Brinkkemper, J. and Solvberg, A. (eds.),
*Information Systems Engineering:*
*State of the Art and Research Themes,*
Lecture Notes in Computer Science,
Springer-Verlag June 2000

# Tropos: A Framework for Requirements-Driven Software Development

**John Mylopoulos**[1]
**University of Toronto**

**Jaelson Castro**[2]
**Federal University of Pernambuco**

**Abstract.** Traditionally, software development techniques have been implementation-driven in the sense that the programming paradigm of the day dictated the design and requirements analysis techniques used. For example, structured programming led to structured analysis and design techniques in the '70s. More recently, object-oriented programming gave rise to object-oriented analysis and design. In this chapter we explore a software development methodology which is *requirements-driven* in the sense that the concepts used to define requirements for a software system are also used later on during design and implementation. Our proposal adopts Eric Yu's *i** framework [1], a modeling framework for early requirements, based on the notions of `actor` and `goal`. We use these notions as a foundation to model late requirements, as well as architectural and detailed design. The proposed framework, named Tropos, seems to complement nicely current proposals for agent-oriented programming platforms.

**Keywords:** software development, software requirements analysis and design, agent-oriented systems, software architectures.

## 1. Introduction

Software development techniques have traditionally been inspired and driven by the programming paradigm of the day. This means that the concepts, methods and tools used during all phases of development were based on those offered by the pre-eminent programming paradigm. So, during the era of structured programming, structured analysis and design techniques were proposed [2, 3], while object-oriented programming has given rise more recently to object-oriented design and analysis [4, 5]. For structured development techniques this meant that throughout software development, the developer can conceptualize her software system in terms of functions and processes, inputs and outputs. For object-oriented development, on the other hand, the developer thinks throughout in terms of objects, classes, methods, inheritance and the like.

[1] Author's full address: Department of Computer Science, University of Toronto, Toronto, Canada; email address: jm@cs.toronto.edu
[2] Author's full address: Centro de Informatica, Universidade Federal de Pernambuco, Recife PE, Brazil; email address: jbc@di.ufpe.br. Partially supported by CNPq grant 203262/86-7.

Using the same concepts to align requirements analysis with software design and implementation makes perfect sense. For one thing, such an alignment reduces impedance mismatches between different development phases. Think what it would be like to take the output of a structured analysis task, consisting of data flow and entity-relationship diagrams, and try to produce out of it an object-oriented design! Moreover, such alignment can lead to coherent toolsets and techniques for developing software (and it has!). As well, it can streamline the development process itself.

But, why base such an alignment on implementation concepts? Requirements analysis is arguably the most important stage of software development. This is the phase where technical considerations have to be balanced against social and personal ones. Not surprisingly, this is also the phase where the most and costliest errors are introduced to a software system. Even if (or rather, when) the importance of design and implementation phases wanes sometime in the future -- thanks to COTS, software reuse and the like -- requirements analysis will remain a critical phase for the development of any software system, answering the most fundamental of all design questions: "what is the system intended for?"

This paper speculates on the nature of a software development framework, named Tropos[3], which is requirements-driven in the sense that it is based on concepts used during early requirements analysis. To this end, we adopt the concepts offered by *i** [1], a modeling framework offering concepts such as `actor`, `agent`, `position` and `role`, as well as social `dependencies` among actors, including `goal`, `softgoal`, `task` and `resource` ones. These concepts are used in a small example to model not just early requirements for an insurance claim management system, but also late requirements, architectural design and detailed design.

The proposed methodology spans four phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an existing organizational setting; the output of this phase is a an organizational model which includes relevant actors and their respective goals;
- Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities;
- Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows;
- Detailed design, where each architectural component is defined in further detail in terms of inputs, outputs, control, and other relevant information.

Section 2 introduces the primitive concepts offered by *i** and illustrates their use with an example. Sections 3, 4, and 5 sketch how the technique might work for late requirements, architectural design and detailed design respectively. Throughout, we assume that the task at hand is to build generic software to support back office claims processing within an insurance company. Finally, section 6 summarizes the contributions of the paper, offers an initial self assessment of the proposed development technique, and outlines directions for further research.

## 2. Early Requirements Analysis with *i**

During early requirements analysis, the requirements engineer is supposed to capture and analyze the

[3] The name "Tropos" is derived from the Greek "tropé", which means "easily changeable", also "easily adaptable."

intentions of stakeholders. These are modelled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non-functional requirements of the system-to-be [6]. In *i** (which stands for ``distributed intentionality''), early requirements are assumed to involve social actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The *i** framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor has about its relationships with other actors. These models have been formalized using intentional concepts such as goal, belief, ability, and commitment (e.g., [7]). The framework has been presented in detail in [1] and has been related to different application areas, including requirements engineering [8], business process reengineering [9], and software processes [10].

A strategic dependency model is a graph, where each node represents an *actor*, and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goal. We call the depending actor the *depender* and the actor who is depended upon the *dependee*. The object around which the dependency centers is called the *dependum*. By depending on another actor for a dependum, an actor is able to achieve goals that it is otherwise unable to achieve, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Figure 1 shows the beginning of an *i** model consisting of two relevant actors for an automobile insurance example. The two actors are named respectively `Customer` and `Insurance Company`. The customer has one relevant goal `CarRepaired`, while the insurance company has goals `Settle claim`, `Maximize profits`, and keep `Happy customer`. Since the last two goals are not well-defined, they are represented in terms of softgoals (shown as cloudy shapes.)

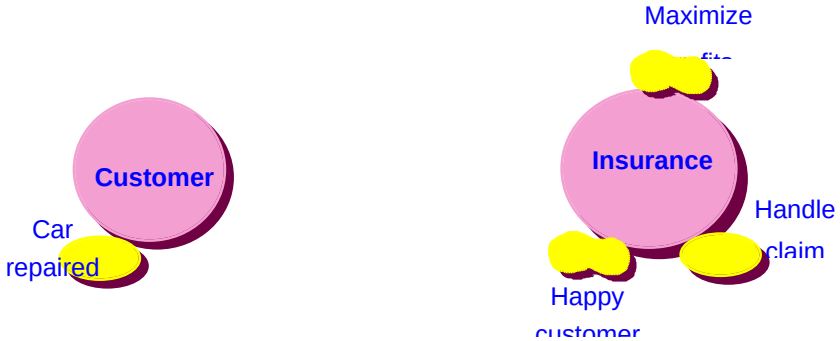


**Figure 1**: "Customers want their cars repaired, while the insurance company wants to maximize profits, settle claims and keep customers happy"

Once the relevant stakeholders and their goals have been identified, a means-ends analysis determines how these goals (including softgoals) can actually be fulfilled through the contributions of other actors. Let's focus on one such goal, namely `Handle claim`.

As shown in figure 2, the analysis is carried out from the perspective of the insurance company, who had the goal in the first place. It begins with the goal `Handle claim` and postulates a task `Handle claim` (represented in terms of a hexagonal icon) through which the goal might be fulfilled. Tasks are partially ordered sequences of steps intended to fulfill some goal. The task we have selected is decomposed into sub-

tasks `Verify policy`, `Prepare offer`, `Finalize deal` which *together* can complete the handling of a claim. It should be noted that the same goal (`Handle claim`) might have several alternative tasks that can fulfill it. Likewise, there may be several alternative decompositions of a task into sub-tasks. Figure 2 only shows one set of decompositions which collectively can fulfill the root goal.

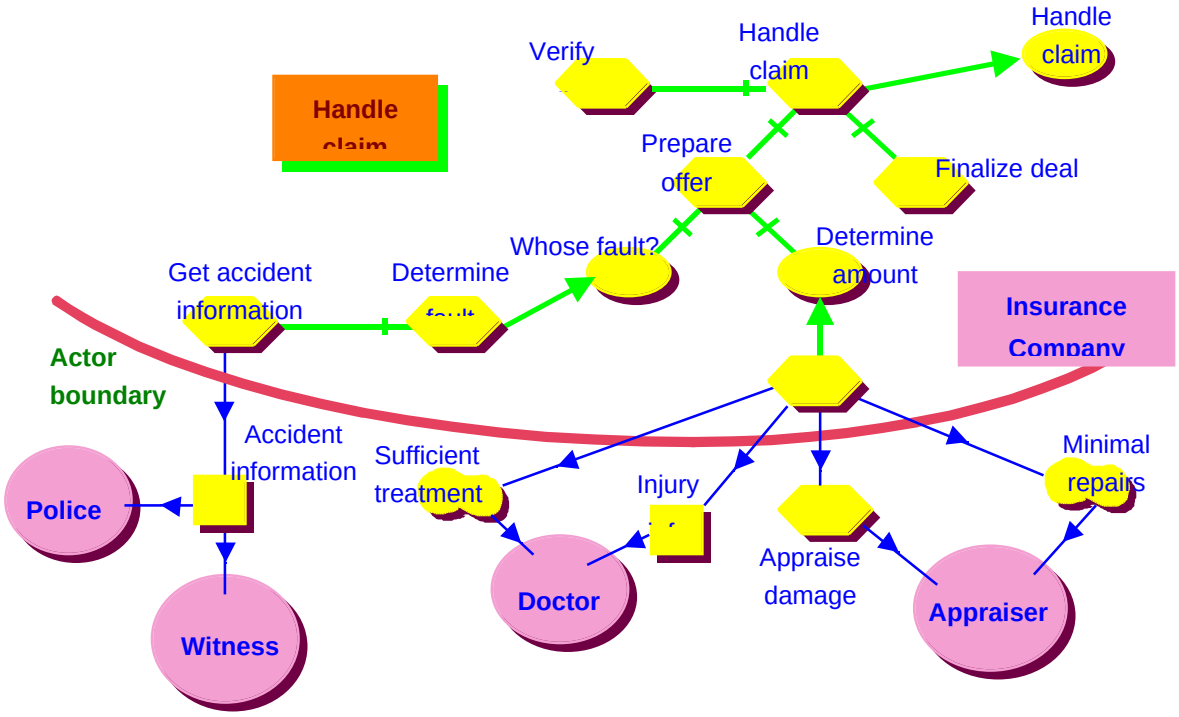


**Figure 2**: Means-ends analysis for the goal `Handle claim`.

Tasks can also be decomposed into goals, whose fulfillment accomplishes the task. For example, the task `Prepare offer` is decomposed into goals `Whose fault?` and `Determine amount`. Representing a task component as a goal means that there might be several possible ways of accomplishing that component.

Decompositions continue until the analysis can identify an actor who can fulfill a goal, carry out a task, or deliver on some needed resource. Such dependencies for the `Handle claim` goal include:

- Resource dependencies on actors `Police` and `Witness`, who are expected to deliver accident information;
- Another resource dependency on `Doctor` for injury information;
- Task dependency on `Appraiser`, who is expected to carry out the standard appraisal task;
- Softgoal dependencies on `Doctor`, who must make sure that the patient receives adequate treatment, and the `Appraiser`, who is expected to minimize the amount of the appraisal.

The result of such means-ends analyses for the initial goals leads to the strategic dependency model mentioned earlier. Fragments of such a model for the insurance claim example are shown in figure 3.

According to this model, the customer depends on the appraiser for a fair appraisal. However, the appraiser

can be expected to act in the interests of the insurance company because of his dependence on the latter for continued employment. The customer, in turn, depends on the body shop to give a maximal estimate, while the body shop depends on the customer for continuing business.

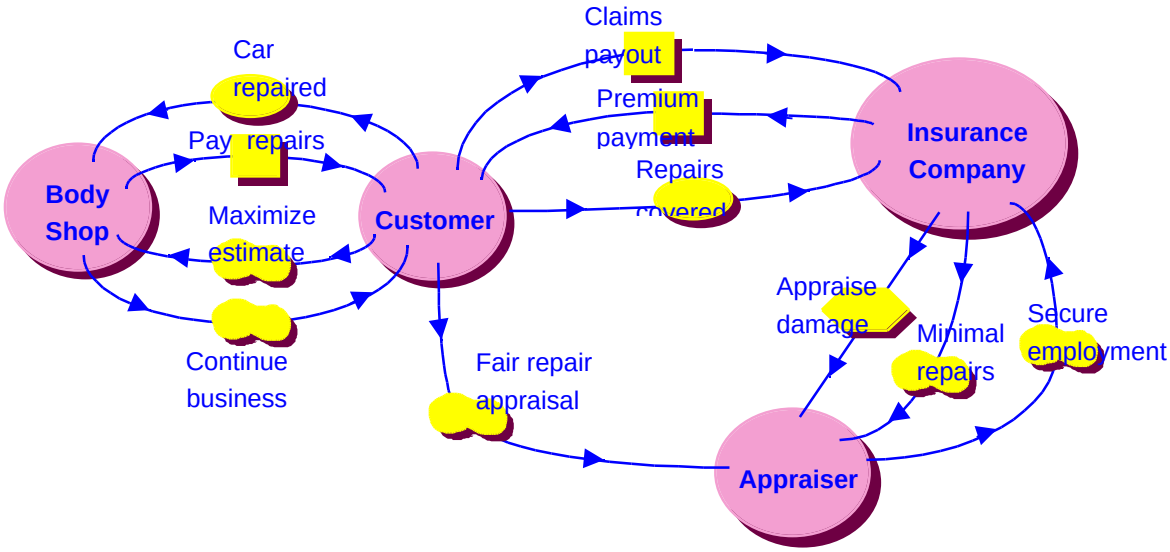


**Figure 3**: Partial strategic dependency model for the handling of insurance claims.

Although a strategic dependency model provides hints about why processes are structured in a certain way, it does not sufficiently support the process of suggesting, exploring, and evaluating alternative solutions. That is the role of the Strategic Rationale model. A strategic rationale model is a graph with four main types of nodes -- goal, task, resource, and softgoal -- and two main types of links -- means-ends links and process decomposition links. A strategic rationale graph describes the criteria in terms of which each actor's selects among alternative dependency configurations.

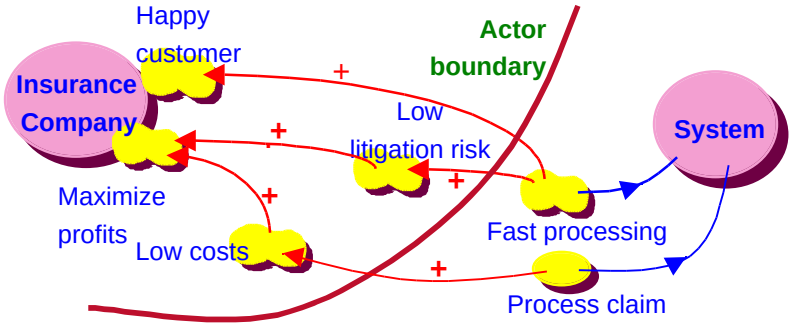


**Figure 4:** The insurance company depends on the system for fast processing of insurance claims

### 3. Late Requirements Analysis

Late requirements analysis results in a requirements specification document which describes all functional and non-functional requirements for the system-to-be. In Tropos, the system is represented as one or more actors which participate in a strategic dependency model, along with other actors in the system's

operational environment. In other words, the system comes into the picture as one or more actors which contribute to the fulfillment of stakeholder goals. For example, the system may be introduced in the strategic dependency model in order to support the goal `Process claim`, as well as the softgoal `Fast processing` of insurance claims, which contributes positively to both the `Maximize profits` and `Happy customer` softgoals (figure 4). Of course, as late requirements analysis proceeds, the system is given additional responsibilities, and ends up as the depender of several dependencies. Moreover, the system is decomposed into several sub-actors which take on some of these responsibilities. To obtain this decomposition, `Process claim` is first reduced into subgoals, such as `Select process` (i.e., what

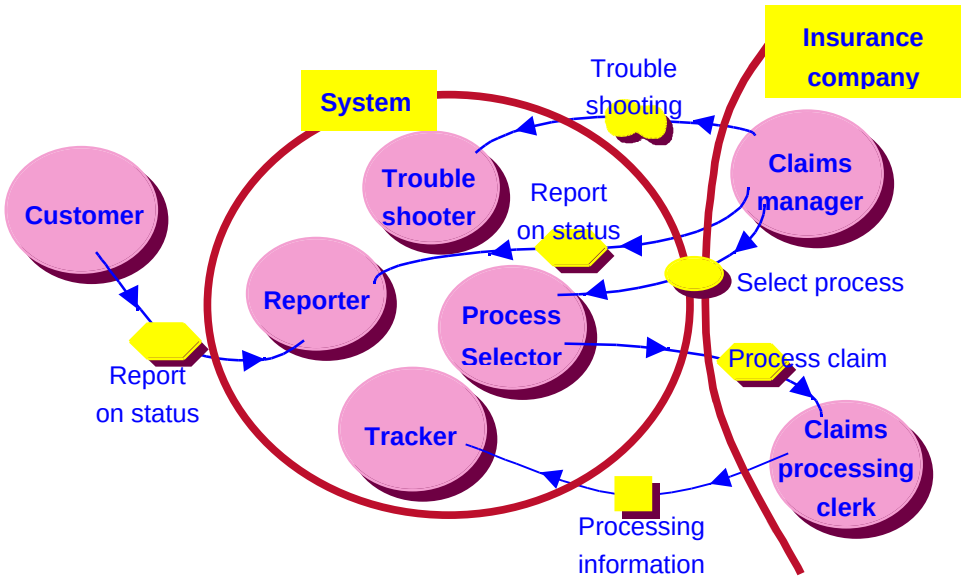


**Figure 5**: The system consists of four actors, each with external dependencies.

sequence of steps will be used to process the claim), `Process claim` and `Report status`, using the kind of means-ends analysis illustrated in figure 2, along with a strategic rationale analysis. The result of this analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been generated. Figure 5 suggests one possible assignment of responsibilities. In particular, `Process Selector` decides what kind of processing will be done for a given claim, and relies on a clerk to carry out this process. We assume that different insurance companies using the software may be processing various types of claims (e.g., large vs small) differently. `Tracker` keeps track of the status of claim and needs information from the processing clerk in order to do so. `Reporter` reports to the claims manager, or the customer on the status of a claim following a given script (hence the task dependency), while `Trouble shooter` is looking for signs of problems ahead. `Tracker` and the `Trouble shooter` are introduced in order to contribute to the fulfillment of the `Fast processing` softgoal.

Resource, task and softgoal dependencies correspond naturally to functional and non-functional requirements. Leaving (some) goal dependencies between system actors and other actors is a novelty. Traditionally, functional goals are “operationalized” during late requirements [6], while quality softgoals are either operationalized or “metricized” [11]. For example, `Fast processing` may be operationalized during late requirements analysis into particular business processes for processing claims. Likewise, a security softgoal might be operationalized by defining interfaces which minimize input/output between the

system and its environment, or by limiting access to sensitive information. Alternatively, the security requirement may be metricized into something like "No more than X unauthorized operations in the system-to-be per year".

Leaving goal dependencies with system actors as dependees makes sense whenever there is a foreseeable need for flexibility in the performance of a task on the part of the system. For example, consider a communication goal "communicate X to Y". According to conventional software development techniques, such a goal needs to be operationalized before the end of late requirements analysis, perhaps into some sort of a user interface through which user Y will receive message X from the system. The problem with this approach is that the steps through which this goal is to be fulfilled (along with a host of background assumptions) are frozen into the requirements of the system-to-be. This early translation of goals into concrete plans for their fulfillment makes software systems fragile and less reusable.

For our example, we have left two goals in the late requirements model. The first goal is `Trouble shooting`, because we propose to implement `Trouble shooter` as an intelligent agent who "learns on the job" (…so to speak…) by using machine learning techniques. Also, `Select process`, because we want to include in the system's architecture a number of components which reflect different types of claims processing done in the insurance industry. So, instead of operationalizing this goal during requirements analysis, we propose to do so during architectural design.

## 4. Architectural Design

Architectural design has emerged as a crucial phase of the design process. Initially a software architecture can be considered to be the realization of early design decisions made regarding the decomposition of the system into components. A software architecture constitutes a relatively small, intellectually graspable model of system structure, and how system components work together. Software architects have developed comprehensive catalogues of software architectural styles (see, for example, [12]). Such styles range from *Independent components* (such as *Events-driven architectures* and *communicating processes*), *Call-and-return* (e.g., *object-oriented systems*, *layered*, *main program* and *subroutine* architectures), *Data flow* (for example, *batch sequential*, *pipes-and-filters*), *Data centered* (e.g., *repository* and *blackboard architectures*), as well as *Virtual machine architectures* (*rule-based systems*, and *interpreters*.) Most of these apply to software systems, even when the basic software components are actors rather than subsystems and modules. For instance, a pipe and filter architectural style corresponds to an agent assembly line, while the blackboard style has been used extensively in the agent programming literature.

Architectures are influenced by system designers as well as technical and organizational factors. During architectural design we concentrate on the key system actors, defined during late requirements analysis, and their responsibilities. There would be a set of desired functionality as well as a number of quality requirements related to performance, availability, usability, modifiability, portability, reusability, testability, etc. The functional requirements can be handled by many standard technologies, such as structured analysis and design, or object-oriented design methods. However, quality requirements are generally not addressed by such techniques [13].

Suppose that in addition to the requirements of figure 5, we have an "easily modifiable" requirement for the `Process selector` actor imposed by the `Claims manager actor`, to make sure that it can accommodate ever changing variations on how an insurance claim is processed. Likewise, in order to fulfill

the requirement of good response time, imposed by the claims processing clerk on process selector, a "good performance" softgoal is introduced. To cope with these goals, the software architect, who is another (external) actor, goes through a means ends analysis comparable to what was discussed earlier. In this case, the analysis involves refining the softgoals to sub-goals that are more specific (and more precise!) and then evaluating alternative architectural styles against them, as shown in figure 6.

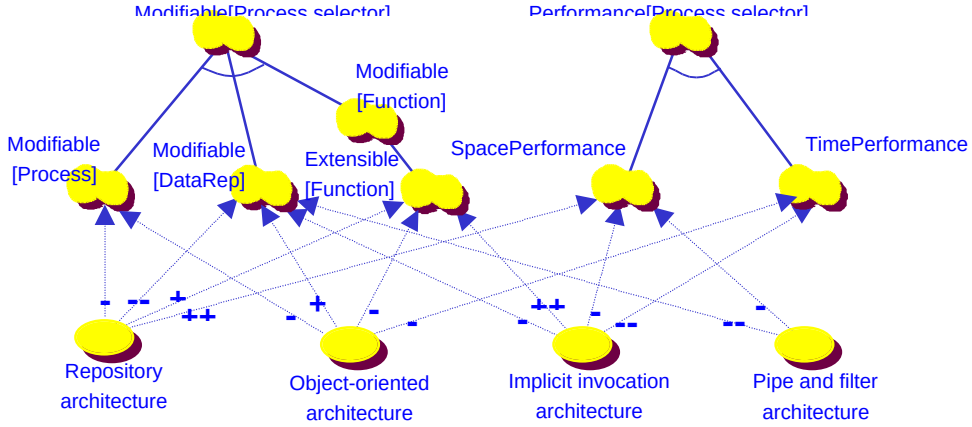


**Figure 6**: A strategic rationale model, from the perspective of the process selector actor.

In the figure, the two softgoals take `Process selector` as argument, meaning that the quality requirements they represent apply specifically to this system component (rather than the whole system). The first of the two softgoals has been AND-decomposed into subgoals `Modifiable[Process]`, `Modifiable[Data representation]`, `Modifiable[Function]`. This analysis is intended to make explicit the space of alternatives for fulfilling the top-level quality softgoals. Moreover, the analysis allows the evaluation of several alternative architectural styles. The styles are represented as goals (saying, roughly, "make the architecture of the new system repository-based/object-oriented/…" ) and are evaluated with respect to the alternative quality softgoals as shown in figure 6. The evaluation results in contribution relationships from the architectural goals to the quality softgoals, labelled "+', "-", "++", etc.

As with late requirements, the interesting feature of the proposed analysis method is that it is goal oriented. Goals are introduced and analyzed during architectural design, and guide the design process.

Apart from goal analysis, this phase involves the introduction of other system actors which will take on some of the responsibilities of the key system actors introduced earlier. For example, to accommodate the responsibilities of the `Reporter` actor of figure 5, we may want to introduce a `Data selector` actor, who selects the data to be presented, a `Transformer` actor, who performs computations that transform the input data to useful information for the `Customer` and the `Claims manager`, and a `Presenter` actor who presents these data in a suitable format. Of course, this analysis is nothing but good old functional decomposition and will not be discussed in any detail here.

An interesting decision that comes up during architectural design is whether fulfillment of an actor's obligations will be accomplished through assistance from other actors, through delegation ("outsourcing"), or through decomposition of the actor into component actors. Going back to the `Reporter` example, the introduction of other actors described in the previous paragraph amounts to a form of delegation. `Reporter` retains its obligations, but delegates subtasks, subgoals etc. to other actors. An alternative

architectural design would have Reporter outsourcing some of its responsibilities to some other actors, so that Reporter removes itself from the critical path of obligation fulfillment. Lastly, Reporter may be refined into an aggregate of actors which, by design, work together to fulfill Reporter's obligations. This is analogous to a committee being refined into a collection of members who collectively fulfill the committee's mandate. It is not clear, at this point, how the three alternatives compare, nor what are their respective strengths and weaknesses.

## 5. Detailed Design

The detailed design phase is intended to introduce additional detail for each architectural component of a software system. In our case, this includes actor communication and actor behaviour. To support this phase, we may be adopting agent communication languages, message transportation mechanisms, ontology communication, agent interaction protocols, etc. from the agent programming community. One possibility, among admittedly many, is adopt one of the extensions to UML [5] proposed by the FIPA (Foundation for Intelligent Agents) and the OMG Agent Work group.

For our example, let's concentrate on the Fast processing goal dependency, which might involve a detailed design on *agent interaction protocols* (AIP). Such a protocol describes a communication pattern among actors as an allowed sequence of messages, as well as constraints on the contents of those messages. To define such a protocol, we use AUML - the Agent Unified Modeling Language [14], which supports templates and packages to represent the protocol as an object, but also in terms of sequence and collaborations diagrams. In AUML inter- and intra-agent dynamics are also described in terms of activity diagrams and state charts.

Figure 7 depicts a protocol expressed as a UML sequence diagram for Select process. When invoked, a Claim manager actor sends a Call-for-Proposal-Process-claim to a Process Selector actor who is willing to participate in processing the claim.

The Process Selector actor can then choose to respond to the Claim manager by a given deadline by submitting a proposal for a suitable Processing clerk actor to deal with the processing (for example an expert on small claims). Alternatively, Process selector may decide to refuse to process the claim or indicate that it does not understand. If a proposal is offered, the Claim manager actor has a choice of either rejecting or accepting the proposal. When Process selector receives a proposal acceptance, it will contact the appropriate Claims process clerk actor and place a request regarding (small) process claims. Based on the returned information, Process selector can inform Claims manager about the proposal's execution. Additionally, the Claim manager actor can cancel the execution of the proposal at any time.

Of course the sequence diagram in Figure 7 only provides a basic specification for an agent claim processing protocol. More processing details are required. For example, a Claims manager actor requests a call for (process claim) proposals (CFP) from a Process selector actor. However, the diagram stipulates neither the procedure used by the Claims manager to produce the CFP request, nor the procedure employed by Process Selector to respond the CFP. Yet, these are clearly important details at this stage of the software development process.

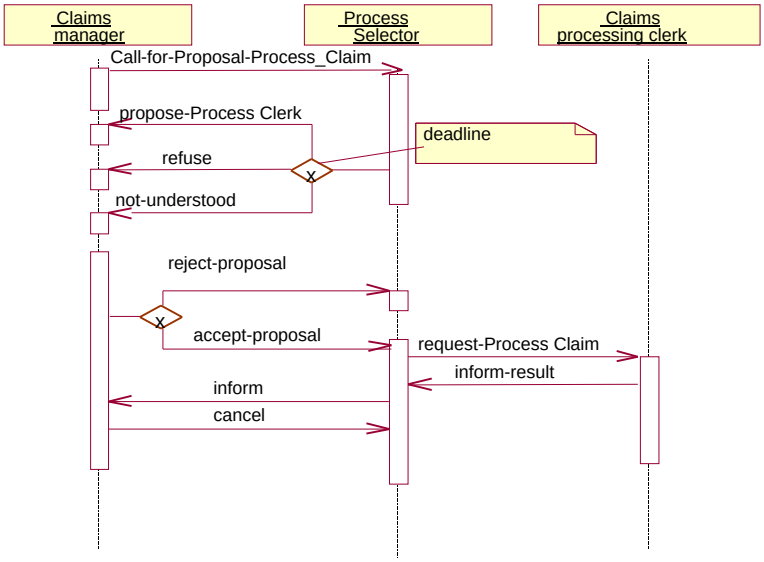


**Figure 7**: An actor interaction protocol for processing claims

Such details can be provided by using *leveling*, i.e., by introducing additional interaction and other diagrams which describe some of the primitive action of the one shown on figure 7. Each additional level can express *intra-actor* or *inter-actor* activity. At the lowest level, specification of an actor protocol requires spelling out the detailed processing that takes place within an actor in order to implement the protocol. Statecharts and activity diagrams can also specify the internal processing of actors who are not aggregates.

## 6. Conclusions and Discussion

We have argued in favour of a software development methodology which is founded on intentional concepts, such as those of `actor`, `goal`, (`goal`, `task`, `resource`, `softgoal`) `dependency`, etc. Our argument rests on the claim that the elimination of goals during late requirements, freezes into the design of a software system a variety of assumptions which may or may not be true in its operational environment. Given the ever-growing demand for generic, component-ized software that can be downloaded and used in a variety of computing platforms around the world, we believe that the use of intentional concepts during late software development phases will become prevalent and should be further researched.

The Tropos project is only beginning and much remains to be done. We will be working towards a modelling framework which views software from four complementary perspectives:

- **Social** -- who are the relevant actors, what do they want? What are their obligations? What are their capabilities?…
- **Intentional** -- what are the relevant goals and how do they interrelate? How are they being met, and by whom?…
- **Process-oriented** -- what are the relevant business/computer processes? Who is responsible for what?…
- **Object-oriented** – what are the relevant objects and classes, along with their inter-relationships?

In this paper, we have focused the discussion on the social and intentional perspectives because they are novel. As hinted earlier, we propose to use UML-type modelling techniques for the others.

Of course, diagrams are not complete, nor formal as software specifications. To address this deficiency, we propose to offer three levels of software specification. The first is strictly diagrammatic, as discussed in this paper. The second involves formal annotations which complement diagrams. For example, annotations may specify that some obligation takes precedence over another. These could be used as a basis for simple forms of analysis. Finally, we propose to include within Tropos a formal specification language for all built-in constructs, to support deeper forms of analysis. Turning to the organization of Tropos models, the concepts of *i** will be embedded in a modeling framework which supports generalization, aggregation, classification and contextualization. Some elements of UML will be adopted as well for modeling the object and process perspectives.

Like other requirements modelling frameworks proposed in the literature, we recognize that diagrams are important for human communication, but are imprecise and offer little support for analysis. Partially formal annotations can help in defining some forms of analysis, and they serve as bridges between informal diagrams and formal specifications. Finally, formal specifications serve as foundation for a formal semantics, as well as a range of analysis techniques, including proofs of correctness, process simulation, goal analysis etc.

Tropos constitutes the last leg of a trilogy on modelling languages. The first language in the trilogy, Taxis [15], was intended as a design language for information systems. Its main novelty was the adoption of semantic network representation techniques to offer a modelling framework which was object-oriented and emphasized taxonomic organization for data, transaction and exception classes. Telos [16] focused on the use of classification to offer meta-modelling facilities where concepts such as `goal`, `activity`, etc. were first defined at the metaclass level before being used at the class level. Telos was intended for software modelling, where one could represent requirements, design, implementation and other information about a software system within a single modelling framework. Tropos is probably the most ambitious undertaking in the trilogy in that it aspires to influence not just the modelling of different types of information about a software system, but also the software development process itself.

Most appropriately, this preview of Tropos has been written on the occasion of Janis Bubenko's sixty-fifth birthday. Janis has made significant research contributions to conceptual modelling, databases, and model-based software development. His early work [17] was an inspiration for our own work on RML [18], and we have benefited by following his research ever since. Just as importantly, Janis has served as role model for younger generations of researchers and academics around the world.

## Acknowledgements

Many colleagues contributed to the ideas that led to this paper. Special thanks to Eric Yu, whose insights helped us focus our research on intentional and social concepts.

The Tropos project includes as co-investigators Eric Yu (University of Toronto) and Yves Lesperance (York University); also Alex Borgida (Rutgers University), Matthias Jarke and Gerhard Lakemeyer (Technical University of Aachen.) The Canadian component of the project is supported in part by the

Natural Sciences and Engineering Research Council (NSERC) of Canada, and the CITO Centre of Excellence, funded by the Province of Ontario.

## References

[1] Yu, E., *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1995.

[2] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.

[3] Yourdon, E. and Constantine, L., Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.

[4] Wirfs-Brock, R., wilkerson, B., Wiener, l., *Designing Object-Oriented Software.* Englewood Cliffs, NJ; Prentice-Hall.

[5] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[6] Dardenne, A., van Lamsweerde, A., and Fickas, S., "Goal–directed Requirements Acquisition," *Science of Computer Programming, 20*, 3-50, 1993.

[7] Cohen, P. and Levesque, H. Intention is Choice with Commitment. *Artificial Intelligence, 32(3).*

[8] Yu, E., "Modeling Organizations for Information Systems Requirements Engineering," *Proceedings First IEEE International Symposium on Requirements Engineering*, San Jose, January 1993, pp. 34-41.

[9] Yu, E., and Mylopoulos, J., "Using Goals, Rules, and Methods to Support Reasoning in Business Process Reengineering", *International Journal of Intelligent Systems in Accounting, Finance and Management 5(1)*, January 1996.

[10] Yu, E. and Mylopoulos, J., "Understanding 'Why' in Software Process Modeling, Analysis and Design," *Proceedings Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[11] Davis, A., Software Requirements: Objects, Functions and States, Prentice Hall, 1993.

[12] Bass, L., Clements. P., Kazman, R., *Software Architecture in Pratice*, SEI Series in Software Engineering, Addison-Wesley, 1998.

[13] Chung, L. K., Nixon, B. A., Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Publishing, 1999.

[14] Parunak, H. Van Dyke, Suater, J., Odell, J., *Engineering Artifacts for Multi-Agents Systems*, ERIM CEC, 1999.

[15] Mylopoulos, J., Bernstein, P., and Wong. H. K. T., "A Language Facility for Designing Data-intensive Applications," *ACM Transactions on Database Systems 5(2)*, 1980.

[16] Mylopoulos, J., Borgida, A., Jarke, M., and M. Koubarakis, M., "Telos: Representing Knowledge About Information Systems," *ACM Transactions on Information Systems*, 1990.

[17] Bubenko, J., "Information Modeling in the Context of System Development," *Proceedings IFIP Congress '80*, 395-411, 1980.

[18]Greenspan, S., Mylopoulos, J., and Borgida, A., "Capturing More World Knowledge in the Requirements Specification," *Proceedings Sixth International Conference on Software Engineering*, Tokyo, 1982.