be utilized in our process elicitation approach.

Leite and Freeman [LeF91] have presented a technique for identifying discrepancies between two different viewpoints (describing requirements), and classifying these. They propose a strategy for requirement elicitation: each participant enters information from different perspectives, which are then analyzed for feedback on data entered, and integrated into views (one for each participant). Views are analyzed for finding discrepancies (missing or wrong facts), which are then discussed with participants for integration of views. They use a rule-based language for viewpoint representation. Their algorithm for finding discrepancies first finds matching rules, and then finds differences in rules. Unmatched rules are classified as missing information. Some heuristics are presented for finding matching rules.

Some of the ideas in Leite and Freeman's technique for finding the matching elements in the different descriptions have been used in our component matching algorithm, with some changes to fit the needs of our process model schema (see Section 6.2.3). However, their approach for resolving the discrepancies is manual.

The second domain that we studied is *Knowledge Engineering*. Elicitation of a software process model can be seen as a knowledge acquisition process where the experts are the software developers from whom software process information (knowledge) is gathered. We could thus hypothesize using some knowledge acquisition techniques for software process elicitation.

Different methods have been proposed for dealing with multiple experts, and handling conflicts amongst them. The solutions proposed range from no conflict resolution at all to specific techniques with some tool support in specific domains.

Leclair [Lec89] has proposed to keep the information from each expert separate (in sub-systems), and let the user choose between solutions proposed by each sub-system, depending on the specific situation. This solution can't be used in eliciting a common, software process model, which requires an agreement on what the actual software process is amongst multiple agents [KeH89].

Another approach is that of Wolf [Wol89], which relies on discussions between experts for conflict resolution before entering the knowledge in the system. Some other methods based on communication (e.g., brainstorming, Delphi method,...) are surveyed in [TuT93]. However, they lack technological support.

A more formal method has been presented by Gaines and Shaw [GaS93]. They have proposed tools for entering information separately from different experts, and for finding consensus, conflicts, correspondences, and contrasts[30] across these different sets of information entered. The descriptions are sets of entities, and a scale ([1..10]) on each attribute for each entity. They have also described a method for eliciting information from multiple experts [ShG89]. The main steps are to first discuss and come to an agreement

---

[30] Definitions used:
Consensus: using the same term for the same concept
Conflict: using the same term for different concepts
Correspondence: using a different terms for different concepts
Contrast: using different terms for the same concept

over a set of entities, then each expert enters his attributes and scales. The attributes are matched between the experts by comparing the scales entered for each entity. Finally the sets of attributes are given to other experts for entering their scales, and these attributes and scales are compared in order to find the consensus, conflicts, correspondences, and contrasts among the descriptions.

This technique cannot be used as-is for software process elicitation because entities in the process can be described by other ways than scales on attributes. Information such as relationships with other entities can be valuable in identifying common components. For example, two activities having the same inputs and outputs are probably more similar than those with different inputs and outputs, but this information cannot be expressed as a scale on an attribute. So we need a method for identifying components that uses all other useful information (relationships and non-numerical attributes), not just scale attributes.

To our knowledge, no other solutions exist to the problem of merging information from different sources.

6.3.5 Summary and analysis of the view merging feature

In this section, we have presented the different types of inconsistencies (related to the entity decomposition, the entity names, the attributes, and the relationships), how they are detected and resolved, and how the final model is built at the same time.

The inconsistencies related to the entity decomposition are the most complex to resolve, covering many different cases (10). A thorough discussion of these cases has been provided, showing how each of these cases are handled.

The last section on related work has shown that no other comparable view merging technique with their tool support exists currently, even in other domains having similar problems (Requirement Engineering and Knowledge Acquisition).

For a given inconsistency (related to entity decomposition – the most complex ones), the time complexity for identifying it (i.e., computing the set of characteristics for a given entity, with respect to a given view) and making the appropriate modifications to the merged model and all views, is as follows:

$t(nbviews, nbents) \in O(nbviews + nbents)$

where: nbviews = number of views

nbents = number of entities

t = the computing time of the algorithm, as a function of the number of views and entities

When identifying an inconsistency, a subset of the eight characteristics are computed. The most complex ones are C6 and C8, requiring to traverse the two views involved twice (so in O(nbents)). The process of modifying the views involves only operations that are performed in constant time, but this is done on each view and on the merged model (so in O(nbviews)). Notice that such computation is performed for each inconsistency.

It should be noted that even if there are no inconsistencies, each entity in each view has to be checked against each other view. In such a case (the best case), the total time complexity is as follows:

$t(nbviews, nbents) \in O(nbviews^2 * nbents^2)$

As for the matching algorithm (see Section 6.2.3), although this is an efficient algorithm (polynomial time), it might take a lot of time on large models containing thousands of entities. A first approach to this problem would be to focus on parts of the model at a time only (e.g., by software development phases). The management of such an approach, or the investigation into approaches and techniques dealing with this problem, however, requires further research work.

6.4 Summary of our specific elicitation techniques

The purpose of this chapter was to provide an insight into the new techniques developed for our view-based elicitation problem: constraint verification, component matching, and view merging. These techniques are the core parts of our process elicitation system V-elicit, as shown in Chapter Five.

The novelty of the techniques presented has been discussed in the specific sections. In some cases, we could use as a basis some other work and modify it to fit our needs. However, the modifications done were quite important, and in the case of the view merging techniques, no other method could be used.

We now have to show that the techniques developed are working and that they are relevant to our problem. Such analysis is provided in the next chapter.

Chapter Seven - Validation

The purpose of this chapter is to show that our system is working as intended (properly implemented), and to compare it with other existing systems (both state-of-the-art and state-of-the-practice).

We have validated our approach and system in three ways: internal validation, external validation, and literature comparison. Internal validation is aimed to show that the V-elicit system functions are *correctly* implemented and that it has been properly documented (e.g., no known logical bugs; all code results from documented design and it is complete; all test cases are successful; etc.). However, internal validation does *not* show the *relevance* of the V-elicit system; this is done through external validation, where the system is put to test against realistic situations involving industrial-scale software processes. During external validation, the V-elicit system is also compared against existing (commercially available) modeling tools. The comparison with state-of-the-art (research) modeling tools is performed in the literature comparison section. Finally, lessons learned are presented in the last section.

7.1 Internal Validation

For internal validation, the following specific issues were verified:

- requirements R1 to R10 stated in Chapter Three have been met (these requirements are summarized in the first column of Table 10)
- the V-elicit system and its techniques are working correctly (i.e., that the tool can detect various inconsistencies and build a merged model)
- the system requirements and design are documented

In section 5.1, when defining the different steps of our elicitation approach, the associated requirements for each step were shown. These steps (see the right-hand side of Table 10)

have all been implemented satisfactorily, and therefore the requirements have all been met.

We have used these steps in many example test cases. In particular, we have shown one of these examples in Section 5.2, with the final result shown in Appendix B. Each type of inconsistency has also been tested successfully, mainly through the examples presented in Chapter Six. From these tests, we can assert that, for all instances and purposes, the V-elicit system and its techniques are working correctly.

System requirements	**V-elicit steps**
R1 : elicit views separately	step 2 : elicit views
R2 : user-definable types of information for the modeling schema	step 1 : plan elicitation
R3 : user-definable types of information for views	step 1 : plan elicitation
R4 : verification of intra-view consistency	step 3 : check views
R5 : identifying common elements across views	step 4 : component matching
R6 : detecting inconsistencies across views	step 5 : view merging
R7 : helping in solving inconsistencies across views	step 5 : view merging
R8 : merging views into final model	step 5 : view merging
R9 : verifying the final model	step 6 : check model, and step 7 : modify model
R10 : checking model against development policies	step 6 : check model

Table 10 - Mapping between system requirements and V-elicit steps

The system has been implemented by the author and numerous programmers (students and research assistants) through many specific projects. The projects focused mainly on one step or feature at a time, giving adequate project management control. The final integration of the project into the entire system was made by the author, after conducting independent testing. In each project, documentation was also carried out. We have linked all these separate documents into an hypertext document, containing also an overall architecture diagram showing the dependencies across the different projects, features, or libraries. Here also, the integration of the documentation into the system documentation was performed by the author after verification of the completeness of the documents.

7.2 External Validation

While internal validation has shown that the V-elicit system is functioning properly, we are also concerned about the relevance of this work in a practical setting, and the advantages of V-elicit over the existing approaches and tools. For such verification of our theory and system, a demo is not sufficient: an empirical study is important [Tic98]. This section describes the case studies performed in order to do such verification.

Our external validation goals are:

G1 – *Process model quality*: to compare the quality of a model developed using V-elicit to those developed using other elicitation approaches and tools. The quality of the models developed is clearly important, as many further technical and business decisions are based on the resultant models.

G2 – *Elicitation process quality*: to compare the process of eliciting a model using V-elicit to those using other elicitation approaches and tools. The rapidity with which the models are developed and the amount of human or other resources used during the elicitation process are clearly important, as slow development or excessive resource consumption renders the tool unusable in a practical setting. Also, the amount of support provided in the elicitation process may affect the process model quality.

G3 - *Tool capability in a practical setting*: to verify that V-elicit can handle large-scale industrial processes. More specifically, we want to verify that constraints can be used to detect intra-view inconsistencies, that the similarity scores do identify most of the common components, and that the types of inconsistencies handled in V-elicit do actually occur in real situations. If the elicitor has to manually do a major part of the matching process, or if the types of inconsistencies managed by the tool do not generally occur in practice then this dismisses the practicality of the tool.

G4 - *Merging capability*: to verify that the system is indeed able to merge views developed by different elicitors (permitting parallel view elicitation). Such characteristic would allow us to elicit a large process model in a relatively short time frame.

G5 - *External validity constraint capability*: to verify that it is possible to define development policies in our constraint language, and validate a model against them.

Each of these goals are discussed in the following subsections. The following case studies are presented for meeting these goals:

Case study	Related goal
Case study #1: Comparison of model quality	G1
Case study #2: Comparison of elicitation processes	G2
Case study #3: Tool capability in a practical setting	G3
Case study #4: Parallel view elicitation	G4
Case study #5: External validity constraints	G5

Table 11 – Case studies and their related goal

Each goal is first refined into specific questions and metrics[31], that are then used for designing the case studies[32] needed to answer the derived questions. Information on how the case studies were executed and how data were gathered is also explained. The results of the case studies determine whether or not the validation goals have been met. These results are presented in specific sections below.

The last section summarizes our findings.

[31] This refinement method (called Goal/Question/Metric or GQM) for planning the metrics to be used in a case study and then for interpreting the results is presented in [BaW84].
[32] The method used here for designing case studies is presented in [FeP97].

7.2.1 Case study #1: Comparison of model quality

The goal here is to compare the quality of a process model produced by V-elicit to those produced by other elicitation approaches and tools (G1 above). Our research hypothesis (see Section 1.1) is that when multiple sources of information are considered in process elicitation, the model quality from V-elicit would be higher than that from other approaches. By "quality", we mean specifically completeness, consistency, and accuracy of the model.

For this case study, we asked six subjects to model three processes each, using either V-elicit or another process modeling tool (3 tools have been compared with V-elicit). We then compared the quality of the model produced across the different tools used.

The following section provides more in-depth information on the measures used for verifying our goals. Section 7.2.1.2 then presents the design of the experiment performed, and Section 7.2.1.3 describes how data was gathered. Finally, the analysis and results are discussed in Section 7.2.1.4.

7.2.1.1 Context for Case study #1

Using the Goal/Question/Metric (GQM) approach [BaW84], we refine our validation goals into measurable factors, which are then used in the case studies performed to verify our research hypothesis. Here are the questions derived from our specific goal (G1):

Q1 - Compared to the process models produced by other elicitation approaches and tools, are the models produced by V-elicit:

- Q1.1 - more complete?
- Q1.2 - more consistent (internally)?
- Q1.3 - more accurate (reflecting reality better)?

Note that these quality factors are considered important and are discussed in the literature [DNR90, Mad91, CKO92, FeH93].

The following metrics are used for answering the questions above in a quantitative way.

M1 - Proportion of the solution model present in the subject's model (*Q1.1*)

M2 - Proportion of the subject's model containing inconsistencies (an inconsistency being a conflicting information inside the model) (*Q1.2*)

M3 - Density of accuracy errors in the subject's model (i.e., entities or relationships representing the process incorrectly) (*Q1.3*)

These metrics are *indirect* ones. That is, they are obtained through calculations using other metrics *directly* available from the models. Such direct metrics used are listed below.

M4 - Number of elements (entities/relationships) in the subject's model

- M4.1 - Number of entities
- M4.2 - Number of relationships

M5 - Number of elements (entities/relationships) in the solution model (remark: depending on the tool used, these numbers may change)

- M5.1 - Number of entities
- M5.2 - Number of relationships within the scope modeled by the subject (i.e., relationships with entities that have not been modeled by the subject are not considered here)

M6 - Number of inconsistencies in the subject's model (remark: these are always related to a single entity)

- M6.1 - Number of inconsistencies related to the model structure (e.g., entities not linked in the model, activities without input or output, improper use of notation element, etc.)
- M6.2 - Number of inconsistencies related to activity decomposition (i.e., relationships shown at one level of decomposition but not

shown in the sub-activities, or not shown at upper level when it should be)

M7 - Number of elements (entities/relationships) in the actual process that are missing in the subject's model.

M7.1 - Number of entities in this case

M7.2 - Number of relationships in this case (not related to the entities involved in M7.1)

M8 - Number of elements (entities/relationships) not present in the actual process that were added to the model

M8.1 - Number of entities in this case

M8.2 - Number of relationships in this case (not related to the entities involved in M8.1)

M9 - Number of elements (entities/relationships) in both the actual process and the model, but that has not been modeled correctly (This is sometimes due to misunderstanding of the process.)

M9.1 Number of entities in this case

M9.2 Number of relationships in this case (not related to the entities involved in M9.1)

The core, indirect, metrics (M1 to M3) are related to the direct metrics (M4 to M9) in the following ways:

- M1 = proportion of solution model present in the subject's model
  = (proportion of the solution's entities modeled) *
  (proportion of the solution's relationships modeled within the scope of the entities modeled)
  $= (1 - \frac{M7.1}{M5.1}) * (1 - \frac{M7.2}{M5.2})$
- M2 = proportion of subject's model (entities only) containing inconsistencies
  $= \frac{M6.1 + M6.2}{M4.1}$

- M3 = density of accuracy errors in the subject's model

$$= \frac{M8.1 + M8.2 + M9.1 + M9.2}{M4.1 + M4.2}$$

The validity of our metrics lies in the fact that they have been derived from specific questions and related goals (using the GQM approach), and that they have been generated from the descriptions provided in papers discussing such quality factors (in [DNR90, Mad91, CKO92, FeH93]).

The measures defined above (M1 to M9) have been gathered during our case study, and the core ones (M1 to M3) have been analyzed. The following section describes the details of this study.

7.2.1.2 Design of Case study #1

Our general goal is to compare the quality of the models produced, as defined in the metrics M1 to M3 (dependent variables), when using different elicitation tools (independent variable). In order to do that, we asked different people (subjects) to model a set of three processes (objects) using one of the tools.

We have used a *randomized complete block design* [Hic93] in which the factor *tool used* is analyzed, blocked by *process modeled.* Our focus is on comparing models that are produced using V-elicit against the ones produced by other tools. However, significant differences can be noted across the models developed using other tools, and across the processes modeled. These effects had to be separated. We are not expecting any effect (or interaction) between the tools used and processes modeled (i.e., we do not expect that some processes may affect differently the results from each tool).

In this section, we describe the details of the case study, providing the characteristics of the tools, elicitors, and models to be elicited.

Hypotheses

For each metric (M) from M1 to M3, our hypotheses to be tested are:

Null hypothesis (H_0): There is no significant difference between the values of the metric M obtained from the subjects using V-elicit and the ones obtained from the subjects using other elicitation tools.

Alternative hypothesis (H_1): The values of the metric M obtained from the subjects using V-elicit are significantly larger (for M1) or smaller (for M2 and M3) than the ones obtained from the subjects using other elicitation tools. Larger values of the completeness metric, and smaller values of the inconsistency and accuracy metrics, mean that the models are of higher quality.

Modeling tools

The choice of the elicitation tools to be used (other than V-elicit) was based on the following criteria:

- notational paradigm (e.g., functional modeling, state-based modeling, etc.): each tool used supported a unique notational paradigm or combination of paradigms, in order to be able to generalize our results. We also had to make sure that these notational paradigms were representative of the ones used in other available tools.
- robustness: the tools should be commercially available, implying that they have been tested and/or used for non-trivial modeling.

Table 12 provides a summary of the tools chosen.

These tools were not running in the same environment as that for V-elicit. They are available on Windows platform only while V-elicit has been developed on a UNIX/X-Windows platform. However, we believe that this did not affect the results because both environments (and computers used) are fast enough to support these tools,

and that the subjects have been trained properly on the tool (within these environments) prior to the case study.

Tools	**Tool 1:Process 98**	**Tool 2: iThink**	**Tool 3: AI0 (IDEF0 notation)**
Company	Scitor Corporation	High Performance Systems Inc.	Knowledge Based System Inc.
Notational paradigm (as described in [CKO92] (Table 2))	system analysis and design, combined with control flow	state transition	system analysis and design, combined with triggers
Aspects covered	activity decomposition, information flow and activity ordering (except activity control[33])	activity ordering, information flow	activity decomposition, information flow, and activity control[33]

Table 12 - Tools used for comparison with V-elicit

Subjects

Six graduate students participated in the case study: a first group of three students modeled the processes using V-elicit, and a second group of three students used one of the three other modeling tools. These students had different backgrounds, as shown in Table 13[34]. However, they all had prior exposure to process modeling concepts through a graduate course and/or readings on that topic. When assigning randomly a tool to each of the students, we made sure that each group of students was composed of people with different background.

[33] "Activity control" refers to the relationship "activity manages activity", as defined in the modeling schema used in V-elicit (see Section 4.1).

[34] The data on the subjects' background has been gathered through interviews with the subjects prior to the case study.

Subject	Background prior to the case study	Tool used
#1	• researcher in software engineering • has experience with Statemate and Petri-Nets, but not for modeling software processes	V-elicit
#2	• researcher in software engineering • has taken a graduate course on software processes • has experience with business processes (as a manager)	V-elicit
#3	• researcher in software engineering • has received several months' industrial experience in process modeling • has taken a course on team software engineering	V-elicit
#4	• researcher in software engineering, doing PSP research • was the teaching assistant for the PSP course at McGill	Process 98
#5	• has taken a graduate course on software processes	iThink
#6	• researcher in software engineering • has taken a graduate course on software processes • has experience with process models and views through his research	AI0

Table 13 - Background of the subjects, and the elicitation tool assigned to them

The subjects were chosen from the graduate students attending the graduate course on software processes (Winter 1998) and from the graduate students working in the area of software engineering at McGill. We asked each potential subject if they would be willing to participate in the case study. Nobody was paid for such participation, it was just done on a voluntary basis. Initially seven persons responded (out of 14), but one of them had to resign because of his summer job. The others who did not answer were all from the graduate course, and not doing any research in software engineering. We were told that in most cases, these people were either away and could not make it for the case study, or did not have enough time for such study. Once they had accepted, we asked them to commit to go through the entire case study.

We believe that the results we got from them are valid since this was on a voluntary basis, and that they were not rewarded on the basis of their results (we just asked them to really do their best in modeling the processes). We motivated them on the basis of: potentially

useful/exciting research results, acknowledgement (indirectly) of their participation in the thesis research, and them learning about processes, models, tools and experimental software engineering.

Pre-case-study training

In order to ensure that the results were not affected by student's varying knowledge of process modeling in general, and of the tool used in particular, all subjects were trained prior to the start of the case study. First, general information on processes and process modeling was presented to them. Then they were shown how to use the specific tool assigned to them. They had to model three simple example processes, containing ten to fifteen entities (activities and artifacts) each: a simplified classic life-cycle model, a testing process (iterations between code fixing and testing), and the general phases of a design process (including architecture development and data design).

Before letting them work on the three case study processes, we verified their knowledge by checking their example models and asking them specific questions on the tools used. The example processes progressively introduced concepts such as entities and relationships, and the different aspects used in modeling: entity decomposition, information flow (input and output of activities), and activity ordering (sequencing, backtracking, and decision making). We made sure that the subjects understood these concepts and how to model them during the training, by checking that the appropriate structure was used in their models, and by asking them to explain their solutions. We were satisfied with their knowledge of the tool and their capacity in handling non-trivial situations (the later was tested by letting them figure out how to model controlled iterations in the testing process example). We believe that the model quality would generally not vary due to their knowledge of the tool or of process modeling concepts in general.

Table 14 indicates the time spent in this training phase for each subject. As one can see, the time spent in showing general modeling concepts is constant among the subjects, except for Subject #6, who knew already about the concepts of views. This initial training phase included an overview of the process modeling goals and concepts (an overview was sufficient since the subjects were already familiar with these topics through their courses and/or research). For the training related to the specific tool, the subjects using V-elicit needed significantly more time to learn how to use the tool, because of the numerous tasks the tool is performing, and the complexity of the concepts of constraints, inconsistencies across views, and view merging. Notice that this later training phase (on the specific tools used) included examples of what a model should contain, and quality issues in modeling, that were easier to introduce using example models in the specific tool used.

Modeling tool	**V-elicit**			**Process 98**	**iThink**	**AI0**
Subject	#1	#2	#3	#4	#5	#6
Time spent showing general modeling concepts (in minutes)	15	15	15	20	15	5
Time spent showing how to use their specific modeling tool (in minutes)	120	100	150	15	45	30
Time spent trying out their specific modeling tool with some examples (in minutes)	110	100	105	45	90	30

Table 14 - Time spent in different phases of the subject's training

Case study processes modeled

Each subject had to model three processes. In order to ensure that the processes used were not biased in favor of any particular tool, we selected the processes from external (neutral) sources. We also made sure that these processes contain information that is not trivial to model (as typically encountered in real situations), such as management activities and their interaction with development activities. Each of these three processes were described in English, from three different partially-overlapping views. We are not concerned here

with situations where only one source of information is available, because of our research hypothesis (that a view-based approach to eliciting software process models would result in high quality models).

One of these processes is the ISPW6[35] example of how software changes are handled in the development process [KFF91]. It has been designed independently by a group of well-known researchers in the field in the early 1990's. The process described is small, but it contains many complex elements that can be found in real settings. We modified it so that it was described from three different views, in order to match the elicitation setting (i.e., using multiple views) that V-elicit is meant for. We identified the activities where each of the three given roles (project manager, design engineers, and quality assurance engineers) were involved. Then we built each view with the set of activities involving the related role only. No information was added or removed from the process during such modification.

The other two processes used in the case study come from industrial-scale processes, elicited independently by a group of several researchers in another project [Mad91a]. One of these processes is a *preliminary analysis phase* of software development, and the other one is a *document review* process. A transcript of the interviews made (one per source of information or agent, for each process) was available to us. Since these processes were too large for our case study (it would have been impossible to ask the subjects to work on the case study for more than a week), we had to simplify them, without loss of generality. Our approach (to avoid biases) was to remove details in activities that were specified in one view only, keeping only the higher level description of such an activity. For example, in the case an analyst described all the details on how to produce one specific document (with no such details in other views), we just kept the general idea that such a document had to be written.

---

[35] ISPW6 – 6th International Software Process Workshop, Hakodate, Hokkaido, Japan, October 1990, published by IEEE Computer Society Press.

Some characteristics of the processes used may affect the results of the case study: the size of the model, the number of views used, and the degree of overlap among the views. For this case study, these characteristics are similar among the processes used. The large-scale (unmodified) version of the industrial processes have also been exercised using V-elicit, as part of additional validation (see Case study #3 below).

7.2.1.3 Data gathering for Case study #1

This section describes how the case study was executed, and how the data was gathered.

Just prior to eliciting the three processes for the case study, each subject was reminded of the importance of trying to do their best in modeling the processes. We specifically insisted on the fact that they should model the entire processes provided, without adding details not specified in the texts. The goal of such emphasis was to ensure that the subjects do not deliberately affect the quality of the model, in favor of a particular tool.

The subjects had to model their processes independently of each other. For the duration of the case study, we specifically asked the subjects not to talk about the case study with the other subjects.

Communication with the author was allowed for predetermined reasons during the case study. For example, the subjects could ask questions about the use of the modeling tool, or request additional information on the processes whenever they felt that some details were confusing. However, the author did not answer any questions related to the quality of the models being developed, even if such questions were asked. Each interaction with the subjects were recorded on paper by the author.

The author looked at how the models were actually developed, and took notes of the elicitation process used, but did not interfere in the process, keeping a role of a discreet observer (unless questions were asked).

The last elicitation step in V-elicit (model verification) could not be performed in our case study. During this step, the elicitor is supposed to show the model to the people who provided the process information, making sure that the elicitor understood and modeled the process correctly. Since the processes come from past projects or literature, such expert (verifier) was not available. Even though the author is quite familiar with the three processes, she could not possibly take that role and guarantee no bias in the results of the case study.

Data on the quality of the models produced was gathered after all subjects had finished developing their models. We first came up with a solution model for each process, and then compared the models produced with the solution model. Each time a quality problem was detected, we first looked at the textual descriptions of the views to see if such understanding of the process could have been possible from the text provided. If this was not the case, only then the error was reported under the appropriate metric, and included in a list of quality problems found. This list was used at the end for verifying again each model, and ensuring that quality problems were consistently identified across models.

7.2.1.4 Data analysis and results of Case study #1

In this section, the data are presented and analyzed for each core metric (M1 to M3), each of them being related to the specific question Q1.1 to Q1.3 (from section 7.2.1.1), respectively.

The technique used for analyzing our results is the "two-way ANOVA" (by process modeled and by subject), followed by an "analysis of means" (Student-Newman-Keuls range test) in the case that the values are significantly different, in order to show which subject (and tool used) has significantly better results [Hic93] (the significance level used throughout this section is 0.05)[36].

[36] The choice of the analysis technique has also been discussed with two experts in statistics.

The values obtained for each core metric is shown in Table 15. The last column indicates the p-value obtained with the ANOVA test, for the factor "subject", and if it is significant enough to reject the null hypothesis. The mean value across the processes is also provided, for each subject.

The results are discussed in the following sub-sections related to the specific questions (Q1.1 to Q1.3).

Tool used	**V-elicit**			**Tool 1**	**Tool 2**	**Tool 3**	**p-value**
Subject	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	
completeness (M1)							0.025
process 1	0.925	0.860	0.698	0.613	0.618	0.562	(signifi-
process 2	1.000	0.857	1.00	0.612	0.844	0.854	cant)
process 3	0.918	0.838	0.869	0.808	0.667	0.752	
mean	0.948	0.852	0.856	0.678	0.709	0.723	
inconsistency (M2)							0.440
process 1	0.100	0.000	0.150	0.000	0.167	0.091	
process 2	0.000	0.067	0.043	0.067	0.000	0.053	
process 3	0.045	0.000	0.043	0.000	0.053	0.240	
mean	0.048	0.022	0.079	0.022	0.073	0.128	
inaccuracy (M3)							0.073
process 1	0.044	0.176	0.079	0.094	0.100	0.125	
process 2	0.033	0.043	0.033	0.041	0.049	0.078	
process 3	0.035	0.070	0.057	0.072	0.093	0.082	
mean	0.037	0.096	0.056	0.069	0.081	0.095	

Table 15 - Data analysis of the case study #1

Completeness

As we can see from Table 15, the difference in model completeness is significant enough to reject the null hypothesis (p-value below the 0.05 significance level).

Additional tests on the means (Student-Newman-Keuls range test) have shown that there is no significant difference between subjects using V-elicit, or between subjects using the

other tools. However, there is a significant difference (at 0.05 level) between subjects using V-elicit and the ones using other tools.

From this, we conclude that, in general, the models developed using V-elicit have less missing information than the ones developed by using other elicitation tools. We believe that this difference comes from the fact that by allowing the elicitor to focus on one view at a time during the modeling process, more information can be extracted from the process.

Consistency

No significant difference has been found across the subjects in terms of model consistency. However, due to the case study settings used and the significant difference in completeness observed above, we would expect that the models produced using V-elicit would be less consistent than the ones produced by the other tools.

First, the different parts of the processes were not of the same complexity: some were more difficult to model than others, and so more error-prone in terms of consistency. For example, the link between management activities and development activities was not as obvious to model using usual links between development activities. In the case of the models produced with V-elicit, more of these complex parts were modeled, compared to the other models produced using other tools. It would then be normal to have an increase in the proportion of inconsistent elements in the V-elicit models.

Second, since the last part of the elicitation process (i.e., model verification with the people providing the process information) could not be carried out, the subjects did not perform constraint verifications on the merged model that are included in such step. We examined the subject's views prior to merging, and we found almost no inconsistencies. Most of the inconsistencies appeared only through the merging operation, which is a complex operation compared to what the other tools support. Many of these

inconsistencies could have been identified in a real setting (potentially for all subjects, not only the ones using V-elicit).

The fact that the V-elicit models were not (significantly) less consistent than the ones produced by other tools indicate that our system handles this issue very well, even better than what we would expect.

Accuracy

This quality factor represents how well the model produced reflects the actual process, and is related to the elicitor's understanding of the process. With just a textual description in hand, people may be tempted to use their own knowledge of similar processes during process modeling, which may not be true for the process at hand. We believe that the best way of ensuring that the elicitor's understanding of the process is correct is through some kind of validation with people involved in the process. As explained earlier, this was not possible in our case study. However, we are interested to see if the tool or the view-based approach has an influence over such a quality factor.

The accuracy metric (M3) is not significant at the 0.05 level, but there are still some significant differences. When applying the Student-Newman-Keuls range test, we can see that subject #1 has significantly more accurate models than the subjects #5 and #6 (using other tools). What is interesting in this difference is that subject #1 is the one with the least process-related experience among the subjects using V-elicit. It seems that a prior experience would affect the subject's understanding of the process to be modeled. Additional research on this relationship is beyond the scope of this thesis.

For reasons similar to that in our analysis of consistency, we should actually expect that the models produced using V-elicit would have more errors related to accuracy. The main reason is that the complex merging process alters the initial views through the selection of the entities and relationships to be kept, and during this selection the elicitor might not

keep an overall view of the process. Additional process elements would not be inserted, but other elements (which were correctly modeled in the views) could become wrongly modeled. The fact that the V-elicit models are actually not less accurate than those from the other tools, and are even more accurate in some cases, is actually encouraging.

Additional tests performed

In order to confirm our results using a *nonparametric test* (i.e., not assuming any specific distribution), we also performed the Friedman test [Dan90] on our data. This test is similar to the two-way ANOVA test, except that ranks are used instead of actual values of the metrics.

Our results from this test were similar to those from the ANOVA tests: the null hypothesis for metrics M1 (completeness) can be rejected at the 0.05 confidence level. For the other metrics (consistency and accuracy), no significant difference have been observed.

Since the models produced using V-elicit are more complete than the models produced using other tools, and that their consistency and accuracy is not affected adversely compared to the other tools (they could even be improved by performing the last step of model verification), we conclude that, in general, the use of V-elicit can improve the overall model quality. Thus, our research hypothesis[37] has been validated through this case study.

[37] Our research hypothesis is that a view-based approach (and its technical support) to eliciting software process models would result in high quality models, especially in terms of their completeness (see Section 1.1).

7.2.2 Case study #2: Comparison of elicitation processes

The goal here is to compare the elicitation process when using the V-elicit system to those when using other tools (G2 above). More specifically, we want to compare (a) the time spent in eliciting the models and (b) the additional resources used (e.g., interaction with an expert, use of paper during the elicitation process, etc.).

In the following section, the measures to be used in this case study are discussed. The design of the experiment performed is integrated with that of case study #1. The reader is referred to Section 7.2.1.2 for details. Section 7.2.2.2 presents how data was gathered. Finally, the analysis and results are presented and discussed in Section 7.2.2.3.

7.2.2.1 Context for Case study #2

In order to define an appropriate set of measures on the elicitation process, we need to be specific about the process issues to be examined. The questions are:

Q2 - How much of the elicitation process is supported or managed by the tool, and how much needs to be carried out outside the tool (e.g., on paper)? Elicitation tasks not supported by a tool could have more variability across multiple elicitation efforts than when they are managed by a tool. A standardized elicitation process is easier to predict.

Q3 - In general, is the elicitation process faster when using the V-elicit tool? If the use of an elicitation tool significantly increases the time spent eliciting a model, it might become unusable in a practical setting. Of course, the quality of the model produced will have to be taken into account in such analysis: it is normal to take more time in order to get a higher quality model.

Q4 - Is the elicitation tool difficult to use? If the tool is very difficult to use, elicitors may not see the benefits of the tool, and they may stop using it.

Based on these questions, the measures are (the related questions are shown in parentheses):

M10 - Total time spent in eliciting a model, in minutes. (*Q2, Q3*)

M11 - Time spent on elicitation tasks not performed using the tool, in minutes. Here we are considering only the tangible tasks (e.g., developing a draft model on paper), not the time spent reading the text for each view and mentally analyzing these views. (*Q2*)

M12 - Percentage of the time spent in elicitation tasks not performed using the tool. (*Q2*)

M13 - Number of times the elicitor has to refer to the tool documentation or ask an expert in order to understand how to use the tool for a particular task. (*Q4*)

7.2.2.2 Data gathering for Case study #2

As the subjects were developing their models, data was gathered through observation: timing of each elicitation task, if the task was performed using the tool or not, and any comment or question the subject had (especially any difficulty encountered with the specific tool). Because the subjects were often working at the same time, we needed an additional way to gather time information, in order to validate such data, and to ensure we did not miss any critical information.

In the case of the V-elicit tool, the system has been instrumented to keep track of timing information: time stamps were added to a file at the beginning and end of each major elicitation step.

For the other tools, such instrumentation was not possible. We thus asked the subjects themselves to record time information as well.

We did not find inconsistencies between the time recorded through observations and the ones recorded by the subjects themselves or by the tool, although the latter approach often provided more details than through observations.

7.2.2.3 Data analysis and results of Case study #2

The process data collected during our case study is presented in Table 16.

Tool used:	**V-elicit**			**Tool 1**	**Tool 2**	**Tool 3**
Subject:	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**
M10 – total elicitation time (in minutes)						
process 1	100	146	192	34	53	53
process 2	181	122	202	76	45	48
process 3	131	184	167	90	35	70
M11 - time (and proportion of time - M12) spent not using the tool in minutes						
process 1	0 (0%)	0 (0%)	0 (0%)	25 (74%)	38 (72%)	0 (0%)
process 2	0 (0%)	0 (0%)	0 (0%)	15 (20%)	23 (51%)	0 (0%)
process 3	0 (0%)	0 (0%)	0 (0%)	15 (17%)	15 (43%)	0 (0%)
M13 – number of times elicitor refers to tool documentation or expert	20	8	14	0	0	0

Table 16 - Information on the elicitation process performed during the case study

As one can see from Table 16 (M10), the elicitation process takes a lot more time when V-elicit is used than when other tools are used. On the other hand, as shown in the first case study, the result is of higher quality. The difference in the elicitation time is due mainly to the fact that with V-elicit, all the three views from each process have to be modeled (separately), even if the information is repeated in multiple views. This is necessary in order to detect any inconsistency across the different descriptions, and take the appropriate decision on how to resolve the inconsistency. In this case study, the three view descriptions were quite short (one page long of plain text for all three views), so

combining them manually using the foreign tools (Tool 1, 2, and 3) was not a difficult task. We believe that with larger processes, this advantage would be diminished significantly. Also, the views were highly overlapping, and the elicitors using V-elicit had to model some information multiple times. Again, in larger models, this overlap is usually not that significant.

Because the other elicitation tools do not have support for views and view merging, the elicitors (subjects #4 and #5) had to draw a first draft on paper of what the model would look like, and then transfer it to the elicitation tool. This is shown with the metrics M11 and M12 in Table 16. There is an exception with subject #6: the tool used in his case (AI0) allows one to list the different entities needed for the model (in a random order) before using them in the graphical model. The graphical editor of the tool AI0 is used for specifying relationships among the listed entities only, not for specifying new entities. With this feature, the elicitor did not feel the need to combine all the information first on paper. Such a global list of entities was produced as the elicitor read through the view descriptions, helping in gathering complete information. An advantage of V-elicit over AI0 is that relationships can also be listed as they are identified in the view descriptions.

After the case study, we asked the elicitors using other elicitation tools to indicate the approach they had used for merging the different views. They admitted that the models were constructed by first modeling the view that seemed most central to the process, and then by adding additional details from other views. We suspect that some of the quality problems inherent in their models could have been caused by such an elicitation approach: the views provided within a single process were sometimes inconsistent, and the most central view did not necessarily contain the right solution to an inconsistency problem. With the V-elicit system, such a problem is minimized greatly, due to its across-view consistency analysis feature that presents to the elicitor the possible solutions to each inconsistency.

One problem we found with the V-elicit system is that it contains so many different elicitation steps and uses so many novel (and seemingly difficult) concepts, that the elicitors had difficulties using the tool. They asked many questions related to how to use the tool during the elicitation process (20, 8, and 14, as indicated by metric M13 in Table 16), even after a quite extensive training period lasting four hours. In comparison, the subjects using other elicitation tools had no apparent difficulties (no questions asked during the elicitation process), even though they spent only one hour and a half to two hours for their training period.

In conclusion, the V-elicit system supports more elicitation activities (M11 and M12) than other tools, but its concepts are more difficult to understand, and the entire elicitation process takes more time than when non-view-based elicitation tools are used. It remains to be seen whether this "learning curve" plateaus out over a long-term use of the tool (or tool of this type) and whether the performance of V-elicit outweighs that of other tools at that time.

7.2.3 Case study #3: Tool capability in a practical setting

For this case study, we want to make sure that V-elicit can handle large-scale industrial processes (goal G3). We have to test our system in a real situation, showing that it can

- check intra-view consistency
- identify the common components across views (or at least provide help in the cases the elicitor is required for such decision);
- identify the actual inconsistencies across views; and
- that the types of inconsistencies handled do exist in real situations.

In case studies #1 and #2 described above, it was not feasible to use industrial-scale processes, and simplifications had to be made to such processes. In case study #3, an actual industrial-scale process was used, defined from three different (actual) agents.

The following sections describe the details of this case study, and the results obtained.

7.2.3.1 Context of Case study #3

The first part of this study involves the verification of the component matching capability of V-elicit. The following specific questions are asked:

Q5 - Are the expected matches really found by V-elicit?

Q6 - Are the entities not supposed to be matched really identified as such by V-elicit?

Q7 - How much of the entities in the final model required assistance by the elicitor for correctly matching them?

The required metrics for answering these questions are the following (with the related question in parenthesis):

M14 - percentage of the expected matches found by V-elicit (Q5)

M15 - percentage of the entities not supposed to be matched, identified as such by V-elicit (Q6)

M16 - percentage of the entities in the final model where the elicitor has been required for correctly matching them (Q7)

In the second part of this study, we want to verify that V-elicit can identify the inconsistencies across views, and that the inconsistencies handled do exist in real situations.

The types of inconsistencies that we want to check are associated with our view merging algorithm (see Section 6.3). The following questions provide more details on the type of inconsistency that we are interested in:

Q8 - Is the "missing element" type of inconsistency found in real situations? (case #1, p. 145)

Q9 - Is the "detail missing" type of inconsistency found in real situations? (case #2, p. 147)

Q10 - Is the "finer decomposition" type of inconsistency found in real situations? (case #3, p. 149)

Q11 - Is the "different grouping" type of inconsistency found in real situations? (case #4, p. 150)

Q12 - Is the "different decomposition" type of inconsistency found in real situations? (case #5, p. 152)

Q13 - Is the "details taken from outside (leaf)" type of inconsistency found in real situations? (case #6, p. 153)

Q14 - Is the "details taken from outside (non-leaf)" type of inconsistency found in real situations? (case #7, p. 155)

Q15 - Is the "different details" type of inconsistency found in real situations? (case #8, p. 156)

For each of these questions, a metric on the number of inconsistencies of each type found during the view merging step is defined (metrics M17 to M24).

The validation of the intra-view consistency checking feature does not require specific questions and metrics, because we are only concerned with the capability of V-elicit to handle this.

The metrics above (M14 to M24) were gathered in our case study. The details of this case study are provided in the next section.

7.2.3.2 Design of Case study #3

The example process for this case study was taken from a previous project in the Software Engineering Lab at McGill University, where a researcher modeled a company's Preliminary Analysis phase of software development [Mad91a].

The information we have is a transcript of the three interviews describing the point of view of three sources of information (agents): an analyst (developing the different documents), a pilot or client representative (providing the information to the analyst, validating the documents, and sometimes writing documents too), and a project manager. The size of each view and the amount of overlap between them is provided in Table 17.

	View 1	View 2	View 3
Number of activities in each view	29	49	29
Number of artifacts in each view	7	23	14
Number of roles in each view	6	7	6
Number of relationships in each view	172	313	162
Total number of unique activities (and % of overlap)	90 (14%)		
Total number of unique artifacts (and % of overlap)	36 (17%)		
Total number of unique roles (and % of overlap)	8 (100%)		

Table 17 - Size and overlap of the views modeled

In our tests, the treatment (i.e., applying the V-elicit system) and the subject (i.e., the elicitor, who was the author herself) were kept constant. The characteristics of the subject are not important here because the V-elicit system is used for performing the view-based elicitation techniques of interest.

It has not been possible to resolve the inconsistencies within or across views by going back to the agents, and the author had to make decisions based on her understanding of the process. However, such an issue can only affect the accuracy of the final model, not the results of this study (i.e., showing that V-elicit can handle inconsistencies in real situations). From the internal validation, we made sure that for each inconsistency found, the system was correctly merging the views, for all possible solution of the inconsistencies. Here we are concerned about the capacity of V-elicit to correctly identify all inconsistencies found in a real situation.

7.2.3.3 Data analysis and results of Case study #3

V-elicit successfully analyzed and merged the three views from the real process given (Preliminary Analysis phase of software development). In the following two parts, details are provided on how well V-elicit has handled the component matching and inter-view consistency verification steps.

Component matching (Q5 to Q7)

Table 18 shows the results of the matching process. As one can see, the system cannot handle all the cases, but it can reduce significantly the number of entities to be matched manually. Also, in the cases where the elicitor had to check the entities and make decisions about the appropriate matches, the use of the similarity scores has reduced considerably the number of alternatives to be considered (for each entity checked by the elicitor with respect to another view, only 3 to 5 possible matches were evaluated, instead of 29 for example - the total number of entities in the other view). It is to be noted that the incorrect identification of the matches were due to close similarity scores with other entities in these cases.

M14 - percentage of the expected matches found by V-elicit	58% (25/43)
M15 - percentage of the entities not supposed to be matched, identified as such by V-elicit	83% (89/107)
M16 - percentage of the entities in the final model where the elicitor has been required for correctly matching them	22% (30/134)

Table 18 – Indication of how well the matching process performed on the industrial process

Using the matches as identified by the system, and the similarity scores computed, the elicitor may also identify matches that were not obvious a priori. During our case study, the author has first identified the matches manually, to check them against the ones found by the system. After considering the matches automatically identified, she realized that two of the ones correctly identified by the system were missing from the ones identified

manually. This shows the usefulness of carefully analyzing the results (matches and similarity scores) of the system.

From the results of this case study, we have identified different situations where V-elicit has difficulties identifying the appropriate matches:

- differentiating between general phases and meeting activities involved in such phases (e.g., a document production phase, and the regular meetings involving all developers)
- differentiating entities described at different levels of abstraction (e.g., if one view contains a prototype, and another one makes the difference between a textual prototype and a graphical prototype)
- differentiating between a general step and a sub-activity being the core part of the step (e.g., a prototyping activity containing a planning activity and a presentation activity, but mainly containing a prototype development activity)

We are now planning to improve the similarity score formula in these areas, and improve the user interface in order to help the elicitor in the assessment of the matches automatically found. Additional case studies similar to this one will be necessary in order to find the optimum solution to this problem. From case study #1, we now know that it is worth exploring the problem of view merging further, because this approach can really result in higher quality models)

Inter-view consistency checking (Q8 to Q15)

Table 19 shows the number and type of the inconsistencies detected by V-elicit. These numbers have been validated by a manual detection of inconsistencies across views: V-elicit did not miss or add inconsistencies compared to what we have found manually.

As one can see, most of the types of inconsistencies handled by V-elicit can actually occur in real situations. It is important to detect them all because each inconsistency that is not resolved properly can lead to a quality problem in the final model. By identifying the

inconsistencies and providing the list of solutions based on the information provided in each view, the elicitor can choose the right solution in a systematic way.

Type of inconsistencies found	**# found** (for a model with 134 entities – see Table 17)
M14 - number of "missing element"	67
M15 - number of "detail missing"	7
M16 - number of "finer decomposition"	3
M17 - number of "different grouping"	8
M18 - number of "different decomposition"	1
M19 - number of "details taken from outside (leaf)"	0
M20 - number of "details taken from outside (non-leaf)"	0
M21 - number of "different details"	1

Table 19 - Number and types of the inconsistencies found across views

The high number of "missing element" type of inconsistency is due to the fact that the overlap across the views is quite small (only 15% for activities for example). Entities that are not in some of the views are often reported as "missing entities", except when other types of inconsistency apply in these cases (such as M15, M16, and M21).

In the case of M19 and M20, we were not expecting to detect such inconsistencies, as explained in Sections 6.3.1.6 and 6.3.1.7. In theory, it can happen that an entity (matched) has all of its sub-entities under other entities in the second view, but in this case the actual definition of the entity cannot be the same (close enough to still be matched).

Conclusion

In this case study, we have successfully modeled an industrial-scale software process using ERD, thus our assumption A1[38] has been validated. The use (and analysis) of the

[38] Assumption A1: A process model can be specified using an entity-relationship diagram (see Section 1.2).

specific view-based techniques have also shown the validity of our assumptions A3 to A5[39].

7.2.4 Case study #4: Parallel view elicitation

The purpose of this case study is to verify that it is indeed possible to merge views developed by different elicitors, and still get a high quality merged model (G4). This would mean that the overall elicitation time could be reduced significantly by performing the view elicitation step concurrently for each view.

Using such capability could also improve the model quality. By assigning only a part of the process model to be developed to an elicitor, the elicitation process is then more manageable, and the elicitor does not lose focus on time-consuming tasks. Also, if the elicitation process is performed over an extensive period of time, the process itself could have changed during that time, and these changes might not be reflected in the model developed. Hence, the model quality could be increased and the elicitation time decreased in such a case by letting multiple elicitors model subparts of the process in parallel, and then merging them.

The following subsections describe our case study and analyze the results.

---

[39] Assumption A3: By using a language based on first-order logic, one can define what an inconsistency is (inside a single view or model), and the inconsistency verification can then be automated.

Assumption A4: The identification of similar components across views can be partly automated through the computation of a similarity score across the components.

Assumption A5: By a careful identification of types of inconsistencies across views, and their possible solutions, the view merging process can be automated using the solutions provided by the elicitor.

(see Section 1.2)

7.2.4.1 Context of Case study #4

For this case study, the following specific questions are addressed:

Q13 - Is it possible to merge views of one process, produced by different elicitors?

Q14 - In the case it is possible to merge such views, do we get similar quality level than when all views are produced by the same elicitor?

In order to answer the first question, we do not need a metric: we are just checking whether or not a merged model has been built from the given views. For the second question, we are using the same quality metrics as in case study #1 (M1 to M3).

The following section describes how this case study was performed.

7.2.4.2 Design of Case study #4

In this case study, we used the views produced by the subjects using V-elicit in the case study #1, and then we merged them. For each of the three processes the subjects had to model, three views were available. We randomly mapped each view to one of the subjects, and built a new set of three views (all views from our new set were from different elicitors).

We then used the V-elicit system to merge such randomly chosen views into a merged model, and measured the quality of that model in the same way as in case study #1. We could then compare the quality of our merged model with the subject's models from case study #1.

This merging task was performed by the author, but this is not expected to have an impact on the results because the part that is expected to introduce more of the quality problems is the view elicitation part, and this was performed by the subjects of the case study #1.

The results of this case study are shown in the next section.

7.2.4.3 Data analysis and results of Case study #4

For all the three processes that were modeled, the V-elicit system was successful in merging the views from different elicitors. Table 20 shows the quality metrics for the models produced in this case study, as well as the original models produced by each subject in case study #1.

Subject	#1	#2	#3	parallel	p-value
completeness (M1)					0.282
process 1	0.925	0.860	0.698	0.842	
process 2	1.000	0.857	1.00	0.853	
process 3	0.918	0.838	0.869	0.771	
mean	0.948	0.852	0.856	0.822	
inconsistency (M2)					0.540
process 1	0.100	0.000	0.150	0.105	
process 2	0.000	0.067	0.043	0.000	
process 3	0.045	0.000	0.043	0.056	
mean	0.048	0.022	0.079	0.054	
inaccuracy (M3)					0.209
process 1	0.044	0.176	0.079	0.118	
process 2	0.033	0.043	0.033	0.012	
process 3	0.035	0.070	0.057	0.031	
mean	0.037	0.096	0.056	0.054	

Table 20 - Quality results, when combining views from different elicitors

We applied the same statistical tests as in case study #1 (two-way ANOVA test and Friedman's test). The last column of Table 20 indicates the p-value associated with the two way ANOVA test. None of the p-values are small enough to reject the null hypothesis (the same hypothesis as in case study #1), indicating that the models from case study #4 were not of significantly different quality than the ones produced by individual elicitors (case study #1).

7.2.5 Case study #5: External validity constraints

For this case study, we want to make sure that development policies can be defined in our constraint language, and that a model can be checked against them (G5). Note that because we are only concerned with the capability of V-elicit here, no specific question/metric is necessary.

We used the book by Davis [Dav95], which describes principles of software development, as our source of independently stated constraints. In this book, the principles are categorized by the development phase in which they apply (e.g., requirement engineering, design, etc.).

We selected approximately seven policies per development phase, for a total of 35 constraints, and we formally specified them in our constraint language. Each constraint has been tested on an example process model. The constraints specified are listed in Appendix E.

The policies were selected based on the amount of interpretation we had to make in order to formalize them: we avoided the ones that required an interpretation that could be different from one organization to another. For example, we did not specify a constraint such as "trust your people", that can be interpreted as "the manager should not be central to all development activities" or as "lower level steps should not all be managed by the top manager".

In some cases, the principle described was not really a development policy, but instead a description of some lessons learned. For example, the principle that states that a prototype reduces the risks associated with the selection of the user interface. Since these are not development constraints, they were not formally specified as such.

For the verification of such constraints on a given model, we have developed example models containing the information to be checked. For this purpose, the modeling schema had to be modified in order to handle the required types of information. This has shown the adaptability of the ERD modeling schema to one's needs (independently specified, from the given development policies), and thus the validity of our assumption A2[40].

The fact that we were able to specify the 35 constraints that were selected, and check the process models against these, demonstrates the capability of V-elicit in handling this type of constraints (and the validity of our assumption A6[41]).

7.2.6 Summary

From the five case studies described above, we noticed that:

- the completeness of the models produced by V-elicit is better than that in models produced by other elicitation tools, and that the consistency and accuracy stays the same even if it was actually expected to be worse in the circumstances mentioned above.
- the time spent eliciting models using V-elicit is generally greater than when using other elicitation tools. However, it is possible to model different views concurrently (by different elicitors), and still obtain high quality models.
- V-elicit has a higher learning curve than other tools, as it contains many novel concepts that most elicitors do not have experience with.
- V-elicit supports more of the elicitation tasks than do other tools, and the elicitors do not need to first produce drafts of models on paper, unlike in other elicitation tools.
- V-elicit can properly merge views from a large-scale process.
- the inconsistencies (across views) handled by V-elicit do occur in real situations.

---

[40] Assumption A2: Using entity-relationship diagrams allows the elicitor to define the types of information a model should contain (see Section 1.2).

[41] Assumption A6: By using a language based on first-order logic, one can formally describe development policies, and their verification on a given model can be automated (see Section 1.2).

- development policies can be formally specified in V-elicit, and models can be checked against them.

While replication of the case studies could lead to firmer results, our results support the argument that V-elicit system is a tangible progress over other elicitation approaches and tools.

From the case studies, we have identified several improvement opportunities:

- The subjects from the case study #1 have found that the use of plain text files when eliciting views had the following drawbacks: it was annoying to re-type the name of the entities when they were involved in multiple relationships, and typing mistakes were frequent but discovered too late (when translating back to V-elicit). More support would be required in such phase, probably by adding a graphical interface.
- In some cases, the commonality analysis function does not correctly detect common components across the views. Three situations that V-elicit has difficulties with have been identified in Section 7.2.3.3. An analysis of the types of information to be used in such cases, and potentially their weights, would be required to improve the commonality analysis function in these situations.
- From the observation of the subjects in the case study #1 (during the "identification of common components" phase), we noticed that they had difficulties in analyzing the similarity scores. A better interface (maybe a graphical one) would be required here, to provide better help and guidance during this task.

These issues are a subject of future research and experimentation.

7.3 Literature comparison

In the previous section, we have compared V-elicit with existing (commercially available) modeling tools. This comparison would not be complete without a comparison with state-of-the-art modeling tools described in the literature.

As discussed in Section 2.2, many process modeling tools and process-oriented development environments have been described in the literature in the last few years. These tools have either been used, or could be used, in industrial elicitation efforts. For a description of each of these tools, the reader is referred to the Appendix D. In this section, we compare those tools with our V-elicit system, on the basis of the support provided for view-based elicitation tasks.

Table 21 shows the results of our comparison. The criteria used in the comparison are the requirements for a view-based elicitation tool presented in Chapter Three. Notice that such requirements have been validated through presentations and discussions with experts in the field. We have added three other requirements ("model simulation", "process model execution", and "process guidance and work coordination"), to show the additional features handled by these tools, but not included in V-elicit since they are not necessary for the elicitation process.

As one can see, although these other tools are "good" for the purposes they were built (e.g., simulation, execution and guidance), they do not generally implement the requirements necessary for view-based elicitation.

In some cases (e.g., Adele-Tempo [BEM94], Articulator [Sca99], Marvel/Oz [BeK98], Merlin [ScW95], Process Weaver [Fer93], and ProcessWise [BGR94]), views of the process are provided during execution, mainly as agendas for developers. The list of activities to be in each of these views can be specified in the language, often as a link between an activity and the role performing it.

Some tools (e.g., Articulator [Sca99], MVP-E [BHM97], Statemate [KeH89], PFV [DPV97], and Funsoft nets [DeG98]) have functions for verifying the consistency and completeness of a model. Examples of such verifications include: use of undefined elements, type mismatches, unconnected or useless elements (e.g., artifact produced that is not used), deadlocks and race conditions, inconsistencies in the refinement of activities,

etc. These verifications are related to the fixed modeling schema provided by such tools, and cannot be specified by the user like in V-elicit.

In the case of MVP-E, some techniques are currently being developed by Verlage [Ver96] for helping in the elicitation of process models from different views: a similarity analysis function to help identifying the common elements across views, and a tentative set of consistency rules to detect inconsistencies between two views. As explained in Section 2.2, the similarity function is based on the MVP-L constructs only, and is not meant for any type of information. Also, the detection of inconsistencies does not handle the differences in the abstraction hierarchies (i.e., the inconsistencies related to entity decomposition).

Only one other tool, OPSIS [ACF96], allows the reconstruction of a model from views. The interfaces (or common elements) of the views are specified by the user in a formal notation. Operators are used to specify the steps in recombining the views (which elements should be kept, which elements should be added to link existing elements, and which label should be kept in case of conflicts among element names). Their approach is used in the context of the modification of an existing model, performed by first decomposing the model into views, and recomposing it after the changes are made. Such manual view-merging approach might be viable for such situation, but it is difficult to apply directly on a set of newly elicited views.

In summary, only V-elicit provides features for defining the modeling schema to be used, and elicit a model from different views, detecting and solving inconsistencies among them.

Tools (see Appendix D for descriptions)	Requirements for view-based elicitation											Other requirements		
	R1 -separate models for views	R2 - user-definable modeling schema	R3 - user-definable views	R4 - intra-view consistency verification	***R5** – identification of commonality across views	***R6** - detection of inconsistencies across views	***R7** - help in resolving inconsistencies across views	***R8** – view merging	R9 – static-semantics correctness of models	***R10a** – specifying development policies	***R10b** – checking model against development policies	model simulation	process model execution	process guidance and work coordination
Adele-Tempo	low		med.										high	high
APEL													high	high
Articulator	low		med.						high			high		
EPOS													high	
Funsoft nets									high			high		
JIL/Little-JIL													high	
Marvel / Oz	low		med.										high	high
Merlin	low		med.										high	high
MVP-E	high		high	high	high	low			high			high	high	
OPSIS	med.		high		low			med.						
Process Weaver	low		med.										high	high
ProcessWise	low		med.										high	high
PFV		med.							high					
SPADE													high	high
Statemate									high			high		
X-elicit		high												
V-elicit	high	high	high	high	high	high	high	high	high	high	high			

Table 21 - Comparison with tools described in the literature

***R** : indicates a key requirement

low/med./high: level of support for a particular requirement R1-R10, simulation, execution or environment

7.4 Lessons learned

We have learned several lessons related to V-elicit development and elicitation of process models. These are listed below.

The quality of process models developed is affected by the perception of the elicitor. We have noticed that even though a textual process description seems clear and unambiguous a priori, different elicitors may understand it in different ways, based on their knowledge of, or experience with, similar processes. Communication and model validation with agents involved in the elicited process is critical to increase the model accuracy.

Entities at different levels of abstraction may have similar descriptions. For example, an activity called "review" may be similar to the sub-activity "review meeting", because it is the core part of the review process. In both cases, the same agents are involved, and the input/output is similar. This kind of situation occurs more often than we expected, making the process of identifying common components across views impossible to fully automate. We believe that the elicitor will always be required for this task. However, the system can help by showing level of similarity between entities (as provided in V-elicit).

Full description of a given entity cannot be assumed by a single agent. Sometimes, one agent may be so absorbed in describing details that obvious tasks or input/output might not be mentioned in his/her view (but others interacting with this agent might be aware of such information). The impact of this is that the model we get from merging the different views may not be fully connected at the bottom level of abstraction, as we might expect. The "internal validation constraints" applied to views should be verified again on the final model.

Document a prototype adequately when it is large. Even though a prototype does not necessarily require full documentation, a large one (like V-elicit, with its 60 KLOC in 275 classes) does need at least documented requirements and design, for a better

understanding of the system and for further development or maintenance involving different people. We did not see that need at the beginning of the development effort, resulting in poor documentation. This then led to difficulties in training new people on that project. We thus decided to re-write documentation properly, in order to remedy to the situation in later development phases.

Chapter Eight - Summary and conclusion

In this thesis, we have presented new techniques for eliciting a software process model from different sources of information (or views): constraint verification, for detecting intra-view inconsistencies (both internal and external validation); component matching, for identifying common components across views; and view merging, for building a single overall model from the views, after detecting and resolving inconsistencies across them. These techniques have been implemented in a prototype system called "V-elicit", which also provides support for the entire elicitation process: elicitation planning, view elicitation, view merging, and verification of the final model.

The six initial technical assumptions (Section 1.2), related to our choice of approaches and techniques, have been verified through the case studies, in the following way:

Technical assumptions	Related case studies
A1. A process model can be specified using an entity-relationship diagram.	#3
A2. Using entity-relationship diagrams allows the elicitor to define the types of information a model should contain.	#5
A3. By using a language based on first-order logic, one can define what an inconsistency is (inside a single view or model), and the inconsistency verification can then be automated.	#3
A4. The identification of similar components across views can be partly automated through the computation of a similarity score across the components.	#3
A5. By a careful identification of types of inconsistencies across views, and their possible solutions, the view merging process can be automated using the solutions provided by the elicitor.	#3
A6. By using a language based on first-order logic, one can formally describe development policies, and their verification on a given model can be automated.	#5

Also, our research hypothesis (Section 1.1), stating that a view-based approach to eliciting software process model would lead to quality process models (mainly their completeness), has been verified through our case study #1.

Additional experimentation has shown that the time spent in modeling the different views separately (in V-elicit) is higher than when developing one model directly (in other tools). However, multiple elicitors can work in parallel on different views, without affecting the quality of the final model, thus reducing the overall elicitation time. The parallel view elicitation would also increase the chances of getting an accurate model in a changing environment (with a high turn-around of agents in the process), by permitting the elicitors to go back quickly to the sources of information when details or input are required. Their input might be required as early as in the intra-view consistency checking step.

Such a system helps in eliciting consistent, complete, and accurate process models in a systematic way. The benefits of a high quality descriptive process model is that any follow-up decisions would have a solid platform. Example decisions include: analysis of a descriptive process model to seek improvement opportunities; generalization of multi-project models to standardize product quality and development cycle-times; assessment and certification of processes; automation of processes; etc. Thus, as can be seen, many of the widely recognized process-oriented activities are based on the ground-work that is presented here. These activities have a positive impact on the quality of the software developed.

Finally, no other tool or approach, to our knowledge, provides such complete technological support for view-based elicitation.

References

[ACF96] Denis Avrilionis, Pierre-Yves Cunin, Christer Fernstrom, "OPSIS: A View Mechanism for Software Processes which Supports their Evolution and Reuse", Proc. of 18th International Conference on Software Engineering, Berlin, Germany, Springer, March 1996, pp. 38-47.

[ADH94] Jean-Marc Aumaitre, Mark Dowson, Del-Raj Harjani, "Lessons Learned from Formalizing and Implementing a Large Process Model", Proc. of Third European Workshop on Software Process Technology, Villard de Lans, France, Springer-Verlag, LNCS #772, February 1994, pp. 227-239.

[ArK94] James W. Armitage, Mark I. Kellner, "A Conceptual Schema for Process Definitions and Models", Proc. of Third International Conference on Software Process, Reston, Virginia, IEEE Computer Society Press, October 1994, pp. 153-165.

[BaW84] Victor R. Basili, David M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, SE-10, November 1984, pp. 728-738.

[BCH95] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost models for future life cycle processes: COCOMO 2.0", Annals of Software Engineering, vol.1 no. 1, November 1995, pp. 57-94.

[BDT96] Alfred Broeckers, Christiane Differding, Gunter Threin, "The Role of Software Process Modeling in Planning Industrial Measurement Programs", Proc. of Third International Metrics Symposium, Berlin, Germany, March 1996.

[BeD92] K. Bernadi, J.C. Derniame, "Software Processes Modeling : What, Who, When", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp.21-25.

[BeK98] Israel Z. Ben-Shaul, Gail E. Kaiser, "Federating Process-Centered Environments: the Oz Experience", Journal of Automated Software Engineering, vol. 5 no. 1, Kluwer Academic Publishers, January 1998, pp. 97-132.

[BEM94] Noureddine Belkhattir, Jacky Estublier, Walcelio Melo, "The Adele/Tempo Experience: An environment to support Process Modeling and Enaction", Software Process Technology, A. Finkelstein and J. Kramer and B. Nuseibeh (Eds), Wiley and Sons, 1994.

[BeT93] James B. Behm, Toby J. Teorey, "Relative Constraints in ER Data Models", Proc. of 12th International Conference on the Entity Relationship Approach, Arlington, Texas, Springer-Verlag, LNCS #823, 1993, pp.46-59.

[BFL95] Sergio Bandinelli, Alfonso Fuggetta, Luigi Lavazza, Maurizio Loi, Gian Pietro Picco, "Modeling and Improving an Industrial Software Process", IEEE Transactions on Software Engineering, vol.21 no.5, May 1995, pp.440-454.

[BGR94] R. F. Bruynooghe, R. M. Greenwood, I. Robertson, J. Sa, R. A. Snowdon, B. C. Warboys, "PADM: Toward a total process modeling system", Software Process Modeling and Technology, Finkelstein Kramer and Nuseibeh editors, Research Studies Press, 1994, pp. 293-334.

[BHM97] U. Becker, D. Hamann, J. Muench, M. Verlage, "MVP-E: A Process Modeling Environment", IEEE TCSE Software Process Newsletter, no. 10, Technical Council on Software Engineering, IEEE Computer Society, 1997.

[BMH96] T. Bruckhaus, N. H. Madhavji, J. Henshaw, I. Jensen, "Impact of Tools on Software Productivity", IEEE Software, vol. 13 no. 5, Sept. 1996, pp. 29-37.

[BMS95] Lionel Briand, Walcelio Melo, Carolyn Seaman, Victor Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization", Proc. of 17th International Conference on Software Engineering, Seattle, Washington, ACM Press, 1995, pp. 133-143.

[BNF96] Sergio Bandinelli, Elisabetta Di Nitto and Alfonso Fuggetta, "Supporting cooperation in the SPADE-1 Environment", IEEE Transactions on Software Engineering, vol. 22, no. 12, December 1996.

[BRB95] N.S. Barghouti, D.S. Rosenblum, D.G. Belanger, C. Alliegro, "Two Case Studies in Modeling Real, Corporate Processes", Software Process: Improvement and Practice, Wiley/Gauthier-Villars, Pilot Issue, vol.1, August 1995, pp.17-32.

[BrB96] Gilles Brassard, Paul Bratley, "Fundamentals of Algorithmics", Prentice Hall, 1996.

[Bro95] Alfred Bröckers, "Process-Based Software Risk Assessment", Proc. of Fourth European Workshop on Software Process Technology, Noordwijkerhout, The Netherlands, Springer-Verlag, LNCS #913, 1995, pp.9-29.

[CDP95] David C. Carr, Ashok Dandekar, Dewayne E. Perry, "Experiments in Process Interface Descriptions, Visualizations and Analyses", Proc. of Fourth European Workshop on Software Process Technology, Noordwijkerhout, The Netherlands, Springer-Verlag, LNCS #913, 1995, pp.119-137.

[CKO92] Bill Curtis, Mark I. Kellner, Jim Over, "Process Modeling", Communications of the ACM, vol.35 no.9, September 1992, pp. 75-90.

[CoW95] Jonathan E. Cook, Alexander L. Wolf, "Automating Process Discovery through Event-Data Analysis", Proc. of 17th International Conference on Software Engineering, Seattle, Washington, ACM Press, 1995, pp. 73-82.

[CRS92] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietman Theobald, Walter Zimmer, "OBST – An Overview", technical report FZI039.1, Forschungszentrum Informatik (FZI), Germany, June 1992.

[Dan90] Wayne W. Daniel, "Applied Nonparametric Statistics", second edition, PWS-KENT publishing company, 1990.

[Dav95] Alan Davis, "201 principles of software development", McGraw Hill, 1995.

[DEA98] Samir Dami, Jacky Estublier, Mahfoud Amiour, "APEL: a Graphical Yet Executable Formalism for Process Modeling", Journal of Automated Software Engineering, Kluwer Academic Publishers, vol. 5 no. 1, January 1998.

[DeG93] Wolfgang Deiters, Volker Gruhn, "Software Process Technology Transfer - A Case Study Based on FUNSOFT Nets and MELMAC", Proc. of 8th International Software Process Workshop, Schloss Dagstuhl, Germany, IEEE Computer Society Press, March 1993, pp. 50-52.

[Dei92] Wolfgang Deiters, "A View Based Software Process Modeling Language", Ph.D. Thesis, University of Dortmund, Dortmund, Germany, December 1992.

[DeO92] Agnes Devarenne, Claire Ozanne, "The Need of a Process Engineering Method", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp.26-30.

[DNR90] Mark Dowson, Brian Nejmeh, William Riddle, "Fundamental Software Process Concepts", technical report no.7-7-5, Software Design & Analysis Inc, April 1990.

[DPV97] Ashok Dandekar, Dewayne E. Perry, Lawrence G. Votta, "A Study in Process Simplification", Software Process: Improvement & Practice, Wiley/Gauthier-Villars, vol.3 no.2, June 1997.

[Dre93] Daniel W. Drew, "Developing Formal Software Process Definitions", Proc. of Conference on Software Maintenance, Montreal, Canada, IEEE Computer Society Press, September 1993, pp.12-20.

[Eas91] Steve Easterbrook, "Handling conflict between domain descriptions with computer-supported negotiation", Knowledge Acquisition, vol.3 (1991), pp. 255-289.

[EBL96] Wolfgang Emmerich, Sergio Bandinelli, Luigi Lavazza, Jim Arlow, "Fine-grained Process Modeling: an Experiment at British Airways", Proc. of Fourth International Conference on the Software Process, IEEE Computer Society Press, December 1996.

[EsB95] Jacky Estublier, Noureddine Belkhatir, "A Generalised Multi-View Approach", Proc. of Fourth European Workshop on Software Process Technology, Noordwijkerhout, The Netherlands, Springer-Verlag, LNCS #913, April 1995, pp. 179-184.

[Fav92] John Favaro, "Process Modelling at the European Space Agency", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp. 159-162.

[FeF85] Paul Feldman, Guy Fitzgerald, "Representing Rules Through Modelling Entity Behavior", Proc. of Fourth International Conference on Entity-Relationship Approach, Chicago, Illinois, IEEE Computer Society Press, October 1985, pp. 189-198.

[FeH93] Peter H. Feiler, Watts S. Humphrey, "Software Process Development and Enactment: Concepts and Definitions", Proc. of Second International Conference on the Software Process, Berlin, Germany, IEEE Computer Society Press, February 1993, pp. 28-40.

[FeP97] Norman E. Fenton, Shari Lawrence Pfleeger, "Software Metrics: A Rigorous and Practical Approach", second edition, International Thomson Publishing, 638 pp., 1997.

[Fer93] Christer Fernström, "Process Weaver: Adding Process Support to UNIX", Proc. of Second International Conference on the Software Process, Berlin, Germany, February 1993.

[FGH93] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications", IEEE Transactions on Software Engineering, vol.20 no.8, August 1994, pp.569-578.

[FKN92] A. Finkelstein, J. Kramer, B. Nuseibeh, I. Finkelstein, M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", Int. Journal of Software Engineering and Knowledge Engineering, World Scientific, vol.2 no.1, March 1992, pp.31-57.

[Fra91] Dennis J. Frailey, "Defining a Corporate-wide Software Process", Proc. of First International Conference on the Software Process, Redondo Beach, California, IEEE Computer Society Press, October 1991, pp.113-121.

[Fra93] Dennis J. Frailey, "Concurrent Engineering and the Software Process", Proc. of Second International Conference on the Software Process, Berlin, Germany, IEEE Computer Society Press, February 1993, pp. 103-114.

[Gal92] Johan Galle, "Applying Process Modelling", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp. 230-236.

[GaS93] Brian R. Gaines, Mildred L. G. Shaw, "Eliciting Knowledge and Transferring it Effectively to a Knowledge-Based System", IEEE Transactions on Knowledge and Data Engineering, vol.5 no.1, February 1993, pp. 4-13.

[Gib94] W. Wayt Gibbs, "Software's Chronic Crisis", Scientific American, vol.271 no.3, September 1994, pp. 86-95.

[GrJ92] Volker Gruhn, Rudiger Jegelka, "An Evaluation of FUNSOFT Nets", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp. 196-214.

[GrS92] Volker Gruhn, Armin Saalmann, "Software Process Validation Based on FUNSOFT Nets", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp. 223-226.

[GrW96] R. M. Greenwood, B. Warboys, "ProcessWeb: Process Support for the World Wide Web", 5th European Workshop on Software Process Technology, Nancy, France, LNCS #1149, 1996, pp. 82-85.

[HeF94] Den Heller, Paula Ferguson, "Motif programming manual for OSF/Motif Release 1.2", O'Reilly & Associates, 1994.

[Hic93] Charles R. Hicks, "Fundamental Concepts in the design of experiments", 4th edition, Saunders College Publishing, 1993.

[HMB94] Dirk Holtje, Nazim H. Madhavji, Tilmann Bruckhaus, WonKook Hong, "Eliciting Formal Models of Software Engineering Processes", Proc. of the 1994 CAS Conference (CASCON'94), Toronto, Ontario, Canada, IBM Canada Ltd. and The National Research Council of Canada, October 1994, pp. 82-98.

[HuK89] Watts S. Humphrey, Mark I. Kellner, "Software Process Modeling: Principles of Entity Process Models", Proc. of 11th International Conference on Software Engineering, IEEE Computer Society Press, May 1989, pp. 331-342.

[Hum93] Watts S. Humphrey, "The Process Evolution Process", Proc. of the International Workshop on the Evolution of Software Processes, Montreal, Canada, January 1993.

[JaM94] David Jacobs, Chris Marlin, "Software Process Representation to Support Multiple Views", Proc. of First Asia-Pacific Software Engineering Conference, Japan, Dec. 1994, also published in International Journal of Software Engineering and Knowledge Engineering, vol.5 no.4, Dec. 1994.

[Kaw92] Peter J. Kawalek, "The Process Modelling Cookbook Orientation, Description and Experience", Proc. of Second European Workshop on Software Process Technology, Trondheim, Norway, Springer-Verlag, LNCS #635, September 1992, pp. 227-229.

[KeH89] Mark I. Kellner, Gregory A. Hansen, "Software Process Modeling: A Case Study", Proc. of 22nd Annual Hawaii International Conference on System Sciences, vol II - Software Track, IEEE Computer Society Press, January 1989, pp.175-188.

[Kel91] Mark I. Kellner, "Software Process Modeling Support for Management Planning and Control", Proc. of First International Conference on the Software Process, Redondo Beach, California, IEEE Computer Society Press, October 1991, pp. 8-28.

[KFF91] Mark I. Kellner, Peter H. Feiler, Anthony Finkelstein, Takuya Katayama, Leon J. Osterweil, Maria H. Penedo, H. Dieter Rombach, "ISPW-6 Software Process Example", Proc. of First International Conference on the Software Process, Redondo Beach, California, IEEE Computer Society Press, October 1991, pp. 176-186.

[KiM93] David H. Kitson, Stephen M. Masters, "An Analysis of SEI Software Process Assessment Results: 1987-1991", Proc. of 15th International Conference on Software Engineering, Baltimore, Maryland, IEEE Computer Society Press, May 1993, pp. 68-77.

[KoN96] Eleftherios Koutsofios, Stephen C. North, "Editing graphs with dotty", technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, June 96.

[KTL92] Herb Krasner, Jim Terrel, Adam Linehan, Paul Arnold, William H. Ett, "Lessons Learned from a Software Process Modeling System", Communications of the ACM, vol.35 no.9, September 1992, pp. 91-100.

[Lec89] Steven R. Leclair, "Interactive Learning: A multiexpert paradigm for acquiring new knowledge", SIGART Newsletter, special issue on knowledge acquisition, #108, April 1989.

[LeF91] J. C. Leite, P. A. Freeman, "Requirements Validation Through Viewpoint Resolution", IEEE Transactions on Software Engineering, vol.17 no.12, December 1991, pp. 1253-1269.

[LHR95] Christopher Lott, Barbara Hoisl, H. Dieter Rombach, "The Use of Roles and Measurement to Enact Project Plans in MVP-S", Proc. of Fourth European Workshop on Software Process Technology, Noordwijkerhout, The Netherlands, Springer-Verlag, LNCS #913, April 1995, pp. 30-48.

[Mad91] Nazim H. Madhavji, "The Process Cycle", Software Engineering Journal, vol.6 no.5, September 1991, pp. 234-242.

[Mad91a] Nazim H. Madhavji, "The Macroscope Project – Software Process Engineering and Evolution", Research Proposal submitted to CRIM, McGill University, June 1991.

[MBB92] N. H. Madhavji, J. E. Botsford, T. W. Bruckhaus, K. El Emam, "Quantitative Measurements based on Process and Context Models", Proc. Workshop on Experimental Software Engineering Issues, Lecture Notes in Computer Science, Springer-Verlag, Dagstuhl, Germany, Sept. 1992, pp. 67-72.

[McB93] Clement L. McGowan, Shawn A. Bohner, "Model Based Process Assessment", Proc. of 15th International Conference on Software Engineering, Baltimore, Maryland, IEEE Computer Society Press, May 1993, pp.202-211.

[MHH94] Nazim H. Madhavji, Dirk Holtje, WonKook Hong, Tilmann Bruckhaus, "Elicit: A Method for Eliciting Process Models", Proc. of Third International Conference on the Software Process, Reston, Virginia, IEEE Computer Society Press, October 1994, pp. 111-122.

[Nej91] Brian Nejmeh, "Strategic Software Process Improvement Planning", technical report no.7-46-1, Software Design & Analysis Inc, March 1991.

[NeR91] Brian Nejmeh, William E. Riddle, "Process Breakdown Structures: An Informal Technique for Software Process Definition", technical report #7-25-7, Software Design & Analysis Inc., March 1991.

[Nej95] Brian A. Nejmeh, "Process Cost and Value Analysis", Communications of the ACM, vol.38 no.6, June 1995, pp. 19-24.

[NWC97] Minh N. Nguyen, Alf Inge Wang, Reidar Conradi, "Total Software Process Model Evolution in EPOS", Proc. of 19th International Conference on Software Engineering, Boston, Massachusetts, IEEE Computer Society Press, 1997, pp.390-399.

[OiB92] Markku Oivo, Victor R. Basili, "Representing Software Engineering Models: The TAME Goal Oriented Approach" IEEE Transactions on Software Engineering, vol.18 no.10, October 1992, pp. 886-897.

[PCC93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Charles V. Weber, "Capability Maturity Model, Version 1.1", IEEE Software, vol.10 no.4, July 1993, pp. 18-27.

[Pen89] Maria H. Penedo, "Acquiring experience with executable process models", Proc. of Fifth International Software Process Workshop, Kennebunkport, Maine, IEEE Computer Society Press, 1989, pp. 112-115.

[Pfl93] Shari Lawrence Pfleeger, "Lessons Learned in Building a Corporate Metrics Program", IEEE Software, vol. 10 no. 3, May 1993, pp. 67-74.

[PhS94] Keith Phalp, Martin Sheppard, "A Pragmatic Approach to Process Modelling", Proc. of Third European Workshop on Software Process Technology, Villard de Lans, France, Springer-Verlag, LNCS #772, February 1994, pp. 65-68.

[Pre97] Roger S. Pressman, "Software Engineering: A Practitioner's Approach", 4th edition, McGraw-Hill, 1997.

[PSV94] Dewayne E. Perry, Nancy A. Staudenmayer, Lawrence G. Votta, "People, Organizations, and Process Improvement", IEEE Software, vol.11 no.4, July 1994, pp. 36-45.

[RHM85] R. A. Radice, J. T. Harding, P. E. Munnis, R. W. Phillips, "A Programming Process Study", IBM System Journal, 24(2), 1985, pp. 91-101.

[Rom93] H. Dieter Rombach, "Practical use of formal process models: first experiences", Proc. of 8th International Software Process Workshop, Schloss Dagstuhl, Germany, IEEE Computer Society Press, March 1993, pp.132-134.

[SaW94] Jin Sa, Brian C. Warboys, "Modelling Processes Using a Stepwise Refinement Technique", Proc. of Third European Workshop on Software Process Technology, Villard de Lans, France, Springer-Verlag, LNCS #772, February 1994, pp. 40-58.

[Sca99] Walt Scacchi, "Experience with Software Process Simulation and Modeling", to appear in Journal of Systems and Software, 1999.

[ScM93] Walt Scacchi, Peiwei Mi, "Experiences in the Modeling, Analysis, and Simulation of Formalized Software Processes", Proc. of 8th International Software Process Workshop, Schloss Dagstuhl, Germany, IEEE Computer Society Press, March 1993, pp. 135-138.

[ScW95] Wilhelm Schäfer, Stefan Wolf, "Cooperation Patterns for process-centred Software Development Environments", Proc. of 7th International Conference on Software Engineering and Knowledge Engineering, Rockville, Maryland, June 1995.

[ShG89] Mildred L. G. Shaw, Brian R. Gaines, "Comparing conceptual structures: consensus, conflict, correspondence and contrast", Knowledge Acquisition, vol.1 (1989), pp. 341-363.

[Sid97] Saeed Siddiqui, "Measuring the impact of process models on achieving a common understanding of a process: A case study", M.Sc. thesis, McGill University, Montreal, Canada, 1997.

[SKV95] I. Sommerville, G. Kotokya, S. Viller, P. Sawyer, "Process Viewpoints", Proc. of Fourth European Workshop on Software Process Technology, Noordwijkerhout, The Netherlands, Springer-Verlag, LNCS #913, April 1995, pp.2-8.

[SuO97] Stanley M. Sutton Jr. and Leon J. Osterweil, "The Design of a Next-Generation Process Language", Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Lecture Notes in Computer Science #1301, September 1997, pp.142-158.

[Tic98] Walter F. Tichy, "Should computer scientists experiment more?", IEEE Computer, May 1998, pp. 32-40.

[TSK95] Toshifumi Tanaka, Kushi Sakamoto, Shinji Kusumoto, Ken-ichi Matsumoto, Tohru Kikuno, "Improvement of Software Process by Process Description and Benefit Estimation", Proc. of 17th International Conference on Software Engineering, Seattle, Washington, ACM Press, 1995, pp. 123-132.

[TuM96] Josée Turgeon, Nazim H. Madhavji, "A Systematic, View-Based Approach to Eliciting Process Models", Proc. of Fifth European Workshop on Software Process Technology, Nancy, France, LNCS #1149, October 1996, pp. 276-282.

[TuT93] Efraim Turban, Margaret Tan, "Methods for knowledge acquisition from multiple experts: an assessment", Int. Journal of Applied Expert Systems, vol.1 no.2, 1993, pp.101-119.

[Ver96] Martin Verlage, "About Views for Modeling Software Processes in a Role-specific Manner", Proc. of the Workshop on Viewpoints, San Francisco, California, USA, ACM Press, October 1996.

[Vis94] Giuseppe Visaggio, "Process Improvement Through Data Reuse", IEEE Software, vol. 11 no. 4, July 1994, pp. 76-85.

[Vot93] Lawrence G. Votta Jr., "Comparing One Formal to One Informal Process Description", Proc. of 8th International Software Process Workshop, Schloss Dagstuhl, Germany, IEEE Computer Society Press, March 1993, pp. 145-147.

[WLM98] Alexander Wise, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton, Jr., "Specifying Coordination in Processes Using Little-JIL", Technical Report 98-38, Department of Computer Science, University of Massachusetts at Amherst, August 31, 1998.

[Wol89] Walter A. Wolf, "Knowledge Acquisition from Multiple Experts", SIGART Newsletter, special issue on knowledge acquisition, #108, April 1989.

[YuM94] Eric S. K. Yu, John Mylopoulos, "Understanding Why in Software Process Modelling, Analysis, and Design", Proc. of 16th International Conference on Software Engineering, May 1994, pp.159-168.

Appendix A - Views used as example for Section 5.2

The following figures represent the entire set of aspects for each of the three views in Section 5.2. As a reminder, these three views were used to illustrate how to elicit a software process model from different views. In Figure 25, Figure 26, and Figure 27, partial information on these views were given. Here, the entire information is presented, through different aspects, as it appears in the V-elicit system.

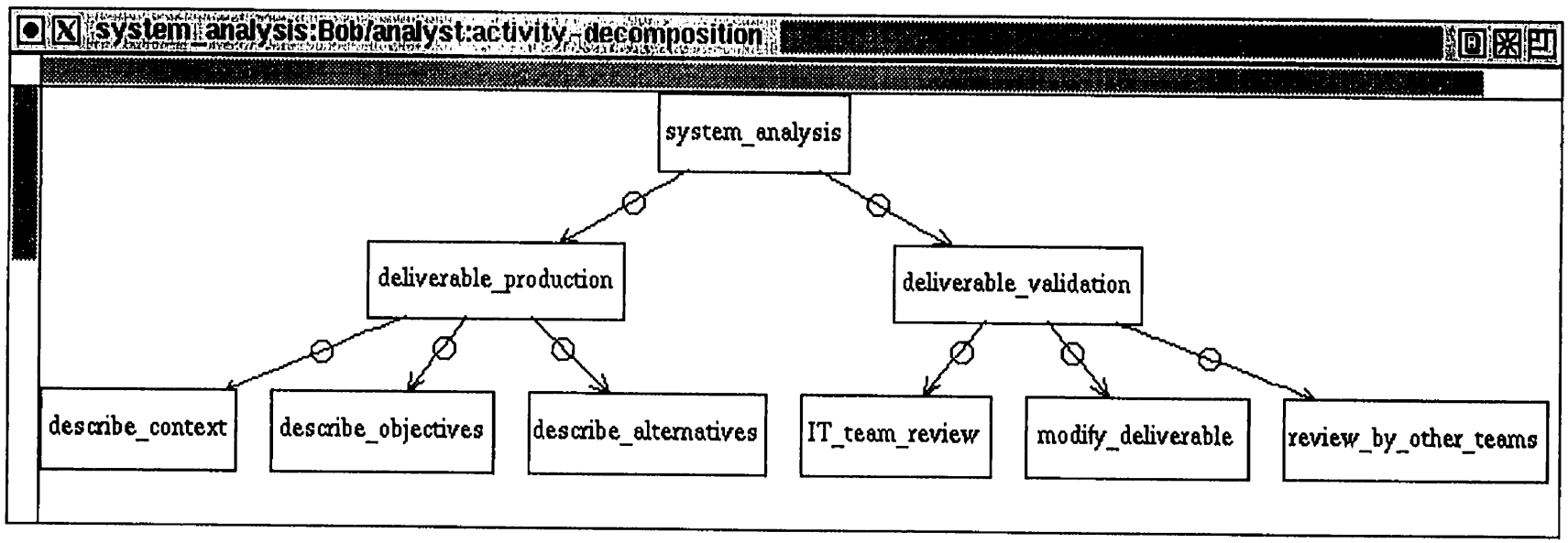


Figure 77 - Bob's activity decomposition aspect

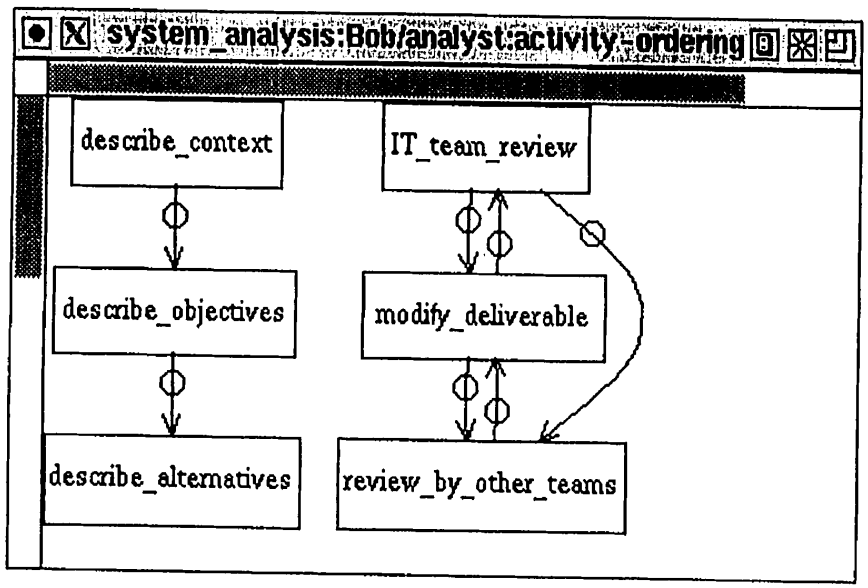


Figure 78 - Bob's activity ordering aspect

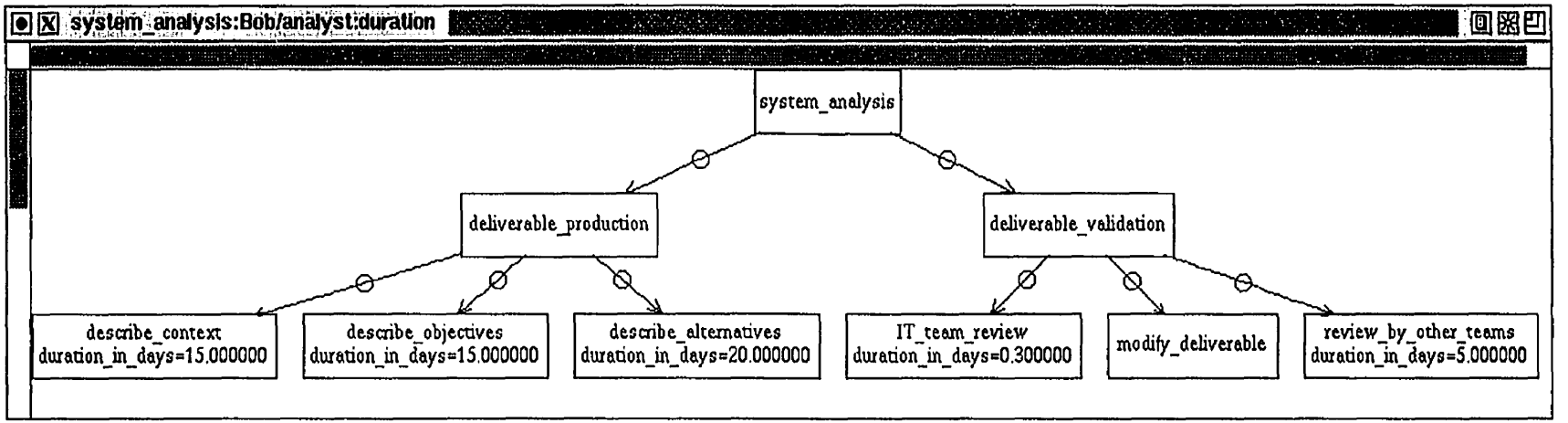


Figure 79 - Bob's activity duration aspect

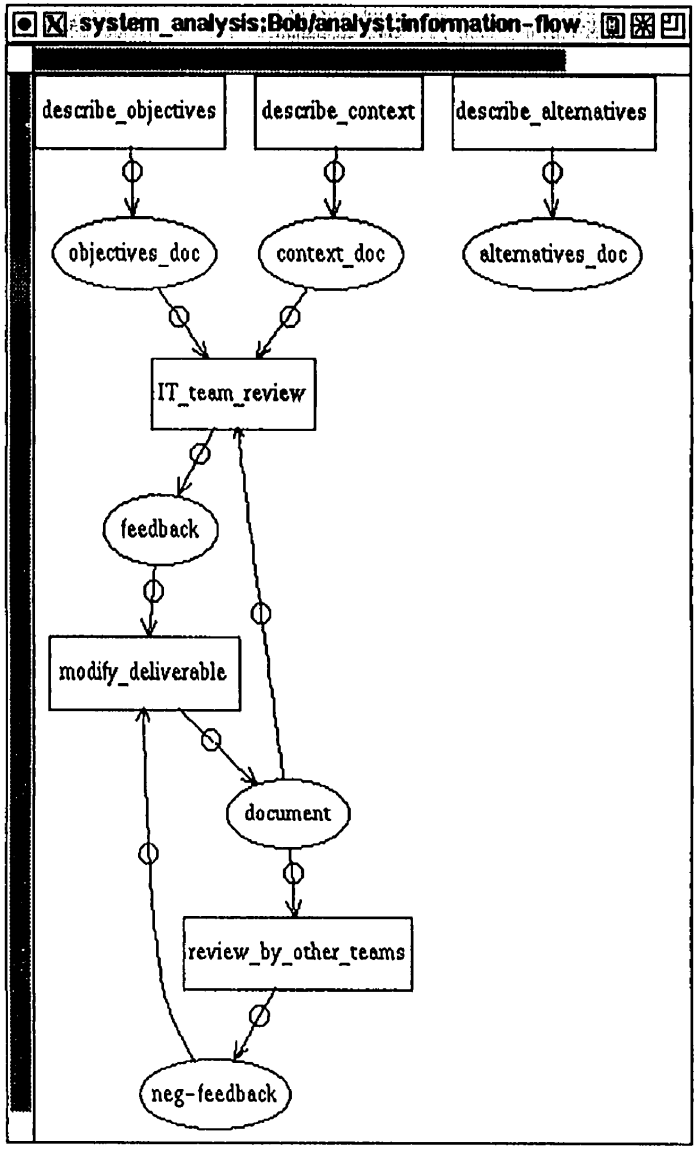


Figure 80 - Bob's information flow aspect

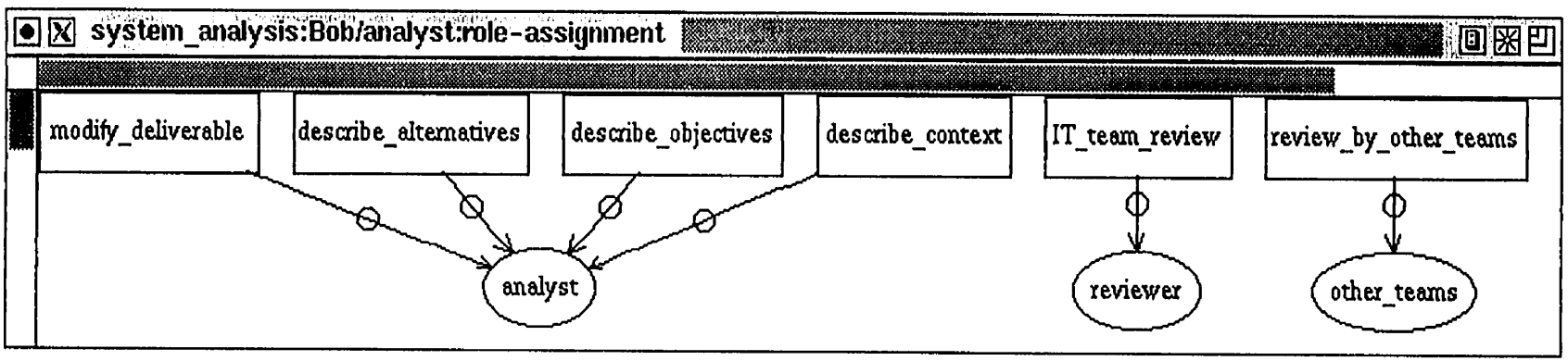


Figure 81 - Bob's role assignment aspect

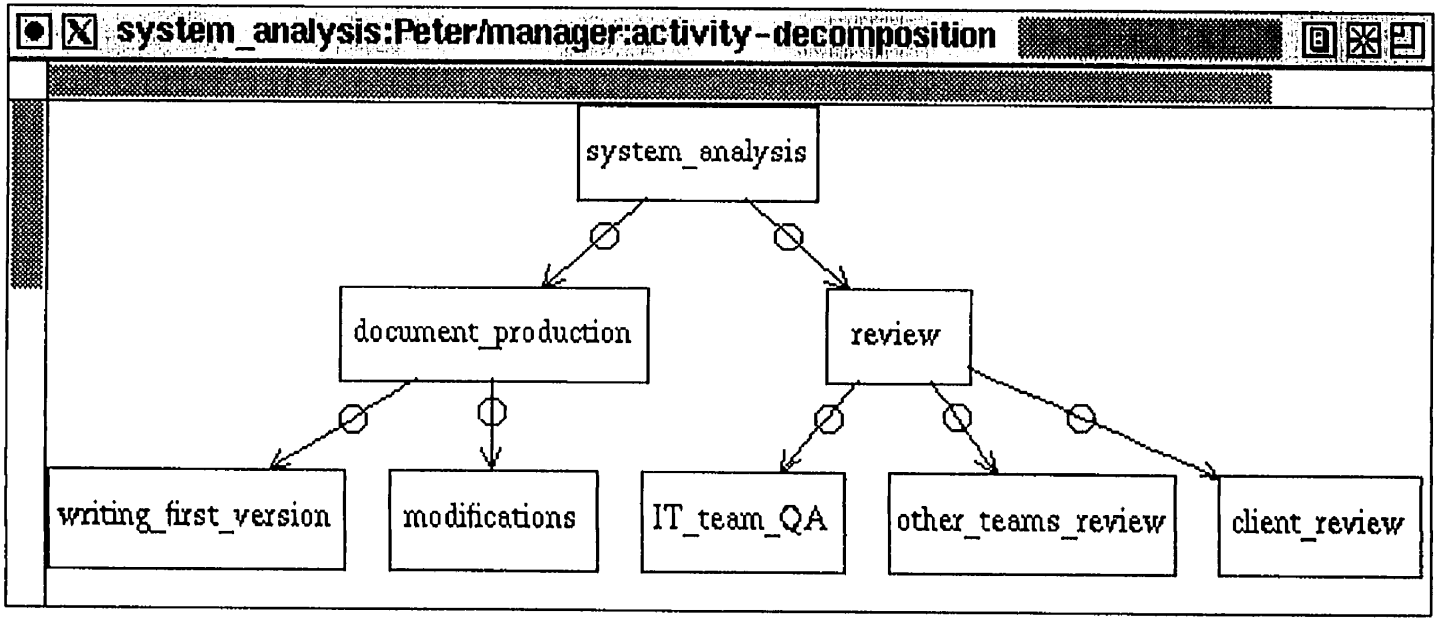


Figure 82 - Peter's activity decomposition aspect

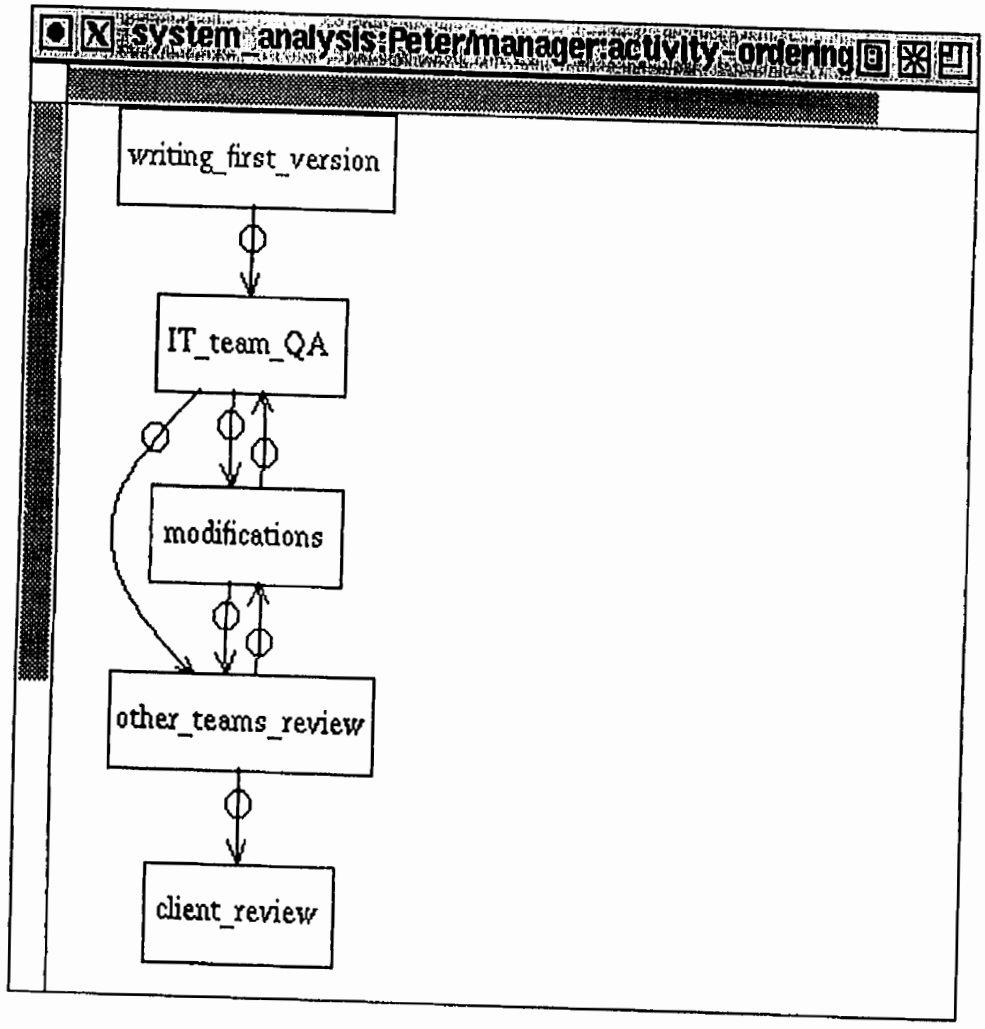


Figure 83 - Peter's activity ordering aspect

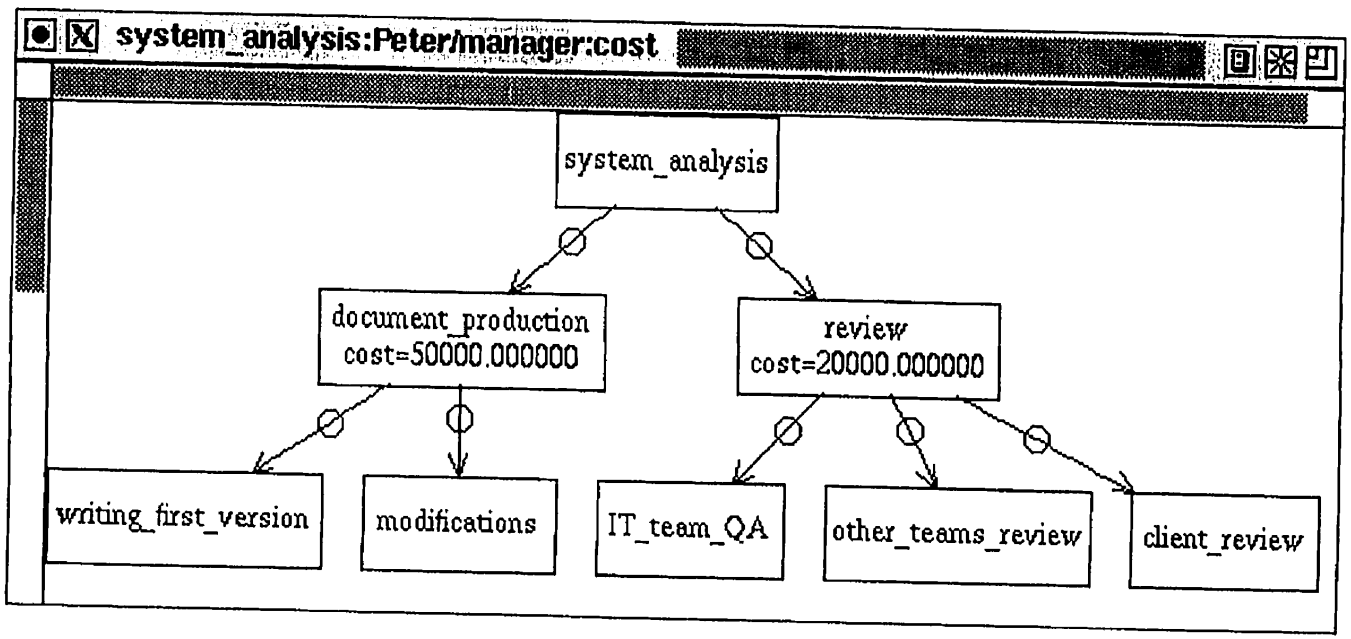


Figure 84 - Peter's cost of activity aspect

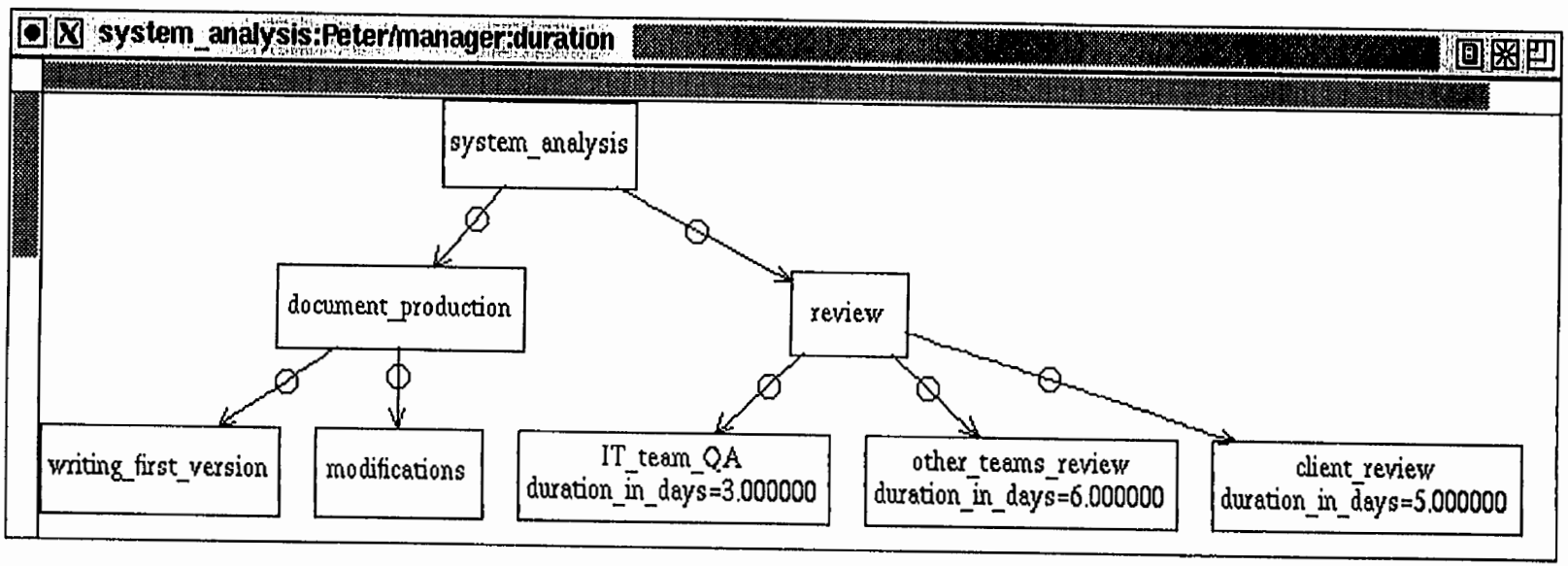


Figure 85 - Peter's activity duration aspect

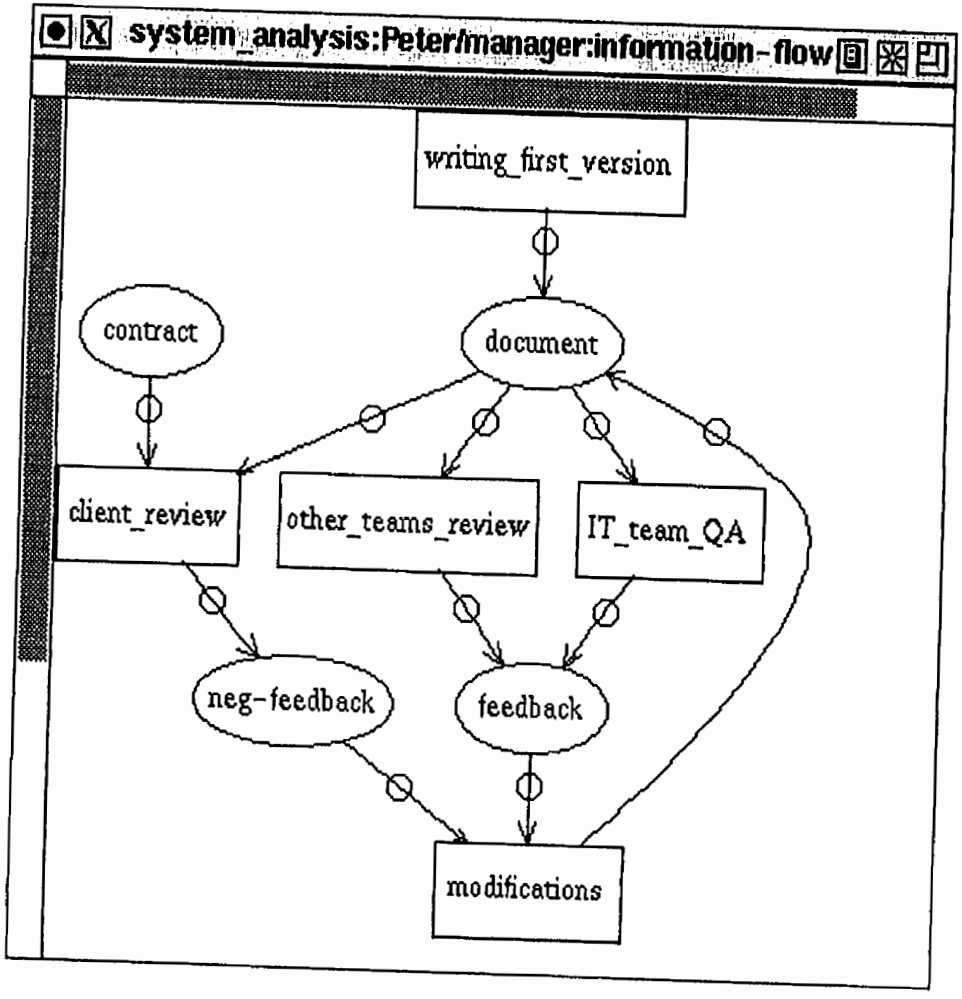


Figure 86 - Peter's information flow aspect

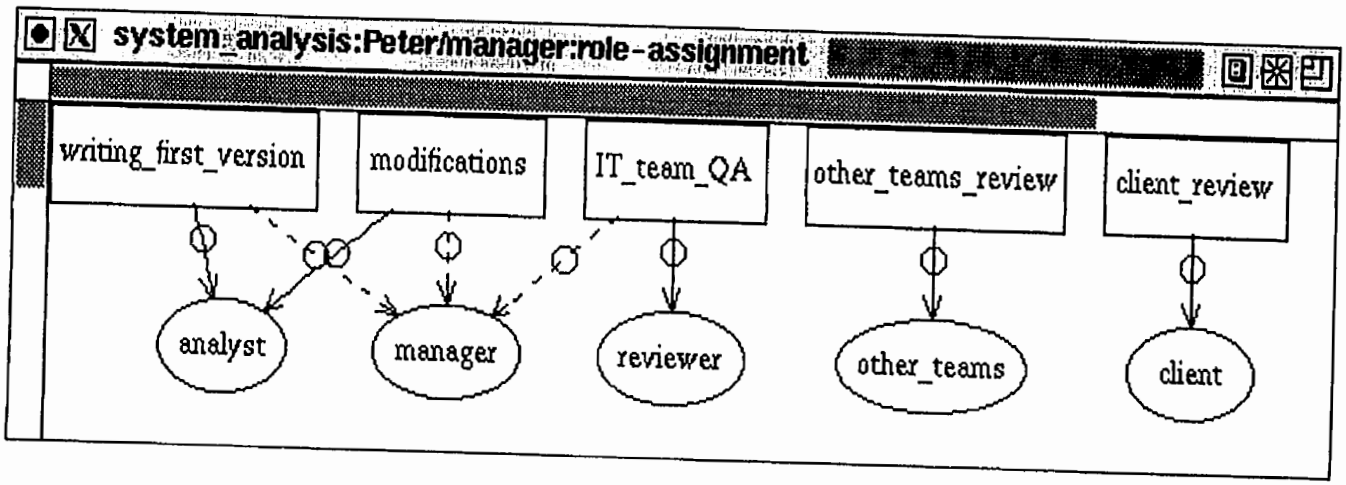


Figure 87 - Peter's role assignment aspect

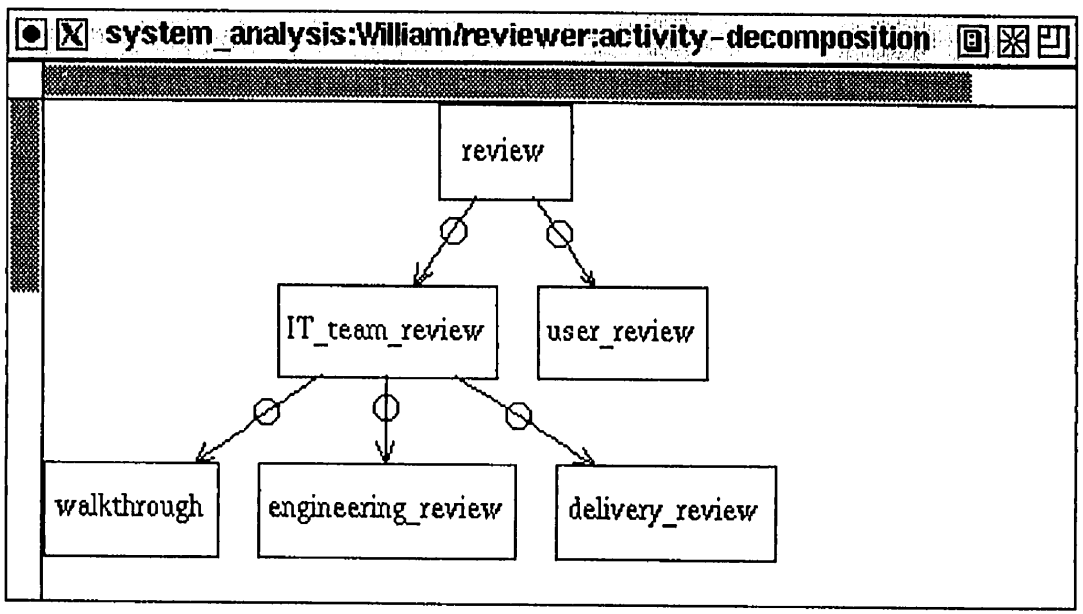


Figure 88 - William's activity decomposition aspect

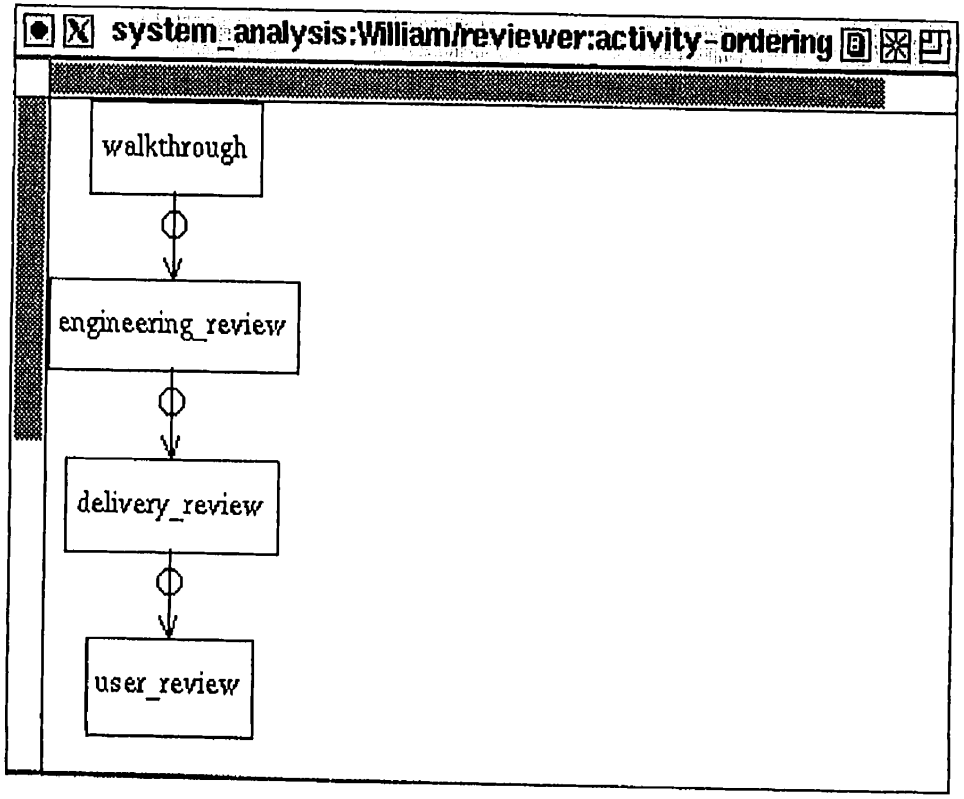


Figure 89 - William's activity ordering aspect

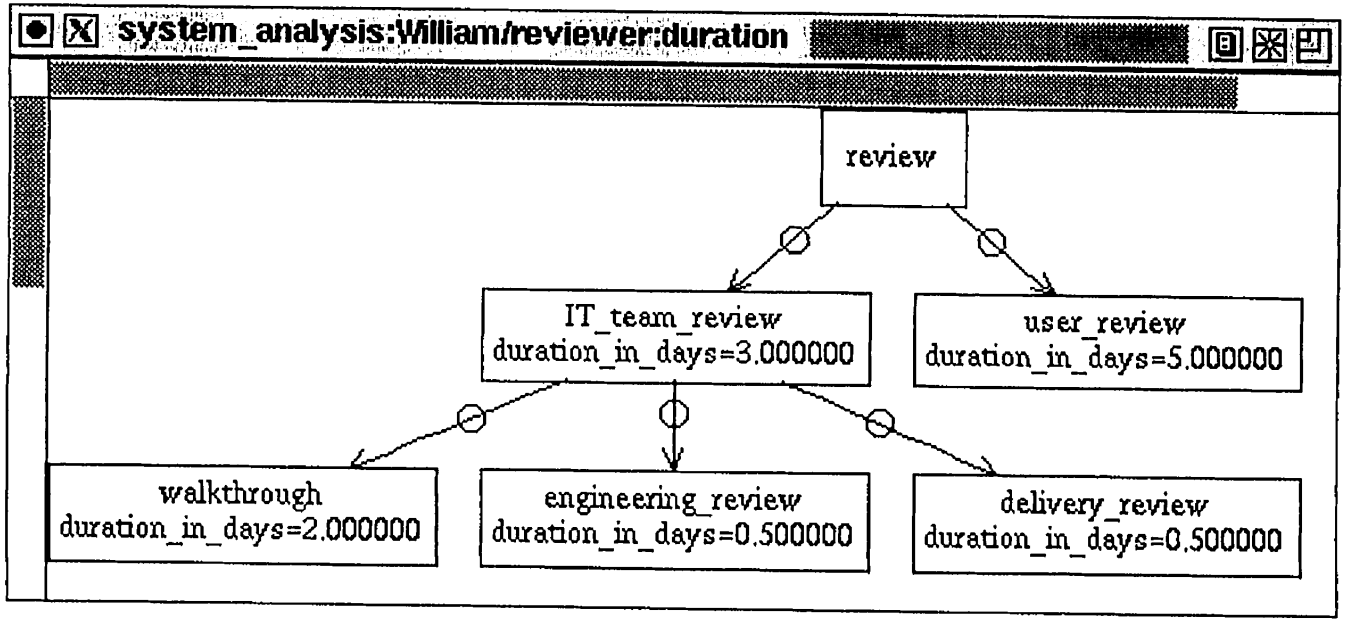


Figure 90 - William's activity duration aspect

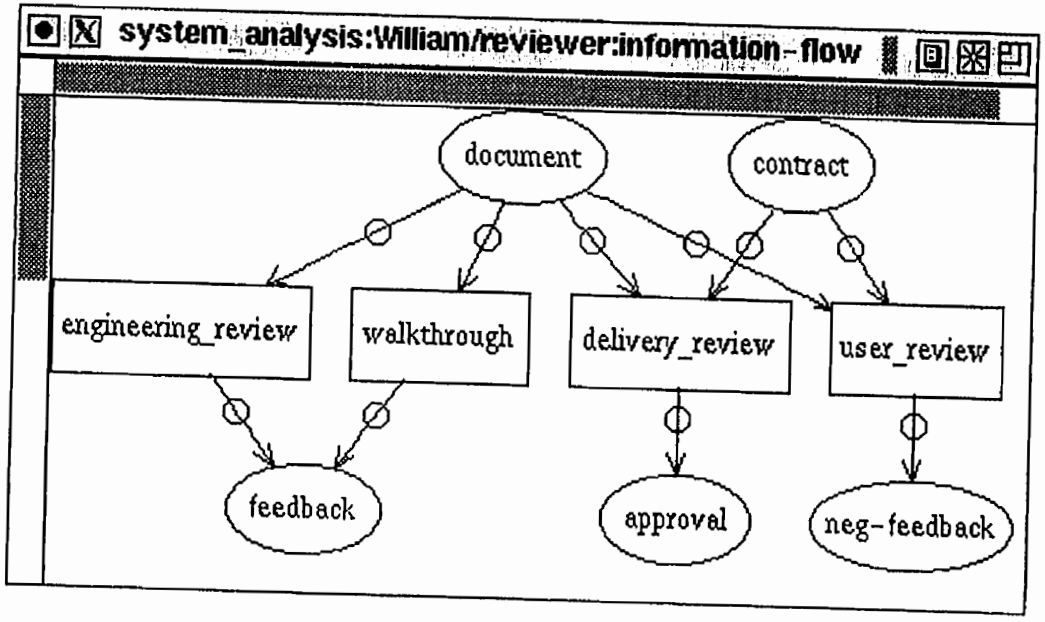


Figure 91 - William's information flow aspect

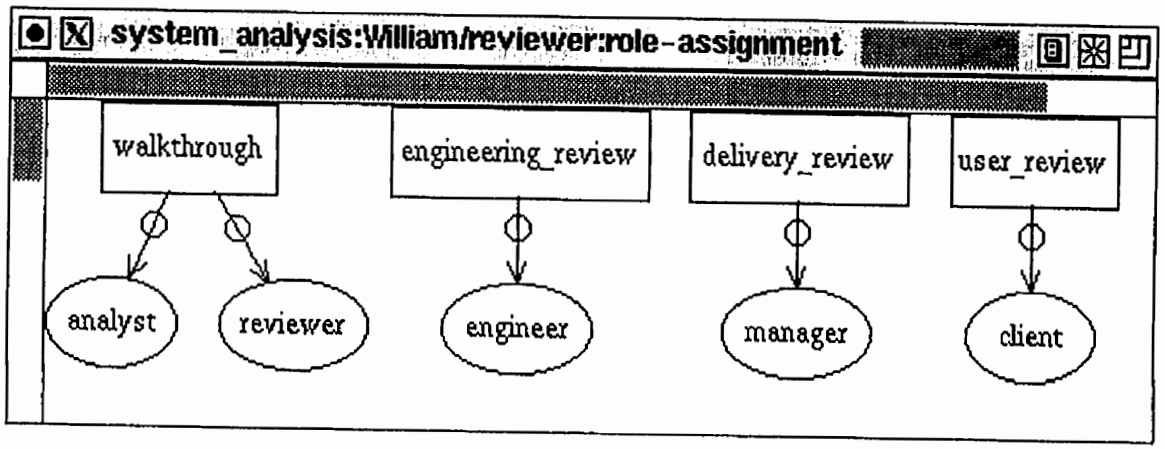


Figure 92 - William's role assignment aspect

Appendix B - Final model after merging the views in Appendix A

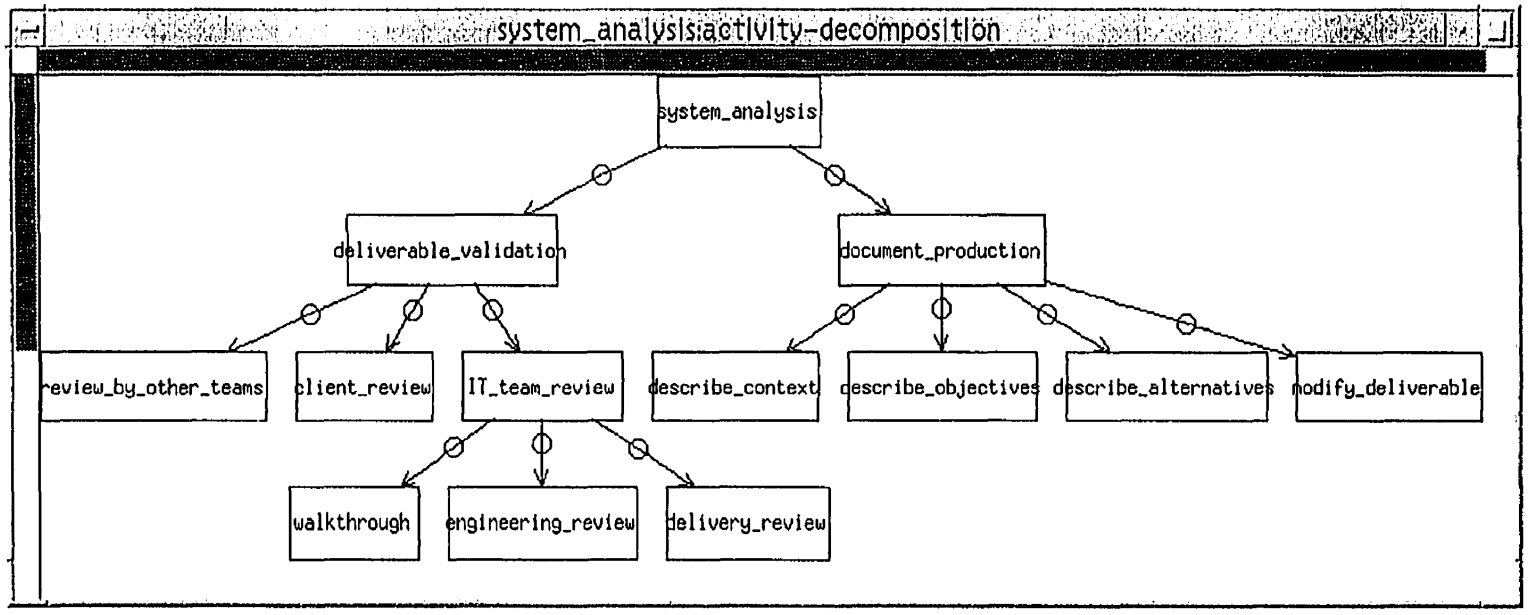


Figure 93 - Activity decomposition aspect of the final model

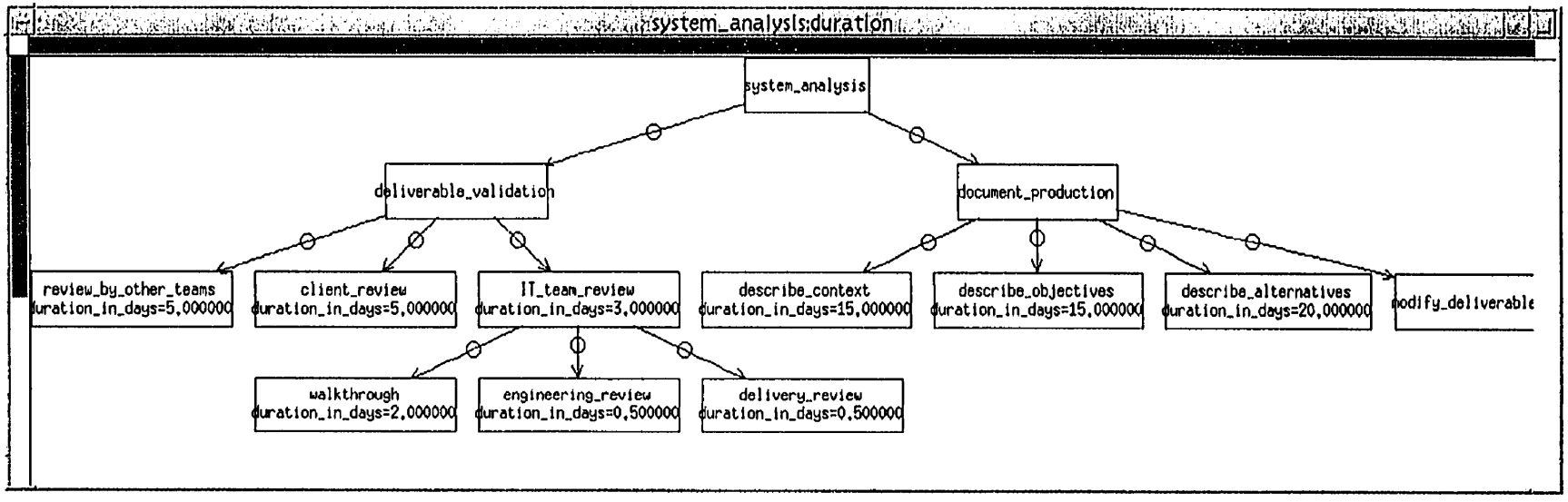


Figure 94 - Activity duration aspect of the final model

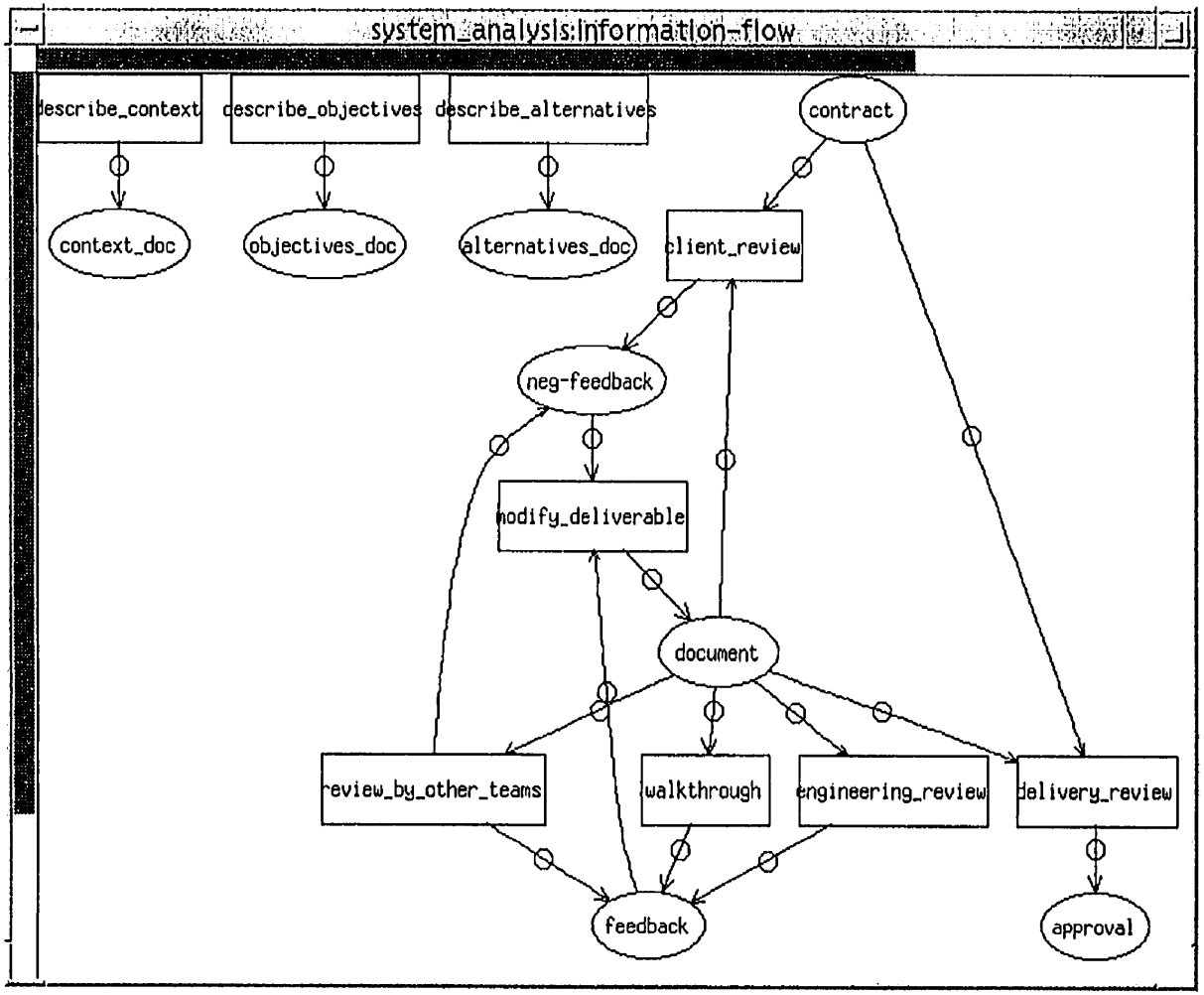


Figure 95 - Information flow aspect of the final model

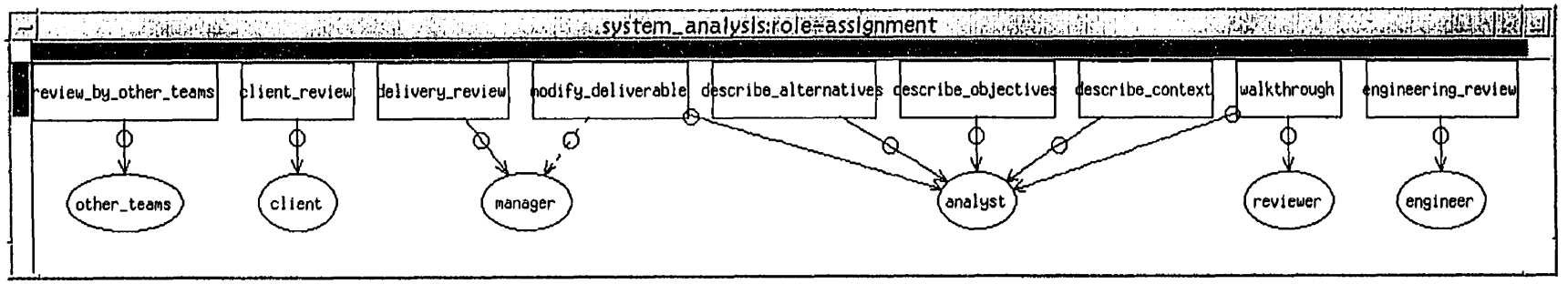


Figure 96 - Role assignment aspect of the final model

Appendix C - Grammar for constraints

```
constraint ::= ForAll( element_in_set, constraint )
             | ThereIs( element_in_set, constraint )
             | condition

element_in_set ::= ( element_variable, list_expression )
                                    // element_variable ∈ list_expression

element_variable ::= entity_variable
                   | relationship_variable

list_expression ::= Set( string )
                                    // use of a set variable (E or R)
                  | SetConstruct( element_in_set, condition )
                                    // { a ∈ A | ... }
                  | TypedEntSet( string, string )
                                    // the first string is the set variable (E) and the second is the entity type
                  | TypedRelSet( string, string, string, string )
                                    // the first string is the set variable (R) and the others make up
                                    // the relationship type
                  | ( list_expression list_expression_operator list_expression )

list_expression_operator ::= Union
                           | Intersection

entity_variable ::= string
                                    // entity variable name

relationship_variable ::= string( entity_variable, entity_variable, string_variable )
                                    // relationship variable name containing entity variables and variable
                                    // for relationship type

string_variable ::= string
                                    // variable for a string

condition ::= True
            | False
            | Not condition
            | ( condition_without_bracket )

condition_without_bracket ::= condition condition_operator condition
                            | num_condition
                            | string_condition
                            | time_condition
                            | list_condition
                            | string_list_condition
                            | char_condition
                            | bool_condition

condition_operator ::= And
                     | Or
                     | Implies
```

```
num_condition ::= num_expression num_condition_operator num_expression

num_expression ::= number
                 | ( num_expression num_expression_operator num_expression )
                 | Round num_expression
                 | Trunc num_expression
                 | Max( string, list_expression )
                                  // maximum of attribute "string" over the given list
                 | Min( string, list_expression )
                 | Sum( string, list_expression )
                 | Mean( string, list_expression )
                 | Sqrt num_expression
                 | Card list_expression
                                  // number of elements of the list
                 | Card str_list_expression
                 | GetInterval( time_expression, time_expression )
                 | GetnValueOf( string, entity_variable )
                                  // get the value of attribute "string" in entity_variable (number)

num_expression_operator ::= +
                          | -
                          | *
                          | /
                          | ^
                          | MOD
                          | DIV

num_condition_operator ::= >
                         | <
                         | <=
                         | >=
                         | ==
                         | !=

char_condition ::= char_expression char_condition_operator char_expression

char_expression ::= `char'
                  | GetcValueOf( string, entity_variable )
                                  // get the value of attribute "string" in entity_variable (character)

char_condition_operator ::= ==
                          | !=

bool_condition ::= ThereIsRel( entity_variable, entity_variable, string_list_expression )
                                  // there is a single relationship between these entities of one of
                                  // the relationship type specified in string_list_expression
                 | ThereIsPath( entity_variable, entity_variable, string_list_expression )
                                  // there is a set of relationships that one can use to go from one entity
                                  // to the other (relationships should be of the types specified
                                  // in string_list_expression)
                 | ThereIsMuliplePaths( entity_variable, entity_variable, string_list_expression )
                                  // there is more than one set ... (see above)
                 | ThereIsDirectedPath( entity_variable, entity_variable, string_list_expression )
                                  // (see above) should go from first entity to second entity
```

```
                | ThereIsMultipleDirectedPaths( entity_variable, entity_variable, string_list_expression )
                | ContainsAtt( string, entity_variable )
                                // check if this entity has a value for attribute "string"
                | IsLeaf( entity_variable )
                                // this entity does not have children (using "is-composed-of"
                                // relationship type)
                | GetbValueOf( string, entity_variable )
                                // get the value of attribute "string"  in entity_variable (boolean)
                | SameEnt( entity_variable, entity_variable )
                                // the two entities have same  name and same type

string_condition ::= string_expression string_expression_operator string_expression

string_expression ::= " string "
                | GetsValueOf( string, entity_variable )
                                // get the value of attribute "string"  in entity_variable (string)
                | GetEntName( entity_variable )
                | GetEntType( entity_variable )
                | GetEntSubType( entity_variable )
                | GetRelType( entity_variable )
                | GetRelDecompBehav( relationship_variable )
                | GetRelTypeKeyword( relationship_variable )

string_expression_operator ::= ==
                            | !=
                            | Contains

time_condition ::= time_expression num_condition_operator time_expression

time_expression ::= Time( string )
                                // time value (yy/mm/dd/mm/ss)
                | GettValueOf( string, entity_variable )
                                // get the value of attribute "string"  in entity_variable (time)

list_condition ::= list_expression list_expression_2
                | element_variableiable IsElementOf list_expression

list_expression_2 ::= list_condition_operator list_expression
                | IsEmpty

list_condition_operator ::= ==
                        | !=
                        | Includes

string_list_condition ::= string_list_expression string_list_expression_2
                | string_variable IsVarElementOf string_list_expression
                                // the value of this variable is in the list
                | string_expression IsValElementOf string_list_expression
                                // the specifies value is in the list

string_list_expression_2 ::= list_condition_operator string_list_expression
                        | IsEmpty

string_list_expression ::= StringSet( string )
                                // string set variable name
```

```
                    | EntType( list_expression )
                            // list of entity types used in the specified list
                    | EntSubType( list_expression )
                    | RelType( list_expression )
                    | RelTypeKeyword( list_expression )
                    | RelDecompBehav( list_expression )
                    | AttName( entity_variable )
                    | AttType( entity_variable )
                    | AttDecompBehav( entity_variable )
                    | ( str_list_expression list_expression_operator str_list_expression )
                    | { "string" enumerated_string_list
                            // enumeration of strings

enumerated_string_list ::= }
                         | , "string" enumerated_string_list
```

Appendix D - State-of-the-art process modeling tools and environments

Adele-Tempo [BEM94]:

Adele was originally a configuration management system, and has then been adapted to a software engineering environment. It can support the modeling of processes (through event-trigger mechanisms) and products (using extended entity-relationship diagrams). When executing the process, each agent uses a Work Environment, showing and controlling the part of the process related to that specific agent. Adele handles data coordination and cooperative work.

APEL [DEA98]:

APEL models are built on top of existing process engines and environments (such as Adele and Process Weaver) that use formalism hard to understand for non process experts. APEL uses a graphical notation for high-level process descriptions, and textual notation for precise details (such as tools used) necessary for the process engine. The static aspects, such as the activities, products, and agents, are modeled in an object-oriented language. The dynamic aspects are specified in control flow, data flow, and state diagrams. A translator is used to generate an executable model (in Adele or Weaver for example) from the high-level descriptions and diagrams.

Articulator [Sca99]:

Articulator is a knowledge-based environment in which software processes can be modeled, analyzed and simulated. A textual modeling notation allows the user to specify objects (resources, agents, and tasks) with their attributes and relationships. Rules are used to specify agent's actions (behavioral information). Two types of simulation are available: knowledge-based simulation (KBS), implemented in Articulator, and discrete-event simulation (DES), using another tool interfacing with Articulator. When KBS is used, the trace ("trajectory") of the simulation is stored, allowing for later queries and analysis, for example going forward or backward

from a specific state. These functionalities can be applied to the entire model or to a subset of a model related to a specific agent.

EPOS [NWC97]:

EPOS is a software process modeling and enactment system. It uses an object-oriented language called SPELL for modeling activities, products, tools, and roles. Pre/post-conditions and code describing the tasks (in a programming language) are stored as attributes of the objects. The model can be instantiated into a task network, that is then executed. Such task network can be modified while being executed. An experience database captures the project history.

Funsoft nets [DeG98]:

Funsoft nets are high-level Petri net notation, extended with elements useful for modeling software processes (e.g., duration of activities, different firing behaviors depending on the number of tokens produced and consumed, etc.). Such process models can be simulated, and validated through the analysis of their static and dynamic properties. The entire approach also includes object models, describing the structure of objects through extended entity-relationship diagrams, and organizational models, showing the organizational entities involved in the process.

JIL / Little-JIL [SuO97, WLM98]:

JIL is an executable process modeling notation similar to a programming language. It has its roots in the modeling language Appl-A. It contains a rich set of constructs for modeling control-flow and coordination. Little-JIL is a higher-level graphical modeling language, that is mapped to the JIL language for execution. Again, the focus is on activity coordination, and it assumes that the agent knows how to perform the different activities (so do not need a description of such activities).

Marvel / Oz [BeK98]:

Marvel is a software development environment using a client-server architecture. A rule-based process modeling language is used to specify tasks, as well as their parameters, preconditions, tools to be used, and effects of their completion (post-conditions). Forward and backward chaining on those rules is used to enforce and automate the process.

The Oz environment is based on similar ideas, but it can support multi-site development. It manages the connection between multiple autonomous and geographically distributed processes. Multiple servers are used, having their own process model and tools. Each server can open connections to remote servers on demand, allowing for coordination across development sites.

Merlin [ScW95]:

Merlin is a process-centered software development environment. Each developer performs his/her work through a working context specifying information on activities, states and documents available. The process descriptions (documents, roles, and activities) and the information on the instantiated process are specified using facts in a PROLOG-like language. The behavioral information is specified in preconditions. The working contexts show the activities that can be executed by that person (i.e., having all preconditions met). Specific rules can also be specified for transactions, indicating how to resolve coordination conflicts such as concurrent access to a document.

MVP-E [BHM97]:

MVP-E is an environment integrating multiple tools used for software process modeling, simulation, and execution. The modeling language used is MVP-L, a formal (textual) notation that is used to describe activities, products, resources, and their attributes. Such attributes can be used in an interface to measurement tools, allowing automatic and manual data collection. Entry and exit criteria are used to model the control flow among activities. Because of the difficulty to view and

understand a model from a textual description, a graphical editor (GEM) has been added. The structural aspects of the models can be analyzed, and consistency can be checked. Functionalities are being added to support view-based modeling, where multiple views would be elicited independently and merged: a similarity analysis function to help identifying the common elements across views, and a tentative set of consistency rules to detect inconsistencies between two views.

OPSIS [ACF96]:

OPSIS is a view mechanism that permits one to extract or merge views from models specified in a Petri-Net type of notation (e.g., Process Weaver). It contains a formal notation and operators for the user to specify how a view should be extracted from a model, and how multiple views should be recombined (possibly after modifying the views). The interface between the views must be specified.

Process Weaver [Fer93]:

Process Weaver is a software development environment providing active process support and process automation. Communication with the developers is done through agendas. The modeling language used has 3 levels: method, cooperative procedures, and work context levels. At the method level, the hierarchy of activities is specified in a graph, and forms are used to capture additional information such as input/output and roles. The control flow information is specified at the cooperative procedures level, using transition nets (Petri nets augmented with preconditions and actions). The information the developer gets (i.e., documents and tools to be used for a task) is modeled at the work context level.

ProcessWise Integrator / ProcessWeb [BGR94, GrW96]:

ProcessWise Integrator is an environment executing a process model, and providing information to the different roles via agendas. The modeling language used is an object-oriented one, consisting of four main types of objects: roles, actions

(activities), entities (artifacts), and interactions. Modifications of the process model can be made while the process is executing.

The user interface of ProcessWise has been moved to WWW (using the Common Gateway Interface) in a new tool called ProcessWeb.

PFV [DPV97]:

PFV (or Process Flowchart Visualization) is a set of tools for modeling software processes in a textual notation, and then visualize them in a flowchart (generated in the graph drawing program "Dot"). It is based on their initial "Interact/Intermediate" tool. The modeling notation includes features for specifying activities and their input/output, decision points, roles, resources, and policies (including pre- and post-conditions). It is also possible to specify additional types of information (e.g., groups and persons). In the graphs generated, colors can be specified for different types of information, making the visualization and understanding easier. Process analysis functions are also provided, including verification of input/output mismatches, identification of sources and sinks, and a variety of summaries.

SPADE [BNF96]:

SPADE is a process-centered software engineering environment. The process is modeled in the language SLANG, a high-level Petri-Net based formalism. The artifacts are kept and maintained in an object-oriented database. Multiple users are supported over a network. Each user interacts with the process through a set of integrated tools.

Statemate [KeH89]:

Statemate is a process modeling and simulation tool. It was originally developed for specifying and designing real-time reactive systems, but its functionalities could be applied to software processes as well. Three perspectives can be modeled in Statemate: functional (activities and information flow), behavioral (through statecharts), and organizational (representing agents and communication). Static

analysis permits the modeler to check the model for consistency, completeness, and correctness. Deadlocks, race conditions, and behavioral ambiguities can also be detected through simulation.

X-elicit [MHH94]:

X-elicit is a front-end elicitation tool, used when gathering software process information. It helps in structuring this information before entering it in another modeling tool such as Statemate (for graphical visualization and analysis). Templates are provided for entering (textual) information in attributes. For example, the template for an activity has attributes "Goal", "Artifact-Input", "Artifact-Output", "Entry-criteria", "Exit-criteria", etc. The type of information to be entered is fully user-definable.

Appendix E – External validity constraints specified

As part of the validation of V-elicit (see Section 7.2.5), we have formally specified development policies from the following book:

Davis, "201 principles of software development", McGraw Hill, 1995.

Here are the 35 constraints specified, from different development phases.

General (8)

#1 - Quality is #1 (i.e., process should include SQA activities)

ThereIs((e,TypedEntSet(E, activity)), GetEntSubType(e)=="SQA")

#5 - Don't try to retrofit quality (i.e., link between development activities and SQA activities should appear at each stage, starting at requirement engineering phase)

For a given stage:

ThereIs((r(e1,e2,t),TypedRelSet(R, activity, verifies, activity)),
(GetEntSubType(e1) == "SQA") and
(GetsValueOf(phase,e2) == "requirement engineering"))

#8 - Communicate with customer/user

ThereIs((r(e1,e2,t),R), (GetEntName(e1) == "customer") and
(GetEntType(e2) == "role"))

#18 - Should develop a short user's manual (e.g., less than 50 pages)

ThereIs((e,TypedEntSet(E, artifact)),
(GetEntName(e)=="user manual") and (GetiValueOf(nbpages,e) < 50))

#23 - Use tools, but be realistic

ThereIs((r(e1,e2,t),TypedRelSet(R, activity, uses, tool)), true)

#32 - Use document standards

ThereIs((r(e1,e2,t),TypedRelSet(R, activity, uses, artifact)),
GetEntName(e2) == "document standard")

#33 - Every document needs a glossary

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "document")),
ThereIs((d2,SetConstruct((e2,TypedEntSet(E,artifact)),
        GetEntSubType(e2) == "glossary")),
ThereIsRel(d1,d2,"artifact contains artifact")))
```

#34 - Every software needs an index

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "software")),
ThereIs((d2,SetConstruct((e2,TypedEntSet(E,artifact)),
        GetEntSubType(e2) == "index")),
ThereIsRel(d1,d2,"artifact contains artifact")))
```

Requirement engineering (7)

#39 - Determine problem before writing requirements

```
ThereIs((a1,SetConstruct((e1,TypedEntSet(E,activity)),
        GetEntName(e1) == "determine problems")),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "write requirements")),
GettValueOf(end_time,a1) < GettValueOf(start_time,a2)))
```

#41 - Fix requirement specification errors now (i.e., make the modifications immediately after finding errors - within one hour)

```
ThereIs((a1,SetConstruct((e1,TypedEntSet(E,activity)),
        GetEntName(e1) == "find requirement error")),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "fix requirement error")),
GetInterval(GettValueOf(end_time,a1), GettValueOf(start_time,a2)) < 60))
```

#43 - Record why requirements were included

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "requirement")),
ThereIs((d2,SetConstruct((e2,TypedEntSet(E,artifact)),
        GetEntSubType(e2) == "rationale")),
ThereIsRel(d1,d2,"artifact stated-for artifact")))
```

#45 - Review requirements

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "requirement")),
ThereIs((a1,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "review")),
ThereIsRel(d1,a1,"artifact is-validated-by activity")))
```

#48 - Use multiple views of requirements

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="gather requirements") and
(Card SetConstruct((e2,TypedEntSet(E,role)),
        (GetEntSubType(e2) == "user") and
        (ThereIsRel(e2,e1,"role communicates-with activity")))
> 1))
```

#50 - Prioritize requirements

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "requirement")),
ContainsAtt(priority,d1))
```

#52 - Separately number every requirements

```
ForAll((d1,SetConstruct((e1,TypedEntSet(E,artifact)),
        GetEntSubType(e1) == "requirement")),
ContainsAtt(number,d1))
```

Design (6)

#62 - Trace design to requirements

```
ForAll((e,TypedEntSet(E, module)),
ThereIs((r(e1,e2,t),TypedRelSet(R, module, comes-from, requirement)),
SameEnt(e,e1)))
```

#63 - Evaluate alternatives

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="evaluate alternative") and
(GetsValueOf(phase,e) == "design"))
```

#64 - Design without documentation is not design (i.e., should have a design document)

```
ThereIs((e,TypedEntSet(E, artifact)),
GetEntName(e)=="design document")
```

#66 - Don't re-invent the wheel (i.e., need an activity to evaluate opportunities to reuse)

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="assess reusability") and
(GetsValueOf(phase,e) == "design"))
```

#68 - Avoid numerous special cases (e.g., no more than 10)

```
ForAll((e,TypedEntSet(E, module)),
Card SetConstruct((e2,TypedEntSet(E,alternative)),
        ThereIsRel(e,e2,"module includes alternative"))
<= 10)
```

#79 - Use efficient algorithms (i.e., should have an activity that analyses the efficiency)

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="analyse efficiency") and
(GetsValueOf(phase,e) == "design"))
```

Coding (7)

#88 - Avoid global variables

```
ForAll((d1,TypedEntSet(E,variable)),
GetEntSubType(d1) != "global")
```

#90 - Avoid side-effects (i.e., use only local variables and parameters)

```
ForAll((r(f1,d1,t),TypedRelSet(R, function, uses, data)),
ThereIs((d2,SetConstruct((e2,TypedEntSet(E,module)),
        ThereIsRel(e2,f1,"module contains function"))),
(ThereIsRel(f1,d1,"function has-parameter data")) or
(ThereIsRel(d2,d1,"module contains data"))))
```

#93 - Use optimal data structures (i.e., need an activity to analyze them)

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="analyze data structures") and
(GetsValueOf(phase,e) == "coding"))
```

#96 - Document before you start coding

```
ThereIs((a1,SetConstruct((e1,TypedEntSet(E,activity)),
        GetEntName(e1) == "documenting")),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "coding")),
GettValueOf(end_time,a1) < GettValueOf(start_time,a2)))
```

#97 - Hand-execute every component

```
ForAll((e,TypedEntSet(E, module)),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "hand-execute")),
ThereIsRel(e,a2,"module used-by activity")))
```

#98 - Inspect code

```
ForAll((e,TypedEntSet(E, module)),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "inspect code")),
ThereIsRel(e,a2,"module used-by activity")))
```

#101 - Don't nest too deep (e.g., no more than 3 levels)

```
ForAll((d1,TypedEntSet(E,code-component)),
GetiValueOf(nesting-level,d1) <= 3)
```

Testing (7)

#107 - Trace tests to requirements

```
ForAll((e,TypedEntSet(E, test)),
ThereIs((r(e1,e2,t),TypedRelSet(R, test, comes-from, requirement)),
SameEnt(e,e1)))
```

#108 - Plan tests long before it is time to test

```
ThereIs((a1,SetConstruct((e1,TypedEntSet(E,activity)),
        GetEntName(e1) == "plan test")),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "testing")),
GettValueOf(end_time,a1) < GettValueOf(start_time,a2)))
```

#109 - Don't test your own software

```
ForAll((r(e1,e2,t),TypedRelSet(R, role, validates, artifact)),
not ThereIsRel(e2,e1,"artifact is-developed-by role")))
```

#110 - Don't write your own test plans

```
ForAll((r(e1,e2,t),TypedRelSet(R, role, develops, test-plan)),
not ThereIsRel(e2,e1,"test-plan is-used-by role")))
```

#115 - Use black-box and white-box testing

```
ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="white-box testing") and
(GetsValueOf(phase,e) == "testing"))

ThereIs((e,TypedEntSet(E, activity)),
(GetEntName(e)=="black-box testing") and
(GetsValueOf(phase,e) == "testing"))
```

#123 - Don't integrate before unit test

```
ThereIs((a1,SetConstruct((e1,TypedEntSet(E,activity)),
        GetEntName(e1) == "unit test")),
ThereIs((a2,SetConstruct((e2,TypedEntSet(E,activity)),
        GetEntName(e2) == "integrate")),
GettValueOf(end_time,a1) < GettValueOf(start_time,a2)))
```

#125 - Analyze causes for errors

```
ForAll((e,TypedEntSet(E, error)),
ThereIs((r(e1,e2,t),TypedRelSet(R, activity, analyse-cause-of, error)),
SameEnt(e,e2)))
```