From Entities and Relationships to Social Actors and Dependencies

John Mylopoulos[1], Ariel Fuxman[1], and Paolo Giorgini[2]

[1] Department of Computer Science, University of Toronto,
6 King's College Road, Toronto, Canada M5S 3H5
{jm, afuxman}@cs.toronto.edu
[2] Department of Computer Science, University of Trento,
Via Sommarive 14, 38050, Povo, Italy
pgiorgini@science.unitn.it

Abstract. Modeling social settings is becoming an increasingly important activity in software development and other conceptual modeling applications. In this paper, we review i* [Yu95], a conceptual model specifically intended for representing social settings. Then, we introduce Tropos, a formal language founded on the primitive concepts of i*, and demonstrate its expressiveness through examples. Finally, we give an overview of a project which uses Tropos to support software development from early requirements analysis to detailed design.

Keywords: conceptual models, semantic data models, entity-relationship model, requirements models, enterprise models, software development methodologies.

1 Introduction

The Entity-Relationship (hereafter E-R) model was proposed by Peter Chen at the first VLDB conference [Che75] as a modeling framework for capturing the meaning of data in a database. Since then, the model has been widely taught and used during structured information system analysis and design. It was also extended to support abstraction mechanisms such as generalization and aggregation [BCN92]. Elements from this extended version can be found in various object-oriented analysis techniques, e.g., OMT [RBP+91], as well as the ever-popular UML [BRJ99]. Unlike many other concepts in Computer Science that enjoyed a short lifespan of practical use, the E-R model has had an enduring impact on software engineering research and practice. Its use spans 25 years, as well as the two dominant software development methodologies of this period (structured and object-oriented software development). There are good reasons for this longevity. The world of software applications has revolved around the notions of static entities and dynamic processes, and the E-R model offers a simple, yet powerful means for modeling the former.

We argue in this paper that the notions of social actor and dependency (among actors) will become increasingly important in software development and conceptual modeling. In software development, agent-oriented programming is gaining popularity because agent-oriented software offers (or, at least, promises) features such as software autonomy, evolvability and flexibility – all of them much-needed in the days of the Internet and e-commerce. In Requirements Engineering, agents and goals are explicitly modeled and analyzed during early requirements phases in order to generate functional and non-functional requirements for a software system (e.g., [DvLF93]). In Conceptual Modeling, more and more organizations invest in models of their business processes, organizational structure and organizational function. Enterprise resource Planning (ERP) technology, as offered by SAP, Oracle, PeopleSoft, Baan et al, includes enterprise models which serve as blueprints for an organization, as well as a starting points for customizing ERP systems. For enterprise models and agent-oriented software alike, the notions of social actor and dependency constitute modeling cornerstones.

This paper reviews a particular social actor model, i*, originally introduced in Eric Yu's PhD thesis [Yu95]. In addition, we propose a formal language, called Tropos, which extends i* and makes it more expressive and amenable to analysis. Section 2 reviews the i* model and discusses similarities and differences with E-R and UML class diagrams. In Section 3, we introduce Tropos and demonstrate its expresiveness through examples. Finally, Section 4 summarizes the contributions of this paper and suggests directions for further research.

2 A Model of Distributed Intentionality

i* offers a conceptual framework for modeling social settings. The framework is founded on the notions of *actor* and *goal*. i* (which stands for "distributed intentionality") assumes that social settings 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 goes through concerning its relationships with other actors. These models have been formalized using intentional concepts from AI, such as goal, belief, ability, and commitment (e.g., [CL90]). The framework has been presented in detail in [Yu95] and has been related to different application areas, including requirements engineering [Yu93], business process reengineering [YML96], and software processes [YM94].

A strategic dependency diagram consists of actors, represented by circles, and social dependencies, represented as directional links. The actors are the stakeholders relevant to the social setting being modelled. Every dependency link has a *depender* actor, who needs something to be done, and a *dependee* actor, who is willing and able to deliver on that something. Thus, dependencies represent commitments of one actor to deliver on what another actor needs. There are four types of dependencies: *goal*, *softgoal*, *task* and *resource*. A goal dependency implies that one actor wants a goal to be achieved, e.g., `RepairCar`, and another actor is willing and able to fulfill this goal. Softgoals are goals that are not formally definable, such as `SecureEmployment`, or `FairEstimate`. Unlike goals, softgoals do not have a well-defined criterion as to whether they are fulfilled. A task dependency implies that one actor wants a task to be performed, e.g., `DoAppraisal`, and another actor is willing and able to carry it out. Finally, a resource dependency means that one actor needs a resource, such as `Premium`, and another actor can deliver on it.

Figure 1 presents a strategic dependency diagram for insurance claims. The diagram includes four actors, named respectively `BodyShop`, `Customer`, `InsuranceCo` and `Appraiser`. These are the relevant stakeholders to the task of handling insurance claims. The diagram also shows the dependencies among these actors. For instance, `Customer` depends on `BodyShop` to repair her car (`RepairCar`, a goal dependency), while `BodyShop` depends on `Customer` for the repairs to be paid (`RepairCosts`, a resource dependency). In addition, `Customer` depends on `BodyShop` to maximize the repairs done to her car (`MaxRepairs`), while `BodyShop` depends on `Customer` for keeping her clients (`KeepClient`). These are both softgoal dependencies. Turning to the dependencies between the customer and the insurance company, `Customer` depends on `InsuranceCo` to cover the repairs (`CoverRepairs`, goal dependency) and pay damage costs (`DamageCosts`, resource dependency); `InsuranceCo` depends on `Customer` to pay the insurance premium (`Premium`) and for continued business (`ContinuedBusiness`, softgoal dependency). `Customer` also depends on `Appraiser` for a fair estimate of the damages on her car (`FairEstimate`). Finally, `InsuranceCo` depends on `Appraiser` to carry out an appraisal (`DoAppraisal`, task dependency), while `Appraiser` depends on `InsuranceCo` for secure employment (`SecureEmployment`).

Superficially, one can view strategic dependency diagrams as variations on entity and relationship diagrams. After all, they are graph-based and nodes represent particular kinds of things while edges represent particular kinds of relationships. However, strategic dependency diagrams come about by asking entirely different questions about the application being modelled. For E-R diagrams the basic question is “what are the relevant entities and relationships?”. For strategic dependency diagrams, on the other hand, the basic questions are “who are the relevant stakeholders” and “what are their obligations to other actors?”. During analysis, we might also want to answer questions such as “is it possible that an insurance claim will never be served?” or “what could possibly happen during the lifetime of a claim?”.

Similar comments apply in comparing strategic dependency diagrams with UML class diagrams. However, we do propose to adopt some of the notational conveniences of class diagrams, such as min/max cardinalities on dependencies and specialization relationships among actors. For instance, we would like to be able to declare different kinds of customers: `CorporateCustomer`, `IndividualCustomer`, `SpecialCustomer`, `ValuedCustomer`, etc. For each of these customer classes, the insurance company may have different insurance claim handling procedures in place, and also different expectations.

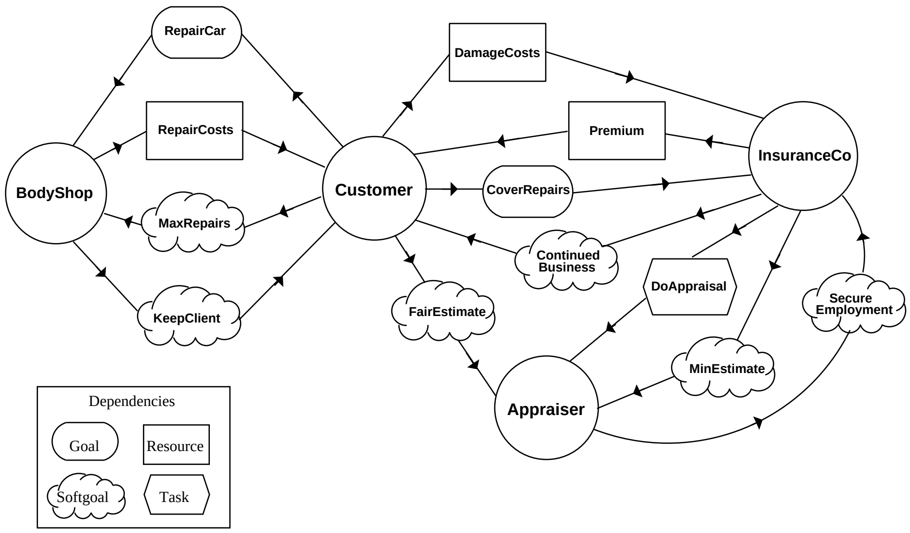

Fig. 1. Handling insurance claims

3 Formal Specification of i* Diagrams

The diagram of Figure 1 only provides a sketch of the social setting being modeled. In order to produce analyzable models, we propose a (new) specification language, called Tropos, founded on the primitives of i*. The language provides constructs for the specification of actors, social dependencies, entities, relationships and actions. Some of its features are inspired by the formal requirements specification language KAOS [DvLF93]. However, KAOS is based on different primitive concepts; in particular, it does not support the notion of social dependency.

The language is structured in two layers. The outer layer declares concepts and has an entity-relationship flavour; the inner layer expresses constraints on those concepts in a typed first-order temporal logic with real time constructs [Koy92]. The notation for temporal operators is summarized in Figure 2.

To begin the formal specification of our example, we note that many of the dependencies of Figure 1 relate to a particular claim. `Claim`, therefore, is an important entity of our model, and can be formulated as follows:

Entity Claim
  Has claimId: Number, date: Date, claimant: Customer, quote[0..n]: Quote, insP: InsPolicy
  Invariant
    $date \leq insP.expirDate$

P	P holds in the *current* state
$\bullet P$	P holds in the *previous* state
$\circ P$	P holds in the *next* state
$\blacklozenge P$	P holds in *current or some past* state
$\Diamond P$	P holds in *current or some future* state
$\blacksquare P$	P holds in *current and all past* states
$\Box P$	P holds in *current and all future* states

Fig. 2. Temporal operators

This entity has a number of attributes, such as identification number (`claimId`), `date`, `claimant` and a list of `quotes` from the bodyshops. It is also associated to a valid insurance policy (..."no valid policy, no claim" your friendly insurance agent would say...):

Entity InsPolicy
  Has insNo: Number, expirDate: Date, car: Car, customer: Customer,
    policyType: PolicyType, premiumAmount: Currency

Actors are defined in terms of their attributes, their initial goals and the actions they are capable of. For instance, the specification for `Customer` is:

Actor Customer
  Has name: Name, addr: Address, phone: Number
  Capable of MakeClaim, Pay
  Goal

$$\forall cl : Claim(cl.claimant = self \Rightarrow (\exists cover : CoverRepairs[cl](\Diamond Fulfil(cover))))$$

This actor can perform the actions `MakeClaim` and `Pay`. Its goal is that all its claims be eventually covered by the insurance company. There are several details to note about the specification. First, the style is "object-oriented", in the sense that `Customer` is considered a class and `self` refers to one of its instances. Second, the variable `cover` is quantified over the set `CoverRepairs[cl]`, which defines the set of all instances of the goal dependency `CoverRepairs` that have the entity instance `cl` among their attributes. Finally, `Fulfil` is a special predicate that states that a goal has been achieved; we will explain it shortly when we introduce the notion of goal modalities.

Similarly, the actor `InsuranceCo` is defined as follows:

Actor InsuranceCo
  Has name: Name, addr: Address, phone: Number
  Capable of AcceptClaim, Pay
  Goal $\forall customer : Customer, \exists business : ContinuedBusiness$
  $(business.Dependee = customer \wedge Fulfil(business))$

The goal of this actor is that all its clients continue their business with the company. Therefore, the softgoal dependency `ContinuedBusiness` should be fulfilled for all the customers. Note that it is possible to refer to a depender or a dependee just as if they were any other attribute of a dependency.

Goal dependencies are defined in terms of their modality, attributes, involved agents and constraints. The most important goal dependency in our example is `CoverRepairs`: the customer depends on the insurance company to cover the cost of repairing her car.

GoalDependency CoverRepairs
  Mode Fulfil
  Depender Customer
  Dependee InsuranceCo
  Has cl: Claim
  Defined
  $Claim(cl) \wedge \bullet\neg Claim(cl) \wedge \Diamond_{\leq 6mo} RunsOk(cl.insP.car)$
  Necessary
  $\exists repair : RepairCar[cl](Fulfil(repair))$
  $\exists damage : DamageCost[cl] \Rightarrow (Fulfil(damage)$
  $\wedge(\exists cost : RepairCost[cl](Fulfil(cost)$
  $\Rightarrow damage.amount \geq cost.amount)))$

The modality of this dependency is `Fulfil`, which means that it should be achieved at least once. There are other modalities in our language, such as `Maintain` (continuing fulfilment) and `Avoid` (continuing prevention of a goal from being fulfilled). The **Defined** clause gives a formal definition of the goal; **Necessary** specifies necessary conditions for it to be achieved. According to its modality, we interpret the definition of `CoverRepairs` as

$$Fulfil(self) \Leftrightarrow \blacklozenge(Claim(cl) \wedge \bullet\neg Claim(cl) \wedge \Diamond_{\leq 6mo} RunsOk(cl.insP.car))$$

This definition states that the customer expects that the car should start running OK in no more than 6 months after the claim is made. We capture the moment in which the claim is made (`Claim(cl)` $\wedge \bullet \neg$ `Claim(cl)`) by using a fluent `Claim(cl)` that is true if and only if `cl` is an instance of the class `Claim` at a particular point in time. The first necessary condition can be interpreted as

$$Fulfil(self) \Rightarrow \blacklozenge(\exists repair : RepairCar[cl](Fulfil(repair)))$$

It states that the car must be repaired by a bodyshop before the customer considers that the claim has been covered. The second necessary condition states that the customer expects to receive the damage costs from the insurance company, and they should be enough to pay the repair costs to the bodyshop.

Resource dependencies are specified in a similar way. The following are the definitions for the resources `DamageCosts` and `Premium`:

ResourceDependency DamageCosts
Mode Fulfil
Depender Customer
Dependee InsuranceCo
Has cl: Claim, amount: Currency
Necessary

$$\exists app : DoAppraisal[cl], \exists min : MinEstimate[app](Fulfil(app) \wedge Fulfil(min) \wedge \exists quote : Quote(quote \in cl.quote \wedge quote.amount \leq app.amount))$$

ResourceDependency Premium
Mode Fulfil
Depender InsuranceCo
Dependee Customer
Has insP: insurancePolicy, dueDate: Date
Necessary

$$\forall cl : Claim(\exists cover : CoverRepairs[cl] (cl.insP = insP \Rightarrow Fulfil(cover)))$$

The necessary condition of `DamageCosts` states that `InsuranceCo` will deliver the resource only if there is an appraisal that minimizes the estimate on the cost of the damages. Furthermore, there must be a quote from a bodyshop that is lower or equal than the value appraised. In the second dependency, the `Customer` will only pay the `Premium` if all its claims have been covered by the insurance company.

As softgoal dependencies cannot be formally defined, we only attempt to characterize them with necessary conditions. For instance, the following is the specification for `MinEstimate`:

SoftGoalDependency MinEstimate
Mode Fulfil
Depender InsuranceCo
Dependee Appraiser
Has appraisal : DoAppraisal
Necessary

$$\neg\exists otherApp : DoAppraisal[appraisal.cl] (otherApp.amount < appraisal.amount)$$

We state that the insurance company expects that the estimate be minimized, in the sense that there should be no other appraisal for the same claim estimating a lower amount. Note that this condition does not fully characterize the softgoal since we are actually not quantifying over all possible appraisals, but rather on the appraisals that exist in our system. Another important softgoal dependency in our example is `ContinuedBusiness`:

SoftGoalDependency ContinuedBusiness
  Mode Maintain
  Depender InsuranceCo
  Dependee Customer
  Necessary
    $\exists pr : Premium(pr.Dependee = self.Dependee$
      $\wedge pr.insP.expirDate > now)$

This softgoal has a different modality from the dependencies presented so far. The necessary condition, which states that the insurance company expects the customer to always be up to date with her payments, is interpreted as follows:

$$Maintain(self)$$
$$\Rightarrow \exists t : Time(\blacksquare_{\geq t}(\exists pr : Premium(pr.Dependee = self.Dependee$$
$$\wedge pr.insP.expirDate > now))$$

Finally, actions are input-output relations over entities. They are characterized by pre- and post-conditions; action applications define state transitions. For instance, the action `MakeRepair` performed by the bodyshop defines a transition from a state in which the car is not working well to another state in which it starts running:

Action MakeRepair
  Performed By BodyShop
  Refines RepairCar
  Input cl : Claim
  Output cl : Claim
  Pre $Claim(cl) \wedge \neg RunsOK(cl.insP.car)$
  Post $RunsOK(cl.insP.car)$

4 Conclusions and Directions for Further Research

We have presented a formal model of actors and social dependencies, intended for modeling social settings. We have also argued that models of social settings will become increasingly important as agent-oriented software becomes more prevalent, and organizational models become more widely used in corporate practice.

We are currently working on a software development methodology which is founded on the Tropos model. The methodology supports the following phases:

– *Early requirements*, concerned with the understanding of a problem by studying an existing organizational setting; the output of this phase is an organizational model which includes relevant actors and their respective dependencies;
– *Late requirements*, where the software system-to-be is described within its operational environment, along with relevant functions and qualities; this description models the system as a (small) number of actors which have a number of dependencies with actors in their environment; these dependencies define the system's functional and non-functional requirements;
– *Architectural design*, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows; within our framework, subsystems are represented as actors and data/control interconnections are represented as (system) actor dependencies;
– *Detailed design*, where each architectural component is defined in further detail in terms of inputs, outputs, control, and other relevant information; our framework adopts elements of AUML [OPB99] to complement the features of i*;
– *Implementation*, where the actual implementation of the system is carried out, consistently with the detailed design; we use a commercial agent programming platform, based on the BDI (Beliefs-Desires-Intentions) agent architecture for this phase.

[MC00] and [CKM00] present the motivations behind the Tropos project and offer an early glimpse of how the methodology would work for particular examples. We are also exploring the application of model checking techniques [CGP99] in analyzing formal specifications such as those presented in this paper. For example, we are studying the possibility of "simulating" formal specifications in order to establish that certain properties will hold for all possible futures, or some future. We are also looking at analysis techniques which can facilitate the discovery of deviations between a formal specification and the modeller's understanding of what is being modelled.

Acknowledgements. We are grateful to our colleagues Eric Yu, Jaelson Castro, Manuel Kolp, Raoul Jarvis (University of Toronto), Yves Lesperance (York University), Fausto Giunchiglia (University of Trento), and Marco Pistore, Paolo Traverso, Paolo Bresciani and Anna Perini (IRST) for contributing ideas and helpful feedback to this research.

This work is funded partly by the Communications and Information Technology Ontario (CITO) and the Natural Sciences and Engineering Research Council (NSERC) of Canada. Also, by the University of Trento, and the Institute of Research in Science and Technology (IRST) of the province of Trentino, Italy.

References

[BCN92] C. Batini, S. Ceri, and S. Navathe. *Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Che75] P. Chen. The Entity-Relationship model: Towards a unified view of data. In D. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases*, September 1975.

[CKM00] J. Castro, M. Kolp, and J. Mylopoulos. Developing agent-oriented information systems for the enterprise. In *Second International Conference on Enterprise Information Systems*, July 2000.

[CL90] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 32(3), 1990.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[Koy92] R. Koymans. Specifying message passing and time-critical systems with temporal logic. In *Springer-Verlag LNCS 651*. Springer-Verlag, 1992.

[MC00] J. Mylopoulos and J. Castro. Tropos: A framework for requirements-driven software development. In J. Brinkkemper and A. Solvberg, editors, *Information Systems Engineering: State of the Art and Research Themes*. Springer-Verlag, 2000.

[OPB99] J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. To be published, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[YM94] E. Yu and J. Mylopoulos. From E-R to A-R – modeling strategic actor relationships for business process reengineering. In P. Loucopoulos, editor, *Proceedings Thirteenth International Conference on the Entity-Relationship Approach*. Springer-Verlag, December 1994.

[YML96] E. Yu, J. Mylopoulos, and Y. Lesperance. AI models for business process reengineering. *IEEE Expert*, 1996.

[Yu93] E. Yu. Modelling organizations for information systems requirements engineering. *First IEEE Int. Symposium on Requirements Engineering*, January 1993.

[Yu95] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Toronto, Canada, 1995.