Software Architectures as Organizational Structures

Manuel Kolp **John Mylopoulos**

Department of Computer Science, University of Toronto, Toronto M5S3G4, Canada
{mkolp, jm}@cs.toronto.edu

Abstract

*This paper proposes a set of concepts for describing a software architecture as an organization. These social structures consist of actors who have goals to fulfil and social dependencies describing their obligations. The framework is an adaptation of i** [Yu95] *proposed as a modeling language for early requirements. Based on this framework, the paper advocates architectural styles for software which adopt concepts from organization theory and strategic alliances literature. The styles are modeled in i* and formalized in terms of Telos metaconcepts. Each proposed style is evaluated with respect to a set of software quality attributes, such as predictability, adaptability and openness. The use of these styles is illustrated and contrasted with a software architecture for mobile robots reported in the literature.*

1. Introduction

We are interested in narrowing the semantic gap between a software architecture and the requirements model from which it was derived . One way to achieve this is to adopt the same concepts for describing requirements and software architectures. This paper reports on an experiment to use concepts from *i**, a modeling framework for early requirements, to model software architectures.

i* offers concepts such as actor, goal, and social dependency intended to model social structures involving social actors, their goals and social inter-dependencies. To adopt this framework for software architectures, we first propose a set of architectural styles inspired by organizational theory and strategic alliance literature, and formalize these as Telos [Myl90] metaconcepts. To guide the selection process among the styles, we evaluate them with respect to a number of software qualities. Finally, we illustrate their use by applying them to two examples of software architectures reported in the literature.

This research is being conducted in the context of the Tropos project [Cas01], which is developing a requirements-driven methodology for software systems.

Section 2 presents our organization-inspired architectural styles described in terms of the strategic dependency model from *i** and specified in Telos. Section 3 introduces a set of desirable software quality attributes for comparing them. Section 4 overviews a mobile robot example while Section 5 sketches the Tropos project within which this research has been conducted. Finally, Section 6 summarizes the contributions of the paper and points to further research.

2. Organizational Styles

Organizational theory (such as [Sco98]) and strategic alliances (e.g., [Yos95]) study alternatives for (business) organizations. These alternatives are used to model the coordination of business stakeholders -- individuals, physical or social systems -- to achieve common goals. Using them, we view a software system as a social organization of coordinated autonomous components (or agents) that interact in order to achieve specific, possibly common goals. We adopt (some of) the styles [Fux01] defined in organizational theory and strategic alliances to design the architecture of the system, model them with *i**, and specify them in Telos [Myl90].

In *i**, a strategic dependency model is a graph, in which each node represents an actor, and each link between two actors indicates that one actor depends on another 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 as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender is adversely affected in its ability to achieve its goals.

The model distinguishes among four types of dependencies -- goal-, task-, resource-, and softgoal-dependency -- based on the type of freedom that is allowed in the relationship between depender and dependee. Softgoals are distinguished from goals because they do not have a formal definition, and are amenable to a different (more qualitative) kind of analysis [Chu00].

For instance, in the structure-in-5 style (Figure 1), the coordination, middle agency and support actors depend on the apex for strategic management purposes. Since the goal *Strategic Management* is not well-defined, it is represented as a softgoal (cloudy shape). The middle agency actor depends on both the coordination and support actors respectively through goal dependencies *Control* and *Logistics* represented as oval-shaped icons. The operational core actor is related to the coordination and support actors respectively through the *Standardize* task dependency and the *Non-operational service* resource dependency.

In the sequel we briefly discuss nine common organizational styles.

The **structure-in-5** (Figure 1) style consists of the typical strategic and logistic components generally found in many organizations. At the base level one finds the operational core where the basic tasks and operations -- the input, processing, output and direct support procedures associated with running the system -- are carried out. At the top of the organization lies the apex composed of strategic executive actors.

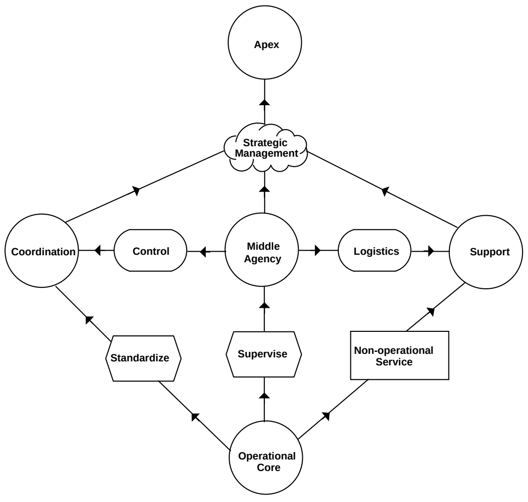

Figure 1. Structure-in-5.

Below it sit the control/standardization, management components and logistics, respectively coordination, middle agency and support. The coordination component carries out the tasks of standardizing the behavior of other components, in addition to applying analytical procedures to help the system adapt to its environment. Actors joining the apex to the operational core make up the middle agency. The support component assists the operational core for non-operational services that are outside the basic flow of operational tasks and procedures.

Figure 2 specifies the structure-in-5 style in Telos [Myl90]. Telos is a language intended for modeling requirements, design, implementation and design decisions for software systems. It provides features to describe metaconcepts that can be used to represent the knowledge relevant to a variety of worlds – subject, usage, system, development worlds - related to a software system. Our styles are formulated as Telos metaconcepts, primarily based on the aggregation semantics for Telos.

The structure-in-5 style is then a metaclass - *StructureIn5MetaClass* - aggregation of five (*part*) metaclasses: *ApexMetaClass*, *CoordinationMetaClass*, *MiddleAgencyMetaClass*, *SupportMetaClass* and *OperationalCoreMetaClass*, one for each actor composing the structure-in 5 style depicted in Figure 1. Each of these five components exclusively belongs (*exclusivePart*) to the composite (*StructureIn5MetaClass*) and their existence depend (*dependentPart*) on the existence of the composite. A structure-in-5 specific to an application domain will be defined as a Telos class, instance of *StructureIn5MetaClass* (See Section 4). Similarly each structure-in-5 component specific to a particular application domain will be defined as a class, instance of one of the five *StructureIn5Metaclass* components.

```
TELL CLASS StructureIn5MetaClass
  IN Class WITH /*Class is here used as a MetaMetaClass*/
        attribute
          name: String
        part, exclusivePart, dependentPart
          ApexMetaClass: Class
          CoordinationMetaClass: Class
          MiddleAgencyMetaClass: Class
          SupportMetaClass: Class
          OperationalCoreMetaClass: Class
  END StructureIn5MetaClass
```

Figure 2. Structure-in-5 in Telos.

The **flat structure** has no fixed structure and no control of one actor over another is assumed. The main advantage of this architecture is that it supports autonomy, distribution and continuous evolution of an actor architecture. However, the key drawback is that it requires an increased amount of reasoning and communication by each participating actor.

The **pyramid** style is the well-known hierarchical authority structure exercised within organizational boundaries. Actors at the lower levels depend on actors of the higher levels. The crucial mechanism is direct supervision from the apex. Managers and supervisors are then only intermediate actors routing strategic decisions and authority from the apex to the operating level. They can coordinate behaviors or take decisions by their own but only at a local level. This style can be applied when deploying simple distributed systems.

Moreover, this style encourages dynamicity since coordination and decision mechanisms are direct, not complex and immediately identifiable. Evolvability and modifiability can thus be implemented in terms of this style at low costs. However, it is not suitable for huge distributed systems like multi-agent systems requiring many kinds of agents.

The **joint venture** style (Figure 3) involves agreement between two or more principal partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. Through the delegation of authority to a specific joint management actor that coordinates tasks and operations and manages sharing of knowledge and resources they pursue joint objectives and common purpose. Each principal partner can manage and control itself on a local dimension and interact directly with other principal partners to exchange, provide and receive services, data and knowledge. However, the strategic operation and coordination of such a system and its partner actors on a global dimension are only ensured by the joint management actor. Outside the joint venture, secondary partners supply services or support tasks for the organization core.

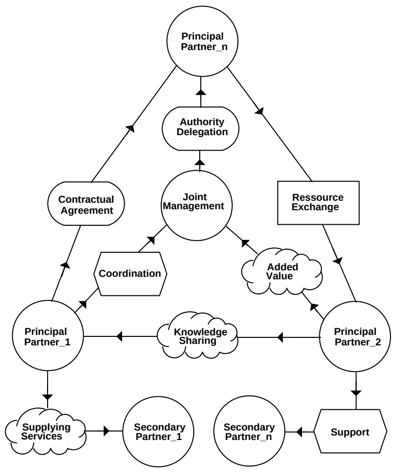

Figure 3. Joint Venture.

The **takeover** style involves the total delegation of authority and management from two or more partners to a single collective *takeover* actor. It is similar in many ways to the joint venture style. The major and crucial difference is that while in a joint venture identities and autonomies of the separate units are preserved, the takeover absorbs these critical units in the sense that no direct relationships, dependencies or communications are tolerated except those involving the takeover.

The **arm's-length** style implies agreements between independent and competitive but partner actors. Partners keep their autonomy and independence but act and put their resources and knowledge together to accomplish precise common goals. No authority is delegated or lost from a collaborator to another.

The **bidding** style (Figure 4) involves competitivity mechanisms and actors behave as if they were taking part in an auction. The auctioneer actor runs the show, advertises the auction issued by the auction issuer, receives bids from bidder actors and ensure communication and feedback with the auction issuer.

The auctioneer might be a system actor that merely organizes and operates the auction and its mechanisms. It can also be one of the bidders (for example selling an item which all other bidders are interested in buying). The auction issuer is responsible for issuing the bidding.

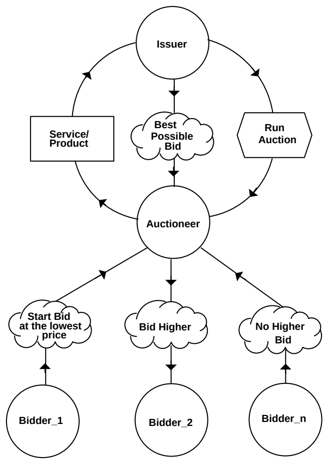

Figure 4. Bidding.

The **hierarchical contracting** style identifies coordinating mechanisms that combine arm's-length agreement features with aspects associated with pyramidal authority. Coordination mechanisms developed to manage arm's-length (independent) characteristics involve a variety of negotiators, mediators and observers at different levels handling conditional clauses to monitor and manage possible contingencies, negotiate and resolve conflicts and finally deliberate and take decisions. Hierarchical relationships, from the executive apex to the arm's-length contractors (top to bottom) restrict autonomy and underlie a venture between the contracting parties.

The **vertical integration** style merges, backward or forward, one or more system actors engaged in related tasks but at different stages of a production process. A merger synchronizes and controls interactions between each of the participants that can be considered intermediate workshops. Vertical integrations take place between exchange partners, actors symbiotically related.

The **co-optation** style involves the incorporation of representatives of external systems into the decision-making or advisory structure and behavior of an initiating organization. By co-opting representatives of external systems, organizations are, in effect, trading confidentiality and authority for resource, knowledge assets and support.

3. Evaluating Architecture

The organizational styles defined in Section 2 can be evaluated and compared using the following software quality attributes identified for architectures involving coordinated autonomous components (e.g., Web, internet, agent or peer-to-peer software systems) :

1 - Predictability. Autonomous components like agents have a high degree of autonomy in the way that they undertake action and communication in their domains. It can be then difficult to predict individual characteristics as part of determining the behavior of a distributed and open system at large.

2 - Security. Autonomous components are often able to identify their own data sources and they may undertake additional actions based on these sources. Protocols and strategies for verifying authenticity for these data sources by individual components are an important concern in the evaluation of overall system quality since there is the danger of hostile external entities spoofing the system to acquire information accorded to trusted domain components.

3 - Adaptability. Components may be required to adapt to modifications in their environment. They may include changes to the component's communication protocol or possibly the dynamic introduction of a new kind of component previously unknown or the manipulations of existing components.

- Coordinability. Autonomous components are not particularly useful unless they are able to coordinate with other components. This can be realized in two ways:

4 - Cooperativity. They must be able to coordinate with other entities to achieve a common purpose.

5 - Competitivity. The success of one component implies the failure of others.

6 - Availability. Components that offer services to other components must implicitly or explicitly guard against the interruption of offered services. Availability must actually be considered a sub-attribute of security [Chu00]. Nevertheless, we deal with it as a top-level software quality attribute due to its increasing importance in multi-agent system design.

7 - Integrity. A failure of one component does not necessarily imply a failure of the whole system. The system then needs to check the completeness and the accuracy of data, information and knowledge transactions and flows. To prevent system failure, different components can have similar or replicated capabilities and refer to more than one component for a specific behavior.

8 - Modularity increases efficiency of task execution, reduces communication overhead and usually enables high flexibility. On the other hand, it implies constraints on inter-module communication.

9 - Aggregability. Some components are parts of other components. They surrender to the control of the composite entity. This control results in efficient tasks execution and low communication overhead, however prevents the system to benefit from flexibility.

	1	2	3	4	5	6	7	8	9
Struct-5	+	+		+	-	+	++	++	++
Pyramid	++	++	+	++	-	+	--	-	
Joint-Vent	+	+	++	+	-	++		+	++
Bid	--	--	++	-	++	-	--	++	
Takeover	++	++	-	++	--	+		+	+
Arm's-Lgth	-	--	+	-	++	--	++	+	
Hierch Ctr			+	+	+	+		+	+
Vert Integr	+	+	-	+	-	+	--	--	--
Coopt	-	-	++	++	+	--	-	--	

Table 1. Correlation catalogue.

Table 1 summarizes the correlation catalogue for the organizational patterns and top-level quality attributes we have considered. Following notations used by the NFR (non functional requirements) framework [Chu00], +, ++, -, --, respectively model partial/positive, sufficient/positive, partial/negative and sufficient/negative contributions.

4. Example

To motivate our styles, we consider an application domain where distributed and open architectures are increasingly important: mobile robots.

The mobile robot example presented in [Sha96] studies notably the layered architecture (Figure 5) implemented in the Terregator and Neptune robots and office delivery robots. According to [Sha96] at the lowest level, reside the robot control routines (motors, joints,...). Levels 2 and 3 deal with the input from the real world. They perform sensor interpretation (the analysis of the data from one sensor) and sensor integration (the combined analysis of different sensor inputs). Level 4 is concerned with maintaining the robot's model of the world. Level 5 manages the navigation of the robot. The next two levels, 6 and 7, schedule and plan the robot's actions. Dealing with problems and replanning is also part of the level-7 responsibilities. The top level provides the user interface and overall supervisory functions.

The following software quality attributes are relevant for the robot's architecture [Sha96]: *Cooperativity*, *Predictability*, *Adaptability*, *Integrity*. Take for instance, consider *Cooperativity* and *Predictability*.

Cooperativity: the robot has to coordinate the actions it undertakes to achieve its designated objective with the reactions forced on it by the environment (e.g., avoid an obstacle). The idealized layered architecture (Figure 5) implemented on some mobile robots does not really fit the actual data and control-flow patterns [Sha96]. The layered architecture style suggests that services and requests are passed between adjacent layers. However, data and information exchange is actually not always straight-forward. Commands and transactions may often need to skip intermediate layers to establish direct communication. A structure-in-5 proposes a more distributed architecture allowing more direct interactions between component.

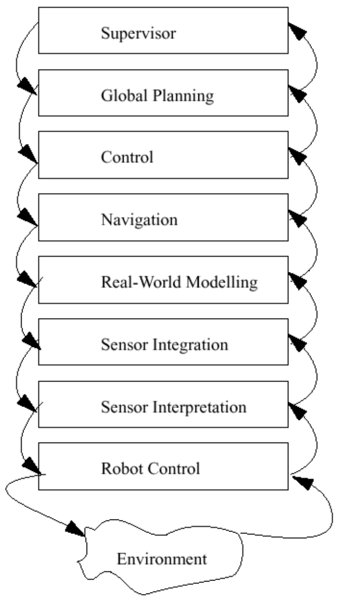

Figure 5. Classical mobile robot layered architecture.

Another recognized problem is that the layers do not separate the data hierarchy (sensor control, interpreted results, world model) from the control hierarchy (motor control, navigation, scheduling, planning and user-level control). Again the structure-in-5 could better differentiate the data hierarchy - implemented by the operational core, and support components - from the control structure – implemented by the operational core, middle agency and strategic apex as will be described in Figure 6.

Adaptability: application development for mobile robots frequently requires customization, experimentation and dynamic reconfiguration. Moreover, changes in tasks may require regular modification. In the layered architecture, the interdependencies between layers prevent the addition of new components or deletion of existing ones. The structure-in-5 style separates independently each typical component of an organizational structure but a joint venture isolating components and allowing dynamic manipulation should be a better candidate. Partner components, except the joint manager, can be added or deleted in a more flexible way.

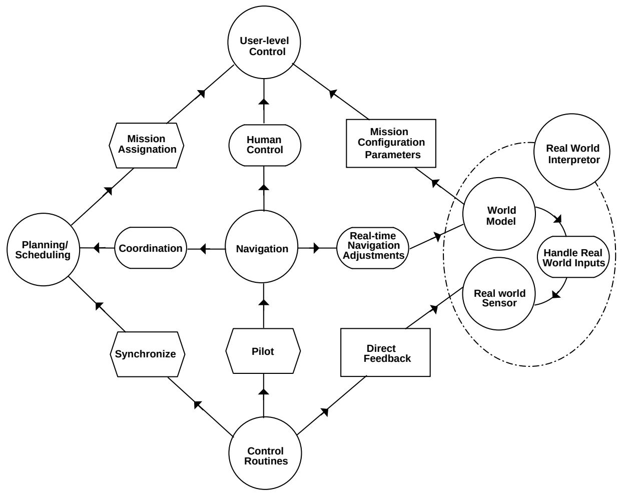

Figure 6. A structure-in-5 mobile robot architecture.

Figure 6 depicts a mobile robot architecture following the structure-in-5 style from Figure 1. The *control routines* component is the *operational core* managing the robot motors, joints, etc. *Planning/Scheduling* is the *coordination* component scheduling and planning the robot’s actions. The *real world interpreter* is the *support* component composed of two sub-components: *Real world sensor* accepts the raw input from multiple sensors and integrates it into a coherent interpretation while *World Model* is concerned with maintaining the robot’s model of the world and monitoring the environment for landmarks. *Navigation* is the *middle agency* component, the central intermediate module managing the navigation of the robot. Finally, the *user-level control* is the human-oriented *strategic apex* providing the user interface and overall supervisory functions.

```
TELL CLASS MobileRobotClass
        IN StructureIn5MetaClass WITH
        attribute  name: String
        part, exclusivePart, dependentPart
              ControlRoutinesClass: OperationalCoreMetaClass
              RealWorldInterpreter: SupportMetaClass
              NavigationClass: MiddleAgencyMetaClass
              PlanningClass: CoordinationMetaClass
              UserLevelControl: ApexMetaClass
END MobileRobotClass
```

Figure 7. Robot structure-in-5 architecture in Telos.

Figure 7 formulates the media robot structure-in-5 in Telos. *MobileRobotClass* is a Telos class, instance of the *StructureIn5Metaclass* specified in Figure 2. This aggregation is composed of five exclusive and dependent parts *ControlRoutinesClass*, *RealWorldInterpreterClass*, *NavigationClass*, *PlanningClass* and *UserLevelControl-Class*, each of them is instance of one metaclass, component of *StructureIn5MetaClass*.

5. A Requirements-Driven Methodology

This research is conducted in the context of Tropos [Cas01], a software system development methodology which is founded on the concepts of actor and goal. Tropos describes in terms of the same concepts the organizational environment within which a system will eventually operate, as well as the system itself. The proposed methodology supersedes traditional development techniques, such as structured and object-oriented ones in the sense that it is tailored to systems that will operate within an organizational context and is founded on concepts used during early requirements analysis. To this end, we adopt the concepts offered by *i** [Yu95], a modeling framework offering concepts like actor, agent, position, role, and social dependencies among actors, including goal, softgoal, task and resource ones.

Tropos spans four phases of software development:

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

6. Conclusion

The paper proposes a set of concepts for specifying software architectures which is inspired by requirements modeling research. As such, we believe that our proposal narrows the gap between a requirements specification and the software architecture to be produced from it. The software architectures produced within our framework are intentional in the sense that components have associated goals that are supposed to fulfil. The architectures are also social in the sense that each component has obligations/expectations towards/from other components. Obviously, such architectures are best suited to open, dynamic and distributed applications, such as those Web, internet, agent, and peer-to-peer software systems.

We are working on formalizing precisely the styles that have been identified and formalizing the sense in which a particular architecture is an instance of such a pattern.

The organizational styles we have described will eventually define a software architectural macrolevel. At a micro level we will be focusing on the notion of patterns like the broker, matchmaker, embassy, mediator, wrapper are more appropriate [Hay99]. Another direction for further work is to relate the architectural styles proposed in this work to extentional, classical architectural components such as (software) components, ports, connectors, interfaces, libraries and configurations.

References

[Cas01] J. Castro, M. Kolp and J. Mylopoulos. "A Requirements-Driven Development Methodology", *Proc. 13th Int. Conf. on Advanced Information Systems Engineering,*CAiSE'01, Interlaken, Switzerland, June 2001.

[Chu00] L. K. Chung, B. A. Nixon, E. Yu and J. Mylopoulos. *Non-Functional Requirements in Software Engineering,* Kluwer Publishing, 2000.

[Fux01] A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. "Information Systems as Social Structures", *Proc. 2nd Int. Conf. on Formal Ontologies for Information Systems,* FOIS'01, Ogunquit, USA, Oct. 2001.

[Hay99] S. Hayden, C. Carrick and Q. Yang. "Architectural Design Patterns for Multiagent Coordination", *Proc. 3th Int. Conf. on Agent Systems '99,*Agents'99, Seattle, USA, May 1999.

[Myl90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. "Telos: Representing Knowledge About Information Systems", *ACM Trans. Info. Sys.*, 8(4), 1990, pp. 325 – 362.

[Sco98] W. Richard Scott. *Organizations : rational, natural, and open systems*, Prentice Hall, 1998.

[Sha96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Yos95] M.Y. Yoshino and U. Srinivasa Rangan. *Strategic alliances: an entrepreneurial approach to globalization*, Boston, Mass., Harvard Business School Press, 1995.

[Yu95] E. Yu. *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.