USING META-MODEL-DRIVEN VIEWS TO ADDRESS SCALABILITY IN i* MODELS

by

Zheng You

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2004 by Zheng You

Revision	Date	By	Remark
4.4	9/1/2004	Jane You	Final revision from Eric. All queries tested again in ConceptBase.
4.3	8/19/2004	Jane You	1. Abstract (first draft) 2. Appendix (first draft) 3. Bibliography (not sure about how to reference some technical reports) 4. Inconsistency of notations in diagrams of chapter 8 is fixed. 5. File formatted according to the SGS request
4.2	8/14/2004	Jane You	All chapters proofread and revised (first round) Parts to add: 1. Abstract 2. Appendix 3. Bibliography 4. Bugs in diagrams to fix
4.01	8/7/2004	Jane You	Chapter 1~4, 9: proofread and revised (suggestions from Eric also implemented) Chapter 5~7: waiting for proofread Chapter 8: proofread yet NOT revised

Abstract

Using Meta-Model-Driven Views to Address Scalability in i* Models

Zheng You

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

This thesis proposes an extension to the i* framework to address scalability issues. The notion of "view" is exploited to selectively present portions of an i* "baseline model", which contains all modeled objects for a given application using i* notations. We first reformulate the i* framework and define four types of views—Actor Class, Strategic Dependency, Strategic Rationale, and Evaluation Results. Next, we define sub view types based on the four types of views and supply a view management framework. The views and sub-views are defined using meta-models, and formalized using the Telos conceptual modeling language. Each view type is associated with a formally defined "selection rule" so that the projection of a specific view from a baseline model can be automated. Relationships among views are depicted in View Maps. Illustrative examples are taken from the London Ambulance Service and the Trusted Computing Group case studies.

Acknowledgements

John for reviewing the thesis

Linda Liu in contributing ideas in the representational constructs

Concept Base team in providing the tool support

Jennifer in contributing the original TCG case study and offering comments on the result

Eric Yu for revising the thesis and comments

Contents

(??Do I provide Table of Figures or not?? The file is a bit too long)

1 Introduction

1.1 Motivation

The *i** framework is a conceptual modeling technique that supports goal- and agent-based reasoning. It was first proposed in Yu's 1994 PhD thesis—Modeling Strategic Relationship for Process Reengineering (Yu 1994). The i* framework was aimed at helping in process modeling, process design, and process analysis from a social and intentional perspective: A Strategic Dependency (SD) model is used to express "the intentional relationships among agents"; whereas a Strategic Rationale (SR) model is used to show "how processes are comprised of intentional elements [of the agents]." Applications of the framework were demonstrated in four areas: requirement engineering, business process reengineering, organizational impact analysis, and software process modeling. In addition to enhancing the argument by working examples, formal constructs of the framework were also presented in (Yu 1994).

A common challenge encountered by users of the i* framework is that the approach is difficult to scale up. Multiple factors may be contributing to the scalability challenge. The i* framework adopts a partial, semi-formal, and qualitative modeling approach that accommodates uncertainty and incompleteness in the real world. While tool support is possible to a certain degree, intensive human interaction is nevertheless required during modeling and analysis. As the size of an application increases, the complexity of modeling and analysis also increases.

The original purpose of the i* framework was to perform process analysis and process redesign (Yu 1994). These two activities require traversing of the modeled structure by i* users; therefore, human decision is required at each step. Moreover, the model evaluation process adopted from the NFR framework (Chung et al. 2000),

used to evaluate the effects of process elements on organizational goals, also requires intensive human interaction. For ease of human interaction, i* models must be visualized. However, any visualization is subject to the constraints of media ability and human comprehension. For example, when visualized, a diagram may be entitled to a limited space, a list may be confined to a finite length, and only two dimensions might be displayed for a matrix in a tabular format. While conceptually an i* model could grow infinitely, it can become intellectually unmanageable beyond a certain size.

We illustrate the scalability challenge in the next section using the London Ambulance Service (LAS) case study.

1.2 The London Ambulance Service Computer-Aided Despatch System

The London Ambulance Service Computer Aided Despatch (LAS-CAD) system is a well known software failure and has been used by the research community as a standard exemplar. It was introduced to the software engineering community at the 8th International Workshop on Software Specification and Design (IWSSD), using the Report of the Inquiry into the London Ambulance Service (LAS-Report 1993) as the primary source of information. Kramer and Wolf (Kramer and Wolf 1996) summarized the results of how several workshop participants handled the exemplar. Others, like Breitman et al. (Breitman et al. 1999) and Letier (Letier 2001) also used the LAS. Breitman et al. (Breitman et al. 1999) surveyed the possibility of the uses of newly—as of 1999—emerged requirements engineering (RE) techniques to identify LAS problems early on; and Letier (Letier 2001) used LAS as a case study for the KAOS goal-oriented requirements approach.

A case study using the i* modeling and evaluation techniques was also performed using a project-specific approach to resolve scalability issues (You 2003). Four i*

models[1] representing different aspects of the LAS case study, encompassing a total of 79 diagrams, were produced, including the evaluation (analysis) diagrams. Approximately 40 different forms of actors were presented in the four models. The study focused on the analysis of user-oriented questions, such as "Why is the manual system not able to meet the performance requirements?" and "How would an automated system help achieve the performance goals?"

The following sample indicates how large and complex an i* model can become. Figure 1.2-1 is a graphical representation of a partial i* model from the LAS case study, which involves only four actors (Ambulance Crew, Resource Allocator, Incident Reviewer, and LAS Management) and part of their inter-relationships. Figure 1.2-2 shows the corresponding formal representation in Telos. Telos is a conceptual modeling language adopted by Yu (Yu 1994) to embed i* concepts. Telos also serves as the internal representation language in the Organization Modelling Environment (OME) tool (OME 2003) supporting i* modeling. Modelers of i* work with the graphical models and do not need to see the Telos code.

[1] In this thesis, we reserve the term "model" for an entire representation (using i* meta-concepts) of a certain organization configuration, and therefore SD and SR, although called "models" in Yu's original thesis, are called "views". The definition of SD, SR, "model" and "view" will be presented in later chapters.

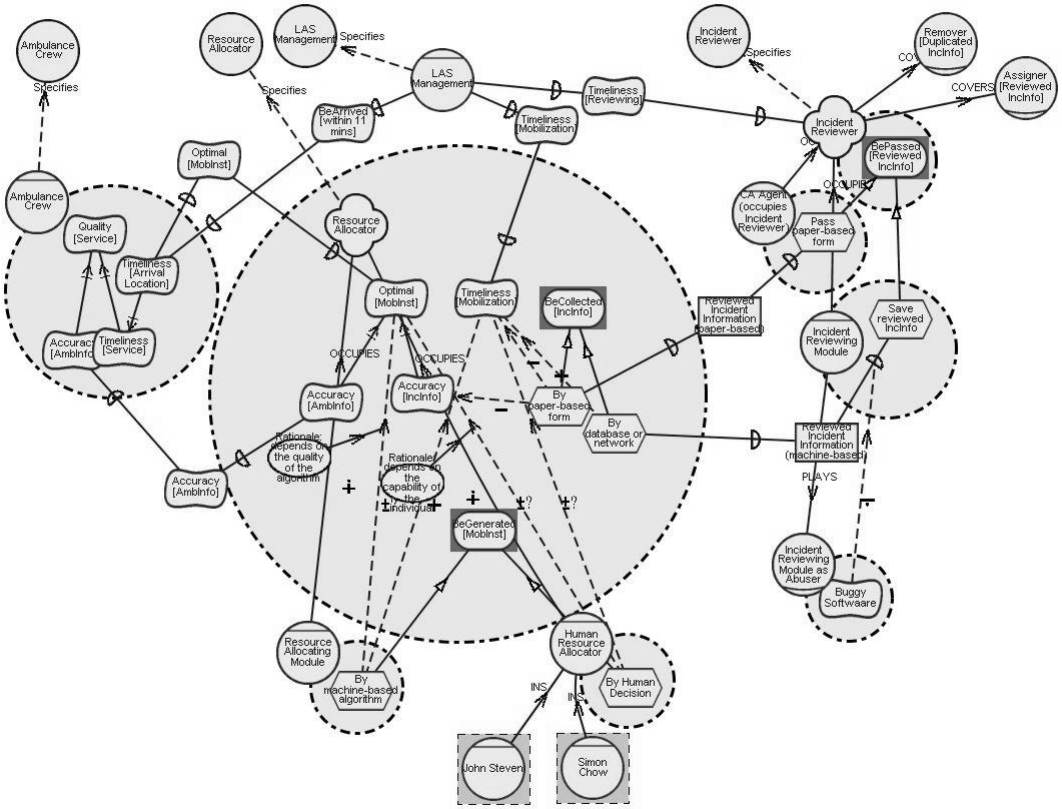


Figure 1.2-1 A partial model from the LAS-CAD case study

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
        name
                displayName : “Ambulance Crew”
        specifiedByLink
                : AmbulanceCrew_Agent
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
        name
                displayName : “Ambulance Crew”
        specifiesLink
                : AmbulanceCrew_PlainActor
        children
                : AC_QualityService
                : AC_TimelinessService
                : AC_TimelinessArrivalLocation
                : AC_AccuracyAmbInfo
```

```
                …
        [outDepLinks
                : AC_TALtoOptimalLink]
END

% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass WITH
        parent
                : AmbulanceCrew_Agent
        outDepLinks
                : AC_TALtoOptimalLink
        links
                : AC_TALtoTS_AndContributionLink
                …
        label
                : UndecidedElementLabel
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
        name
                displayName : “Resource Allocator”
        specifiedByLink
                : ResourceAllocator_Position
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
        name
                displayName : “Resource Allocator”
        specifiesLink
                : ResourceAllocator_PlainActor
        occupiedByLinks
                : RAMOccupiesRA
                : HRAOccupiesRA
        children
                : RA_OptimalMobInst
                : RA_TimelinessArrivalLocation
                : RA_AccuracyAmbInfo
                : RA_BeGeneratedMobInst
        [inDepLinks
                : OptimaltoOptimalLink_RA]
                        …
END

% occupies link from agent Resource Allocation Module to position Resource Allocator %
TELL SimpleClass RAMOccupiesRA IN OccupiesLinkClass WITH
        from
                : ResourceAllocationModule_Agent
        to
                : ResourceAllocator_Position
END

% agent Resource Allocation Module %
```

```
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass WITH
        occupiesLinks
                : RAMOccupiesRA
        children
                : RA_BeGeneratedMobInst_ByAlgorithm
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
        occupiesLinks
                : HRAOccupiesRA
        children
                : RA_BeGeneratedMobInst_ByHumanDecision
END

% dependency link from softgoal Timeliness [Arrival Location] inside agent Ambulance Crew to softgoal
dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
        from
                : AC_TimelinessArrivalLocaltion
                [: AmbulanceCrew_Agent]
        to
                : AC_OptimalMobInst_RA
END

% dependency link from softgoal dependum Optimal [MobInst] to softgoal Optimal [MobInst] inside position
Resource Allocator %
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
      from
: AC_OptimalMobInst_RA
         to
                : RA_OptimalMobInst
                [: ResourceAllocator_Position]
END

% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass, SoftGoalElementClass WITH
        inDeplinks
                : AC_TALtoOptimalLink
        outDepLinks
                : OptimaltoOptimalLink_RA
        label
                : UndecidedElementLabel
END

% softgoal Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
        parent
                : ResourceAllocator_Position
        inDepLinks
                : OptimaltoOptimalLink_RA
        label
                : UndecidedElementLabel
                …
```

END

Figure 1.2-2 Partial representation of the model in TELOS showing the size of the underlying constructs

Our experience with the LAS case study indicates that it is difficult indeed to accommodate all elements of a model in one representation that is still intellectually comprehensible. Although Figure 1.2-1 contains only 82 elements out of some 400, some i* users may already found this partial model difficult to read.

The LAS case study is considered to be only a medium-scale application. In fact, an i* model can increase in size and complexity to the extent that communications via the models become impossible – let alone the resolving of practical questions. In the literatures on i*, various ad-hoc practices have been used to reduce the large model into segments. The research reported in this thesis aims to introduce systematic methods to deal with scalability issues of i* models.

1.3 Research Objectives and Approach

Objectives

The objective of this research is to seek a systematic method to break down a large and complex i* model into segments that are self-contained, and comprehensible to humans. Thus, when using these segments in combination, users of i* are able to achieve the same as they could with the entire model. Meanwhile, we also intend to offer a systematic approach to maintain the connections among these segments.

Approach

We found that we need to reformulate the i* framework before new guidelines to deal with scalability can be introduced. Thus, the approach taken is, first, to provide a generic and formalized representation of the i* framework. The missing

representational constructs for some of the graphical notations – such as *role*[2] – are clarified, and the inconsistency in the formal constructs between Yu's original thesis and the Organization Modelling Environment (OME) tool are aligned. During this process, we did *not* introduce major new concepts to i* since our objective is not to redesign the i* framework but, rather, to resolve the scalability issues that arise while using i* in practice.

After the existing i* concepts had been clearly presented, a framework extension that contains different types of *views* (a projection over a model according to some criteria) and that supports view management was proposed. The views and sub-views are defined using meta-models, and formalized using the Telos conceptual modeling language. Each view type is associated with a formally defined "selection rule" so that the projection of a specific view from a baseline model can be automated. This formalization makes the view extension extensible, and makes it economic to maintain: New view types can be added by specifying a new class in Telos, and a view can be updated by changing its associated selection rule. Relationships among views are depicted in View Maps.

Then we studied the details of each type of view in the extension. Every view type is presented based on the following four aspects: an informal description of what type of elements from an i* model is to include; a sample view based on the LAS case study showing the elements actually qualified; brief justifications for the strengths and constraints of the view; and the formalized selection rule used to derive this type of view from an i* model.

The view extension and the selection rules were further validated in the research. The extension was validated against a larger and more complex case study—Trusted Computing Group, a previous study which had to cope with complexity in the absence of a systematic method. The rules were translated into ConceptBase, a

[2] We use italics to highlight the first mention of a concept in a section. In most cases, we do not highlight the same element again in the same chapter.

deductive object base supporting Telos data models, query classes and tested for validity.

1.4 Related Work

When real-world applications increase in size and complexity, the various models that try to abstract the applications grow accordingly. Diagrams serve as the vehicle of communication and comprehension of these models, and "the usefulness of any diagram is inversely proportional to the size and model depicted" (Feldman and Miller 1986). Not surprisingly, all modeling techniques—whether intended to model concepts, processes, states, or intentions—experienced scalability problems. Solutions to these problems had been developed by various research and industry groups to enhance communication among analysts, designers, and domain experts; to coordinate efforts contributed by distributed teams; and to manage large and complex projects using qualitative guidelines.

In this section, we first summarize the approaches taken in techniques closely related to i*—KAOS and EKD. KAOS is a goal-oriented requirements acquisition process (Lamsweerde 2003), and EKD is an enterprise knowledge modeling process that embraces goal- and agent-oriented elements (Bubenko et al. 2001). We also survey some well-established modeling techniques in their approaches to dealing with large-scale applications. These well-established techniques include Conceptual Models (Feldman and Miller 1986; Garlson et al. 1990; Harel 1988), State-Chart diagrams (Harel 1988), and the SADT approach. Some of these techniques have been adapted to modeling frameworks such as IDEF—the NASA standard, and UML—the de-facto industrial standard for object modeling.

1.4.1 Scalability handling in KAOS and EKD

Neither KAOS (Lamsweerde 2003) nor EKD (Bubenko et al. 2001) have claimed to have any problem with scalability, including their built-in diagrammatic representation of the models. One reason for the smooth process is that KAOS and

EKD have simpler semantics than i*, since both allow only “AND” and “OR” decomposition of a goal. Thus, the corresponding goal model follows a strict tree structure, which can be easily expanded or contracted at each node. Partial details of a model can always be obtained by selecting a sub-tree, and the connections to the rest of the model are maintained by the edges between parent node and its off-springs. The i* framework (Yu 1994), on the other hand, encompasses richer semantics at the meta-level by allowing cross-relationships among elements and, therefore, its diagrammatic form exhibits a network graph structure. Typically, it is more complicated in separating elements in a network graph than in a tree structure.

Despite the major differences in meta-level concepts, KAOS, EKD and the proposed view extension share some common strategies in terms of project management. These strategies include organizing a project into sub-models (term used in KAOS and EKD) or views (term used in this thesis), introducing hierarchies to modeled contents, and applying queries to facilitate information access.

Both KAOS (Lamsweerde 2003) and EKD (Bubenko et al. 2001) have multiple sub-models, each focusing on a specific perspective, and each grouping a set of closely related meta-concepts. For example, there are goal centered models to address stakeholder intentions, process models to address dynamic issues, and agent models to address agent responsibilities. In the first part of our view extension, we followed a similar approach and categorized the meta-level concepts in the i* framework into four groups, which we call views. Views differ from sub-models in that our view extension enforces strict consistency among different types of views that are derived from the same underlying i* model. Changes in the underlying model shall be reflected in all related views. Sub-models in KAOS or EKD are typically constructed separately and, thus, are loosely coupled.

KAOS uses supports from its GRAIL tool (Lamsweerde 2003) to preserve model consistency and maintain one hierarchy for each type of modeled elements including concept, diagram, and model. Each entry in any of these hierarchies is uniquely identified by a combination of their type and name. EKD achieves a similar

functionality in its KETH tool (Bubenko et al. 2001) by introducing hierarchies to the repository of knowledge. Since these hierarchies might be built by different human users, Janie et al. suggest that synonyms be replaced by a common (unique) term throughout the entire organization (Bubenko et al. 2001). In the second part of the view extension, hierarchies of views are introduced. These hierarchies are visualized in a built-in type of diagram, which we call *view map*, offered by the extension. We suggest each view be identified with a unique ID. We provide basic guidelines for building the hierarchy according to view types and the view decomposition procedure. But hierarchies in KAOS and EKD depend completely upon human decision and vary from project to project, so there lack reusable guidelines.

Both GRAIL and KETH (tools for KAOS and EKD) provide text search engines. The search engine is to help users locate specific information without having to browse the whole hierarchy. In our view extension, selection rules are formulated in First Order Logic for each view, and they are Telos-compatible. These rules select modeled elements from an i* model based on their types, which correspond to i* meta-level concepts. Thus, our solution can be fully automated.

In brief, even though KAOS and EKD are considered more as requirements engineering (RE) processes, and i* is considered as RE notations, when comes to scalability issues, they do share common approaches as far as managing a real-world project is concerned.

1.4.2 Scalability handling in Object-Oriented and SADT modeling techniques

Over the years, research on scalability-related problems has been conducted on functional modeling (IDEF0 1993), conceptual schema modeling (Feldman and Miller 1986; Harel 1988; Garlson et al. 1990; Gandhi et al. 1992; Campbell et al. 1996), and dynamic feature modeling techniques (Harel 1988; Damm and Harel 2001; Douglass 2003). Each technique has built-in meta-level concepts on which a

set of well-defined rules relies to abstract important information from details. Applying these rules enhances the capability of dealing with large complex models by a specific approach.

Our view extension is inspired by these early researches mentioned in the previous paragraph. The influences appear in three major directions. First, views of i* are represented (graphically) and decomposed in a similar manner as of IDEF0. Next, the two-level abstraction offered in the original i* framework conforms to what was proposed in the higraph-based visual formalization. Finally, focusing on representation is the approach embraced by both this thesis and other conceptual modeling researches (Feldman and Miller 1986; Garlson et al. 1990; Campbell et al. 1996; Castano 1998).

IDEF0, derived from Structured Analysis and Design Technique (SADT), is a well-formed graphical language that focuses on functional modeling of a system (IDEF0 1993). Each IDEF0 model is generated by decomposing a single system function step-by-step, and scalability issues are addressed by a set of rigorous and precise rules along this decomposition process. Auxiliary techniques—such as a *consistent naming convention* and a *reference structure*—are applied. The former mitigates reader confusion among various elements in the model, while the latter provides an overview of a project and allows quick access to a reader-interested part. This research follows the same approach by introducing a view extension to i*, which provides built-in support for a reference structure over the views. The reference structure follows a tree-like topology, and each node in the reference structure corresponds to a view (in i* view extension) or a diagram (in IDEF0). Every node should be uniquely referenced across the entire application, and each may have parent or child nodes according to the reference structure.

Even though the fact is not explicitly stated, influences from the *higraph*-based visual formalism presented in (Harel 1988) can be found in most conceptual schema (Garlson et al. 1990; Gandhi et al. 1992; Campbell et al. 1996) and dynamic feature modeling techniques (Damm and Harel 2001; Douglass 2003). This visual formalism

introduces hierarchies into *flat* models. In a higraph-based model, blobs denoting elements at a certain level of abstraction are connected by hyperedges – implying connecting multiple basic modeling elements. In the application provided in (Harel 1988), blobs are mapped to states, and hyperedges are mapped to events. A state, or parent blob, can contain sub-states, or sub-blobs; this semantic makes it possible to introduce hierarchy into state-charts. Later, Harel extended this approach to Live Sequence Charts (LSC) (Damm and Harel 2001). Both approaches were adopted by UML in resolving scalability issues (Douglass 2003). Similarly, in (Garlson et al. 1990), the concepts of complex entity, complex attribute, and complex relationship were defined to introduce hierarchy into a flat E-R model. A suitable analogy would be the complex entities and attributes to parent blobs, and complex relationships to hyperedges. The original i* framework (Yu 1994) applied a 2-level abstraction hierarchy over i* models. Actors in the Strategic Dependency (SD) view can be treated as a parent blob which contains internal elements that are shown only in the Strategic Rationale (SR) view. Contribution-links appearing in the SD view are hyperedges in that they may combine multiple links from different internal elements towards some same external elements.

Conceptual schema, such as class diagrams and ER charts, are extensively used for modeling data. Algorithms (Feldman and Miller 1986; Campbell et al. 1996; Castano 1998) and proofs (Garlson et al. 1990) were employed to explore possible means in abstracting the flat-structured conceptual models into a nested style. Authors of the methods claim that they took a “reverse-engineering” approach by focusing on reformulating an existing model rather than constructing a new one. Our view extension follows a similar philosophy. We reduce models in a “flat” manner and do not introduce abstract elements in views, yet other approaches try to define abstract elements (at a higher abstraction level) that correspond to some basic elements (at the flat structure level). Moreover, our selection rules are based purely on the types of i* meta-concepts and can be fully automated, while the other approaches require intensive human interaction (Feldman and Miller 1986).

In brief, our view extension presented in this thesis is influenced by the scalability-handling techniques applied and proposed in a number of existing modeling methods. Yet we have encountered different challenges and thus led to adaptations. One reason is that i* embraces a richer set of meta-concepts so that meta-model driven rules can be defined to partition elements according to their types. Another is i* introduces intentional and social aspects to a model, which are not accommodated in other formalities.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 reviews the original i* framework presented in Yu's 1994 thesis, and the formal constructs used in the Organization Modelling Environment (OME) tool (OME 2003). Chapter 3 presents the first part of the proposed view extension, which is a reformulation of the i* framework based on a consolidation of the changes made to i* over the past 10 years. Graphical notations of new concepts are synthesized from previous literature of our research group, and formal constructs of some newly introduced concepts are presented for the first time. Chapter 4 presents the second part of the proposed view extension, which is described from three aspects: its features and the view map; its formal constructs; and critical concepts related to the selection rules. Chapters 5 to 7 describe in detail selection rules associated with each view. Examples from the LAS case study are presented to illustrate the use of each type of view. Chapter 5 focuses on Actor Class views; Chapter 6, on Strategic Dependency views; and Chapter 7, on Strategic Rationale views. Chapter 8 validates the proposed extension over the existing Trusted Computing Group (TCG) case study, and Chapter 9 draws conclusions and proposes relevant future work.

2 The Original i* Framework

In this chapter, we summarize the modeling features of the i* framework and review its formal constructs from Yu (Yu 1994). Examples from the London Ambulance Service (LAS) case study are cited to illustrate various meta-level concepts.

2.1 Modeling Features

2.1.1 The Strategic Dependency Model

Actors are strategic in i*: they have "motivations, intents, and rationales behind [their] actions" (Yu 1994). An actor can be further differentiated into *roles*, *agents*, and *positions*. A role is "an abstract actor embodying expectations and responsibilities." An agent represents a physical actor – human or machine – who can *play* different roles. A position represents a group of responsibilities that can be *occupied* by one agent; as well, a position can *cover* more than one role. There is also a defined aggregation (PART) relationship among the same type of actors, and an instantiation (INSTANCE) relationship between two agents. The graphical notations of the two relationships were briefly introduced in one example (Yu 1994). Figure 2.1-1 shows graphical notations of various forms of actors. A plain circle (e.g., **Ambulance Crew**[3]) denotes a (plain) actor; a circle with a curved line across the bottom denotes a role (e.g., **Remover [Duplicated IncInfo]**); a flower shape denotes a position (e.g., **Resource Allocator**); and a circle with a bar across the top denotes an agent (e.g., **Incident Reviewing Module**).

[3] We use bold to highlight the first mention of an element in the models. In most cases, we do not highlight the same element again.

The Strategic Dependency (SD) model is used to express the "intentional description of a process in terms of a network of dependency relationships among actors." Dependency relationships are represented by dependable elements, and actors depend on one another for *goals* to be achieved, *tasks* to be performed, *softgoals* to be satisficed, and *resources* to be furnished. The symbol "D" in the *dependency link* indicates the direction of dependency. Yu also "call[s] the depending actor the *depender*, the actor who is depended *dependee*[, and] the object around which the dependency relationship centers *dependum* (Yu 1994). Figure 2.1-1 shows the graphical notation of the different dependency types.

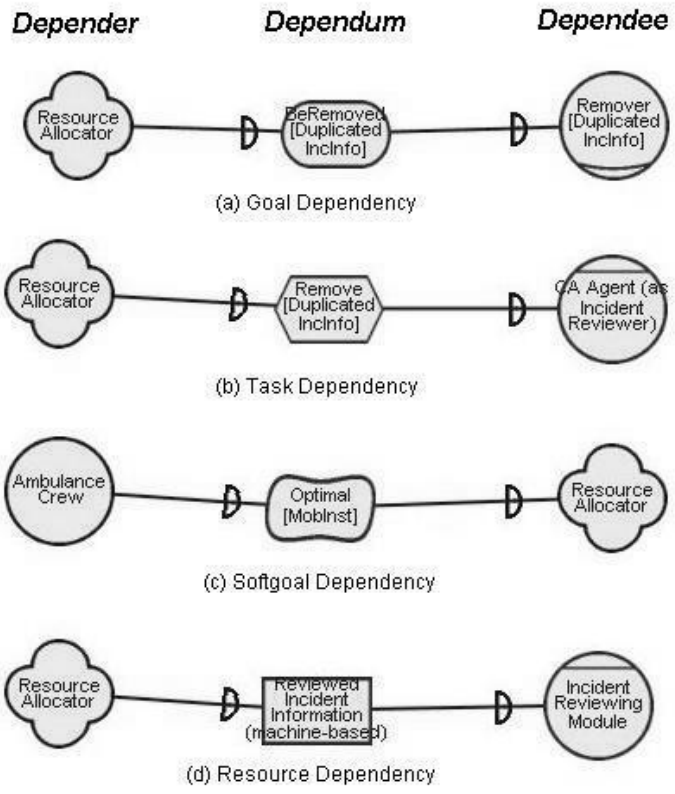


Figure 2.1-1 Dependency types

Figure 2.1-2 shows a partial SD model from the LAS case study. This model shows the dependency relationship among actors **Resource Allocator, Ambulance Crew, Incident Reviewer, and LAS Management**. Relationships among these actors are also presented. For example, either a Resource Allocation

Module or a Human Resource Allocator *occupies* the position of Resource Allocator. The latter depends on the Ambulance Crew to ensure the **Accuracy** of Ambulance Information (AmbInfo), and, in turn, the Ambulance Crew depends on the Resource Allocator to provide **Optimal** Mobilization Instruction (**MobInst**).

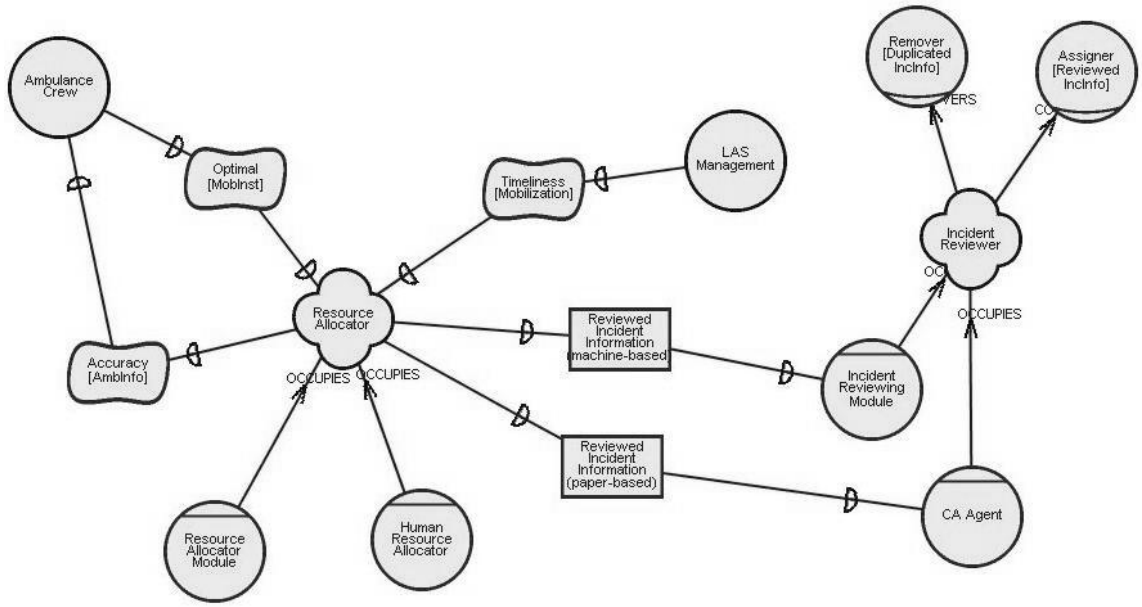


Figure 2.1-2 A partial SD model from the LAS case study

2.1.2 The Strategic Rationale Model

The **Strategic Rationale** (**SR**) model is aimed to "provide the intentional description of processes in terms of process elements and the rationales behind them." This implies that the layout of the reasoning structure internal to an actor, based on inter-actor relationships presented in the SD model, is represented in the SR model. In this internal structure, intentional elements – goals, tasks, resources, and softgoals – are connected by intentional links (Yu 1994).

Two classes of intentional links are defined in (Yu 1994). *Task decomposition* link, denoted by ⊣, expresses "a task in terms of its decomposition into sub-components." (Yu 1994) distinguished (semantically but not graphically) among four types of *task decomposition* links according to the type of sub-components.

A task can be decomposed to a sub-goal via a *subgoal* decomposition link, to a sub-task via a *subtask* decomposition link, to a sub-resource via a *resourceFor* decomposition link, and to a softgoal via a *softgoalFor* decomposition link. For example, in Figure 2.1-3, task **Provide [Optimal MobInst]** is decomposed to softgoals **Accuracy [AmbInfo]** and **Accuracy [IncInfo]** via two softgoalFor links, respectively.

Several types of *means-ends* links, denoted by ⇒, were also defined and the "arrowhead points from the means to the end." A goal specified as the end can be achieved by means specified as tasks through *goal-task* means-ends links (GTLink). For example, goal **BeCollected [IncInfo]** can be achieved by information passed either task **By database or network** or task **By paper-based forms** (Figure 2.1-3). Similarly, a resource specified as the end can be furnished by means specified as tasks through *resource-task* links (RTLink). A softgoal can be satisficed by means specified as tasks or softgoals through *softgoal-task* (STLink) and *softgoal-softgoal* (SSLink) links, respectively. A softgoal-link can contribute positively (denoted by **+**) or negatively (**-**) to the softgoal specified as the end, and they are shown graphically as curved arrows. For example, task **Provide [Optimal MobInst]** contributes positively to softgoal **Optimal [MobInst]** through the softgoal-task (means-ends) link, and softgoal **Timeliness [Arrival Location]** contributes positively to softgoal **Timeliness [Service]** through a softgoal-softgoal (means-ends) link (Figure 2.1-3). The framework also allows task-task links that specified tasks as both the end and the means. (Yu 1994)

Figure 2.1-3 shows the process elements (activities, plans) and initiatives behind the intentions of position **Resource Allocator**. This internal structure can help us select among alternative activities or plans. For example, achieving the top-level goal **BeCollected [IncInfo]** requires only one of the two alternatives—collect incident information **By paper-based forms** versus **By database or network**—being performed. Selecting the former will result in the top-level softgoal **Timeliness [Mobilization]** being harmed – via the negative contribution

link from the former, while selecting the latter will not. If timeliness is a major concern of Resource Allocator, the latter alternative (collect incident information by database or network) thus needs to be chosen. We see from the example that by using the SR model, users may obtain a better understanding of how the top-level goals can be achieved, and how these goals relate to each other.

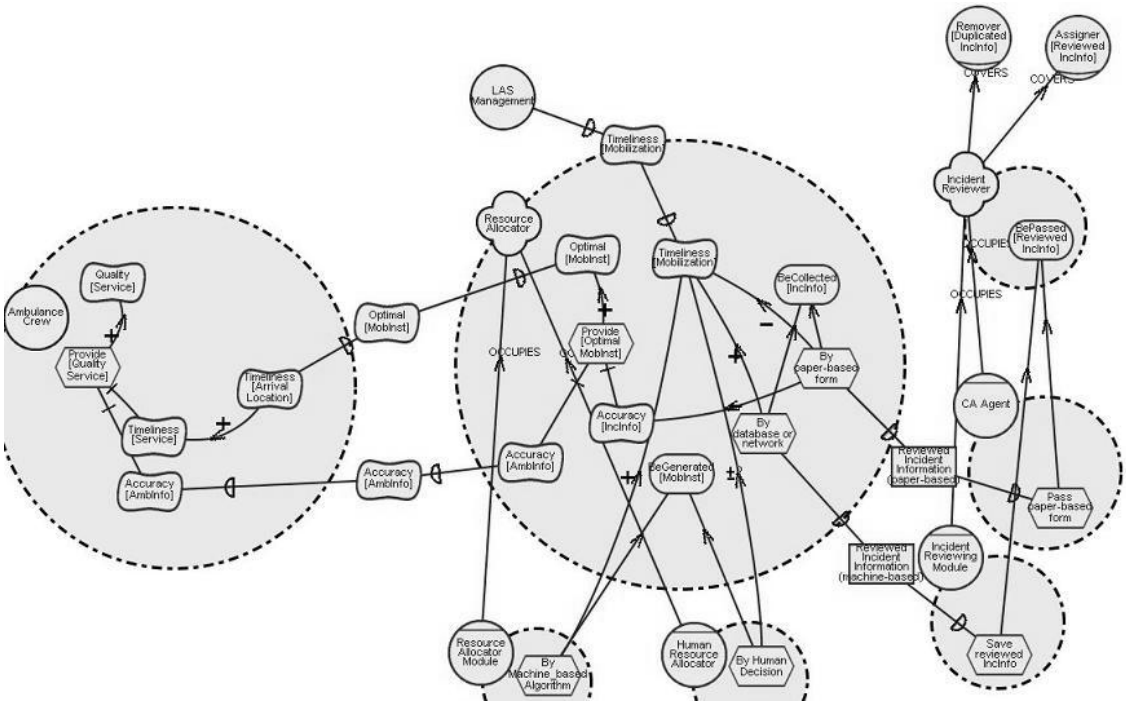


Figure 2.1-3 A partial SR model from the LAS case study

2.2 Representational Constructs

Meta-level concepts of the i* framework, and their relationships, are embedded into the conceptual modeling language Telos (Koubarakis et al. 1989), which results in "an object-oriented representational framework with classification, generalization, aggregation, attribution, and time" (Yu 1994). Two levels of classes are involved in this formalization: Concepts from the i* framework are defined at the meta-class level in Telos, and *domain class* are defined as instances of some meta-class and at the simple-class level (Yu 1994). Figure 2.2-1 shows the definition of the meta-class AgentElementClass and one of its instances at the domain level, specified as a simple class. Text quoted by

%% are comments. In order to distinguish the objects internal to an actor, we prefix such objects with the acronyms of actors. For example, we prefix softgoal **Quality [Service]** inside agent Ambulance Crew as **AC_QualityService**, where AC is the acronym for Ambulance Crew. We apply this naming convention throughout this thesis.

```
% Telos representation of concept agent %
TELL MetaClass AgentElementClass … WITH
       attributes
              name : String;
              children: IntentionalElementClass
END

% Telos representation of domain class AmbulanceCrew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass ISA
AmbulanceCrew_Actor WITH
       name
              displayName : “Ambulance Crew”
       children
              : AC_QualityService
              : AC_TimelinessService
              : AC_TimelinessArrivalLocation
              : AC_AccuracyAmbInfo
              …
END
```

Figure 2.2-1 Definition of meta-level class AgentElementClass and a domain class that instantiates it denoting the class of agent Ambulance Crew from the LAS case study

However, the formal constructs shown in Yu’s original thesis and the Organization Modelling Environment (OME) tool differ in class and attribute design. For example, Yu formulated a goal dependency using an instance of GoalDependsClass, while OME using one instance of GoalElementClass and two instances of DependencyLinkClass. The OME tool style conforms to Yu’s original proposal since the two are equivalent in semantics: all i* semantics are naturally implemented in the OME tool. We favor the OME tool style in that it is

widely used and provides a measure to verify the validity of the models so that human interference can be minimized.

2.2.1 The Strategic Dependency Model

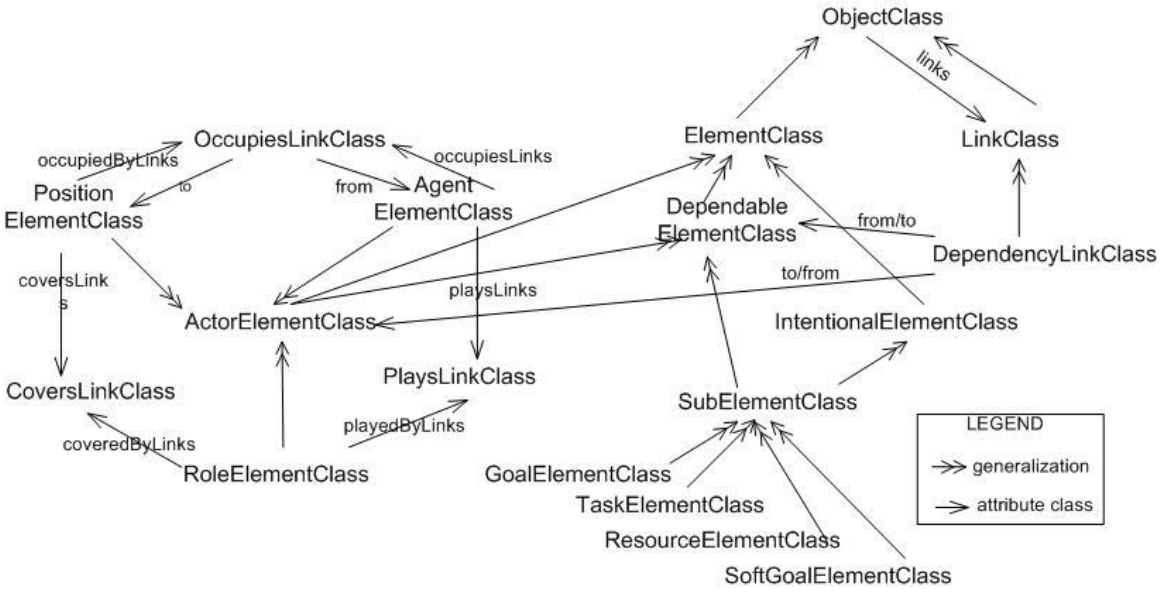


Figure 2.2-2 A partial meta-model of the SD model in Yu's thesis

Figure 2.2-2 shows a partial meta-model of the SD model adapted from Yu's original thesis. There are two categories of objects in the SD meta-model: the **Element**(meta)**Class** and the **Link**(meta)**Class**. An instance of LinkClass (e.g., **`AC_TALtoOptimalLink`** in Figure 2.2-3) shall have some instances of ElementClass as its two critical attributes *from* and *to*. The instance of ElementClass that is specified as *from* (e.g., **`AmbulanceCrew_Agent`**) denoting the source element from where the link starts, and similarly *to* where the link ends (e.g., **`AC_OptimalMobInst_RA`**). An instance of ElementClass (e.g., **`AmbulanceCrew_Agent`**) may have some instances of LinkClass (e.g., **`AC_TALtoOptimalLink`**) as its attribute *links*.

Figure 2.2-3 shows the formal representation of some of the elements that appear in Figure 2.1-2. Text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation.

```
%the actor Ambulance Crew%
TELL SimpleClass AmbulanceCrew_Actor IN AgentElementClass WITH
       name
              displayName : “Ambulance Crew”
       links
              : AC_TALtoOptimalLink
END

%the position Resource Allocator%
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
       links
              : OptimaltoOptimalLink_RA
                     …
END

%The dependency link from Ambulance Crew to the softgoal dependum Optimal
[MobInst]%
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
: AmbulanceCrew_Agent
to
: AC_OptimalMobInst_RA
END

%The dependency link from the softgoal dependum Optimal [MobInst] to Resource
Allocator%
TELL SimpleClass OptimaltoOptimalLink_RA IN OutgoingDependencyLinkClass WITH
       from
: AC_OptimalMobInst_RA
to
: ResourceAllocator_Position
END

%The softgoal dependum Optimal [MobInst]%
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
       links
              : AC_TALtoOptimalLink
              : OptimaltoOptimalLink_RA
END
```

Figure 2.2-3 Representation of a partial SD model from the LAS case study

2.2.2 The Strategic Rationale Model

In Yu's thesis, the meta-model of SR includes every segment shown in the SD model plus those shown in Figure 2.2-4. This meta-model conforms to the intuitive description of the SR model in Section 2.1.2.

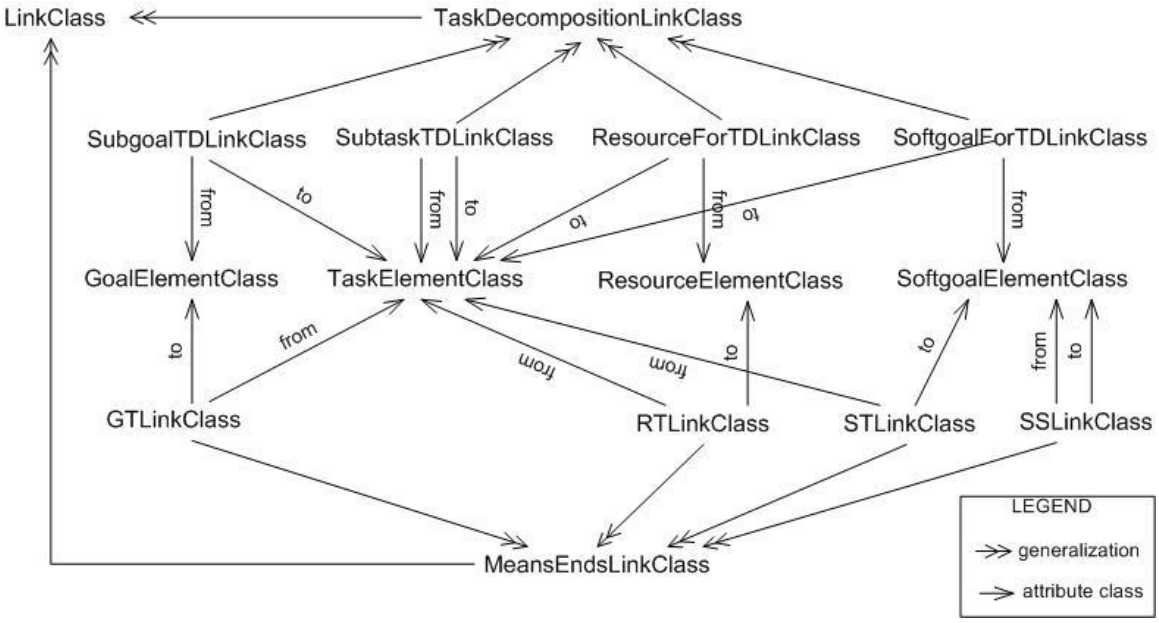


Figure 2.2-4 Partial meta-model for the SR model

Figure 2.2-5 shows the formal representation of some of the elements that appear in Figure 2.1-3. The text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation.

In SR models, both the *from* and *to* attributes for an instance of DependencyLinkClass (e.g., **`AC_TALtoOptimalLink`**) can represent some instances of IntentionalElementClass (e.g., `from` **`AC_TimelinessArrivalLocaltion`** `to` **`AC_OptimalMobInst_RA`**), while in the SD model, one of them must be an instance of ActorElementClass (e.g., the same link from **`AmbulanceCrew_Agent`** `to` **`AC_OptimalMobInst_RA`**).

```
%actor Ambulance Crew%
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      name
```

```
              displayName : “Ambulance Crew”
       children
              : AC_QualityService
              : AC_TimelinessService
              : AC_TimelinessArrivalLocation
              : AC_AccuracyAmbInfo
              …
END

%softgoal Timeliness [Arrival Location] inside boundary of actor Ambulance Crew%
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass WITH
       parent
              : AmbulanceCrew_Agent
       links
              : AC_TALtoOptimalLink
              : AC_TALtoTS_AndContributionLink
              …
END

%position Resource Allocator%
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
       children
              : RA_OptimalMobInst
              : RA_TimelinessArrivalLocation
              : RA_AccuracyAmbInfo
              : RA_BeGeneratedMobInst
                     …
END

%agent Resource Allocation Module%
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass WITH
       children
              : RA_BeGeneratedMobInst_ByAlgorithm
END

%agent Human Resource Allocator%
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
       children
              : RA_BeGeneratedMobInst_ByHumanDecision
END
```

```
%The dependency link from softgoal Timeliness [Arrival Location] in the boundary
of actor Ambulance Crew to the softgoal dependum Optimal [MobInst]%
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
       from
              : AC_TimelinessArrivalLocaltion
       to
              : AC_OptimalMobInst_RA
END

%the dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside boundary of position Resource Allocator%
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
       from
              : AC_OptimalMobInst_RA
       to
              : RA_OptimalMobInst
END

%softgoal dependum Optimal [MobInst]%
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
       inDeplinks
             : AC_TALtoOptimalLink
       outDepLinks
             : OptimaltoOptimalLink_RA
END

%softgoal Optimal [MobInst] inside the boundary of position Resource Allocator%
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
       parent
              : ResourceAllocator_Position
       links
              : OptimaltoOptimalLink_RA
       …
END
```

Figure 2.2-5 SR presentation in Telos

2.3 Summary

This chapter outlines in brief features of the original i* framework described by Yu (Yu 1994). These features are graphically presented using two models: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model.

Meta-level concepts such as “actors” and “dependencies” are introduced in the SD model, while intentional links such as “means-ends” and “decomposition” are explained in the SR model. Graphical notations of these concepts are illustrated using samples from the LAS case study.

We omit the concept of *dependency strength* originally presented by Yu, because this concept does not play a role in our view extension, nor was it widely referenced in previous literatures. Nevertheless, dependency strength could be used in the future as a criterion in simplifying complex i* models.

Formal constructs of the meta-level concepts were adapted into Telos using the OME tool style, which differs from what was presented by Yu (Yu 1994). Sample domain classes from the LAS case study were cited in demonstrating these formal constructs.

3 Reformulating the i* Framework Using the Concept of View

Over the past 10 years, new concepts were introduced to the i* framework and existing concepts were refined. The definition of the Goal-oriented Requirements Language (GRL) framework elaborates on the incorporation of concepts from the NFR framework into the i* framework, anticipated by Yu (Yu 1994). The latest GRL version was presented in 2003 (GRL 2003).

Besides the definition of GRL, one major milestone was the separation of the actor diagram from the SD diagram, another was the release of the Organization Modelling Environment (OME) tool which implemented the meta-model of i*.

Yu (Yu 1994) formally proposed three specified types of actors – roles, agents, and positions—and three intentional links—plays, covers, occupies. Two other types of links—Instances and PART—were well established in OO modeling, so Yu just gave their graphical notation yet not emphasized. It was not until 1997 that the concept of *agents* (one type of specified actors) was explicitly depicted (Yu 1997; Chung et al. 1997). Liu and Yu first emphasized graphical notations for role, agent, position, and the links among them (Dubois et al. 1998). They refined this line of concepts and their graphical notations, built the specified actors hierarchy, and formalized graphically three types of links (is-A, INS, and is-Part-of) among these specified actors (Yu and Liu 2000). However, in their 2000 publication, various types of actors and the three types of links were shown in the SD model. In 2002, specified actors and the links among them were first shown separately in a so-called Actor Diagram (Liu et al. 2002).

The OME tool (version 2) was released in 1998; OME version 3 (the current version is 3.13) supports GRL, i*, NFR, and other kinds of frameworks. Some new graphical notations that had not appeared in publications were added

recently. These new notations smooth the merging of NFR approach into GRL. The GRL framework implemented in the current OME tool supports specified actors and their corresponding links, which are initially specified in i* but omitted in the standard submission of GRL. These effects result in the differences in modeling features between the OME tool and Yu's original thesis.

Changes made to i*, as discussed in previous paragraphs, appeared in various literature produced by the i* research group. Lacking adequate explanations, these changes confused readers unfamiliar with the concepts. For example, such terms as diagram and model were often interchanged (meaning some partial i* model) in different publications, and diagrams (models) were normally presented in an ad-hoc sequence convenient to the specific publication.

In this chapter, we attempt to consolidate what has happened over the past 10 years. The main objective is to collect, synthesize and organize concepts scattered throughout existing literatures. Minor adjustments are made to existing concepts to improve accuracy (of each of them) and consistency (among all of them). As a first step, modeling constructs are organized in four types of views, in correspondence to the two types of models (SD and SR) by Yu (Yu 1994). This paves the way for scalability issues to be addressed in subsequent chapters.

Section 3.1 summarizes the reformulated framework and briefly justifies our view extension; Section 3.2 discusses the reformulated i* framework in detail; Section 3.3 presents the formal constructs of the reformulated i* framework; and Section 3.4 discusses the relationships among the four types of views.

3.1 Introduction

We reformulate the i* framework by refining the concept of *model* and by introducing the new concept of *view*. Initially, SD and SR are called "models" by Yu (Yu 1994), but in this thesis we reserve the term *model* for the collection of i* objects structured according to i* syntax and semantics. A model contains information in both SD and SR, and we call a domain i* model *the baseline model*. In most cases, an i* *model* describes a particular configuration (e.g., from

one type of viewpoint, at a certain period of time, and for a specific project) among organizational actors.

A *view* is a partial presentation of that type of configuration. In this sense, SD and SR are called "views" in our extension. In fact, the extension distinguishes among four types of views: an **Actor Class (AC)** view for focusing on various forms of actors and the associations among the different forms of each actor, a **Strategic Dependency (SD)** view focusing on inter-actor dependency relationships, a **Strategic Rationale (SR)** view focusing on "the rationales that actors have about adopting one configuration or another" (Yu 1994), and an **Evaluation Results (EVLR)** view helping in the decision-making process over alternative system configurations.

We reformulate the baseline model in this way for the following reasons.

First, the SD view is an abstract form of the SR view. Inter-actor dependencies and external links and elements in the SD view can be obtained from its corresponding SR view. From the formal construction of i* models, we can affirm that the SD and SR views share a majority of concepts in their meta-models, with SR having some extra concepts representing internal rationale. Thus, any SD view can be obtained by collapsing actors' internal structures in the corresponding SR view, and each collapsed actor in the SD view inherits all the external dependencies that are originally connected to its internal elements. In this sense, we consider it more appropriate to treat them as views that project over the same model instead of sub-models.

Second, a distinguished AC view makes actor analysis easier. In most of the early literature, the SD view was used to identify stakeholders and perform basic actor analyses within an organization. Questions such as "How does a plain actor map to a specified one?" and "What are the relationships among the specified ones (*actor associations*)?" were not emphasized. It appeared straightforward with the examples shown in early literature, when there was no need to distinguish among different forms of actors. Yet social configuration for a

medium-sized organization (e.g., 500 employees) can increase in complexity and, thus, accommodation of actor associations (e.g., 300 “plays”, “covers”, or “occupies” links) in the initial SD models becomes difficult. Showing dependency relationships for multiple specified forms of the same actor (e.g., position Resource Allocator and agent Resource Allocator Module) at the same time also appears difficult. Thus, we decide to abstract these sets of information into a new type of view—Actor Class. Separation of the actor associations from dependencies does not affect our analysis. The former focuses on understanding which set of actors have something in common; the latter, on reflecting how an organization functions among the inter-actions of actors who basically do not share internal rationales.

Finally, the Evaluation Results (EVLR) view accommodates concepts imported from the NFR framework. After the collaboration of i* and NFR, a model evaluation process employing a qualitative label propagation algorithm was implicitly adopted by i*. In accordance with this action, we distinguished the EVLR view to present the results of the evaluation process. The evaluation process uses the SR view to run the algorithm, so each EVLR view is built on top of its corresponding SR view. However, users may use the same SR view to perform different evaluations that differ in various assumptions, so one SR view normally corresponds to a set of EVLR views.

3.2 Realigned Graphical Notations

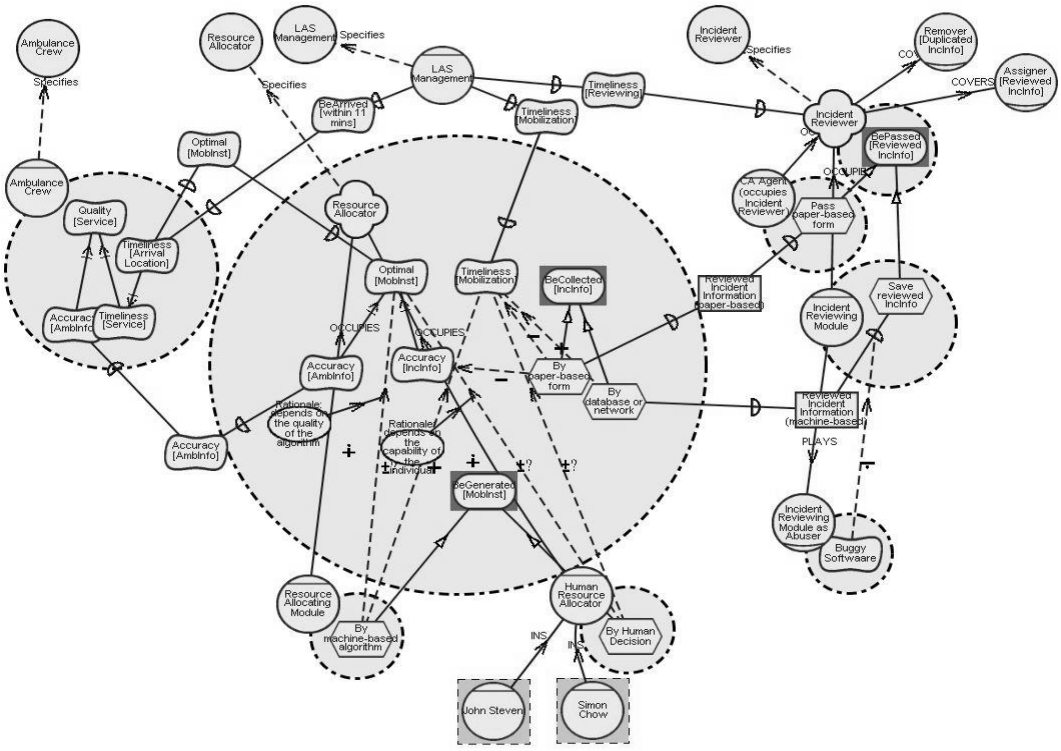


Figure 3.2-1 A partial baseline model showing some structures related to plain actor Ambulance Crew, LAS Management, Resource Allocator, and Incident Reviewer from the LAS case study

The baseline model, which consists of i* domain classes, serves a centralized repository for information elicited for a specific configuration of an organization. Because multiple evaluation results can be obtained from the same baseline model structure, each baseline model contains one basic model structure and several sets of evaluation results that are distinguished by initial values or human decisions for label assignments. For simplicity, we refer to the basic model structure as the baseline model as long as no confusion will incur. Figure 3.2-1 shows a sample of the baseline model structure from the LAS case study. We use this sample as the baseline model of the sample views shown in Section 3.2.1~3.2.4.

3.2.1 The Actor Class view

As defined by Yu (Yu 1994), the i* framework supports the concept of strategic actors. Actors can be *plain* or *specified*. A role, a position, an agent, or an agent instance[4] (the term "agent instance" will be discussed later in this section) is called a specified actor. A plain actor is an actor of unspecified type, i.e., the modeler does not say whether it is a role, a position, an agent, or an agent instance. Since such an unspecified actor can appear as an element in a model, we give it the special term "plain actor", to distinguish it from the general notion of actor (see Section 4.4.1 for more details). Besides, we define six relationships—*plays, occupies, covers, is-A, INS, and is-Part-of*—among actors as *actor associations* (Koubarakis et al. 1989; Yu and Liu 2000). This type of overall information forms the Actor Class (AC) view.

In addition to clarifying existing actor types and actor associations, we introduce new concepts into i* framework, and they are: two new association types—*specifies* and *complete composition*, one specified actor type *agent instance*, and the *external relationship inheritance rule* along actor associations.

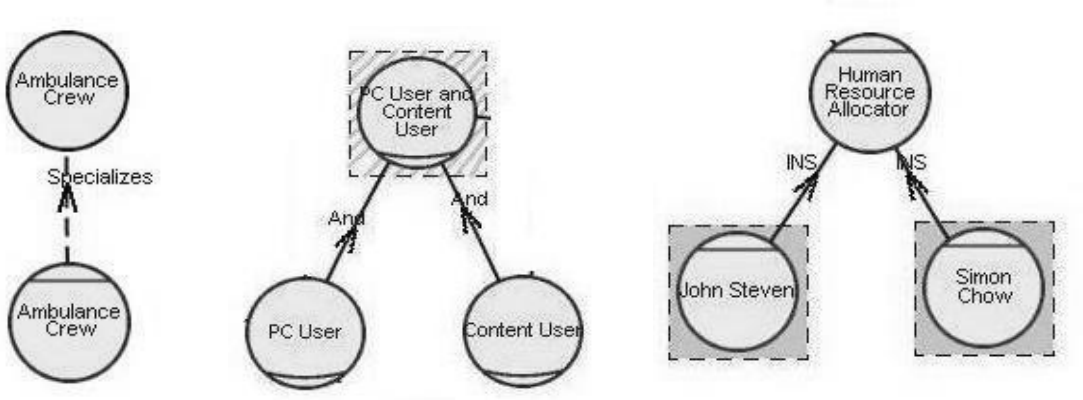
The "specifies" relationship originates from a specified actor to point to its corresponding plain actor. Graphically, it is denoted by a dashed arrow line labeled "specifies", with the arrow pointing to the plain actor (Figure 3.2-2(a)). We call the former the *direct specified actor* of the latter. This link reflects a form of generalization similar to "is-A" between a plain actor and its specified form. The "is-A" relationship, however, can only apply between actors of the same specified type. For example, the role "Government as PC User" can only specializes (via an "is-A" link) the role "PC User". The "specifies" relationship is needed in enforcing the external relationship inheritance rule between a plain and its specified forms.

---

[4] Instances of other forms of actor types such as role instance are also possible. We leave this part of semantic for future research.

The “complete composition” relationship is added as a specialized form of the “is-Part-of” relationship, which implies that the union of the *parts* is exactly the same as the *whole*. As with “is-Part-of”, this new relationship can only apply among actors of the same specified type. Graphically, it is denoted by a solid arrow line labeled “And” with the arrow pointing to the “whole” and the “whole” is highlighted using a dash-filled rectangle with dashed-border (Figure 3.2-2(b)). This graphical notation is not to be confused with the “And” contribution (Section 3.2.3), which can only apply between two intentional elements. The “complete composition” relationship applies a rigorous scope of the responsibilities and opportunities of the “whole”, basing on those of its “parts”. In other words, any property of the “whole” must be found in one of its “parts”. Therefore, a more accurate consistency can be enforced along this type of aggregation relationship.

We distinguish *agent instances* from agents in that they have different semantic implications. An agent instance reflects a domain-object level actor such as a human individual (e.g., **John Steven**), a physical organization (e.g., USA Government), a specific machine, and the like. An agent reflects the classification (at the domain-class level) of the domain-object level instances. For example, agent **Human Resource Allocator** denotes the group of individuals who are thus classified. Moreover, this change affects the syntax of the INS link. In this reformulation, only an agent instance may instantiate (via an INS link) an agent. Graphically, we distinguish an agent instance from an agent by highlighting the former using a filled rectangle with dashed border (Figure 3.2-2(c)).



(a) Specifies (b) Complete composition (c) Agent instances

Figure 3.2-2 newly introduced graphical notations

An *external relationship inheritance rule* is defined over the reformulated actor associations discussed previously in this section. The "specifies" link imply that the source (a specified actor) and the target (the corresponding plain actor) share the exact same set of external relationships. The "is-A", "plays", "occupies", "covers", and "INS" links all imply that the actor serving as the source of such a link inherits all external relationships from its corresponding target, but not vice versa. For example, in Figure 3.2-3, position **Incident Reviewer** "covers" both role **Remover [Duplicated IncInfo]** and role **Assigner [Reviewed IncInfo]**. Suppose role Remover [Duplicated IncInfo] has an external dependency G1 and role Assigner [Reviewed IncInfo] has G2, and G1 differs from G2. According to the external relationship inheritance rule, position Incident Reviewer has both G1 and G2 as external dependencies. The "complete composition" and "is-Part-of" links imply that the actor serving as the target of such a link inherits external relationships from its corresponding sets of source actors. For example, the roles **PC User** and **Content User** (source actors) are each a part of the combined role **PC User and Content User** (the target).

By applying the external relationship inheritance rules, we can specify external relationships at a single actor, and these relationships can be referenced by associated actors through an inheritance network along actor associations. By this means, redundant external relationships can be avoided in an i* baseline

model, which leads to SD views showing no redundant dependencies from one actor to some different specified forms of another actor.

Figure 3.2-3 shows the AC view projected from the baseline model shown in Figure 3.2-1. By omitting dependency links and internal elements, the diagram appears clearer and more readable. Actor associations stand out: Position **Resource Allocator** can be occupied by either a **Resource Allocation Module** or a **Human Resource Allocator**; and position Incident Reviewer covers role **Remover [Duplicated IncInfo]** and role **Assigner [Reviewed IncInfo]**.

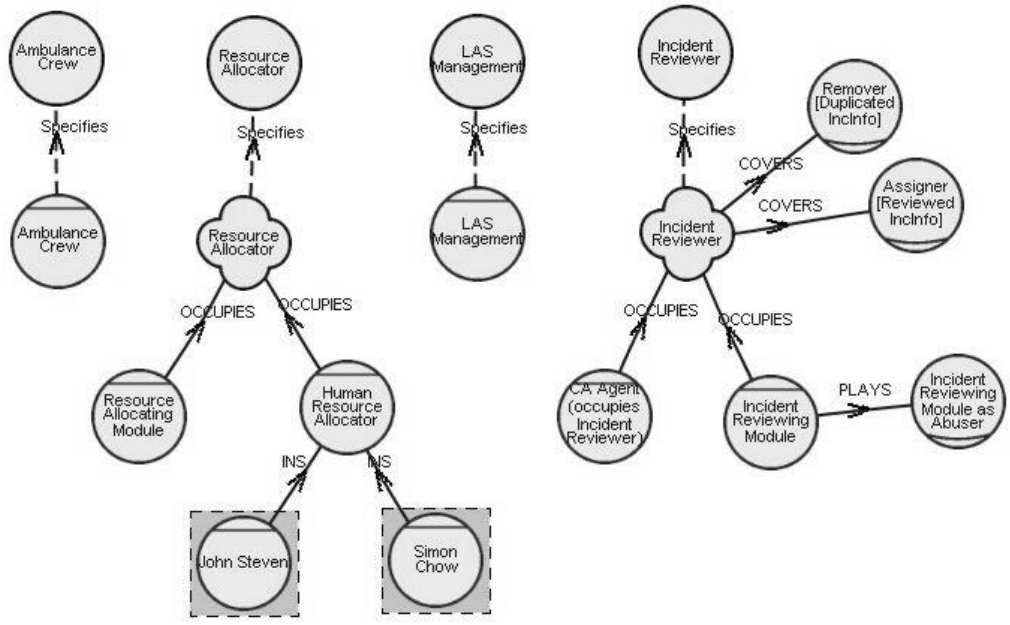


Figure 3.2-3 Sample Actor Class view from the LAS case study

3.2.2 The Strategic Dependency view

The Strategic Dependency (SD) view corresponds to the SD model described in (Yu 1994). Some minor changes originating from (Yu and Liu 2000; Liu et al. 2003) are applied, including the removal of the actor associations and the addition of contribution links that target some external elements – dependum or link. The purpose of the SD view is thus to express the "intentional description

of a process in terms of [not only] a network of dependency relationships among actors" (Yu 1994), but also to express the intertwined negative or positive effects towards those dependency relationships among actors. The details of the representation of those negative or positive effects will be discussed in the next section.

Our reformulation also introduces intentional links that end at an external element (a dependum or a link), which we call *external links* (see Section 4.4.5 for more details). In addition, since the annotations (critical, open) of dependencies are not widely emphasized in various i* modeling practices, we omit that aspect in this thesis.

Figure 3.2-4 shows the SD view extracted from the baseline model of the LAS case study (Figure 3.2-1). Position **Resource Allocator** (*depender*) depends on agent **Ambulance Crew** (*dependee*) to ensure the **Accuracy** of Ambulance Information (**AmbInfo**) (*dependum*); in turn, agent Ambulance Crew depends on the Resource Allocator to provide **Optimal** Mobilization Instruction (**MobInst**). The Resource Allocator depends on either a **CA Agent** or the **Incident Reviewing Module** to supply **Reviewed Incident Information**. If the **Incident Reviewing Module** plays an **Abuser** role, it will *hurt* (an *external correlation link*) the *incoming dependency* from the Resource Allocator.

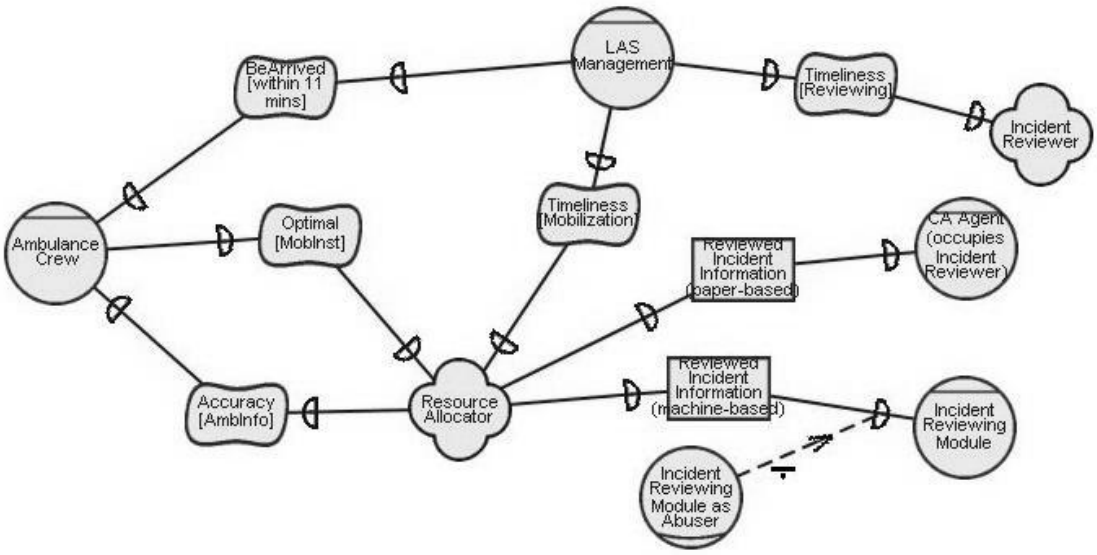


Figure 3.2-4 Sample SD view from the LAS case study

3.2.3 The Strategic Rationale view

The Strategic Rationale (SR) view experienced major changes in the graphical notations when i* evolved into GRL in 2001. Our view extension follows what was defined in (GRL 2003). GRL refined the notion of *belief* and *decision point*. It also distinguished *correlation links* from contribution links and defined *labels* for contribution and correlation links.

Although logically defined by Yu (Yu 1994), the graphical notation of *belief* was not presented until the introduction of GRL (GRL 2003). As stated in GRL, "[b]eliefs make it possible for domain characteristics to be considered and properly reflected into the decision making process, and hence facilitating later review, justification and change of the system, as well as enhancing traceability." Since beliefs are held by some stakeholders, it shall not appear as a dependum and, hence, shall never appear in the SD view. Belief and the other four that appear in the SD view—goal, task, softgoal, resource—are called *intentional element* in total. The graphical notation of a belief is shown in Figure 3.2-5.

Figure 3.2-5 Graphical notation of belief

GRL (GRL 2003) distinguishes among four classes of *intentional links*. A goal (ends) can be achieved by different tasks (means), and this relationship is expressed by the *means-ends* link (the original GTLink). A task (or goal) can be decomposed into sub-components—sub-goals, sub-tasks, sub-softgoals, and sub-resources. This relationship is expressed by the *decomposition* links. This link type remains the same as what was initially defined by Yu (Yu 1994). *Contribution* (combination of the original STLink and SSLink) and *correlation* (newly added type) links are used to express a direct or indirect effect from a

descendent to an ancestor softgoal. Graphical notations of the four classes are shown in Figure 3.2-6.

(a) Means-ends (b) Decomposition (c) Contribution (d) Correlation

Figure 3.2-6 Intentional link types

Moreover, an effect could be positive (*make*, *help*, or *some+*), equal, unknown, or negative (*break*, *hurt*, or *some-*). In GRL (GRL 2003), *make* implies a sufficiently positive effect; *help*, a partially positive effect; and *some+*, a positive effect with unknown extent. Similarly, *break* implies a sufficiently negative effect; *hurt*, a partially negative effect; and *some-*, a negative effect with unknown extent. *Equal* implies an identical effect, while *unknown* implies a possible positive or negative effect. In addition, direct effects to a softgoal could be *AND* or *OR*, meaning all the off-springs must be met or only one of the off-springs need to be met for the corresponding softgoal to be satisficed. Graphical notations of these effect labels are presented in Figure 3.2-7 for contribution links and in Figure 3.2-8 for correlation links. Alternatively, words (e.g., BREAK) can be used to label the links instead of the symbols (e.g.,).

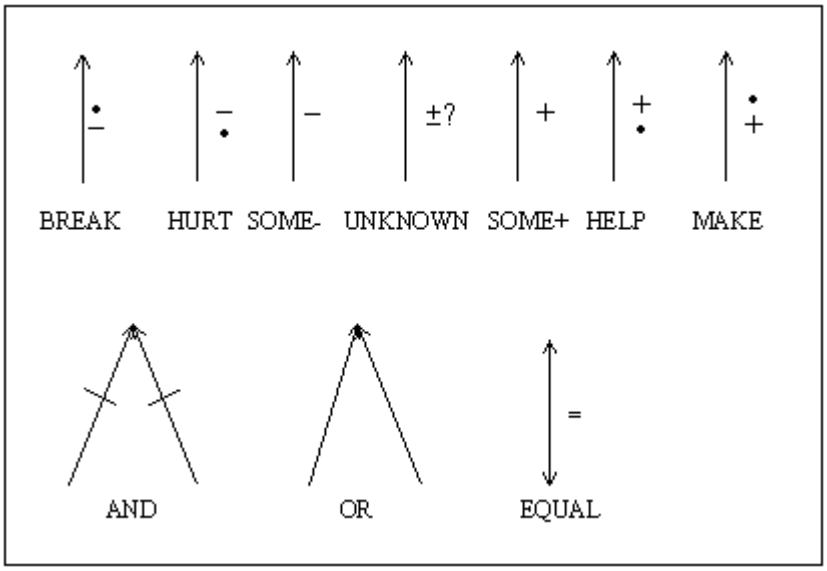


Figure 3.2-7 Effects of contribution links

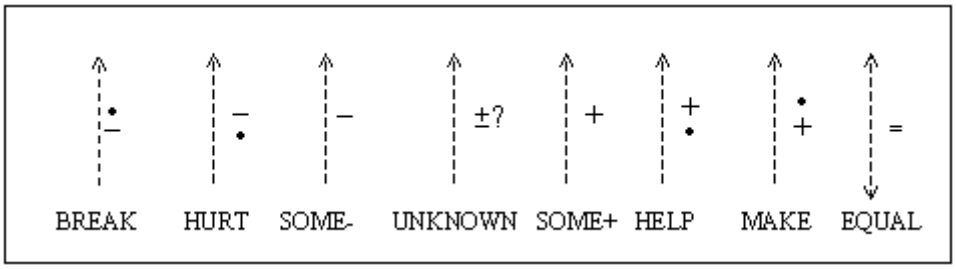


Figure 3.2-8 Effects of correlation links

Liu and Yu defines the notation of *decision point* in the i* framework (Liu et al. 2003). A decision point is a goal that requires more than one task. Graphically, it is denoted by a goal highlighted using a solid-filled solid-border rectangle. Figure 3.2-9 shows goal **BeCollected [IncInfo]** as a decision point since it can be achieved **by** using either **paper-based forms** or **machine-based mechanisms**. Since this notation does not affect our view extension, we only denote it graphically.

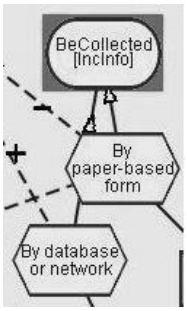


Figure 3.2-9 graphical notation of decision point

Figure 3.2-10 shows the SR view corresponding to our baseline model from the LAS case study (Figure 3.2-1). The view shows the intentional elements that are required to achieve top-level goals of the position **Resource Allocator**. We call intentional elements that reside internal to an actor as *internal elements*. For example, softgoal **Optimal [MobInst]** and **Timeliness [Mobilization]** are internal elements to position Resource Allocator.

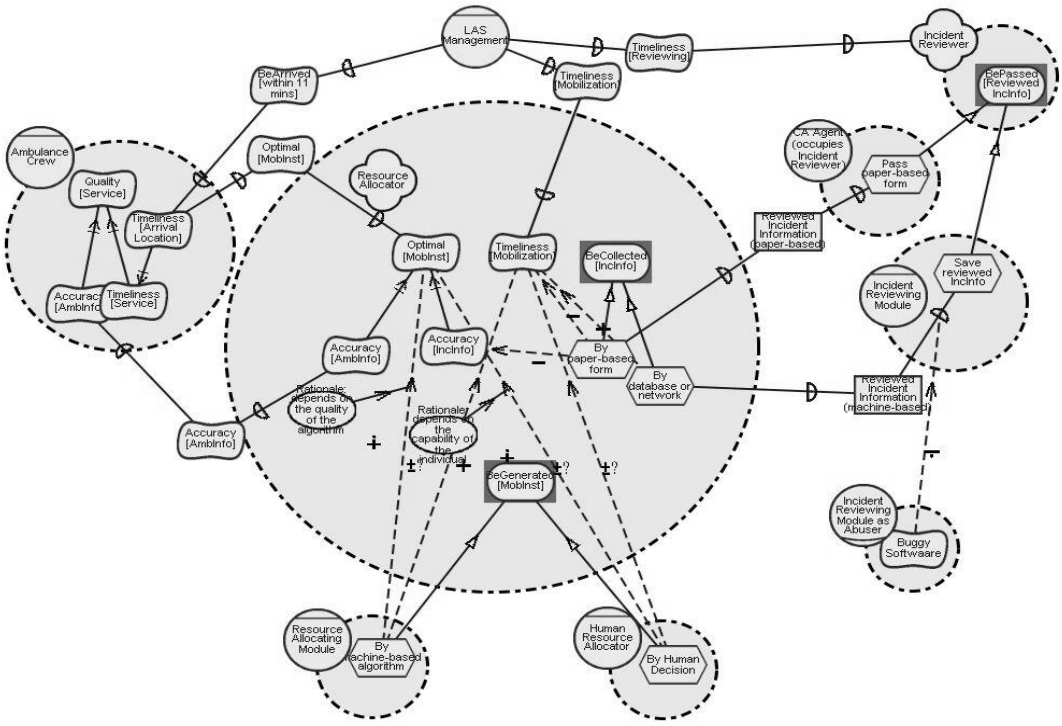


Figure 3.2-10 Example of the SR view from the LAS case study

3.2.4 The Evaluation Results view

The Evaluation Results (EVLR) view presents graphically the results of the evaluation process over an i* model. A qualitative evaluation process of i* models was adapted from the NFR framework (Chung et al. 2000) in GRL (GRL 2003), its purpose is to assess the feasibility of certain alternatives in achieving organizational level goals.

The evaluation process labels each process element according to some initial assumptions of leaf nodes in the SR view. A *leaf node* is an intentional element that normally has no incoming intentional links; a *top level node* is one that normally has no outgoing intentional links. The evaluation process propagates labels from leaf nodes step-by-step to top-level nodes, from internal elements to their incoming dependums, and from that dependum to the internal elements that reside inside the corresponding depender.

The original label propagation algorithm is defined for the NFR framework (Chung et al. 2000), and has been adapted to the richer i* notations throughout the literature (e.g., Liu et al. 2003) and in case studies (e.g., Horkoff 2004). In this thesis, we do not define the propagation rules, because the topic itself deserves further research and a uniformed label propagation algorithm in i* is yet to be defined. Consequently, scalability issues specific to this type of view is not studied in detail. However, we summarize some basic notations that are generically accepted in the EVLR view.

GRL distinguishes among six types of intentional element labels, each denoting a qualitative level of the satisficeability of the node; they are *Satisficed*, *Weakly Satisficed*, *Conflict/irresolvable*, *Undecided*, *Weakly Denied*, and *Denied*. Figure 3.2-11 shows their graphical notation.

(a) Satisficed (b) Weakly Satisficed (c) Undecided

(d) Weakly Denied (e) Denied (f) Conflict

Figure 3.2-11 Label types

The current OME tool distinguishes the labels from the way they are assigned. A *starting label* is a label assigned to a node (normally leaf node) by the modeler, and we highlight the corresponding node with a dashed-border solid-filled rectangle (Figure 3.2-12(a)). An *automated label* is a label that propagates automatically from a node's descendents to it, and, hence, there is no graphic change to the corresponding node (Figure 3.2-12(b)). A *human-decision-involved label* is a label that is assigned by the modeler according to what is contributed by its descendents, and it is denoted by highlighting the corresponding node with a solid-border solid-filled rectangle (Figure 3.2-12(c)). This notation appears graphically the same as the decision point, so we recommend that this not be used to highlight decision point in the EVLR view. An *imported label* is a label that is propagated from previous evaluation steps that are not shown in the current diagram, and is denoted by highlighting the corresponding node with a dashed-border dashed-filled rectangle (Figure 3.2-12(d)). As mentioned in the previous paragraph, these graphical notations do not play a critical role in our view extension, so we define them only graphically.

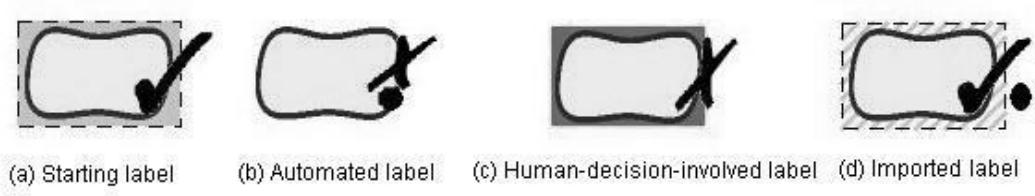


Figure 3.2-12 methods of label assignment

Figure 3.2-13 shows the EVLR view obtained by performing the evaluation process using the sample SR view from the LAS case study (Figure 3.2-10). During the evaluation, four process elements were selected to assign the starting

labels: softgoal **Accuracy [AmbInfo]** was considered weakly satisficed, task generate mobilization information **By Machine-based Algorithm** and **Pass paper-based form**, and softgoal **Buggy [Software]** were considered satisficed initially. No human decision is involved in the label propagation process nor any imported labels from other segments of the baseline model that are not visualized in this view. According to the label propagation algorithm adapted in (Liu et al. 2003), the weakly satisficed label of softgoal Accuracy [AmbInfo] contributes a weakly denied label to both the top-level softgoal **Quality [Service]** through an AND link and the incoming dependum **Accuracy [AmbInfo]** from the Resource Allocator. The former label, together with the undecided label propagated from softgoal Timeliness [Service] via another AND link, makes the label of Quality [Service] undecided. Following a similar procedure, the labels are propagated step-by-step until all top-level nodes are labeled.

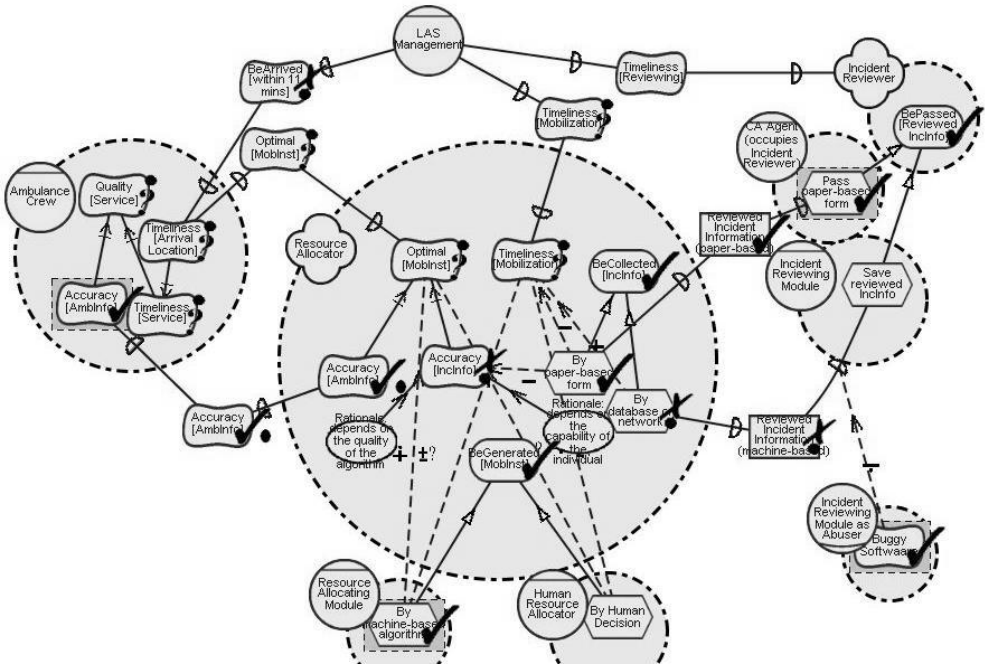


Figure 3.2-13 Sample Evaluation Results view based on an SR view from the LAS case study

3.3 Representational Constructs

We use here the approach discussed in the original framework (Section 2.2) to embed the reformulated i* framework into Telos. The OME style is again selected in presenting the formal constructs.

3.3.1 The Actor Class view

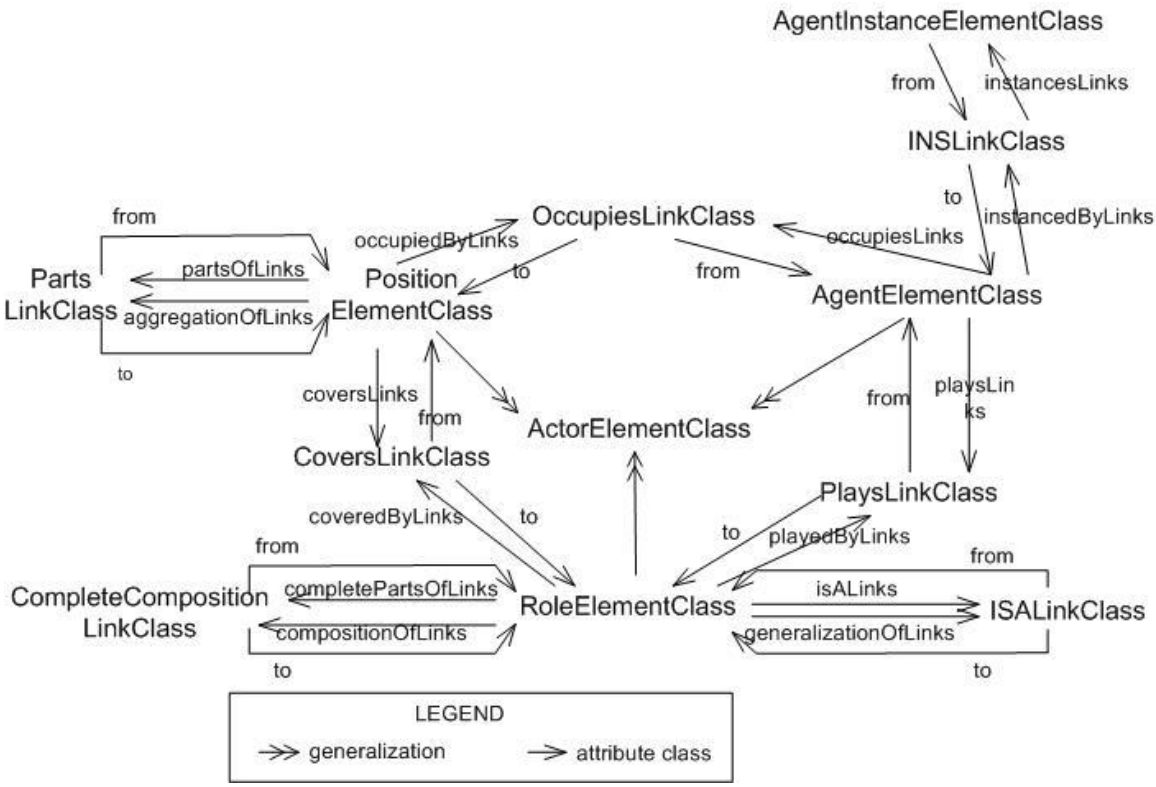


Figure 3.3-1 A partial meta-model of the Actor Class view

Figure 3.3-1 shows a partial meta-model of the AC view. The relationship between ISALinkClass and RoleElementClass applies to all other element classes shown in the meta-model, but we omitted them for the sake of simplicity. So does the relationship between the following pairs: PartsLinkClass and PositionElementClass, and CompleteCompositionLinkClass and RoleElementClass. The formal definition of the “specifies” link will be given in Section 4.4.1 since we consider it more appropriate to put that link in our view extension.

Note that an instance of the INSLinkClass always has an instance of AgentInstanceElementClass (e.g., **John Steven**) as its attribute *from* and an instance of AgentElementClass (e.g., **Human Resource Allocator**) as attribute *to*. In this thesis, we distinguish the two concepts explicitly in i* semantics for the first time.

Figure 3.3-2 shows the formal representation of some of the elements that appear in the AC view shown in Figure 3.2-3. The text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation. Note that the link names do not show in the graphical presentation of the view.

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
      name
            displayName : “Ambulance Crew”
      specifiedByLink
            : ACASpecifiesACPA
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      specifiesLink
            : ACASpecifiesACPA
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass ACASpecifiesACPA IN SpecifiesLinkClass WITH
      from
            : AmbunalceCrew_Agent
      to
            : AmbulanceCrew_PlainActor
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
      name
            displayName : “Resource Allocator”
      specifiedByLink
            : RAPSpecifiesRAPA
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
      specifiesLink
            : RAPSpecifiesRAPA
      occupiedByLinks
            : RAMOccupiesRA
```

```
            : HRAOccupiesRA
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass RAPSpecifiesRAPA IN SpecifiesLinkClass WITH
      from
            : ResourceAllocator_Position
      to
            : ResourceAllocator_PlainActor
END

% occupies link from agent Resource Allocation Module to position
Resource Allocator %
TELL SimpleClass RAMOccupiesRA IN OccupiesLinkClass WITH
      from
            : ResourceAllocationModule_Agent
      to
            : ResourceAllocator_Position
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass
WITH
      occupiesLinks
            : RAMOccupiesRA
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
      occupiesLinks
            : HRAOccupiesRA
END
```

Figure 3.3-2 Actor Class view representation in Telos

3.3.2 The Strategic Dependency view

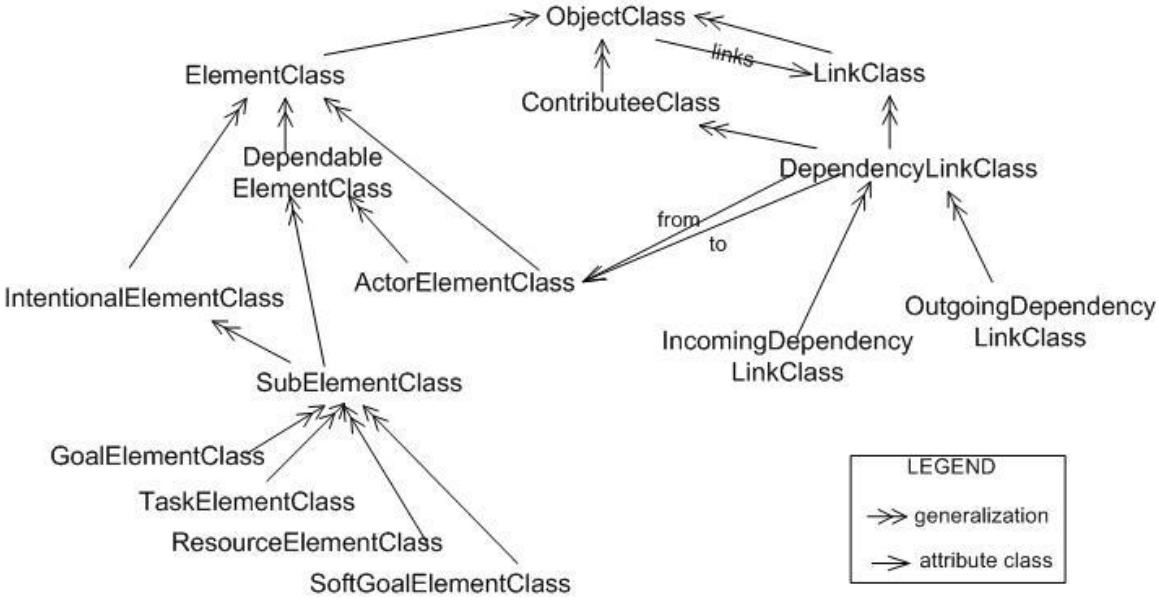


Figure 3.3-3 A partial meta-model of the SD view

Figure 3.3-3 shows a partial meta-model of the SD view. In the OME style we followed, a more rigid hierarchy was introduced into the meta-model to enforce the application of i* semantics. For example, OME introduced the concept of DependableElementClass, whose instance can have an instance of DependencyLinkClass as its attribute *links*. An instance of ContributionLinkClass that ends (*to*) at an instance of DependencyLinkClass is considered as a construct in the SD view. This type of link was included only in the SR view by Yu (Yu 1994). In addition, our view extension distinguishes between *incoming dependencies* (instances of IncomingDependencyLinkClass) and *outgoing dependencies* (instances of OutgoingDependencyLinkClass). Details regarding these dependency links are discussed later in Section 4.3.4.

Figure 3.3-4 formally represents some of the elements that appear in the SD view shown in Figure 3.2-4. The text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation. The attributes quoted using [square bracket] are *calculated* attributes. They are calculated based on the information obtained from the

baseline model and are not originally specified in the given element. For example, the outgoing dependency link AC_TALtoOptimalLink was initially specified as a link of an internal softgoal AC_TimelinessArrivalLocation of agent Ambulance Crew. However, in SD view, it is abstracted as a link of its parent—agent Ambulance Crew.

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
      name
            displayName : “Ambulance Crew”
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      name
            displayName : “Ambulance Crew”
      [outDepLinks
            : AC_TALtoOptimalLink]
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
      name
            displayName : “Resource Allocator”
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
      name
            displayName : “Resource Allocator”
[inDepLinks
: OptimaltoOptimalLink_RA]
                  …
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass
WITH
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
END

% dependency link from softgoal Timeliness [Arrival Location] inside
agent Ambulance Crew to softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
[: AmbulanceCrew_Agent]
to
: AC_OptimalMobInst_RA
END
```

```
% dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
        from
: AC_OptimalMobInst_RA
 to
[: ResourceAllocator_Position]
END

% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
END
```

Figure 3.3-4 SD view representation in Telos

3.3.3 The Strategic Rationale view

We argued previously (Section 3.1) that SR view is the detailed form of a SD view, so modeling constructs for the SR view is a superset of those for the SD view. The same analogy applies to the formal constructs between SR and SD. Thus, it appears sufficient for us to just show the representational constructs in the SR view that are not covered in the SD meta-model.

We use two diagrams to exhibit the meta-model for the SR view. Figure 3.3-5 focuses on presenting the hierarchy of element classes in the SR view while Figure 3.3-6 focuses on showing various link classes that are supported in the SR view. Besides,

Figure 3.3-5 shows the hierarchy of element classes. There are five meta-level classes that have corresponding graphical notations: GoalElementClass, TaskElementClass, ResourceElementClass, SoftgoalElementClass, and BeliefElementClass. Others are intermediate classes that only help implementation of i* semantics. For example, the inheritance relationship from GoalElementClass and TaskElementClass to DecomposableElementClass enforces a rule in i* that only a goal (instance of GoalElementClass) or a task (instance of TaskElementClass) can be decomposed. Another example is the use

of SubElementClass and IntentionalElementClass. From the partial meta-model of the SD view (Figure 3.3-3), we know that a sub-element (instance of SubElementClass) is dependable while an intentional element (instance of IntentionalElementClass) is not. BeliefElementClass does not subclass SubElementClass, so a belief (instance of BeliefElementClass) is not dependable. This semantic implies that a belief shall never be a dependum.

Figure 3.3-6 focuses on showing various link classes and their semantics that are supported in the SR view. For example, a means-ends link (instance of MeansEndsLinkClass) can only starts from a task (instance of TaskElementClass) and ends at either a goal, a task, or a resource (specified instance of EndsElementClass). Besides, to distinguish dependency from other non-actor-association links, we group the four types of links—means-ends, decomposition, contribution, and correlation—into *intentional links* (instances of IntentionalLinkClass).

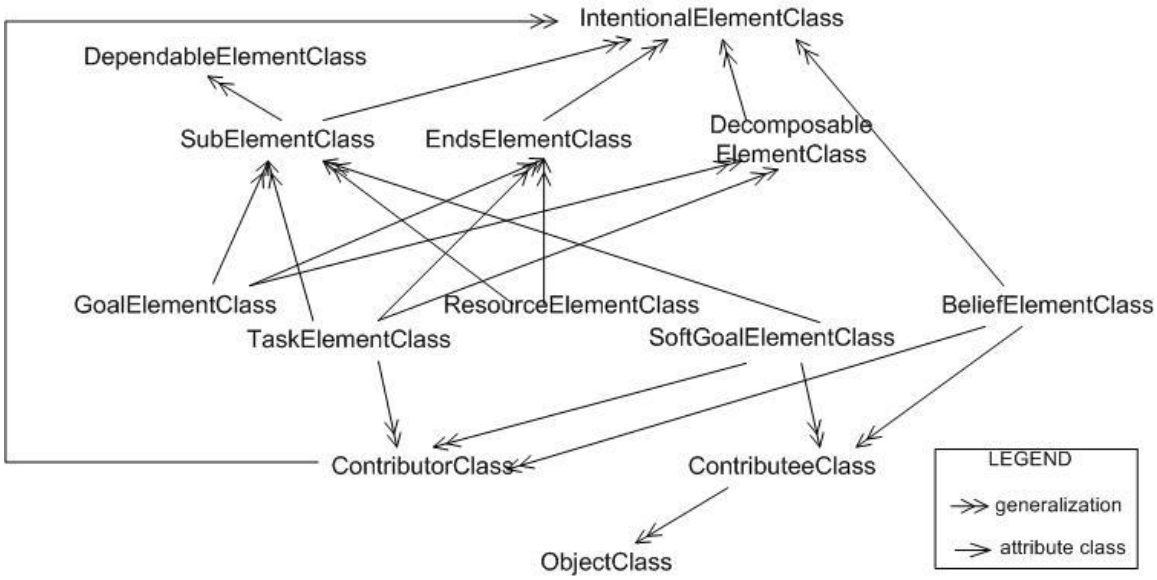


Figure 3.3-5 A partial schema showing Element hierarchy in the SR view

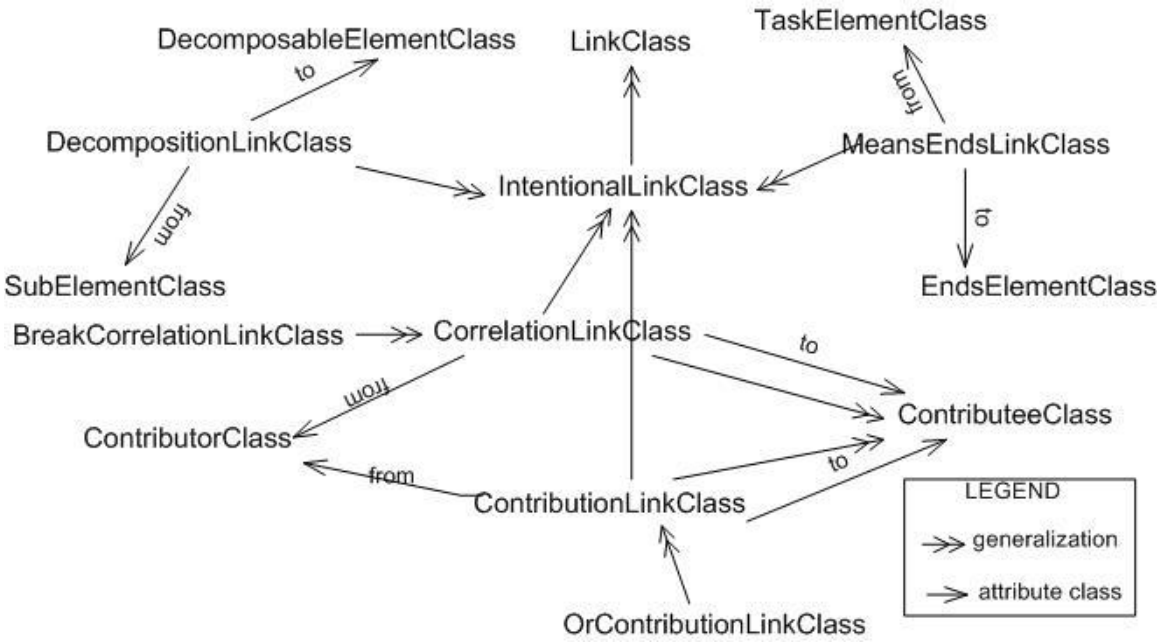


Figure 3.3-6 A partial meta-model showing the links supported by SR view

Figure 3.3-7 shows the formal representation of some of the elements that appear in the SR view shown in Figure 3.2-10. The text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation.

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
      name
            displayName : “Ambulance Crew”
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      name
            displayName : “Ambulance Crew”
      children
            : AC_QualityService
            : AC_TimelinessService
            : AC_TimelinessArrivalLocation
            : AC_AccuracyAmbInfo
            …
END

% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
parent
            : AmbulanceCrew_Agent
```

```
      outDepLinks
            : AC_TALtoOptimalLink
      links
            : AC_TALtoTS_AndContributionLink
            …
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
      name
            displayName : “Resource Allocator”
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
      name
            displayName : “Resource Allocator”
      children
      : RA_OptimalMobInst
: RA_TimelinessArrivalLocation
            : RA_AccuracyAmbInfo
            : RA_BeGeneratedMobInst
                  …
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass
WITH
      children
            : RA_BeGeneratedMobInst_ByAlgorithm
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
      children
            : RA_BeGeneratedMobInst_ByHumanDecision
END

% dependency link from softgoal Timeliness [Arrival Location] inside
agent Ambulance Crew to softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
: AC_TimelinessArrivalLocaltion
to
: AC_OptimalMobInst_RA
END

% dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
        from
: AC_OptimalMobInst_RA
 to
: RA_OptimalMobInst
END
```

```
% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
END

% softgoal Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
parent
: ResourceAllocator_Position
inDepLinks
: OptimaltoOptimalLink_RA
…
END
```

Figure 3.3-7 SR view representation in Telos

3.3.4 The Evaluation Results view

The i* framework supports a set of qualitative labels. We formalize them using a set of simple classes, each of which corresponds to an instance of the meta-class IntentionalElementLabelClass. For example, the weakly denied label ( ) is represented by simple class WeaklyDeniedElementLabel. The formal representation of these modeling constructs is shown in Figure 3.3-8.

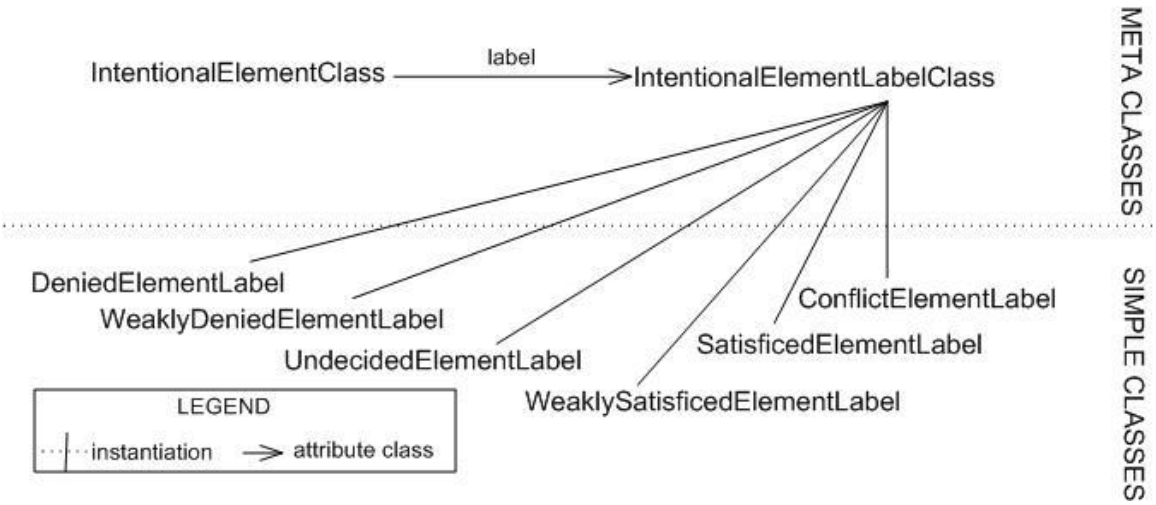


Figure 3.3-8 Formal representation of labels in Telos

Figure 3.3-9 shows the formal representation of two elements that appear in the EVLR view shown in Figure 3.2-13. Each of the two elements has “label” as its attribute, and each is assigned an **UndecidedElementLabel**. The text quoted by %% on top of each simple class denotes the name of the corresponding element shown in the graphical representation.

```
% softgoal Timelines [Arrival Location] inside agent Ambulance Crew%
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
      parent
            : AmbulanceCrew_Agent
      outDepLinks
            : AC_TALtoOptimalLink
      links
            : AC_TALtoTS_AndContributionLink
            …
      label
%an Undecided label is assigned to this element %
            : UndecidedElementLabel
END

TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
      label
      %an Undecided label is assigned to this element %
            : UndecidedElementLabel
END
```

Figure 3.3-9 Evaluation results in TELOS representation

3.4 Discussion

The four views derived from the same baseline model share some common elements and these elements serve as connectors among the views. Given a baseline model, the information contained in it can be partitioned into three basic views: Basic AC view, Basic SR view, and basic EVLR view. Actors (plain or specified) show in both the AC and SR view, yet the former contains actor associations while the latter focuses on dependencies. Any SD view can be viewed as an abstraction of its corresponding SR view. Any EVLR view contains

all elements in its corresponding SR view along with label assigned to the elements as attributes during an evaluation process.

The inter-view relationship can be seen more clearly in the underlying Telos representation. We use Figure 3.4-1 to show the formal constructs of a partial baseline model, denoting parts belonging to different views using different special effect. Then we show separately the corresponding formal representations in different views.

In Figure 3.4-1, we *italicize* the attributes that belong *only* (meaning do not belong to the SR view) to the AC view; we **bold** the attributes that belong to both SD and SR views; and the attributes without special effects belong to *only* the SR view. For the calculated attributes in the SD view, we put them in [square bracket]. Intentional elements are assigned labels in the EVLR view, so we underline those attributes shown *only* in the EVLR.

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
      name
            displayName : “Ambulance Crew”
      specifiedByLink
            : ACASpecifiesACPA
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      name
            displayName : “Ambulance Crew”
      specifiesLink
            : ACASpecifiesACPA
      children
            : AC_QualityService
            : AC_TimelinessService
            : AC_TimelinessArrivalLocation
            : AC_AccuracyAmbInfo
            …
      [outDepLinks
            : AC_TALtoOptimalLink]
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass ACASpecifiesACPA IN SpecifiesLinkClass WITH
      from
            : AmbunalceCrew_Agent
      to
            : AmbulanceCrew_PlainActor
```

```
END

% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
parent
            : AmbulanceCrew_Agent
      outDepLinks
            : AC_TALtoOptimalLink
      links
            : AC_TALtoTS_AndContributionLink
            …
      label
            : UndecidedElementLabel
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
      name
            displayName : “Resource Allocator”
      specifiedByLink
            : RAPSpecifiesRAPA
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
      name
            displayName : “Resource Allocator”
      specifiesLink
            : RAPSpecifiesRAPA
      occupiedByLinks
            : RAMOccupiesRA
            : HRAOccupiesRA
      children
      : RA_OptimalMobInst
: RA_TimelinessArrivalLocation
            : RA_AccuracyAmbInfo
            : RA_BeGeneratedMobInst
[inDepLinks
: OptimaltoOptimalLink_RA]
                  …
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass RAPSpecifiesRAPA IN SpecifiesLinkClass WITH
      from
            : ResourceAllocator_Position
      to
            : ResourceAllocator_PlainActor
END

% occupies link from agent Resource Allocation Module to position
Resource Allocator %
TELL SimpleClass RAMOccupiesRA IN OccupiesLinkClass WITH
      from
```

```
            : ResourceAllocationModule_Agent
      to
            : ResourceAllocator_Position
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass
WITH
      occupiesLinks
            : RAMOccupiesRA
      children
            : RA_BeGeneratedMobInst_ByAlgorithm
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
      occupiesLinks
            : HRAOccupiesRA
      children
            : RA_BeGeneratedMobInst_ByHumanDecision
END

% dependency link from softgoal Timeliness [Arrival Location] inside
agent Ambulance Crew to softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
: AC_TimelinessArrivalLocaltion
[: AmbulanceCrew_Agent]
to
: AC_OptimalMobInst_RA
END

% dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
        from
: AC_OptimalMobInst_RA
 to
: RA_OptimalMobInst
[: ResourceAllocator_Position]
END

% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
      label
            : UndecidedElementLabel
END

% softgoal Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
parent
```

```
: ResourceAllocator_Position
inDepLinks
: OptimaltoOptimalLink_RA
      label
            : UndecidedElementLabel
…
END
```

Figure 3.4-1 The Telos representation of a segment from the LAS baseline model

The following diagram shows the corresponding SD view of Figure 3.4-1. Only actors and their dependency links are included, and the non-bolded attributes are calculated ones.

```
% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass ISA
AmbulanceCrew_Actor WITH
      name
            displayName : “Ambulance Crew”
      outDepLinks
            : AC_TALtoOptimalLink
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass ISA
ResourceAllocator_Actor WITH
      name
            displayName : “Resource Allocator”
inDepLinks
: OptimaltoOptimalLink_RA
                  …
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass ISA
ResourceAllocator_Actor WITH
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass ISA
ResourceAllocator_Actor WITH
END

% dependency link from softgoal Timeliness [Arrival Location] inside
agent Ambulance Crew to softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
: AmbulanceCrew_Agent
to
: AC_OptimalMobInst_RA
END

% dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside position Resource Allocator %
```

```
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
        from
: AC_OptimalMobInst_RA
 to
: ResourceAllocator_Position
END

% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
END
```

The corresponding AC view of Figure 3.4-1 show below keeps only actors and their associations.

```
% plain actor Ambulance Crew %
TELL SimpleClass AmbulanceCrew_PlainActor IN ActorElementClass WITH
      name
            displayName : “Ambulance Crew”
      specifiedByLink
            : ACASpecifiesACPA
END

% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      name
            displayName : “Ambulance Crew”
      specifiesLink
            : ACASpecifiesACPA
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass ACASpecifiesACPA IN SpecifiesLinkClass WITH
      from
            : AmbunalceCrew_Agent
      to
            : AmbulanceCrew_PlainActor
END

% plain actor Resource Allocator %
TELL SimpleClass ResourceAllocator_PlainActor IN ActorElementClass WITH
      name
            displayName : “Resource Allocator”
      specifiedByLink
            : RAPSpecifiesRAPA
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass WITH
      name
            displayName : “Resource Allocator”
```

```
      specifiesLink
            : RAPSpecifiesRAPA
      occupiedByLinks
            : RAMOccupiesRA
            : HRAOccupiesRA
END

% Specifies link from position Resource Allocator to plain actor
Resource Allocator %
TELL SimpleClass RAPSpecifiesRAPA IN SpecifiesLinkClass WITH
      from
            : ResourceAllocator_Position
      to
            : ResourceAllocator_PlainActor
END

% occupies link from agent Resource Allocation Module to position
Resource Allocator %
TELL SimpleClass RAMOccupiesRA IN OccupiesLinkClass WITH
      from
            : ResourceAllocationModule_Agent
      to
            : ResourceAllocator_Position
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass
WITH
      occupiesLinks
            : RAMOccupiesRA
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass WITH
      occupiesLinks
            : HRAOccupiesRA
END
```

The corresponding SR view of Figure 3.4-1 shown below keeps actors, their external dependencies, and their internal structures.

```
% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass ISA
AmbulanceCrew_Actor WITH
      name
            displayName : “Ambulance Crew”
      children
            : AC_QualityService
            : AC_TimelinessService
            : AC_TimelinessArrivalLocation
            : AC_AccuracyAmbInfo
            …
END

% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
```

```
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
parent
            : AmbulanceCrew_Agent
      outDepLinks
            : AC_TALtoOptimalLink
      links
            : AC_TALtoTS_AndContributionLink
            …
      label
            : UndecidedElementLabel
END

% position Resource Allocator %
TELL SimpleClass ResourceAllocator_Position IN PositionElementClass ISA
ResourceAllocator_Actor WITH
      name
            displayName : “Resource Allocator”
      children
      : RA_OptimalMobInst
: RA_TimelinessArrivalLocation
            : RA_AccuracyAmbInfo
            : RA_BeGeneratedMobInst
            …
END

% agent Resource Allocation Module %
TELL SimpleClass ResourceAllocationModule_Agent IN AgentElementClass ISA
ResourceAllocator_Actor WITH
      children
            : RA_BeGeneratedMobInst_ByAlgorithm
END

% agent Human Resource Allocator %
TELL SimpleClass HuamnResourceAllocator_Agent IN AgentElementClass ISA
ResourceAllocator_Actor WITH
      children
            : RA_BeGeneratedMobInst_ByHumanDecision
END

% dependency link from softgoal Timeliness [Arrival Location] inside
agent Ambulance Crew to softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_TALtoOptimalLink IN DependencyLinkClass WITH
from
: AC_TimelinessArrivalLocaltion
to
: AC_OptimalMobInst_RA
END

% dependency link from softgoal dependum Optimal [MobInst] to softgoal
Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass OptimaltoOptimalLink_RA IN DependencyLinkClass WITH
        from
: AC_OptimalMobInst_RA
 to
: RA_OptimalMobInst
```

```
END

% softgoal dependum Optimal [MobInst] %
TELL SimpleClass AC_OptimalMobInst_RA IN DependumElementClass,
SoftGoalElementClass WITH
      inDeplinks
            : AC_TALtoOptimalLink
      outDepLinks
            : OptimaltoOptimalLink_RA
      label
            : UndecidedElementLabel
END

% softgoal Optimal [MobInst] inside position Resource Allocator %
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
parent
: ResourceAllocator_Position
inDepLinks
: OptimaltoOptimalLink_RA
      label
            : UndecidedElementLabel
…
END
In the EVLR, a label attribute is associated with corresponding
intentional element tokens.
TELL SimpleClass RA_OptimalMobInst IN SoftgoalElementClass WITH
parent
: ResourceAllocator_Position
inDepLinks
: OptimaltoOptimalLink_RA
      label
            : UndecidedElementLabel
…
END
```

Besides what was formally proposed in this thesis, we uniquely named each simple class in our sample. Naming convention is beyond the scope of this research so we will not enforce the use of any specific style. The style chosen proved to be sufficient in identifying elements from the LAS case study, but we do not guarantee it will generalize to other applications.

4 Managing i* Models Using Views

As a sub-step in our view extension to effectively represent large-scale and complex i* models, we separate meta-concepts in the Actor Class (AC) view from the Strategic Dependency (SD) view. However, for a sufficiently large-scale application, a basic (AC, SD, or SR) view itself can become complex, and difficult to comprehend. So we need to break down each basic view until the information contained in a view is readily comprehensible.

While scaling down a complex baseline model into multiple views, the number of views can grow. The approach itself introduces a new line of complexity into representing and traversing the model. As a result, we introduce a **view extension** as a separate project management framework alongside the core i* framework. The purpose of this view extension is to offer a reference structure so that users can maintain a relationship among various views and locate information effectively from other views.

Section 4.1 explains the features of the view extension; Section 4.2 presents the representational constructs of the view extension; Section 4.3 defines related meta-concepts that are used in the selection rules; and Section 4.4 briefly summarizes contributions of our view extension.

4.1 View Extension Features

We use a **View Map (VM)** to visualize relationships among various views in the reference structure. Unique names are given to models, views, links and elements to provide a referencing structure. This strategy is important to support cross diagram references and, thus, minimize manual efforts (given the fact that these references have to be maintained manually at present).

In the reformulated i* framework (Section 3), four types of views—AC, SD, SR and EVLR—are defined. To address scalability, our extension further

distinguishes among various sub view types. The views are defined using Telos: Use meta-classes to encode view types (e.g., BasicViewClass), and use simple (domain) classes to encode an actual view (e.g., theBasicACView) obtained from an existing baseline model. In this regard, adding or deleting or updating a view type can follow a systematic and formal approach. Thus, it is easier for users to maintain and evolve over time this view extension and to make use of tool support.

Elements in a view are not selected arbitrarily; rather, a *selection rule* is bound to each specified view type. Applying a selection rule to the baseline model or some intermediate view, we find that the resulting elements constitute a corresponding sub-view of the input element. Selection rules are defined in Telos-compatible First Order Logic (FOL) and can be implemented using Telos queries (instances of QueryClass). See Appendix for more details regarding the translation from FOL formula to O-Telos classes.

The reformulated i* framework discussed in chapter 3 distinguishes the baseline model from views. Our extension distinguishes between *basic* and *partial* views. For any real-world application, one or more i* models can be constructed according to different social settings, different view-points, or different time periods. We define each of these models as the baseline model for the specific settings and viewpoints. Corresponding to the four view types, four basic views are derived from each baseline model, one for each view type. Basic views are derived according to the type of meta-level concepts each specific view type support (see Section 3.3 for more details). Partial views, corresponding to one or more sub- view type, are derived from a basic view or another partial view according to the selection rules associated with the sub-view type.

4.2 View Map

In a view map, we use a heavy-border box to denote a *basic* or an *initial view* (the view all other views are based on in a view map), and we use a regular-

border box to denote a *derived view* (views other than the initial one in a view map). The *decomposition* from one view into multiple child views is denoted by branches; this type of reduction is total. In other words, the union of modeled elements in child views is equivalent to the set of modeled elements in the parent view. The *projection* over one view to a sub-view is denoted by dashed arrow-lines. The view decomposition and projection links connect sub-views of type AC, SD, and SR. In the EVLR view, we use a solid arrow-line to denote the direction of *label propagation*. Figure 4.2-1 shows the graphical notations of the concepts.

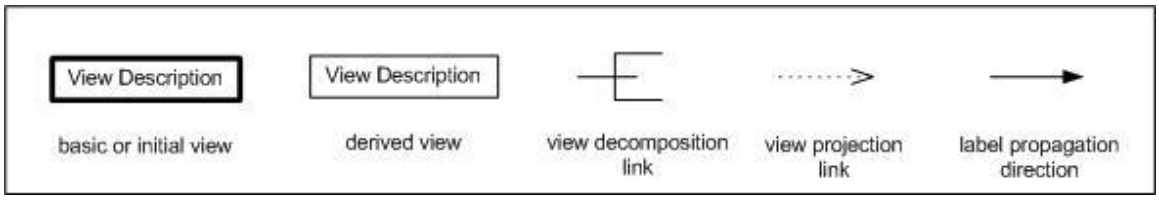


Figure 4.2-1 Graphical notations in View Map

Figure 4.2-2 illustrates the generic view map that fits for all i* models. For any i* model constructed for a given organizational configurations, **The Baseline Model** can be decomposed into four basic views: **The Basic AC View**, **The Basic SD view**, **The Basic SR view**, and **The Basic EVLR view**.

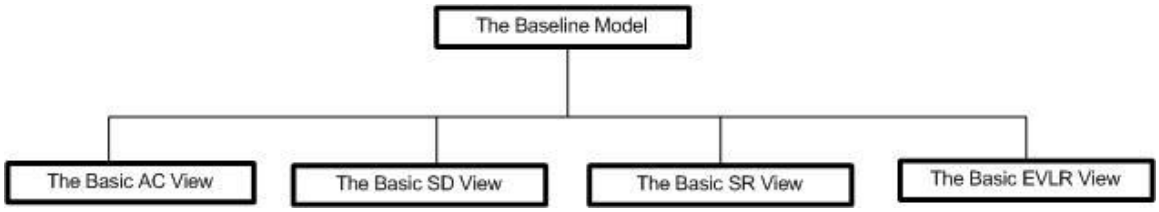


Figure 4.2-2 Generic View Map showing relationship of the baseline model and the basic views

4.3 Representational Constructs

Each type of view is defined by a meta-level view class, and concrete views in an application are instances of the meta-level view classes. Selection rules are

encoded in query classes and are attached as the *selectionRule* attribute to each specific type of view class.

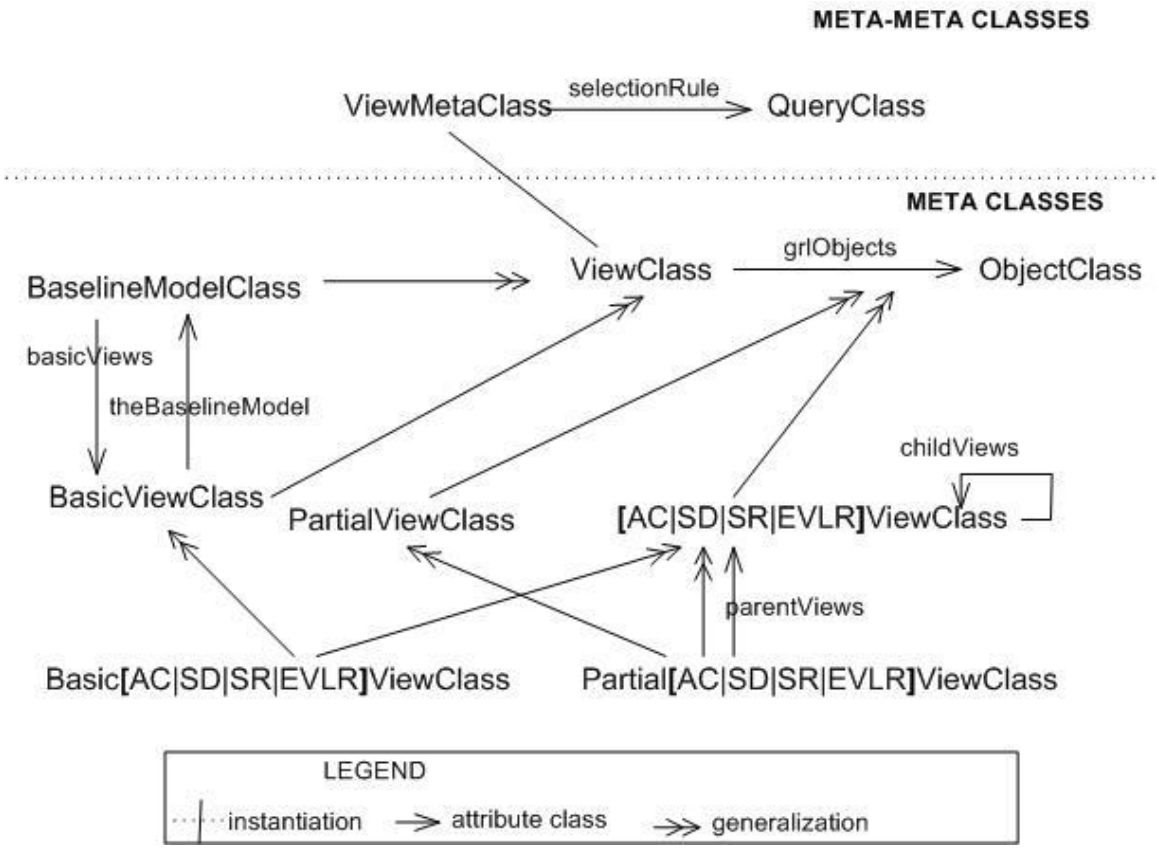


Figure 4.3-1 A partial meta-model of the view extension showing meta-level relationship among the baseline model class and other view classes

Figure 4.3-1 shows the part of the meta-model that defines the relationship among a baseline model and its child views. Formally, we consider a baseline model as a specific view (the whole); an instance of BaselineModelClass takes an instance of a BasicViewClass as attribute *basicViews*, while the latter takes the former as its attribute *theBaselineModel*. Besides, the above figure also shows two lines of specializing view classes: One of them is in accordance with the four view types, and the other is in accordance with the distinction between basic and partial. After combination, we obtained eight sub- view classes, including BasicACViewClass, BasicSDViewClass, BasicSRViewClass, BasicEVLRViewClass, PartialACViewClass, PartialSDViewClass, PartialSRViewClass, and PartialEVLRViewClass. We use short-hand style

Basic[AC|SD|SR|EVLR]ViewClass in Figure 4.3-1 to reference the four basic view classes.

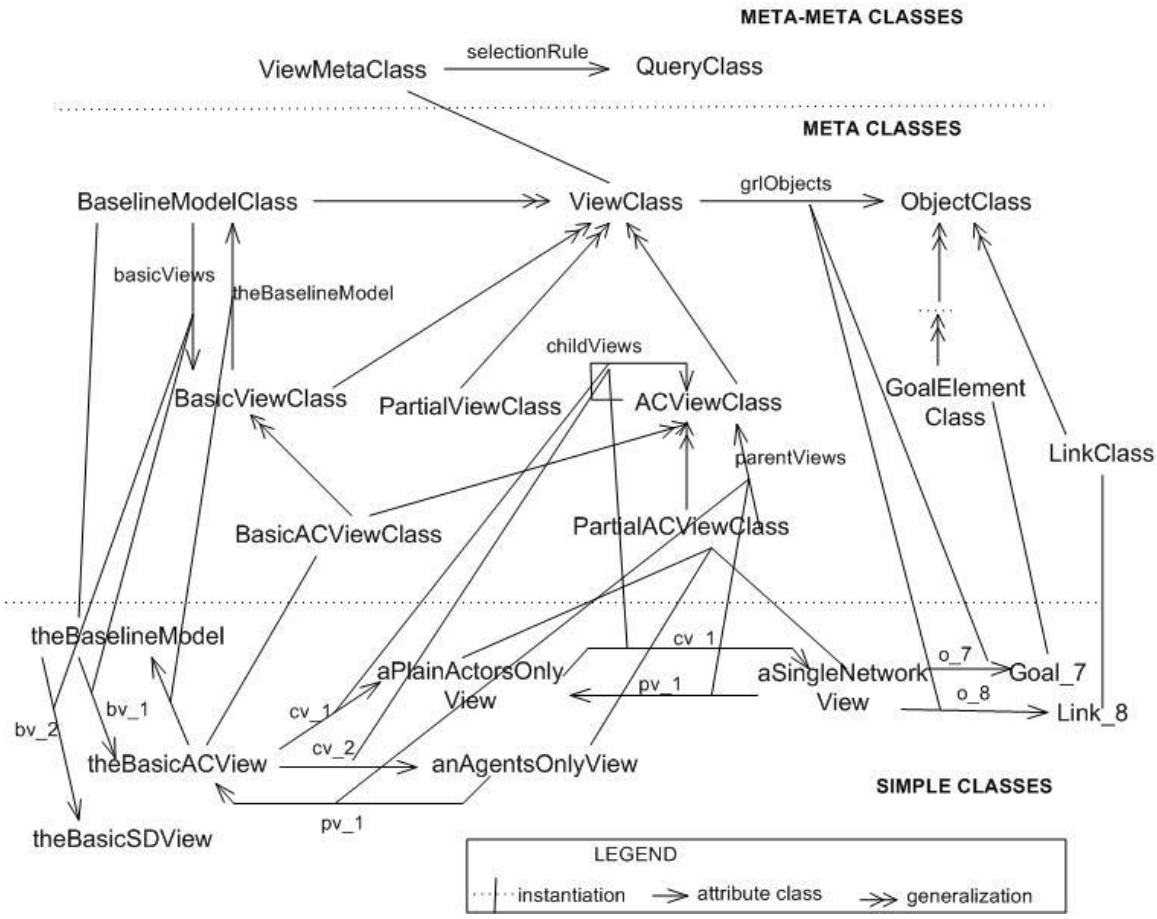


Figure 4.3-2 A partial meta-model of the view extension showing the hierarchy of inheritance

Figure 4.3-2 shows the relationships among the meta-level classes and concrete views residing in an i* model. Each i* model corresponds to a singleton instance of BaselineModelClass—**theBaselineModel**. Instances of any BasicXXViewClass are also singletons, and here "XX" stands for one of AC|SD|SR|EVLR. For example, **theBasicACView** is the singleton instance of meta-class BasicACViewClass. Each view is constituted by a sub-set of domain classes existed in the baseline model. For example, **aSingleNetworkView** (indirect instance of PartialACViewClass) contains Goal_7 (instance of

GoalElementClass) and Link_8 (instance of LinkClass) as contents of its attribute *grlObjects*.

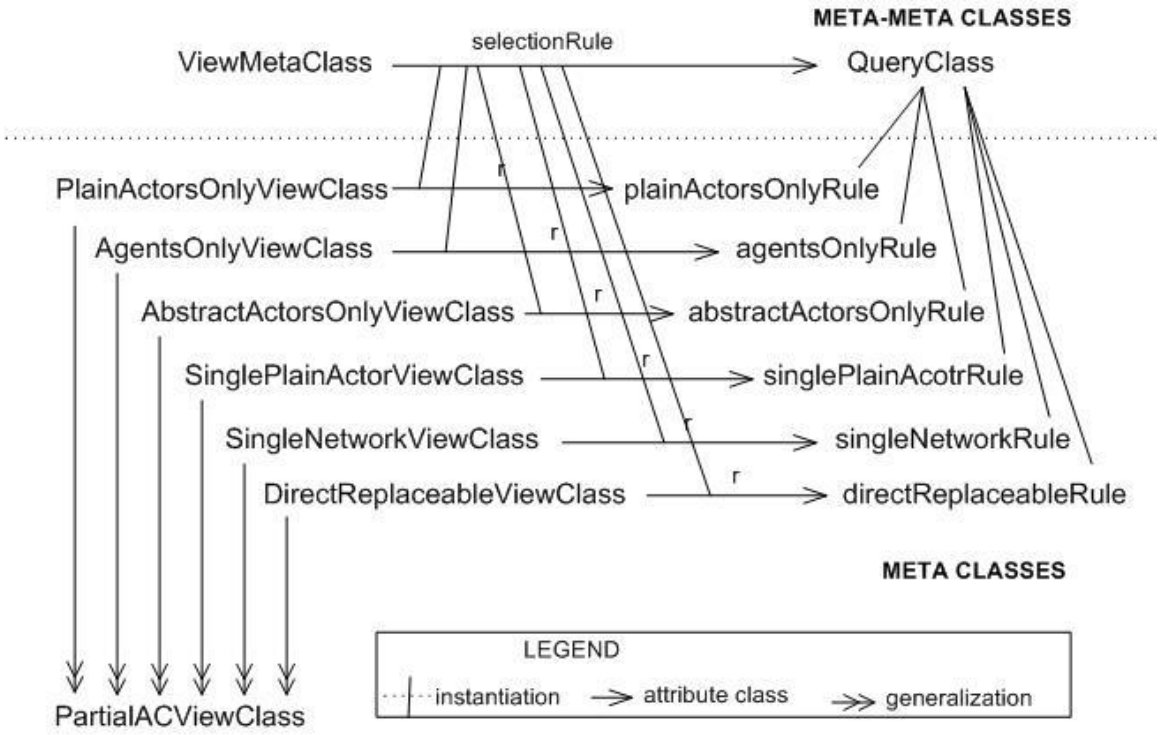


Figure 4.3-3 A partial meta-model of the view extension showing meta-level relationships among different types of AC view classes

Figure 4.3-3 shows a partial meta-model of the view extension concerning AC sub- view types. Query classes assigned to different types of AC views are manifested. For example, **plainActorsOnlyRule** (instance of QueryClass) is assigned to the **PlainActorsOnlyViewClass** as its attribute *selectionRule*. Each partial view (e.g., aPlainActorsView) of a given type (e.g., Plain-Actors-Only view type) corresponds to the resulting set of elements following the execution of the query (e.g., plainActorsOnlyRule) attached to the view type.

Figure 4.3-4 and Figure 4.3-5 shows the similar meta-model of the view extension concerning SD and SR views, respectively.

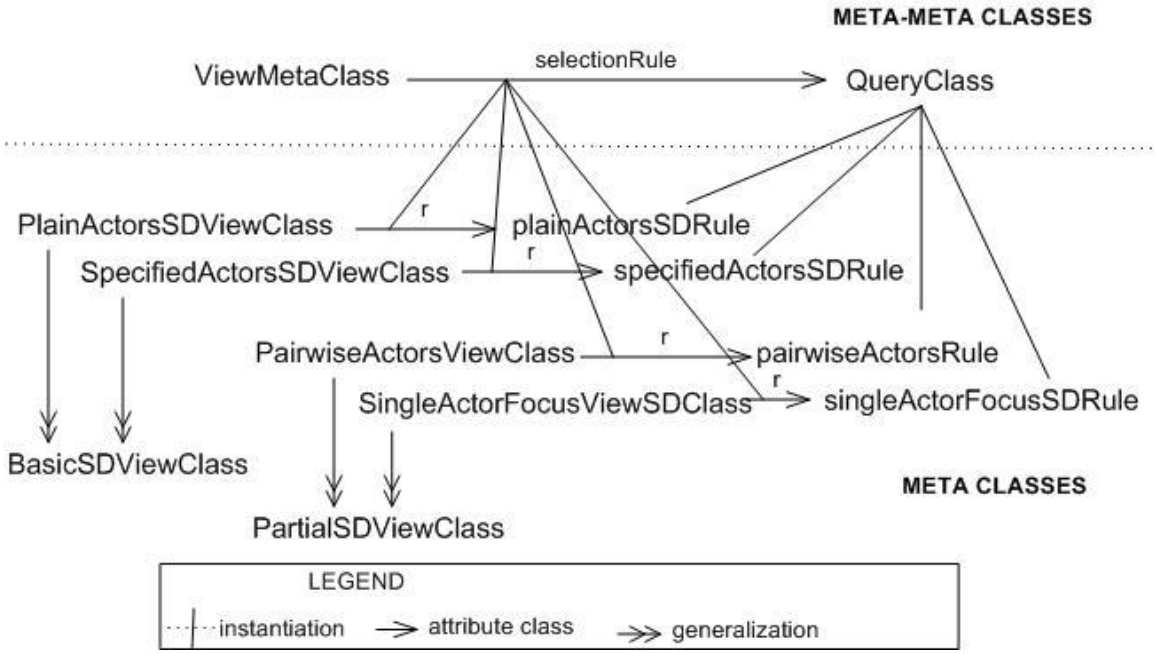


Figure 4.3-4 A partial meta-model of the view extension showing meta-level relationships among different types of SD view classes

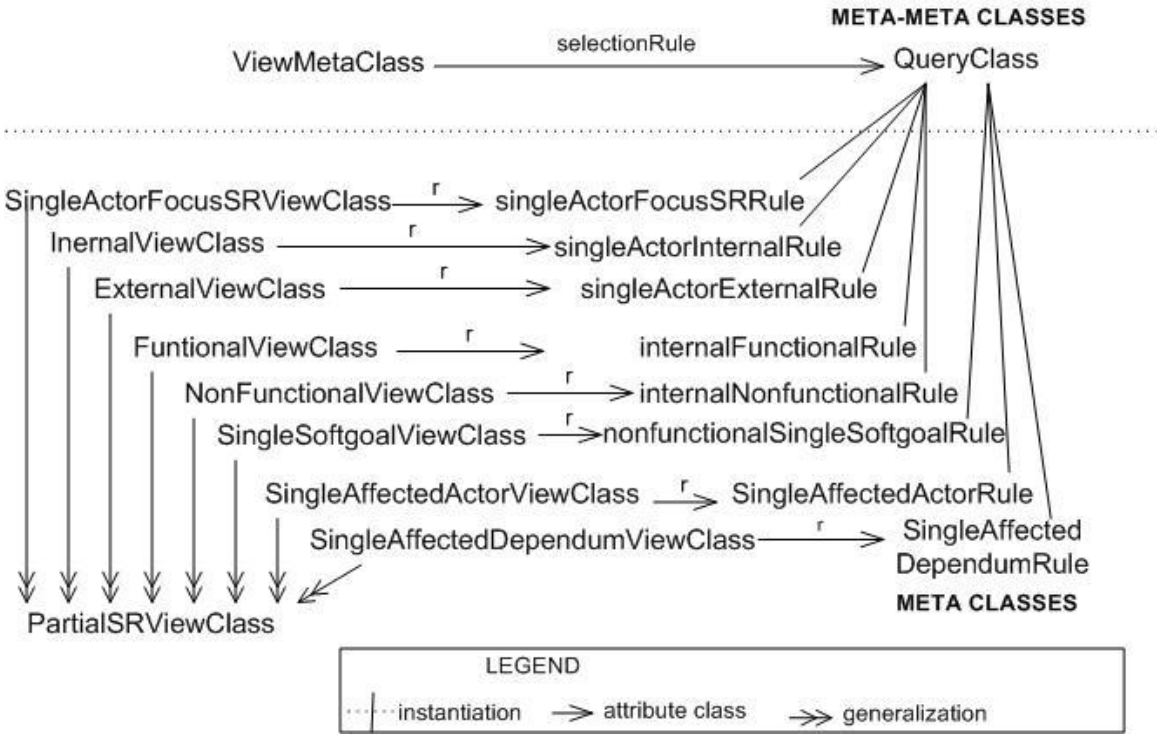


Figure 4.3-5 A partial meta-model of the view extension showing meta-level relationships among different types of SR and EVLR view classes

4.4 Meta-concepts Essential to Selection Rules

In previous sections, we introduced the view types and their corresponding representational constructs in Telos. In this section, we define some critical concepts that are extensively referenced in the selection rules. Most of the concepts come in pairs, as follows: *plain* vs. *specified actor* (Section 4.3.1), *parent* vs. *children* (Section 4.3.3), *incoming* vs. *outgoing dependency* (Section 4.3.4), and *ancestor* vs. *descendent* (Section 4.3.6); the exceptions are *actor association* (4.3.2) and *external link* (Section 4.3.5).

Concepts discussed in this section are derived from existing meta-concepts in our reformulated i* framework, and some of them have been defined informally in Section 3.2, along with the description of the graphical notations. We emphasize in this section the formal constructs related to these concepts: without exception, they are described in a Telos compatible First Order Logic (FOL) form.

4.4.1 Plain and specified actor

Our extension implements the concept *plain actor* explicitly using meta-class PlainActorElementClass, and the concept *specified actor* using meta-class SpecifiedActorElementClass. PlainActorElementClass is equivalent to only ActorElementClass, while SpecifiedActorElementClass is equivalent to the generation of RoleElementClass, PositionElementClass, AgentElementClass, and AgentInstanceElementClass. Among specified actors, we distinguish between *abstract actors* (instances of AbstractActorElementClass) and *physical actors* (instances of PhysicalActorElementClass) for the former represents the classification of similar instances while the latter represents a single instance. AbstractActorElementClass is equivalent to RoleElementClass and PositionElementClass, and AgentElementClass, while PhysicalActorElementClass to AgentInstanceElementClass. Figure 4.4-1 shows the partial meta-model that relates to our extended actor types.

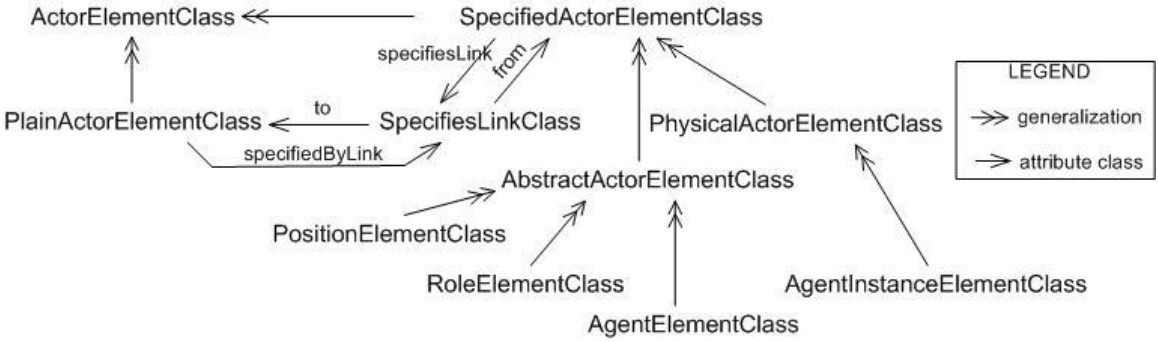


Figure 4.4-1 A partial meta-model showing relationships among extended actor types in our extension

4.4.2 Actor association

We define actor associations informally as the general form of eight relationships among actors, as follows: “plays”, “occupies”, “covers”, “is-A”, “INS”, “is-Part-of”, “specifies”, and “complete composites”. Now we formally present these concepts as subclasses of ActorAssociationLinkClass. Figure 4.4-2 shows the part of meta-model related with association links.

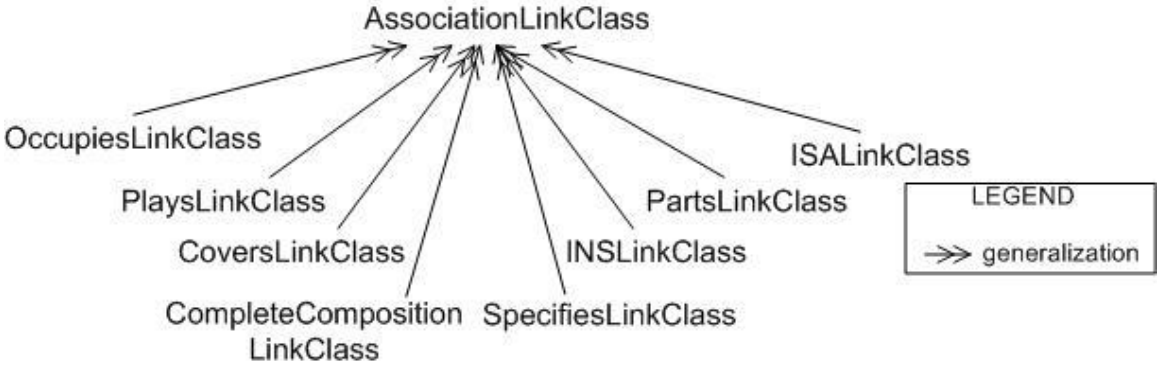


Figure 4.4-2 Partial meta-model showing association link classes

4.4.3 Parent versus children

The i* semantic has natural support for one level of abstraction between a strategic actor and its internal rationales. These internal rationales are modeled using intentional elements (goals, tasks, softgoals, resources, and beliefs) that are connected by intentional links (means-ends, decomposition, contribution, and

correlation). Our extension defines the relationship discussed above as *parent-children*[5]. In other words, a strategic actor can have intentional elements as its *children*, while, in turn, these intentional elements have that actor as their *parent*. The partial meta-model related to these concepts is demonstrated in Figure 4.4-3.

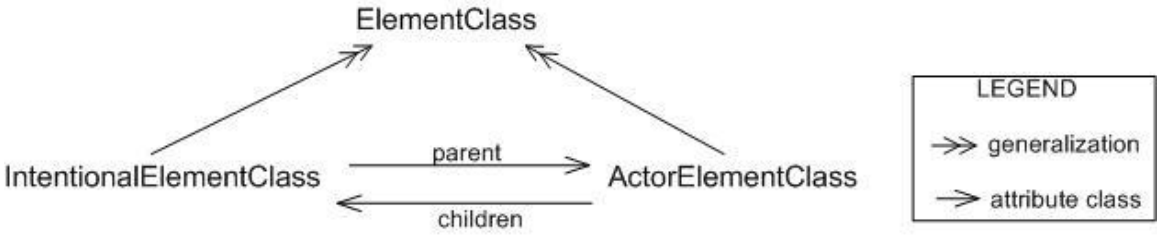


Figure 4.4-3 Partial meta-model showing the parten-children relationship

For example, in the underlying representation of a partial model shown in Figure 4.4-4, we see that simple class **AmbulanceCrew_Agent** (denoting agent Ambulance Crew) has simple class **AC_TimelinessArrivalLocation** (denoting softgoal Timeliness [Arrival Location]) assigned to its attribute *children*. The latter, in turn, has the former assigned to its attribute *parent*.

```
% agent Ambulance Crew %
TELL SimpleClass AmbulanceCrew_Agent IN AgentElementClass WITH
      children
                  : AC_QualityService
                  : AC_TimelinessService
                  : AC_TimelinessArrivalLocation
                  : AC_AccuracyAmbInfo
                  …
END

% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
      parent
            : AmbulanceCrew_Agent
            …
END
```

Figure 4.4-4 Partial Telos representation showing the parent-child relationship

[5] This choice of terms follows from OME version 3 tool and does not imply there will be multiple layers of parent-children relationship in the present reformulated i* framework.

Formally, we identify the parent and children of a given element using Telos queries. The parent of a given intentional element can be obtained by executing the *find_parent* query. Children of a given actor element are also called the internal elements. We use query *find_internal_elements* to retrieve the set of internal elements. The symbol "§" denotes *for all those* in the FOL formula specified in this thesis (see the appendix for more details regarding the rules in translating queries expressed in our FOL format into O-Telos query classes).

Query1

find_parent(e:IntentionalElementClass)::=

§a:ActorElementClass·e.parent=a

Query2

find_internal_elements(a:ActorElementClass)::=

§e:IntentionalElementClass·(e ∈ a.children)

4.4.4 Incoming versus outgoing dependency

For a specific actor, or an intentional element internal to the actor, or a dependum (external to all actors), we can distinguish the incoming and outgoing dependencies according to the direction of the dependency links. An incoming dependency comes from a depender to a dependum or from a dependum to a dependee. An outgoing dependency goes from a depender toward a dependum or from a dependum to a dependee. We formalize the distinctions explicitly using IncomingDependencyLinkClass and OutgoingDependencyLinkClass. Instances of these two meta-classes are referenced by intentional elements (instances of IntentionalElementClass) as attributes *inDepLinks* and *outDepLinks*, respectively. Figure 4.4-5 shows the part of the meta-model that deals with these relationships.

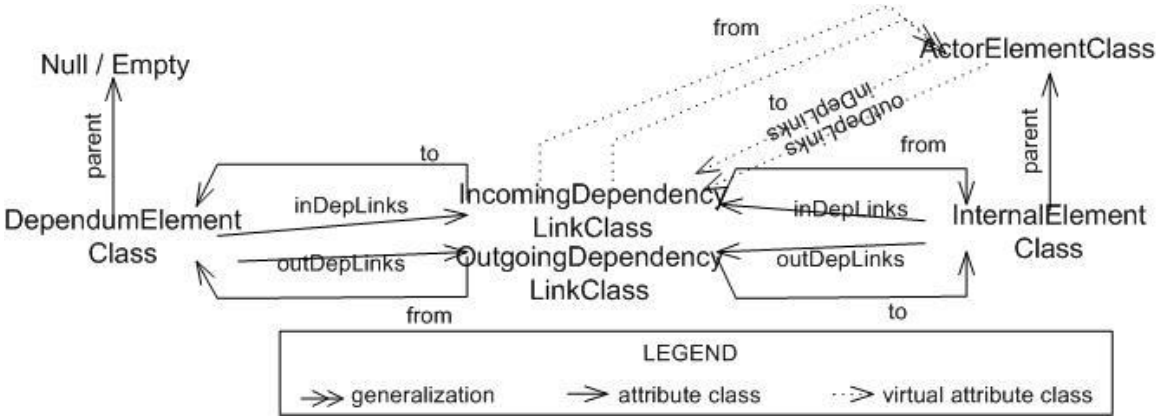


Figure 4.4-5 Partial meta-model showing incoming and outgoing dependency links

Examining Figure 4.4-5, we observe that the virtual *from/to* attribute (the one to ActorElementClass) of the dependency links applies only to SD views while their origin (the one to InternalElementClass) applies only to SR views. So do the inDepLinks and outDepLinks attribute of ActorElementClass and IntentionalElementClass.

Formally, we identify the incoming and outgoing dependencies of a given actor element using Telos queries. The incoming dependencies can be obtained by executing the *find_incoming_dependencies_to_actor* query. The outgoing dependencies are obtained by executing the *find_outgoing_dependencies_from_actor* query.

Query3

find_incoming_dependencies_to_actor(a:ActorElementClass)::=

§l:DependencyLinkClass·

l.to=a ∨ (∃e:InternalElementClass· e.parent=a ∧ l.to=e)

Query4

find_outgoing_dependencies_from_actor(a:ActorElementClass)::=

§l:DependencyLinkClass·

l.from=a ∨ (∃e:InternalElementClass· e.parent=a ∧ l.from=e)

As a by-product of the above definition, we can formally define dependum element and internal element by attaching deductive rule to SubElementClass. In

the formula below, name of meta classes (e.g., DependumElementClass) are shown as the left-hand operand of "::=" (equivalent to), and its definition (e.g., "e:SubElementClass with 'dependum_rule'") as the right-hand operand. The previously defined meta-class on which this new one will be based (e.g., SubElementClass) appears after the semicolon and before the word "with" in the definition. The corresponding deductive rule (e.g., "dependum_rule") follows the word "with" and is placed in "quotation marks". This pattern applies to all the definition of meta-classes using a deduction rule.

Def1

DependumElementClass::= e: SubElementClass with "dependum_rule"

dependum_rule::=

$\neg(\exists a$: ActorElementClass $\cdot$ e.parent = a)

Def2

InternalElementClass::= e: IntentionalElementClass with "internal_rule"

internal_rule::=

$\exists a$: ActorElementClass$\cdot$e.parent = a

We also define queries to locate the dependers and dependees for a given dependum (instance of DependumElementClass). There are two levels of dependers: the actor level (shown in SD view) and the element level (shown in SR view). We construct different queries for them in our extension. In FOL, they are as follows:

Query5

find_depender_actor(de:DependumElementClass)::=

$\S a$:ActorElementClass$\cdot$ $\exists l$:DependencyLinkClass$\cdot$

(l.from=a $\vee$ ($\exists e$:InternalElementClass$\cdot$ e.parent=a $\wedge$ l.from=e)) $\wedge$ l.to=de

Query6

find_depender_element(de:DependumElementClass)::=

$\S e$:InternalElementClass$\cdot$ $\exists l$:DependencyLinkClass$\cdot$ l.from=e $\wedge$ l.to=de

Query7

find_dependee_actor(de:DependumElementClass)::=

§a:ActorElementClass· ∃l:DependencyLinkClass·

(l.to=a ∨ (∃e:InternalElementClass· e.parent=a ∧ l.to=e)) ∧ l.from=de

Query8

find_dependee_element(de:DependumElementClass)::=

§e:InternalElementClass· ∃l:DependencyLinkClass· l.from=de ∧ l.to=e

4.4.5 External links

To distinguish dependency from other non-actor-association links, we group the four types of links—means-ends, decomposition, contribution, and correlation—into intentional links (see Section 3.3.3 for detail). Intentional links normally connect elements inside an actor boundary; however, they sometimes extend their target outside the actor boundary, and we call these intentional links *external links*.

We define external links using a query *find_all_external_links*.

Def3

ExternalLinkClass::=l:IntentionalLinkClass with “external_rule”

external_rule::= (l in find_all_external_links())

The query is defined recursively. We first define a sub-query *find_direct_external_links*. Informally, a *direct external link* is one that originates from an element within an actor’s boundary and ends at a dependency link outside the actor’s boundary. Formally, the query is defined as follows:

Query9

find_direct_external_links()::=

§l:IntentionalLinkClass·

∃a:ActorElementClass, dl:DependencyLinkClass, e:IntentionalElementClass·

l.from=e ∧ e.parent=a ∧ l.to=dl

Then we define an *external link* recursively—informally, it is:

1. A direct external link; or
2. Any link that ends at an external link

Formally, query find_all_external_links is expressed as:

Query10

find_all_external_links()::=

§l:IntentionalLinkClass· l∈ find_direct_external_links() ∨

(∃l2:IntentionalLinkClass· l.to=l2∧(l2 ∈ find_all_external_links()))

4.4.6 Ancestor versus descendent

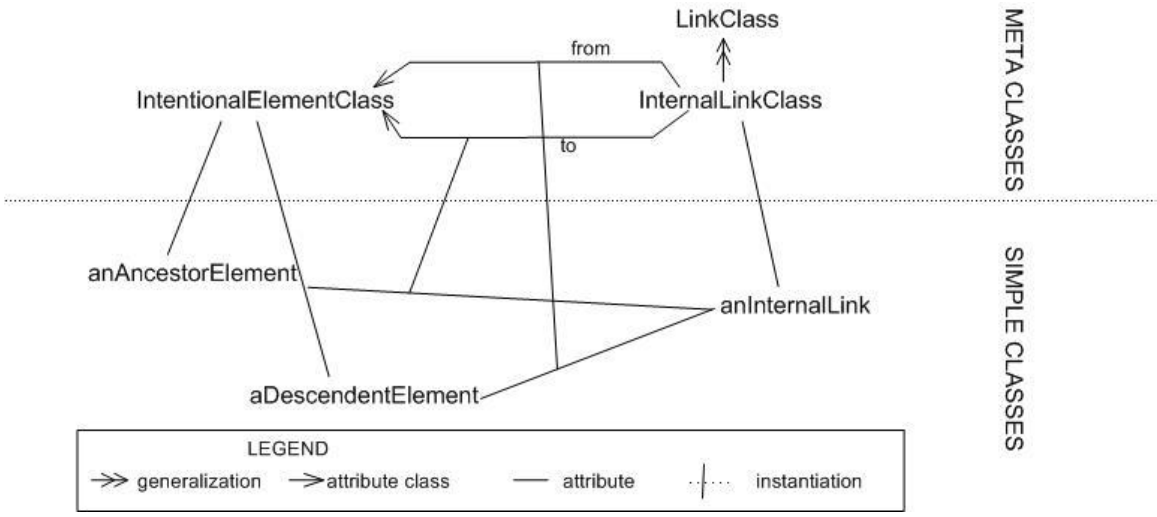


Figure 4.4-6 Partial meta- and domain-model showing the ancestor-descendent relationship

As explained in the previous sections, an actor's internal rationales that are modeled using intentional elements are connected by intentional links. We derive the ancestor-descendent relationship using i* meta-concepts shown in Figure 4.4-6. Links in i* are all directed, and its source and destination are denoted by two attributes, *from* and *to*, respectively. As a result, we define the element at the source end as a direct *descendent* of the one at the destination end, and, in turn, the latter is a direct *ancestor* of the former.

```
% softgoal Timeliness [Arrival Location] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessArrivalLocation IN SoftGoalElementClass
WITH
```

```
       parent
             : AmbulanceCrew_Agent
      links
             : AC_TALtoTS_AndContributionLink
             …
END

% softgoal Timeliness [Service] inside agent Ambulance Crew %
TELL SimpleClass AC_TimelinessService IN SoftGoalElementClass WITH
      parent
             : AmbulanceCrew_Agent
      links
             : AC_TALtoTS_AndContributionLink
             …
END

% and_contribution link from softgoal Timeliness [Arrival Location] to
Timeliness [Service] inside agent Ambulance Crew %
TELL SimpleClass AC_TALtoTS_AndContributionLink IN
AndContributionLinkClass WITH
      from
             : AC_TimelinessArrivalLocation
      to
             : AC_TimelinessService
END
```

Figure 4.4-7 Telos representation of partial model showing the descendent-ancestor relationship

Figure 4.4-7 shows how a direct descendent-ancestor relationship is identified from the underlying Telos representation of an i* model. In this case, softgoal **Timeliness [Arrival Location]** is a direct descendent of softgoal **Timeliness [Service]**. For a more generalized definition, we say an intentional element *e* is a descendent (or ancestor) of *ie* if and only if the former fulfills the following conditions:

1. *ie* and *e* share the same parent;
2.
    a. *e* is a direct descendent (or ancestor) of *ie*; or
    b. there exists an intentional element *e1* such that *e1* is a descendent (or ancestor) of *e* and *ie* is a direct descendent (or ancestor) of *e1*.

Formally, we define those using Telos queries as follows:

Query11

find_direct_descendants(ie: IntentionalElementClass) ::=

§ e: IntentionalElementClass· ∃l:DependencyLinkClass· l.to=ie ∧ l.from=e

Query12

find_all_descendants(ie: IntentionalElementClass) ::=

§ e: IntentionalElementClass· e∈ find_direct_descedent(ie) ∨

(∃d:IntentionalElementClass· e.parent=d.parent ∧ d∈ find_all_descendants(ie)

∧ e∈ find_all_descendants(d))

Query13

find_direct_ancestors(ie: IntentionalElementClass) ::=

§ e: IntentionalElementClass· ∃l:DependencyLinkClass· l.from=ie ∧ l.to=e

Query14

find_all_ancestors(ie: IntentionalElementClass) ::=

§ e: IntentionalElementClass· e∈ find_direct_ ancestors(ie) ∨

(∃d:IntentionalElementClass· e.parent=d.parent ∧ d∈ find_all_ ancestors(ie)

∧ e∈ find_all_ancestors(d))

4.5 Summary

In this chapter, we presented an extension for tackling the scalability issues in representing an i* model. Scalability issues are resolved through the use of views and their attached selection rules. A type of built-in diagram—View Map—is offered in the extension to visualize a reference structure of multiple views derived from the same i* model. The selection rules are built upon a set of meta-concepts that originated from the reformulated i* framework and that was formalized in the view extension.

The extension was embedded in Telos, and the extension was specified independently from the Telos constructs of the core i* framework. Partial meta-models were used to illustrate view classes in our extension, as well as some meta-concepts. We present the formal definitions in a Telos compatible First Order Logic (FOL) form so that these rules can also be implemented using other conceptual modeling languages.

5 Actor Class views

The Actor Class (AC) view allows use of the i* model focusing on actor associations and actor analysis—studying the social and intentional structure among various actors and their specified forms within an organization. However, a Basic AC view (the one derived from a baseline model) can still appear complex. Therefore, it should be scaled down to make each partial view, when visualized, more comprehensible.

We define six partial AC view classes in our view extension; their meta-level constructs have been discussed in Chapter 4. In this chapter we present domain examples (as instances) of each partial view class and define the selection rule attached to it.

Each view type is presented from these four perspectives, and each perspective forms a subsection: Informal Description, Example, Justifications, and Selection Rule. An informal description consists of giving the reader a brief idea of what kinds of elements are qualified for a specific partial view. A domain example from the LAS case study is used to further clarify the idea. We then provide explanation of why that view type is desirable and outline some context of use for it. Last, we provide formal definition of the selection rule attached to each partial view class, which is embedded in Telos and presented using Telos compatible First Order Logic (FOL). The transformation from this FOL form to O-Telos, a Telos compatible conceptual modeling language, is provided in the Appendix.

Section 5.1 gives an overview of the relationship between different types of AC views using a generalized View Map; Section 5.2 presents the Basic AC view and six partial AC views from the 4 aspects discussed in the previous paragraph; Section 5.3 summarizes the results of this chapter.

5.1 Overview

In addition to the Basic AC view, we define six types of partial AC views: Single-Network view, Single-Plain-Actor view, Abstract-Actors-Only view, Plain-Actors-Only view, Agents-Only view, and Direct-Replaceable view. Figure

5.1-1 shows the relationships between different types of views. Each view has a selection rule attached to it, and some of them require input arguments (e.g., Actor <n>). The application of a rule (e.g., singlePlainActorRule) over **Any AC view** (the original view) will result in the corresponding partial AC view (e.g., **Single-Plain-Actor <n> View**).

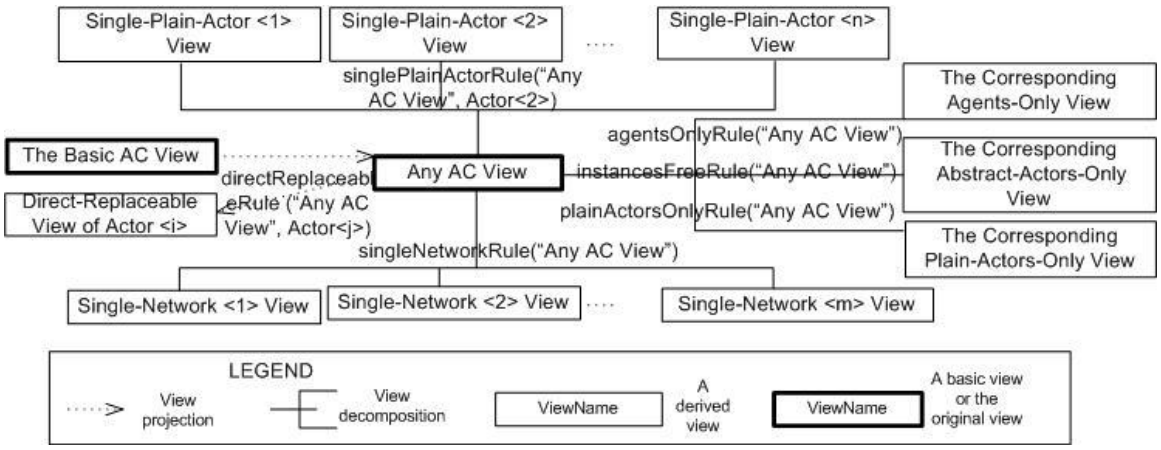


Figure 5.1-1 A generic view map showing a parent AC view and its possible children

Looking at the above diagram, for any AC view, we see that it can be *decomposed* in three ways: by plain actors, by connected networks, or by meta-concept types. A view-decomposition implies the *parent view* (e.g., **Any AC View**) is equivalent to the union of the *child views* (e.g., **Single-Network <1> View**) resulting from the decomposition. For example, suppose there are *n* (where *n* is a positive integer) plain actors in an AC view, then elements in it are partitioned into *n* Single-Plain-Actor views, each containing exactly one plain actor. Moreover, every element contained in the parent view is contained by at least one of the child views. A parent view can also be *projected*, and so result in a child view (e.g., **Direct Relationship View of Actor <i>**) that reflects only partial information from it.

5.2 Details of the AC Views

5.2.1 Basic Actor Class View

Informal Description

The Basic Actor Class View enumerates all actors (plain and specified) and their association links. The association links include the “plays,” “isA,” “is-Part-of,” “covers,” “occupies,” and “INS.” We also need to include the “specifies” and “And” (complete composition) links from our view extension.

The parent view of the Basic Actor Class View is the Baseline Model, so we normally use the latter as the original view over which the selection rule is to be applied.

Example

Since our purpose in this section is to demonstrate the use of various AC view types, we choose four plain actors out of ten from basic AC view of the London Ambulance Service (LAS) case study (You 2003). This partial basic AC view includes just enough elements to show our approach. Figure 5.2-1 visualizes the part of interest. Plain actors that are selected are as follows: Ambulance Crew, LAS Management, Resource Allocator, and Incident Reviewer. This AC view will be used as the original view that the sub-views are derived from throughout this chapter.

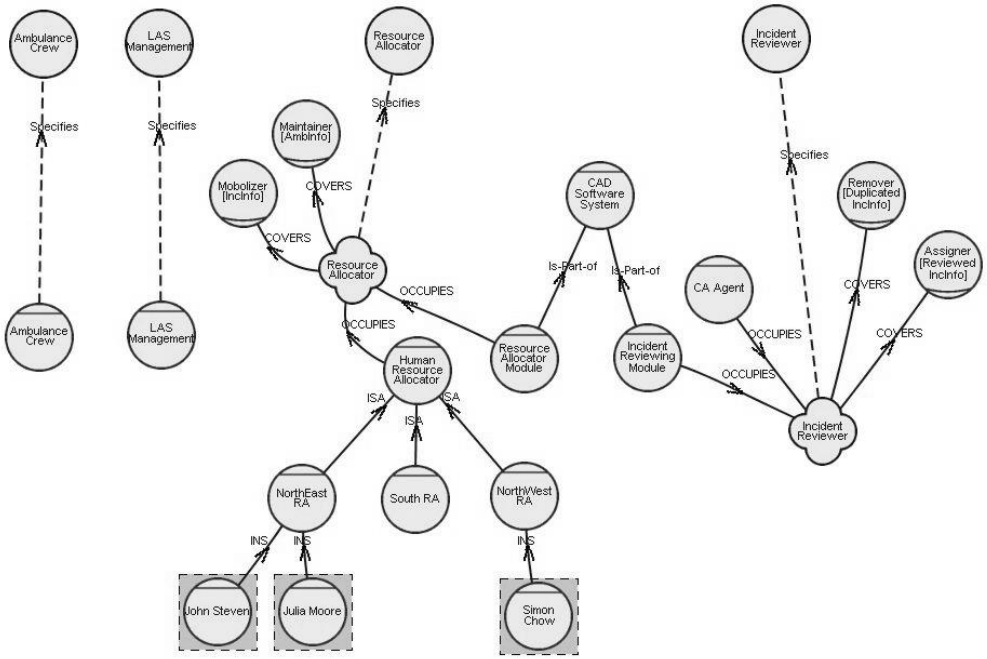


Figure 5.2-1 A partial Basic Actor Class View from the LAS-CAD case study (our original view)

Justifications

As argued previously, a distinguished **Actor Class** (AC) view makes actor identification and actor analysis easier. Yu (Yu 1994) and most of the early literature on the subject did not emphasize on questions such as "how does a plain actor map to a specified one?" and "what are the relationships between the specified ones (which we call *actor associations*)?" The issue appeared adequate with the examples shown in early literature—when there was no such need to distinguish among different forms of actors. Yet social configuration for a medium-size organization (e.g., 500 employees) can become too complex to be expressed in the original SD models. Thus, for ease of communication, it is desirable to have an AC view separate from a SD view.

Separation of the actor associations from dependencies appears natural since these entities focus on different type of analysis: the former on a vertical

hierarchy among a plain actor and its specified forms; the latter on a horizontal dependency network among (normally) actors originated from different plain actors. The associations help perform actor analysis, while the dependencies help perform process analysis. The purpose of actor analysis is to identify actors from the application domain; the purpose of process analysis is to identify process elements (such as goal or task).

Therefore, separation of the AC view is recommended for all application domains that have more than 20 actors (based on our previous experience), or any application domain that has complex social associations among stakeholders.

Selection Rule

Formally, we obtained the corresponding Basic AC View out of a Baseline Model by applying the following query **theBasicActorClassView** over the latter:

Query15

theBasicActorClassView(m:BaselineModelClass)::=

§o:ObjectClass· o∈ m ∧

o∈ {a | a in ActorElementClass} ∪ {l | l in AssociationLinkClass}

In the formulae above, operator “in” denotes “instantiation”. For example, expression “*a* in ActorElementClass” means “object *a* is an instance of class ActorElementClass.”

In the definition of selection rules for partial views, we define for simplicity only the element objects—instances of meta-classes suffixed by “-ElementClass”—in the queries. Whenever link objects—instances of meta-classes suffixed by “-LinkClass”—are not defined explicitly, it implies that a link object, say *l*, should be selected if and only if it satisfies the following conditions:

1. *l* exists in the parent view (e.g., the baseline model *m*); and

2. Elements assigned as both the "from" and "to" attributes of *l* are selected into the child view (e.g., the basic AC view class derived from *m*).

Formally, we define a generic query as one to find all link objects for a given set of element objects as follows:

Query16

%pv: parent view; cv: child view

find_all_links(pv:ViewClass, cv: ViewClass)::=

§ l: LinkClass· (l∈ pv) ∧(∃e1, e2:ElementClass· e1,e2∈ cv ∧ l.from=e1 ∧ l.to=e2)

This rule applies to all definitions of selection rules throughout this thesis, so we will not repeat it later. But in this section, since link type "l in AssociationLinkClass" has been specified in the rule, this rule does not apply.

5.2.2 Single-Network view

Informal Description

A Single-Network view presents a group of specified actors that are connected with association links. Since plain actors are not included in this view, the "specifies" association which ends at a plain actor shall not appear, either.

Given a parent AC view and a specified actor within that view, objects that satisfy one of the following conditions should be selected into this view:

1. The specified actor, say *a*;
2. A specified actor that is connected by an association link with *a*;
3. A specified actor that connects to any previously selected actors in the view.

Example

Figure 5.2-2shows three Single-Network views that are derived from the original AC view. With the plain actors removed, elements in the original view formed 3 networks. Networks 1 and 2 have only one agent each: **Ambulance**

Crew (Figure 5.2-2 (a)) and **LAS Management** (Figure 5.2-2 (b)), respectively. Network 3 combined the specified actor associated with plain actors **Resource Allocator** and **Incident Reviewer**. In most cases, each network corresponds to the set of specified actors for a single plain actor. In Figure 5.2-2 (c), which appears a special case, the two sets of specified actors are joined by agent **CAD Software System**, which appears as the aggregation of the agent **Resource Allocation Module** (specified Resource Allocator) and the agent **Incident Reviewing Module** (specified Incident Reviewer).

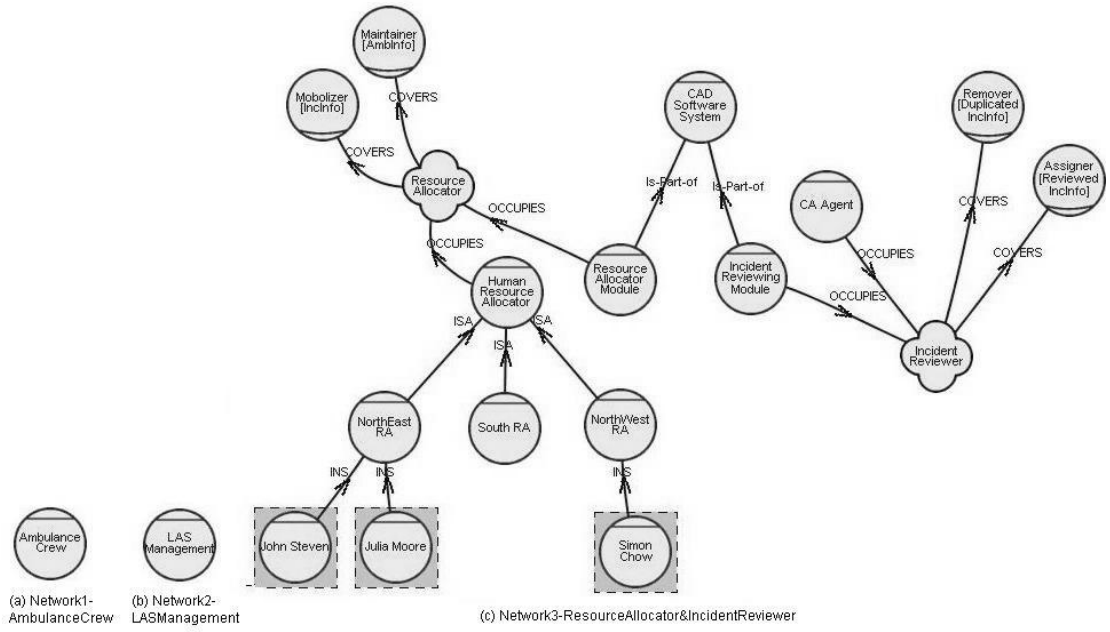


Figure 5.2-2 Single-Network views derived from the original view

Justifications

In most organizations, human resource staff want to identify the responsibilities related with a given position (job profile), and when somebody is hired to take the position, they then keep track of this relationship. This information can be modeled in i* as follows: the responsibilities as roles, the position as a position, and employees as agent instances. When we try to use an i* model in analyzing the situation, the question to answer becomes “What actors share similar responsibilities?” The next possible set of questions might be “How

much commonality do they share?" and "How can they work with each other in an organization?" To answer these questions efficiently, we need to single out only the specified actors that have association links among them.

Grouping specified actors in connected networks appears natural when considering questions listed in the foregoing. The purpose of an AC view is to present actors and their associations; in an organization, this kind of work is normally done in a plain-actor-by-actor manner. Users of the i* model may explore all possible variations of one plain actor, study the possible roles it may cover, the positions that are designed to fulfill it, and the actual class of individual who are considered as this plain actor. One may even assign employees in an organization to the plain actors. Thus, it makes sense to group specified forms of a plain actor in one view.

The Single-Network view can be used to scale down the complexity of the original view, yet not lose information in addressing questions related to a single plain actor.

Selection Rule

Formally, we obtain the corresponding Single-Network view out of any given AC view by applying the following query **singleNetworkRule**.

singleNetworkRule (v:ACViewClass, a:ActorElementClass)::=

§o:ObjectClass· o∈ v ∧o∈ {a, find_all_associated_actors(a) }

Query17

find_direct_associated_actors(a:SpecifiedActorElementClass)::=

§a1:SpecifiedActorElementClass· ∃l:AssociationLinkClass·

l.from=a∧l.to=a1 ∨ l.from=a1∧l.to=a

Query18

find_all_associated_actors(a:SpecifiedActorElementClass)::=

§a1:SpecifiedActorElementClass· a1∈ find_direct_associated_actors(a) ∨

$(\exists a2{:}SpecifiedActorElementClass \cdot a1 \in find\_direct\_associated\_actors(a2) \wedge a2 \in find\_all\_associated\_actors(a))$

5.2.3 Single-Plain-Actor view

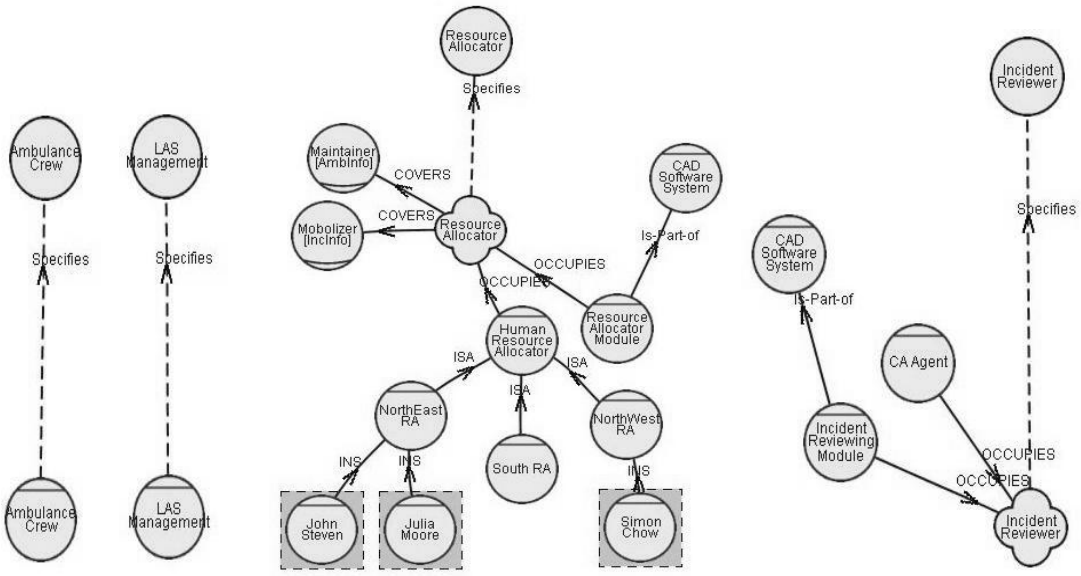
Informal description

A Single-Plain-Actor view presents the family of specified actors who can inherit all external relationships from a given plain actor.

Given a parent AC view and a plain actor within that view, objects satisfying one of the following conditions should be selected into this view:

1. The given plain actor, say *a*;
2. The specified actor that connected with *a* via a Specifies link, which we call the *direct specified actor*, say *dsa*, of *a*;
3. Any specified actors that have a non-is-Part-of link to *dsa*; or any specified actor that has an is-Part-of link from *dsa;*
4. Any specified actors that have a direct non-is-Part-of link *to* or an is-Part-of link *from* any previously selected actors in this view.

Example



(a)AmbulanceCrew&LAS **(b) Resource Allocator** **(c) Incident Reviewer**

Figure 5.2-3 Single-Plain-Actor views derived from the original view

Figure 5.2-3 shows all Single-Plain-Actor views that can be derived from the original AC view. There are four plain actors in the original view, and thus we have four Single-Plain-Actor views. The views for plain actor Ambulance Crew and LAS Management appear extremely simple, so we show them in one diagram (which contains two views). Note that agent **CAD Software System** appears in both the partial view for plain actor **Resource Allocator** and **Incident Reviewer**, and this implies that it can inherit external relationships from both of the plain actors.

Justifications

The modeling process of i* is iterative. Typically, modelers identify plain actors (AC view); next, their dependencies (SD view); and sometimes, internal rationales (SR view) of the plain actors. When more information and a deeper understanding of the application are obtained, modelers differentiate plain actors into their specified forms and sometimes build a network of the specified forms surrounding the plain actor. Subsequently, plain actors in the SD views shall be substituted with one of its specified forms. Thus, showing all candidates for that transition becomes a request from the modeler. The Single-Plain-Actor view is thus designed in response to this modeler's request, i.e., this type of view helps obtain various SD views based on different forms of the actor.

Presenting all the specified forms that can inherit external relationships from a plain actor in one view appears natural in partitioning. The substitute of plain actors in the SD view is done in a plain-actor-by-plain-actor manner. Users of the i* model may explore all possible variations of one plain actor, and choose one from the candidates before moving on to work on another plain actor. Switching views are not necessary for finding the right substitute for a single plain actor.

Even though we do not claim that our view extension supports the i* modeling process. According to earlier discussion in this section, the Single-Plain-Actor

view may help maintain connection between the abstract information (e.g., a SD view showing relationships among plain actors) and the particulars (e.g., the corresponding SD view substituting each plain actor with its specified form). Abstract information is typically collected at an earlier modeling stage. At a later stage, when a better understanding of the application domain is developed through the model refining process, generic information are then refined to particulars. Displaying connections between an actor's generic form and various specified ones helps maintain the consistency when selecting a specified actor to stand in for the corresponding plain one in a SD view. Therefore, this view offers one systematic approach for modelers to follow in refining i* models.

Selection Rule

Formally, we obtain the corresponding Single-Plain-Actor view out of a given AC view by applying the following query **singlePlainActorRule;** we pass the selected plain actor (*a*) as an input argument to the query.

singlePlainActorRule (v:ACViewClass, a:PlainActorElementClass)::=

§o:ObjectClass· o∈ v ∧ o∈ {a, a1=find_direct_specified_actors(a),
find_all_replacing_actors(a1) }

Query19

find_direct_specified_actors(a:PlainActorElementClass)::=

§ta:SpecifiedActorElementClass· ∃l:SpecifiesLinkClass·l.from=ta∧l.to=a

Query20

find_direct_replacing_actors(a:SpecifiedActorElementClass)::=

§a1:SpecifiedActorElementClass· ∃l:AssociationLinkClass·
((l in PartsLinkClass) ∨ (l in CompleteCompositionLinkClass)) ∧
l.from=a ∧ l.to=a1) ∨
((l in ISALinkClass) ∨ (l in INSLinkClass) ∨ (l in PlaysLinkClass) ∨
(l in CoversLinkClass) ∨ (l in OccupiesLinkClass)) ∧
l.from=a1 ∧ l.to=a)

Query21

find_all_replacing_actors(a:SpecifiedActorElementClass)::=

§a1:SpecifiedActorElementClass· a1∈ find_direct_replacing_actors(a) ∨

(∃a2:SpecifiedActorElementClass· a1∈ find_direct_replacing_actors(a2) ∧

a2∈ find_all_replacing_actors(a))

5.2.4 Abstract-Actors-Only view

Informal description

An Abstract-Actors-Only view presents only abstract actors including roles, positions, agents, and any association links among them.

Example

Figure 5.2-4 shows the corresponding Abstract-Actors-Only view of the original AC view. We see that all plain actors and agent instances have disappeared in this view.

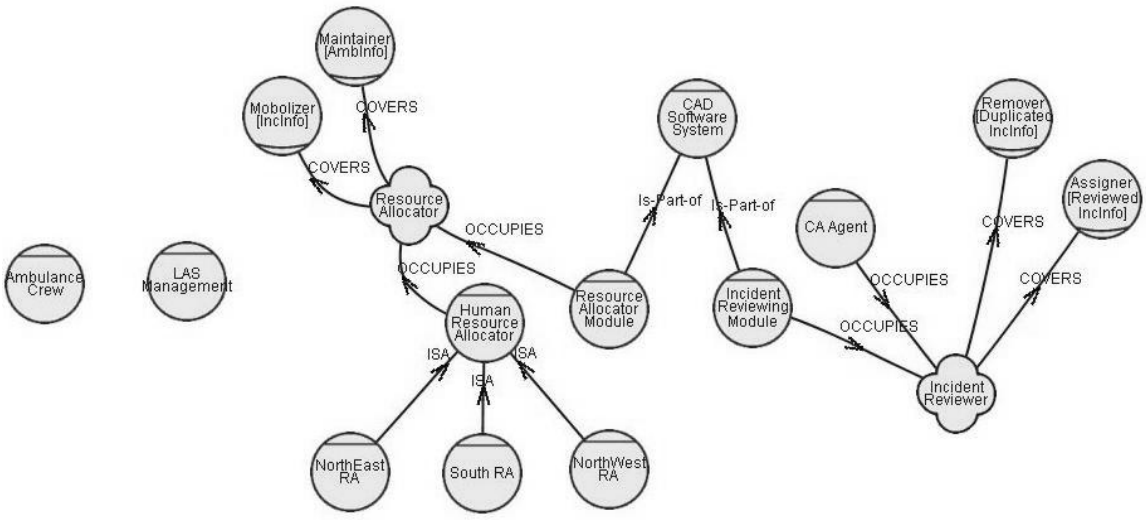


Figure 5.2-4 Abstract-Actors-Only view derived from the original view

Justifications

The Abstract-Actors-Only view focuses on the relationship between the abstract actors, ignoring the abstraction of plain actors and the instantiation of agents. This view may help when an organization has hundreds or thousands of employees, devices, and machines—especially when the individual agent instances in an organization are easily classified to a relatively small number of agents. Under this circumstance, we strongly recommend this view be used to let the user focus on understanding relationships between different forms of actors.

Another advantage of this view is its reusability. Since some organizations from the same industry field may share certain organizational restructures, this kind of view may be reused in a second or third application. For example, every hospital should have the role of doctor, position Principle, agent Emergency, and so forth. Reusable modeling patterns can save time and resource.

Selection Rule

Formally, we obtain the corresponding Abstract-Actors-Only view out of a given AC view by applying the following query **abstractActorsOnlyRule**:

abstractActorsOnlyRule(v:ACViewClass) ::=

§o:ObjectClass· o∈ v ∧o∈ find_all_abstract_actors ()

Query22

find_all_abstract_actors()::=

§a:SpecifiedActorElementClass· (a in AbstractActorElementClass)

5.2.5 Plain-Actors-Only view

Informal description

A Plain-Actors-Only view presents plain actors, their direct specified actors, and the “specifies” links that connect them.

Example

Figure 5.2-5 shows the corresponding Plain-Actors-Only view of the original AC view. We can see that all specified actors have disappeared—except the one that initiates the "specifies" link. Given our external relationship inheritance rule, we need to specify just one direct specified actor for each plain actor. Therefore, this view normally contains only (2*number of plain actors) actor elements.

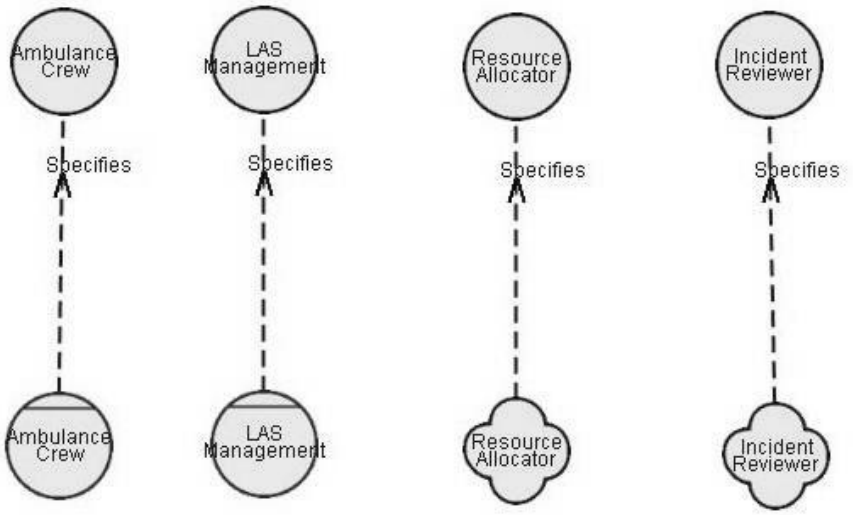


Figure 5.2-5 Plain-Actors-Only view derived from the original AC view

Justifications

Normally, at the beginning of a modeling process or when dealing with higher management personnel, details of an application are not a great concern. Thus, overview questions such as "How many stakeholders are there in an organization?" and "Who are the stakeholders?" may be asked. The Plain-Actors-Only view supplies just enough information for dealing with such questions.

This grouping appears natural in that it may work only on certain phases of the modeling process or in addressing only certain levels of management requirements. Modeling is done in a phase-by-phase manner, so plain actor information required in the beginning phase is not required in a later one. Different management group requires different levels of abstract information, so

detailed (or maybe complex) specified actor information is not required at the CEO level. Therefore, showing only plain actors in a view does not incur much overhead in performing higher abstraction level actor analysis.

Selection Rule

Formally, we obtain the corresponding Plain-Actors-Only view out of a given AC view by applying the following query **plainActorsOnlyRule**:

plainActorsOnlyRule (v:ACViewClass)::=

§o:ObjectClass· o∈ v ∧o∈ {find_all_plain_actors( ) ,

{find_direct_specified_actors(a) %Query19% | a ∈ find_all_plain_actors()} }

Query23

find_all_plain_actors()::=

§a:ActorElementClass· (a in PlainActorElementClass)

5.2.6 Agents-Only View

Informal description

An Agents-Only view presents agents, agent instances, and the association links that connect them.

Example

Figure 5.2-6 shows the corresponding Agents-Only view of the original AC view. We can see that this view contains only agents (e.g., **LAS Management**), agent instances (e.g., **John Steven**), and instantiation links (e.g., the **INS** links between agent and agent instances) among them.

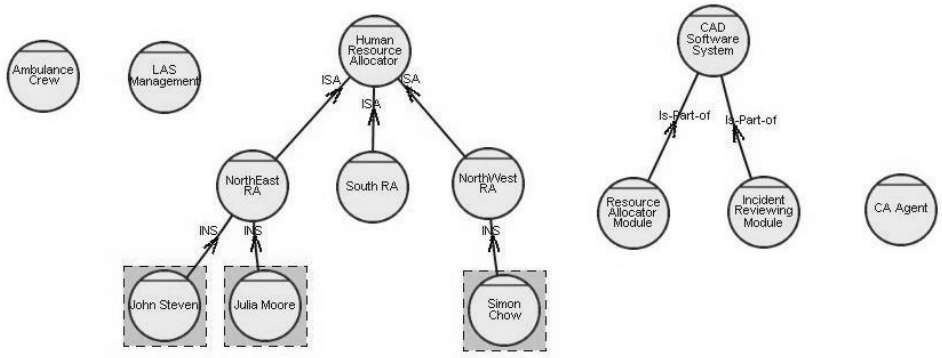


Figure 5.2-6 View showing only the agents for the LAS case study

Justifications

When tackling social issues (organization modeling), sometimes we need only analyze the relationships between physical participant classes. The Agents-Only view can help study the static hierarchy among employees, and may help model organization layout; therefore, this view may help process staff layout in an organization.

However, this view is not necessary when an organization's process can be clearly addressed using the Abstract-Actors-Only view.

Selection Rule

Formally, we obtain the corresponding Agents-Only view out of a given AC view by applying the following query **agentsOnlyRule**:

agentsOnlyRule(v:ACViewClass)::=

§o:ObjectClass· o∈ v ∧o∈ find_all_agents()

Query24

find_all_agents()::=

§a: SpecifiedActorElementClass·

(a in AgentElementClass) ∨ (a in AgentInstanceElementClass)

5.2.7 Direct-Replaceable view

Informal description

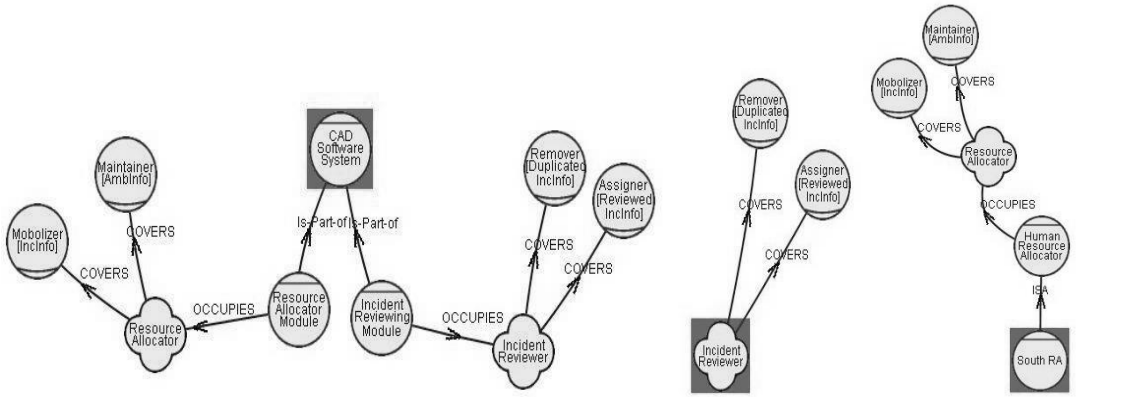
A Direct-Replaceable view presents the family of specified actors whose external relationships can be inherited by a given specified actor, and we call the former a *direct replaceable* to the latter. This direct substitution implies that in any SD view, the given specified actor can stand in for any of the replaceables. There may be external relationships that belong to the given actor directly but not to its replaceables in the SD view.

Given a parent AC view and a specified actor within that view, objects satisfying one of the following conditions should be selected into this view:

1. The given specified actor, say *a*;
2. Any specified actor that has any link other than "is-Part-of" from *a* to it; or any specified actor that has an "is-Part-of" link to *a;*
3. Any specified actor that has a direct link other than "is-Part-of" *to* or an "is-Part-of" link *from* any of the previously selected actors in this view.

Example

Figure 5.2-7 shows Direct-Replaceable views projected over the original AC view. In (a), direct replaceables of agent **CAD Software System** are presented. In (b) and (c), the direct replaceables of position **Incident Reviewer** and agent instance **South RA**, respectively, are shown. The given specified actor is highlighted using a solid rectangle.



(a) CAD Software System **(b) Incident Reviewer** **(c) South RA**

Figure 5.2-7 Direct-Replaceable Views projected over the original AC view

Justifications

The Direct-Replaceable view provides an overview of the family of actors that has a subset of external responsibilities and vulnerabilities to a given actor. This family draws a scope which the given actor can cover. For example, when introducing a new automated system to some organization, we want to know "what responsibilities of which positions occupied by which type of agents are to be implemented in the system." To answer such a question, we need to find out the corresponding actors whose external responsibilities can be covered by the system-to-be; we can use the Direct-Replaceable view of the system-to-be to answer it[6].

Furthermore, this type of view simplifies the SD view by allowing external dependencies to be specified in one place (as some attribute of a single actor). For example, the two agents Human Resource Allocator and Resource Allocating Module share most of the external dependencies (Figure 1.2.1). Under this circumstance, we specify these dependencies to their general form—plain actor Resource Allocator.

[6] Here we assume that an i* model exists for the given organization

Studying the scope of a single specified actor may appear inefficient, yet, in reality, model users study responsibilities in an actor-by-actor manner. Thus, we assume little overhead incurred in using this view. In addition, omitting the plain actor from this view shall not harm its comprehensibility, since this kind of responsibility scope analysis is normally performed at a more detailed level. Abstract level plain actor information appears not relevant.

Selection Rule

Formally, we obtain the corresponding Direct-Replaceable view out of a given AC view by applying the following query **directReplaceableRule**. We pass the selected specified actor (*a*) as an input argument to the query.

directReplaceableRule(v:ACViewClass, a:SpecifiedActorElementClass)::=

§o:ObjectClass· o∈ v ∧ o∈{{a}, find_all_replaceable_actors(a) }

Query25

find_direct_replaceable_actors(a:SpecifiedActorElementClass) ::=

§a1:SpecifiedActorElementClass· ∃l:AssociationLinkClass·

((l in PartsLinkClass) ∨ (l in CompleteCompositionLinkClass)) ∧

l.from=a1 ∧ l.to=a)

∨

((l in ISALinkClass) ∨ (l in INSLinkClass) ∨ (l in PlaysLinkClass) ∨

(l in CoversLinkClass) ∨ (l in OccupiesLinkClass)) ∧

l.from=a ∧ l.to=a1)

Query26

find_all_replaceable_actors(a:SpecifiedActorElementClass) ::=

§a1:SpecifiedActorElementClass· a1∈ find_direct_replaceable_actors(a) ∨

(∃a2:SpecifiedActorElementClass· a1∈ find_direct_replaceable_actors(a2) ∧

a2 ∈ find_all_replaceable_actors (a))

5.3 Summary

In this chapter, we presented relationships between the Basic AC view and six types of partial AC views. Each of the views was also explored in detail. The AC views studied in this section are: the Basic AC view, the Single-Network view, the Single-Plain-Actor view, the Abstract-Actors-Only view, the Plain-Actors-Only view, the Agents-Only view, and the Direct-Replaceable view.

View relationships were illustrated using a generic View Map that fits for all applications. View decomposition and projection directions were also shown.

The AC views were presented formally and informally. An informal description gives the reader a basic idea of what kinds of elements are qualified for a specific partial view. The formal definition of the selection rule, which is attached to each view class, makes it possible to automate these views in an i* modeling tool. Some discussion about the benefits and limitations of the each view type are also included. An original AC view obtained from the LAS case study was used as the running example to demonstrate the results of decomposition and projection over it.

6 Strategic Dependency Views

The purpose of the Strategic Dependency (SD) view is to express the "intentional description of a process in terms of a network of dependency relationships between actors" (Yu 1994), and to express the intertwined negative or positive contributions towards those dependency relationships, among actors.

The Basic SD view should, by definition, include all types of actors and all dependency links or external intentional links among them. However, when a view is visualized, it is normally redundant to show different forms of actors that are basically related to the same plain actor in one diagram, since these actors share most of the external relationships. For example, agent **Resource Allocator Module** and position **Resource Allocator** from the LAS case study both depend on an **Ambulance Crew** to supply accurate ambulance information (AmbInfo). Therefore, we normally present an SD view by selecting one actor (or more non-overlapping ones) representing each plain actor. In addition, each type of Basic SD view can still appear complex. Therefore, we need to scale down the view to make each partial view, when visualized, more comprehensible.

We define two basic and two partial SD view classes in our view extension, and we discussed their meta-level constructs in Chapter 4; in this chapter we present domain examples (as instances) of each view class and define the selection rule attached to it. We adopt the same pattern as used in the AC views, and explore each partial view from these four perspectives: Informal Description, Example, Justifications, and Selection Rule.

Section 6.1 uses generalized View Maps to give an overview of the relationship between different types of SD views; Section 6.2 presents two Basic SD views and two partial SD views from the four aspects mentioned in the previous paragraph; Section 6.3 summarizes the results of this chapter.

6.1 Overview

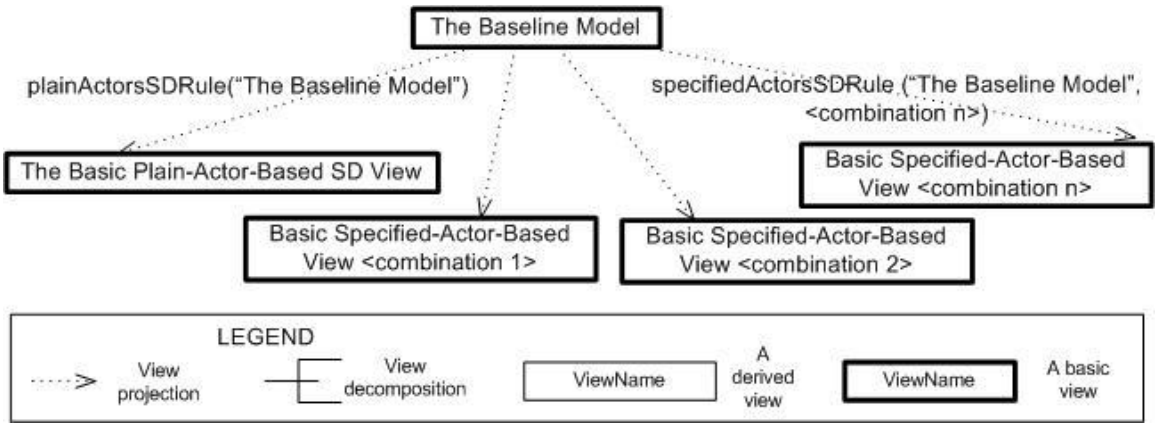


Figure 6.1-1 Generalized view map showing relationships between different forms of Basic SD views

Figure 6.1-1 presents the relationship between different forms of Basic SD views. Each Basic SD view corresponds to one Plain Actor SD view. Several Specified Actor SD views can be derived from the Baseline Model, and the derivation process requires actor association information so that external relationships for a selected actor can be calculated following the external relationship inheritance rule. For example, if agent **Resource Allocator Module** is showing in some SD view standing in for plain actor **Resource Allocator**, then it will inherit all the external relationships from position Resource Allocator (following the "plays" link), and recursively from plain actor Resource Allocator (following the "specifies" link). Since all these forms of Basic SD views share the same external relationships pattern, we do not distinguish them again when they are scaled down further into partial views.

Any basic or partial SD view, regardless of the form of actors shown, can be further decomposed into views smaller in size and simpler in inter-actor relationships than the original. We illustrate this point using Figure 6.1-2. Our first approach is to decompose an SD view (e.g., **Any SD View**) into Single-Actor-Focus views (e.g., **Single Actor<1> View**). An SD view can also be

decomposed into Pair-wise-Actors views (e.g., **Pair-wise Actor<1>, Actor<n> View**) for the selected other actor pairs (e.g., **{Actor<1>, Actor<n>}**).

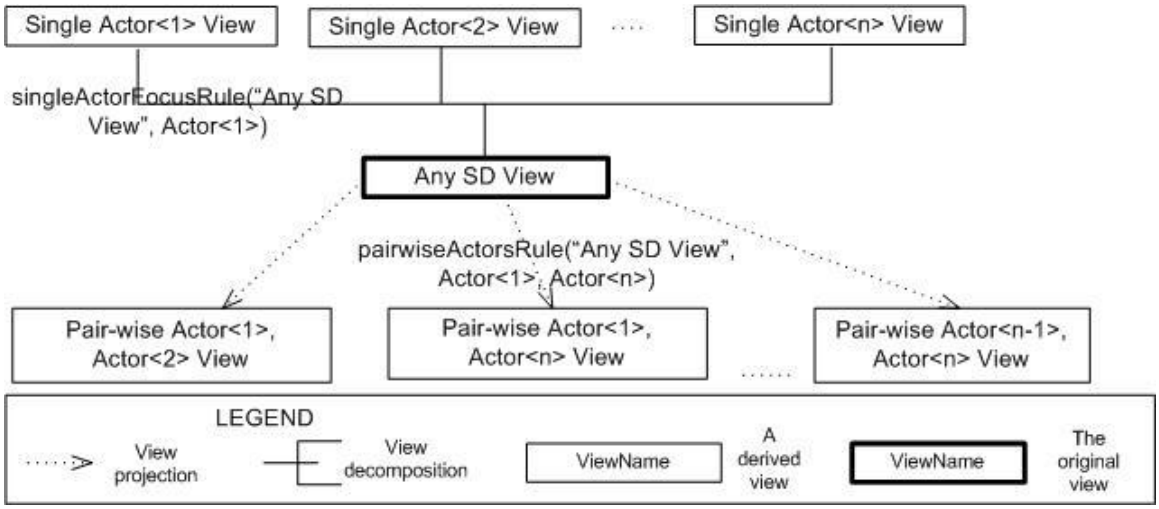


Figure 6.1-2 Generalized view map showing possible decomposition of "Any SD View"

6.2 Details of the SD Views

6.2.1 Plain- versus Specified-Actor-Based SD View

Informal Description

Both the Plain-Actor-Based and the Specified-Actor-Based SD view are designed to present inter-actor external relationships.

A Plain-Actor-Based SD view includes all plain actors as well as the external dependency and contribution links among these plain actors.

A Specified-Actor-Based SD view includes selected specified actors that cover the responsibilities of all plain actors in the Plain-Actor-Based form. External dependency and contribution links among the selected specified actors are also included in this view.

Example

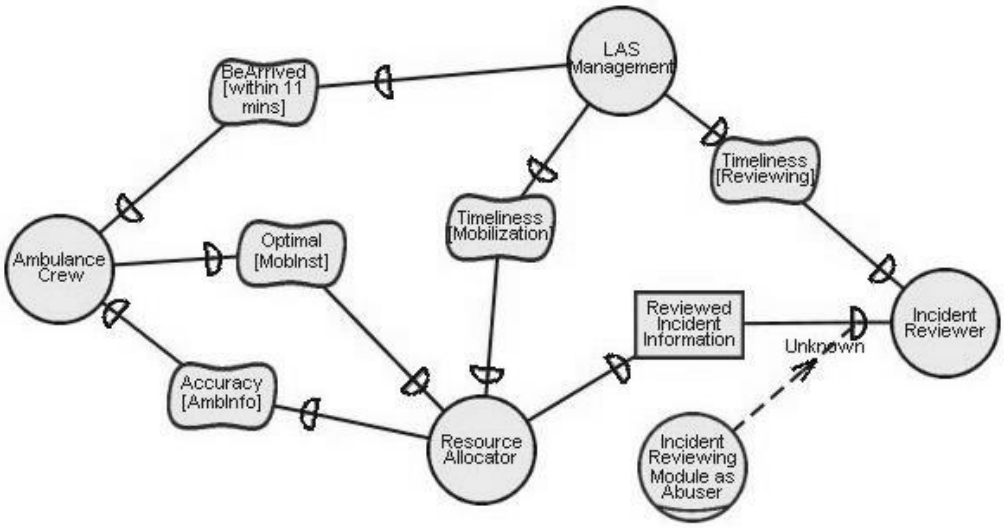


Figure 6.2-1 Partial Plain-Actor-Based SD view from the LAS case study

Figure 6.2-1 shows the external relationships between four plain actors (**Ambulance Crew**, **LAS Management**, **Resource Allocator** and **Incident Reviewer**) from the LAS case study, corresponding to the Plain-Actor-Based form of an SD view. Given the external relationship inheritance rule along actor associations, we can use the information from the corresponding actor associations shown in Figure 6.2-2 to substitute the plain actors with one of its specified forms.

Figure 6.2-3 presents the same part of the underlying model, yet in the Specified-Actor-Based form. From Figure 6.2-2, we know that position **Incident Reviewer** specifies plain actor **Incident Reviewer**, and from the inheritance rule discussed in our reformulated i* framework we know that the former inherits all external relationships from the latter. Thus, the position Incident Reviewer also has the external dependency Timeliness [Incident Reviewing]. All other substitutes of actors shown in Figure 6.2-3 adopted a similar one-to-one manner—as described previously. Except for plain actor Incident Reviewer who was replaced by 3 specified forms (position **Incident Reviewer**, agent **Incident Reviewing Module**, and agent **CA Agent**).

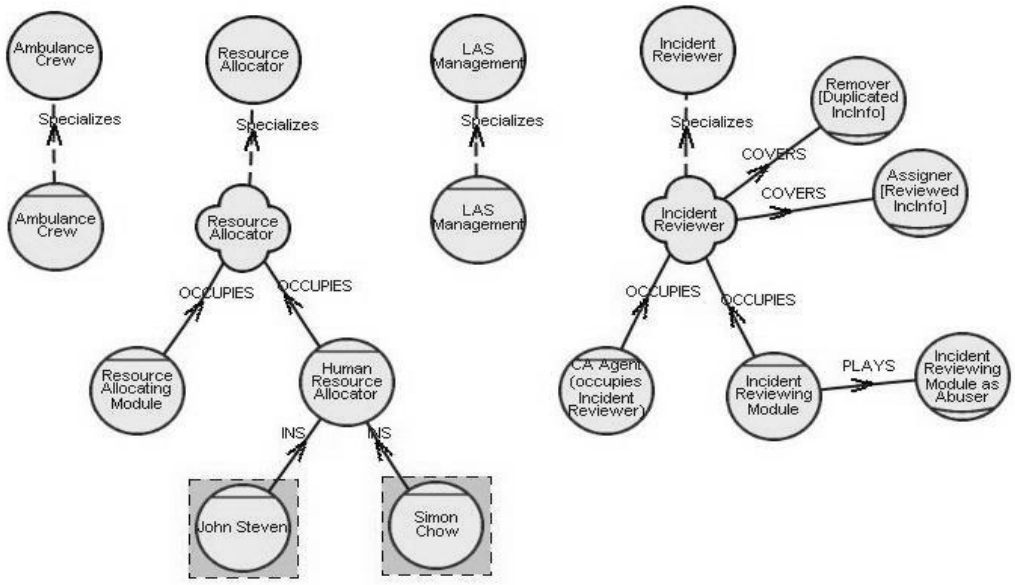


Figure 6.2-2 Partial Basic AC view from the LAS case study showing the associations of the four plain actors

Abstract external relationships must be instantiated as well. In Figure 6.2-3, the abstract external resource dependency **Reviewed Incident Information** is replaced by two resource dependencies: **Reviewed Incident Information (paper-based)** and **Reviewed Incident Information (machine-based)**, each directing to one of the two agents. The softgoal dependency **Timeliness [Incident Reviewing]** was redirected to position Incident Reviewer. The external correlation link, starting from role **Incident Reviewing Module as Abuser,** was also refined to affect only the machine-based resource dependency towards agent **Incident Reviewing Module;** its label changes from *Unknown* to *Hurt*. The label of the abstract correlation link is set to *Unknown* because position Incident Reviewer is an abstract form of the two agents (**CA Agent** and **Incident Reviewing Module**), yet the correlation link affects only one of them and, thus, the combined effect is unknown.

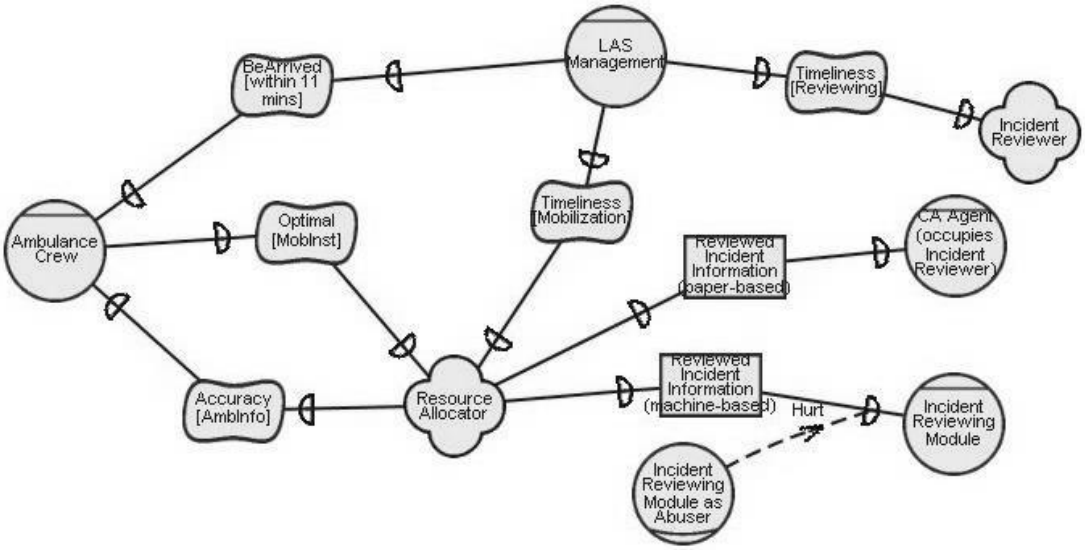


Figure 6.2-3 The Specified-Actor-Based SD view corresponding to the Plain-Actor-Based SD view

Justifications

Both the Plain-Actor-Based view and the Specified-Actor-Based SD view are designed to present inter-actor external relationships. The Plain-Actor-Based view assumes the highest level of abstraction: showing stakeholders in a plain actor form and external relationships in a generic form. The Specified-Actor-Based view assumes more detail: replacing plain actors with their specified forms and refining the generic external relationships according to the set of specified actors selected.

The separation of these two views appears natural since they serve different purposes, and different levels of detail are required at different times. For example, in an organization, the CEO may need very brief information, so the very abstract form of information would be required; but an on-site manager may need to know the exact and specified employee assignments, so a specified form would be a must.

The two views shown in this section could be more useful during the modeling process; however, we do not study this issue in this thesis. The modeling process

is an ongoing one, and sometimes different levels of information are required for storage in the same model. Using our approach, a more abstract SD view can be systematically detailed into a concrete one with the help of actor associations from the AC view, without duplication of any external dependencies. The Single-Plain-Actor views (Section 5.2.3) and Direct-Replaceable views (Section 5.2.7) are designed to serve this systematic refinement of SD views (see the corresponding sections for more detail).

Nevertheless, external relationships should be consistently mapped between the Plain-Actor-Based form and various Specified-Actor-Based forms. Precautions are required when performing this conversion (mapping). There are three general cases for this mapping:

1. The relationship is mapped as is. (e.g., Optimal [MobInst], Accuracy [AmbInfo]).

2. An abstract relationship is decomposed or analyzed. (e.g., Reviewed Incident Information mapped to two resource-dependums; the Unknown correlation link is analyzed to just affect the machine-based Incident Information and is refined to Hurt).

3. When two plain actors are replaced by a specified one that covers both of them, the external relationships between them become internal and will not be included in the Specified-Actor-Based view.

Selection Rule

We need to identify clearly what type of i* objects are qualified for the SD view in general, so we first give the definition of a generic Basic SD view. Formally, we can obtain the corresponding Basic SD view from a Baseline Model by applying the following query **theBasicStrategicDependencyView**:

theBasicStrategicDependecyView(m:BaselineModelClass)::=

§o:ObjectClass· o∈ m ∧o∈ { {a | a in ActorElementClass},
{e | e in DependumElementClass}, {l | l in DependencyLinkClass},

{l | l ∈ find_all_external_links()} }

For any given SD view, we can obtain its corresponding Plain-Actor-Based view by applying the query **plainActorsSDRule**:

plainActorsSDRule (v:SDViewClass)::=

§o:ObjectClass· o∈ v ∧ o∈ {A=find_all_plain_actors(), find_inter_dependums(A) , find_inter_dependencies(A), find_all_inter_external_links(A)}

For any given SD view and a set of selected specified actors, we obtain its corresponding Specified-Actor-Based view by applying the query **specifiedActorsSDRule**:

specifiedActorsSDRule(v:SDViewClass, A={a1,…,an}:ActorElementClass) ::=

§o:ObjectClass· o∈ v ∧ o∈ { A, find_inter_dependums (A),
find_inter_dependencies(A), find_all_inter_external_links(A) }

Query27

find_inter_dependums(A={a1,…,an}:ActorElementClass) ::=

§e:DependumElementClass·
∃l1,l2:DependencyLinkClass;a1,a2:ActorElementClass· (a1, a2 ∈ A) ∧
(l1.from=e=l2.to) ∧ (l1.to=a1 ∨ l1.to.parent=a1) ∧
(l2.from=a2 ∨ l2.from.parent=a1)

Query28

find_inter_dependencies(A={a1,…,an}:ActorElementClass) ::=

§l:DependencyLinkClass· ∃a:ActorElementClass,b:DependumElementClass·
(a∈ A) ∧ (b∈ find_inter_dependums(A)) ∧
(l.from.parent=a ∧l.to=b ∨ l.to.parent=a ∧l.from=b)

Query29

find_direct_inter_external_links(A={a1,…,an}:ActorElementClass) ::=

§l:IntentionalLinkClass· ∃dl:DependencyLinkClass·
dl∈ find_inter_dependencies(A) ∧

(∃e:IntentionalElementClass·e.parent∈ A ∧ l.from=e ∧ l.to=dl)

Query30

find_all_inter_external_links(A={a1,…,an}:ActorElementClass)::=

§l:IntentionalLinkClass· l.from.parent ∈ A ∧

(l ∈ find_direct_inter_external_links(A) ∨

(∃l2:IntentionalLinkClass· l2∈ find_all_inter_external_links(A) ∧ l.to=l2))

6.2.2 Single-Actor-Focus view

Informal Description

A Single-Actor-Focus view centers on a single actor and can apply to both SD and SR views. In case of an SD view, the view presents the selected actor, the dependums to which it connects, the external links that affect those dependums, the depender/dependee actors of the dependums, and the originator of the external links. External links that are originated from the selected actor and the links to which these external links end at are also included in this view.

For clarity, we restate here the informal definition of an *external link*. An intentional link that ends at a dependency link is an external link, and a link that starts from an actor and ends at an external link is an external link, also. The formal definition of external link can be found in Section 4.4.5.

Example

Figure 6.2-4 shows the Single-Actor-Focus view of position **Resource Allocator** (the given actor) from the LAS case study. This view includes softgoal dependum **Optimal [MobInst]** (a dependum) and agent **Ambulance Crew** (a depender to the dependum). This view also includes the Hurt correlation link (an external link) and role **Incident Reviewing Module as Abuser** who exerts a partially negative (Hurt) effect on Resource Allocator's outgoing resource dependency **Reviewed Incident Information** (machine-based).

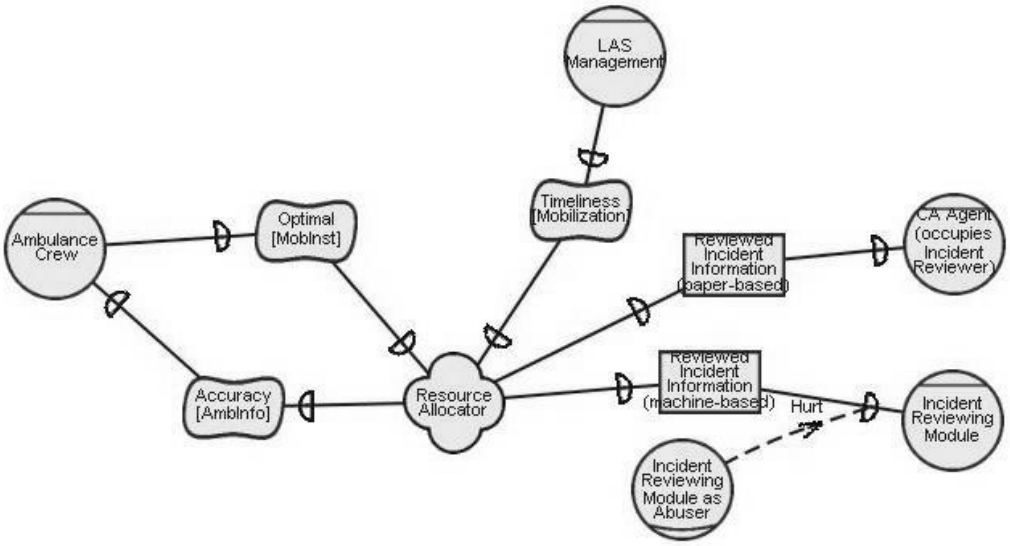


Figure 6.2-4 Single-Actor-Focus view for position Resource Allocator from the LAS case study

Justifications

In (Yu 1994), one use of the SD view is to perform node analyses, studying the "confluence of various incoming and outgoing [external relationships]… at an actor…" From the outgoing dependencies, we can determine what opportunities are available for an actor to achieve certain goals, and what vulnerabilities could make the achievement of those goals fail. From the incoming dependencies, we learn the responsibilities that other actors require of this actor. External links to the dependency links (or dependums) indicate the extra difficulty or help this actor receives from the initiator of the link. The formation of this view corresponds to the activities performed by i* model users.

Presenting SD views in a single actor form does not introduce a large overhead to the analysis. All external relationships surrounding the given actor are included in this view, so questions related to the given actor can be answered without consulting information not presented in it. Therefore, we suggest that when inter-actor relationships in an SD view grow complex (lots of cross-over of links), i* users apply this approach.

Selection Rule

Formally, we obtain the corresponding Single-Actor-Focus view from a given SD view by applying the following query **singleActorFocusSDRule**. We pass the selected actor as an input argument (*a*) to the query.

singleActorFocusSDRule(v:SDViewClass, a:ActorElementClass) ::=

§o:ObjectClass· o∈ v ∧o∈ {a,
find_incoming_dependencies_to_actor(a), %Query 3
find_incoming_dependeums_to_actor(a),
find_indirect_incoming_dependencies_to_actor(a),
find_dependers_to_actor(a),
find_outgoing_dependencies_from_actor(a), %Query 4
find_outgoing_dependums_from_actor(a),
find_indirect_outgoing_dependencies_from_actor(a),
find_dependees_from_actor(a),
find_externallinks_to_incoming_dependency(a),
find_externallinks_originator_to_incoming_dependency(a),
find_externallinks_to_indirect_outgoing_dependency(a),
find_externallinks_originator_to_indirect_outgoing_dependency(a),
find_externallinks_from_actor(a),
find_externallinks_to_externallinks_from_actor(a),
find_externallinks_target_from_actor(a) }

Query31

find_incoming_dependums_to_actor(a:ActorElementClass)::=

§d:DependumElementClass· ∃l:DependencyLinkClass·
l.from=d ∧ l ∈ find_incoming_dependencies_to_actor(a)

Query32

find_indirect_incoming_dependencies_to_actor(a:ActorElementClass)::=

§l:DependencyLinkClass· ∃de:DependumElementClass
l.to=de ∧ de ∈ find_incoming_dependums_to_actor(a)

Query33

find_dependers_to_actor(a:ActorElementClass)::=

§a1:ActorElementClass·

∃d:DependumElementClass, l:DependencyLinkClass·

d ∈ find_incoming_dependums_to_actor(a) ∧

l.to=d ∧ l ∈ find_outgoing_dependencies_from_actor(a1)

Query34

find_outgoing_dependums_from_actor(a:ActorElementClass)::=

§d:DependumElementClass· ∃l:DependencyLinkClass·

l.to=d ∧ l ∈ find_outgoing_dependencies_from_actor(a)

Query35

find_indirect_outgoing_dependencies_from_actor(a:ActorElementClass)::=

§l:DependencyLinkClass· ∃de:DependumElementClass

l.from=de ∧ de ∈ find_outgoing_dependums_from_actor(a)

Query36

find_dependees_from_actor(a:ActorElementClass)::=

§a1:ActorElementClass·

∃d:DependumElementClass, l:DependencyLinkClass·

d ∈ find_outgoing_dependums_from_actor(a) ∧

l.from=d ∧ l ∈ find_incoming_dependencies_to_actor(a1)

Query37

find_externallinks_to_incoming_dependency(a:ActorElementClass)::=

§l:IntentionalLinkClass· ∃dl:DependencyLinkClass·

l.to=dl ∧ dl∈ find_incoming_dependencies_to_actor(a)

Query38

find_externallinks_originator_to_incoming_dependency(a:ActorElementClass)::=

§a:ActorElementClass· ∃l:IntentionalLinkClass·

l∈ find_externallinks_to_incoming_dependency(a) ∧

(∃e:IntentionalElementClass·l.from=e ∧ e.parent=a)

Query39

find_externallinks_to_indirect_outgoing_dependency(a:ActorElementClass)::=

§l:IntentionalLinkClass· ∃dl:DependencyLinkClass·

l.to=dl ∧ dl∈ find_indirect_outgoing_dependencies_from_actor(a)

Query40

find_externallinks_originator_to_indirect_outgoing_dependency(a:ActorElementClass)

::=

§a:ActorElementClass· ∃l:IntentionalLinkClass·

l∈ find_externallinks_to_indirect_outgoing_dependency(a) ∧

(∃e:IntentionalElementClass·l.from=e ∧ e.parent=a)

Query41

find_externallinks_from_actor(a:ActorElementClass)::=

§l:IntentionalLinkClass·∃e:IntentionalElementClass·

l.from=e ∧ e.parent=a ∧ l∈ find_all_external_links()

Query42

find_externallinks_target_from_actor(a:ActorElementClass)::=

§l0:LinkClass·∃l:IntentionalLinkClass·

l∈ find_externallinks_from_actor (a) ∧ l.to=l0

Query43

find_externallinks_to_externallinks_from_actor(a:ActorElementClass)::=

§l0:LinkClass·∃l:IntentionalLinkClass·

l∈ find_externallinks_from_actor (a) ∧ l0.to=l

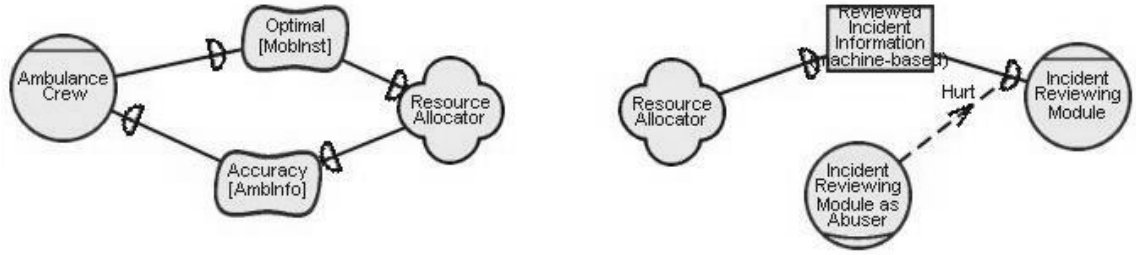
6.2.3 Pair-wise-Actors View

Informal Description

A Pair-wise-Actors view presents two selected actors and the external relationships between them. This view also applies to both the SD and the SR view.

Example

Figure 6.2-5(a) shows the Pair-wise-Actors view between position **Resource Allocator** and agent **Ambulance Crew**, and Figure 6.2-5(b) shows the view between position **Resource Allocator** and role **Incident Reviewing Module as Abuser**. Note that in (b), agent Incident Reviewing Module appears just for added clarity and it can be ignored.



(a) Ambulance Crew vs. Resource Allocator **(b) RA vs. Incident Reviewing Module as Abuser**

Figure 6.2-5 Pair-wise-Actors SD views from the LAS case study

Justifications

Even though this view can sometimes dramatically simplify representation, we do not recommend excessive use of the view—because applying it can create a combinatorial explosion problem (number of different pairs of actors). Thus, this view should be used conservatively and selectively, so we give these guidelines:

1. The number of total actors is manageable (say < 20).
2. There are significant requests that the relationships between some pair of actors be addressed.

3. Choose only the pairs that require this level of analysis.

Selection Rule

Formally, we obtain the corresponding Pair-wise-Actors view from a given SD view by applying the following query **pairwiseActorsRule**. We pass the selected actor pair {a0, a1} as the input arguments to the query.

pairwiseActorsRule(v:[SDViewClass | SRViewClass], {a0, a1}:ActorElementClass) ::=

§o:ObjectClass· o∈ v ∧ o∈ { {a0, a1},

find_inter_dependums({a0, a1}), %Query27

find_inter_dependencies({a0, a1}), %Query28

find_all_inter_external_links({a0, a1}) } %Query30

6.3 Summary

We presented in this chapter various views we can use to simplify the Basic SD views. We defined the Plain-Actor-Based and Specified-Actor-Based views to represent the inter-actor relationship network. These two types of basic views are at different levels of abstraction and, thus, contain different levels of detail. We also defined two types of partial Strategic Dependency (SD) views in our view extension.

The relationship between different view types was illustrated using generalized view maps. Two View Maps are presented: one for explaining the relationship between different forms of Basic SD views, and another for explaining the relationship between the basic view and the partial views.

We presented the SD views from both informal and formal aspects. An informal description gives the reader a basic idea of what kinds of elements are qualified for a specific partial view. The formal definition of the selection rule attached to each view class makes it possible to automate these views in an i* modeling tool. We included also some justification for each view.

7 Strategic Rationale Views

The Strategic Rationale (SR) view aims to "provide the intentional description of processes in terms of process elements and the rationales behind them." In other words, the layout of the reasoning structure internal to an actor, based on its relationship with others presented in the SD model, is represented in the SR model. (Yu 1994)

The Basic SR view should, by definition, include all types of elements involved in the SD view (actors, dependency links, and external links), and intentional elements and intentional links inside the boundary of each actor. However, when the view is visualized, it is extremely hard to show all information contained in the Basic SR view just by using one diagram for most real-world projects. The modeling tool could get out of memory when the diagram reaches a certain size. Even though a huge diagram is produced, it would be difficult for users to retrieve information. As a result, the Basic SR view needs to be communicated using a set of inter-connected smaller views.

We scale down the Basic SR view first by Single-Actor-Focus views. Since any SR view shares information external to actors with its corresponding SD view, we can focus on a single actor each time, and proceed to other actors through the external connection. In some cases, even a Single-Actor-Focus view could appear complex. Therefore, we need to further scale it down to make each sub-view, when visualized, more comprehensible.

We define in our view extension seven new partial SR view classes—besides the Single-Actor-Focus and Pair-wise-Actors view defined in the previous chapter. The meta-level constructs of these view classes were discussed in Chapter 4; in this chapter we present domain examples (as instances) of the view classes and define the selection rule attached to each of them. We adapt the same pattern as used in the AC views, and

explore each partial view from these four perspectives: Informal Description, Example, Justifications, and Selection Rule.

Section 7.1 gives an overview of the relationship between different types of SR views using a generalized View Map; Section 7.2 presents the basic Single-Actor-Focus SR view and 7 newly defined partial SR views from the four aspects discussed in the previous paragraph; Section 7.3 summarizes the results of this chapter.

7.1 Overview

The relationship between the Basic SR view and a Single-Actor-Focus SR view appears the same as the one presented in the SD views. Since we discussed that in the previous chapter we do not repeat it here; furthermore, we use a Single-Actor-Focus SR view as our original view.

Figure 7.1-1 shows a generalized hierarchy of the decomposition of a Single-Actor-Focus SR view. Any such view (e.g., **Single Actor<i> SR View**) can be further decomposed into an Internal (e.g., **Single Actor<i> Internal View**) and an External view (e.g., **Single Actor<i> External View**). An Internal view can be further decomposed into a Functional (e.g., **Internal-Functional Elements View**) and a Non-functional view (e.g., **Internal-Non-functional Elements View**), and the Non-functional view can again be decomposed into a set of Single-Softgoal views (e.g., **Single-Softgoal<j> View**). An External view can be decomposed into a set of Single-Affected-Dependum views (e.g., **Single Dependum<1> View**) or Single-Affected-Actor views and (e.g., **Effects to Actor<m> View**).

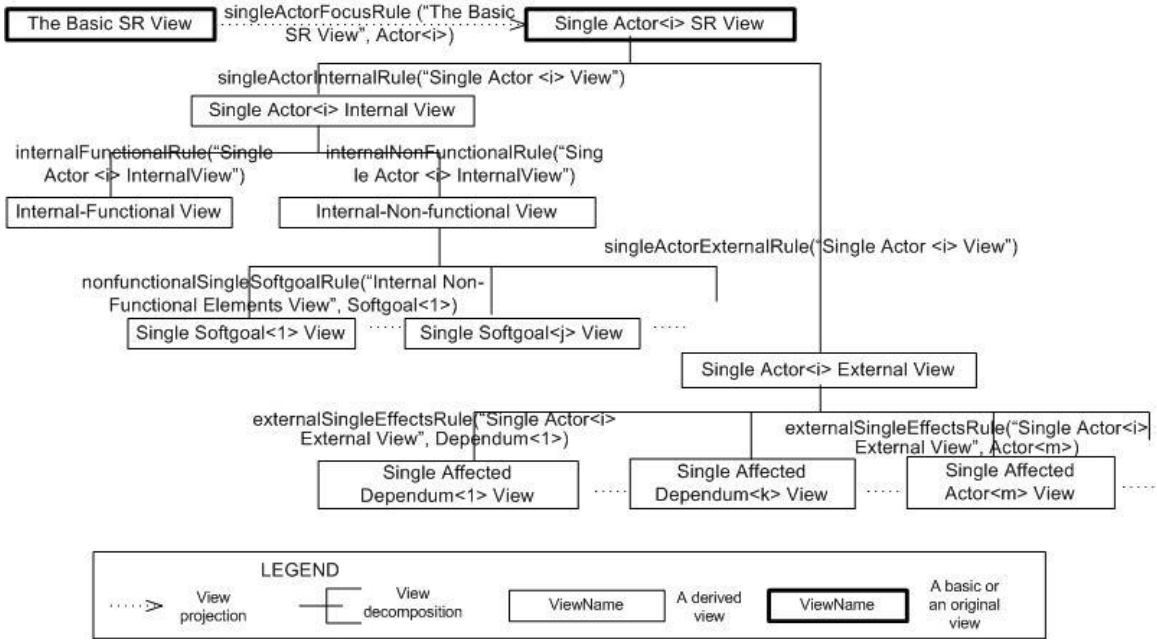


Figure 7.1-1 Generalized view map showing decomposition hierarchy from a Single-Actor-Focus SR view to its sub-views

The decomposed hierarchy of SR views can be used in a reverse direction to perform the evaluation process across different EVLR views in a systematic manner. Figure 7.1-2 shows an example of how this idea can be applied. The sample shows the label propagation direction from Single-Affected-Dependum views and Single-Softgoal views to External and Internal views, respectively. From the External and Internal views to the Single-Actor-Focus view for actor "Actor<i>", and then propagate to the Single Actor View for another actor (e.g., **Actor<x>**). However, sometimes we cannot finish label elements in one Single-Actor-Focus view before we move to another one, and iteration among different actors may become frequent. This issue itself deserves further research; yet it does not affect our approach, so we disregard it in this thesis.

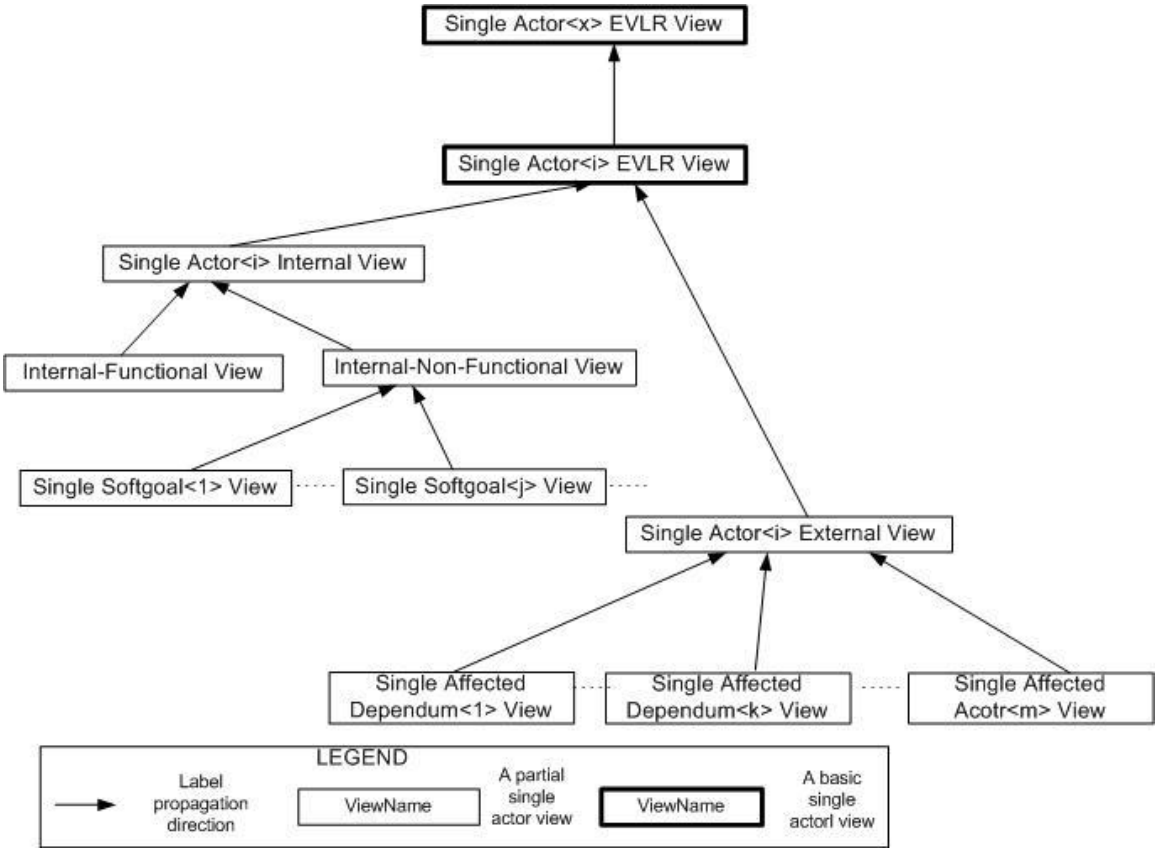


Figure 7.1-2 Generalized view map showing the label propagation direction for the evaluation process using the hierarchy of SR sub-views

7.2 Details of SR Views

Since any SR view shares information external to actors with its corresponding SD view, we can focus on a single actor at each time and proceed to other actors through the external information. Moreover, the purpose of an SR view is to systematically study the internal rationales behind some external relationships of an actor. We thus use the Single-Actor-Focus SR view as our original view in this section.

7.2.1 Single-Actor-Focus SR View

Informal Description

A Single-Actor-Focus view centers on a single actor and can apply to both SD and SR views. In the case of an SR view, the view presents these elements included in the corresponding SD view: the selected actor, the dependums to which it connects, external links that affect those dependums, the depender/dependee actors of the dependums, and the originator of the external links. In addition, the internal goal-oriented structure, including intentional elements and links internal to the selected actor, are presented only in the SR version.

Example

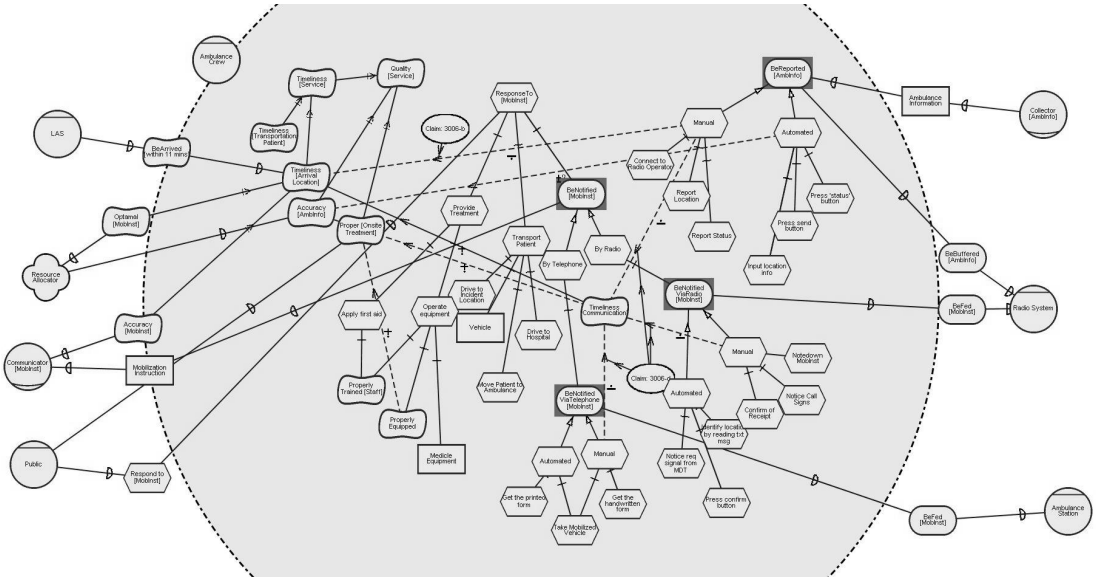


Figure 7.2-1 Single-Actor-Focus SR view showing internal rationales of agent Ambulance Crew from the LAS case study (the original view)

Figure 7.2-1 shows an example of a Single-Actor-Focus SR view from the LAS case study. From the figure, we learnt that the agent Ambulance Crew has three top-level intentional elements: softgoal **Quality [Service]**, task **RespondTo [MobInfo]**, and goal **BeReported [AmbInfo]**. The view also enumerates detailed elements and routines in achieving the top-level intentions. For example, we

know from the means-ends links that AmbInfo can be reported (goal BeReported [AmbInfo]) either manually (task **Manual** as the means to achieve goal BeReported [AmbInfo]) or automatically (task **Automatic** as the means to achieve goal BeReported [AmbInfo]). To report manually, an Ambulance Crew need to **Connect to Radio Operator**, **Report Location**, and **Report Status**. Similar information can be obtained for the Automatic report process.

Since our purpose in this section is to demonstrate the use of various view types, completeness of a model is not critical. Thus, we choose as our starting point this Single-Actor-Focus view, which includes just enough elements to show our approach. This SR view will be used as the original view from which the sub-views derive throughout this chapter.

Justifications

The Single-Actor-Focus SR view does not introduce much overhead to the analysis process. Normally, the analysis of actor's internal structure is taken in an actor-by-actor manner—especially when the internal structure of an actor appears complex (multiple top-level intentions, deep decomposition tree structures). Node analysis questions and others regarding a given actor can be answered by simply exploring the actor's internal structure. Moreover, all external relationships from the selected actor towards other actors are kept in this view, so whenever information from other actors is required, users can trace into other Single-Actor-Focus views without confusion.

Another benefit is, given current tool support, each diagram has to be drawn separately and there is no support for underlying structures. If one extended actor appears in different SR diagrams, with the model not yet stable, then significant overhead is incurred because multiple diagrams must be fixed for any tiny change to that actor's internal rationale. By decomposing the Basic SR view into a set of Single-Actor-Focus views, changes internal to an actor can be localized to a sub-view and with a single entry. Even any external dependency changes can be limited to *n* diagrams, where *n* is the number of actors involved in this change.

Admittedly, while each SR diagram is simplified – focusing on a single actor and its dependencies, the number of SR diagrams increased from 1 to *m*, where *m* is the number of actors in the system. For this reason, we suggest users maintain a view reference structure (using view map) for various decomposed SR views.

Each evaluation results (EVLR) view corresponds to an SR view, so decomposing the Basic SR view may also affect the presentation of the EVLR views. Figure 7.2-2 and Figure 7.2-3 show two Single-Actor-Focus EVLR views from the LAS case study. The weakly denied label of softgoal dependum **BeArrived [within 11 mins]** is propagated from the Single-Actor-Focus view for agent **Ambulance Crew** (Figure 7.2-2) to the view for agent **LAS Management**. The imported label is highlighted in Figure 7.2-3 using a dashed rectangle.

Since our focus of this thesis is to provide a means of representing an i* model, we do not define here new label propagation algorithms. The EVLR views vary according to different algorithms, so we demonstrate in this section one way in which some decomposed EVLR views can be used. We do not discuss in the thesis the generic scale-down rules for EVLR views.

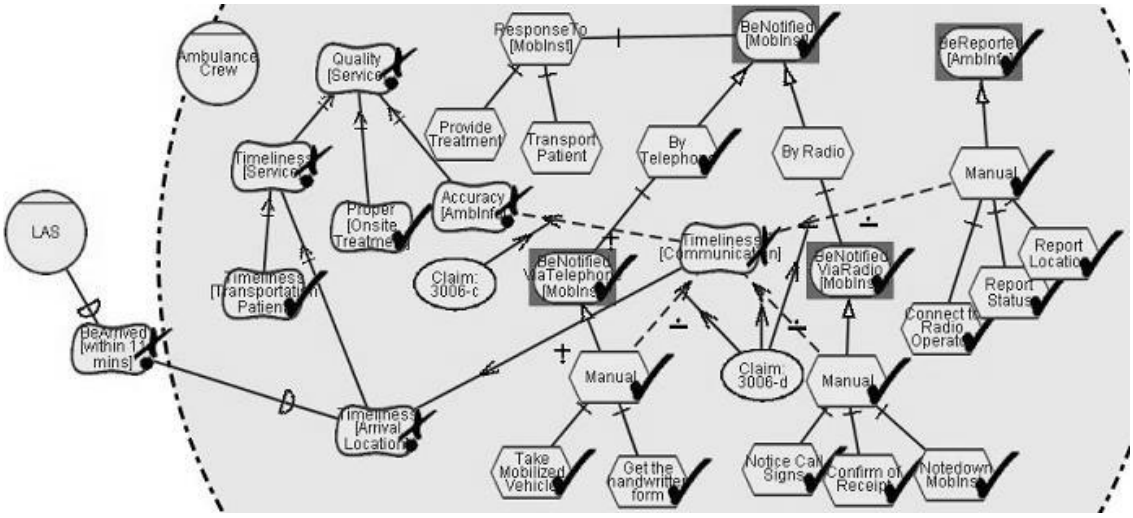


Figure 7.2-2 Sample Single-Actor-Focus EVLR view showing evaluation results for agent Ambulance Crew from the LAS case study

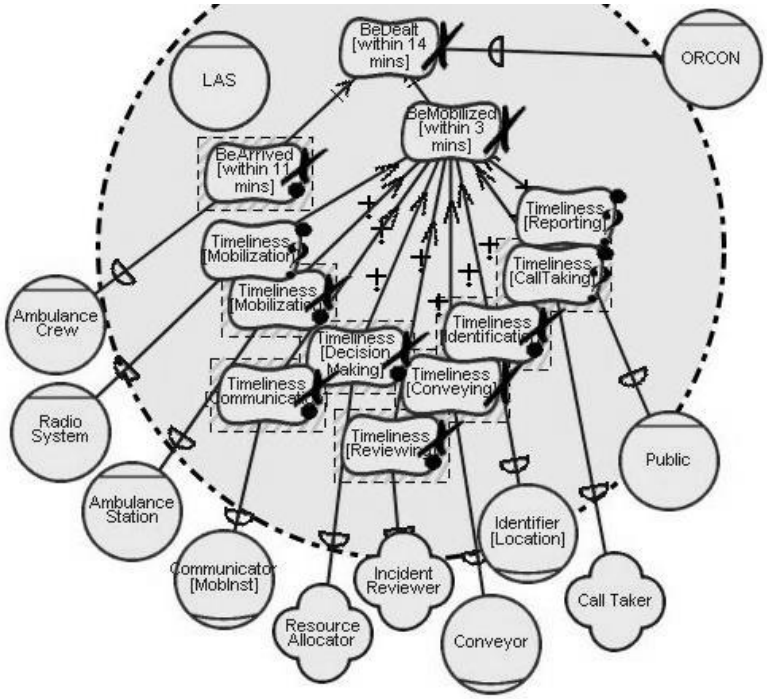


Figure 7.2-3 Sample Single-Actor-Focus EVLR view showing evaluation results for agent LAS (Management) from the LAS case study

Selection Rule

Formally, we obtain a Single-Actor-Focus view for a given actor from any multi-actor SR view by applying the query **singleActorFocusSRRule**. We pass the selected actor (*a*) as an input argument to the query. This one is similar to the singleActorFocusSDRule, except if includes one extra query—find_internal_elements. We give here the definition of the rule and extra query, yet we omit the definition for the sub queries already defined in previously (Section 6.2.2).

singleActorFocusSRRule(v:SRViewClass, a:ActorElementClass) ::=

§o:ObjectClass· o∈ v ∧ o ∈ {singleActorFocusSDRule(v, a),

find_internal_elements(a) } %Query2

7.2.2 Single-Actor-Internal or External View

Informal Description

A Single-Actor-Internal view presents the specified single actor and its internal goal structure, formed by internal elements and internal links.

A Single-Actor-External view presents the specified single actor, its external relationships, actors served as depender or dependee to it, and actors whose external relationships affect or are affected by the selected actor.

Example

Figure 7.2-4 shows the Single-Actor-Internal view of agent Ambulance Crew derived from the original view, and Figure 7.2-5 shows the corresponding External view. In the former, internal structures of agent Ambulance Crew remain the same as its parent view (the original view); in the latter, only internal elements that have an external relationship are kept. For example, goal **BeReported [AmbInfo]** is shown in the external view, while the two means to achieve it are omitted.

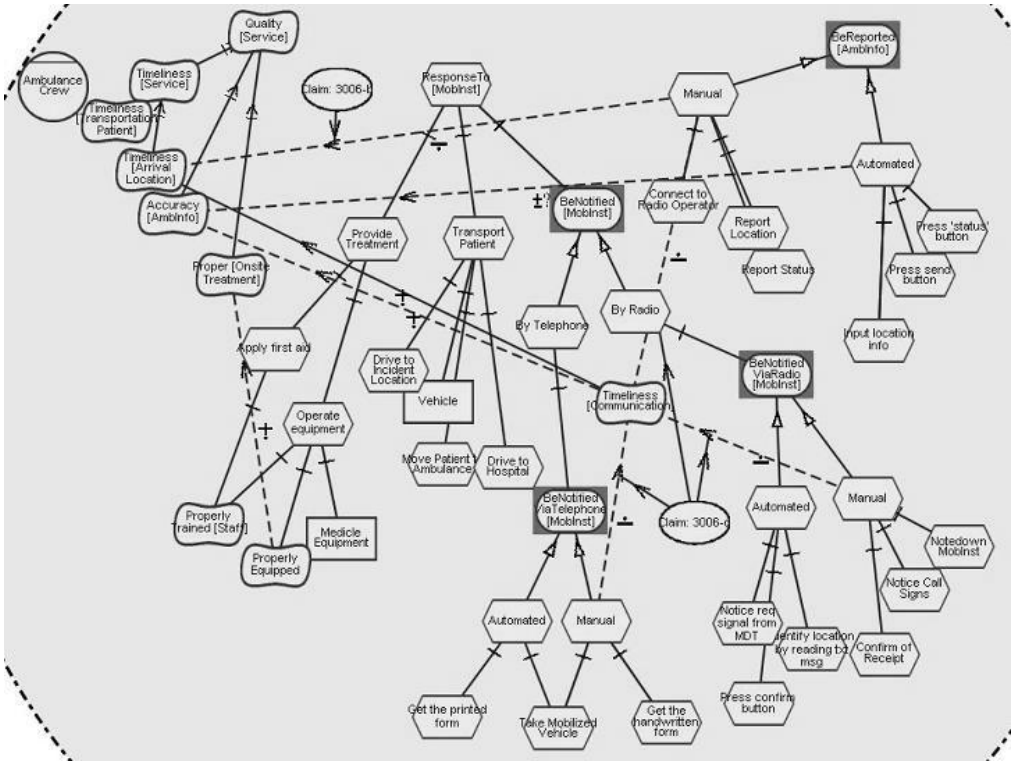


Figure 7.2-4 Single-Actor-Internal view derived from the original view

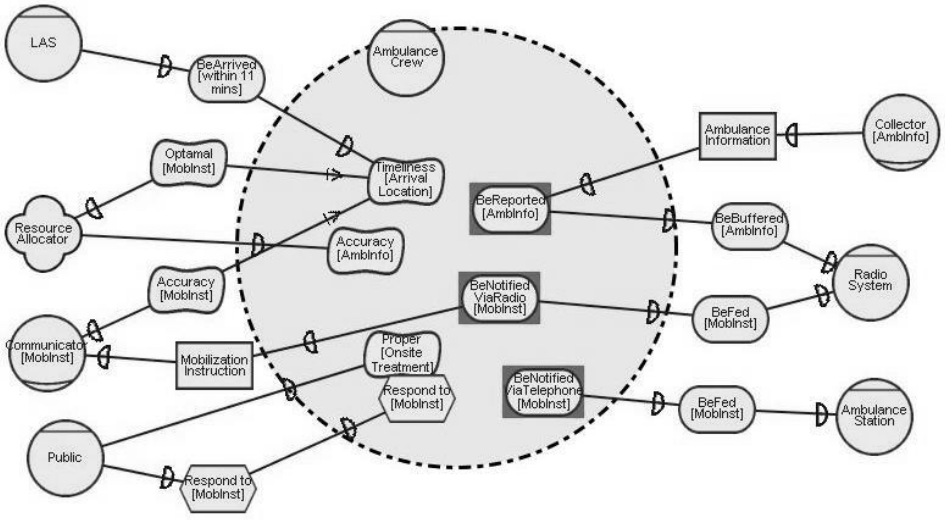


Figure 7.2-5 Single-Actor-External view derived from the original view

Justifications

In some cases, even a Single-Actor-Focus view could appear complex (e.g., our original view). Therefore, we need to scale it down further so as to make

each sub-view, when visualized, more comprehensible. The first step we take is to separate internal rationales from the external ones.

This separation appears natural for i* models.

Answering questions that relate to the internal process elements and routines does not require external relationship information. From the internal view, we can still find out what top-level intentions the actor has, what the alternatives that will achieve those intentions are, and what the routines of each alternative are. For example, using elements shown in Figure 7.2-4, we can also find out the two alternative routines available to achieve goal **BeReported [AmbInfo]**. In this light, external relationships of an actor are not relevant.

The elements included in the external view appear sufficient for linking internal elements from an actor to the ones that reside in another. When tracing to other Single-Actor-Focus views, the user needs to know only which internal element is connected with which dependum, and which dependum is connected to which actors other than the selected one. For example, from Figure 7.2-5 we know that role **Collector [AmbInfo]** depends on goal **BeReported [AmbInfo]** to furnish resource **Ambulance Information**. If we want to identify which internal element of role Collector [AmbInfo] requires that piece of information, we shift to the Single-Actor-Focus view of the role, locate the same dependum, and follow the incoming dependency link “to” the dependum to locate the internal depender. For this purpose, the internal goal structure of an actor does not appear critical.

Selection Rule

Formally, we obtain a Single-Actor-Internal view from a Single-Actor-Focus view (for actor *a*) by applying the query **singleActorInternalRule**, and a Single-Actor-External view from a Single-Actor-Focus view by applying the query **singleActorExternalRule**.

singleActorInternalRule(v_a:SingleActorFocusSRViewClass)::=

§o:ObjectClass· o∈ v_a ∧o∈ {a, find_internal_elements(a)}

singleActorExternalRule(v_a:SingleActorFocusSRViewClass)::=

§o:ObjectClass· o∈ v ∧

o∈ { singleActorFocusSDRule(v, a), find_internal_connectors(a)}

Query44

find_internal_connectors(a:ActorElementClass)::=

§e:IntentionalElementClass· e.parent=a ∧

(∃l1:DependencyLinkClass· l1.from=e ∨ l1.to=e) ∨

(∃l2:IntentionalLinkClass· l2.from=e ∧ l2∈ find_externallinks_from_actor(a))

7.2.3 Internal-Non-functional and Functional View

Informal Description

An Internal-Non-functional view presents the selected actor, its top-level softgoals, and all the descendents (reasoning structure) of these softgoals. An Internal-Functional view presents the selected actor, its top-level non softgoals, and all the descendents towards these (reasoning structure) non softgoals.

For clarity, we restate here the informal definition of *descendent:* A descendent of a given element is a sub-element either that has a direct intentional link to the given element or whose direct ancestor is a descendent of the given element. The formal definition of ancestor and descendent can be found in Section 4.4.6.

Example

Figure 7.2-6 shows an example of the Internal-Non-functional view derived from the Single-Actor-Internal view for agent Ambulance Crew (Figure 7.2-4), and Figure 7.2-7 shows the corresponding Internal-Functional view. In Figure 7.2-6, top-level softgoal **Quality [Service]** and all its descendents are shown in a

separate view from the other two top-level intentions (task **Response To [MobInst]** and goal **BeReported [AmbInfo]**) that are shown in Figure 7.2-7.

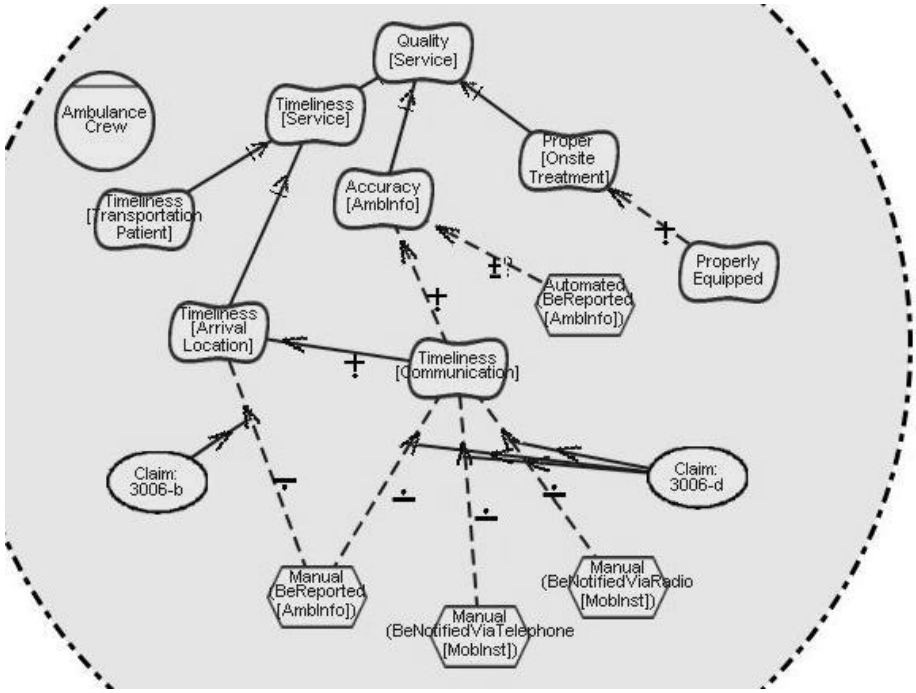


Figure 7.2-6 Single-Actor-Internal-Non-functional view derived from the Single-Actor-Internal view for agent Ambulance Crew

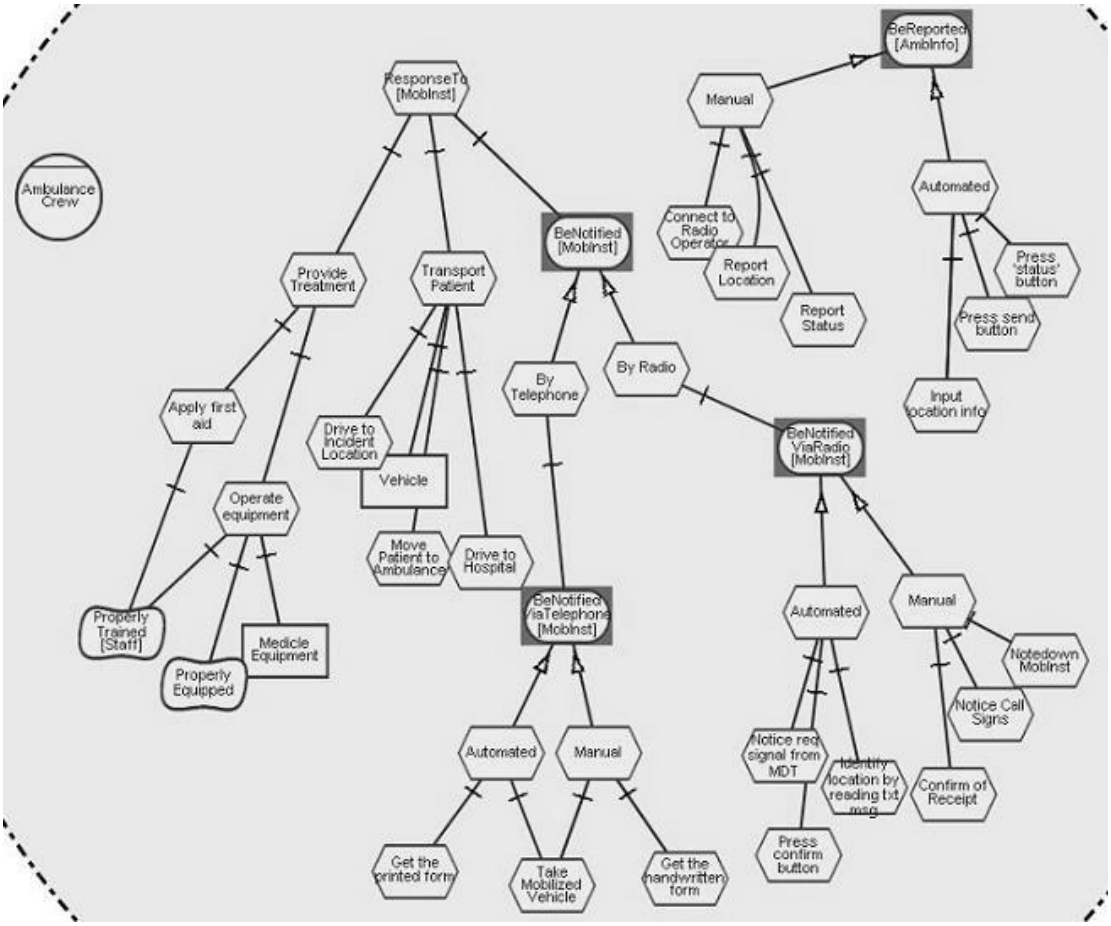


Figure 7.2-7 Single-Actor-Internal-Functional view derived from the Single-Actor-Internal view for agent Ambulance Crew

Justifications

In some cases, a Single-Actor-Internal view still appears complex (e.g., the Single-Actor-Internal view derived from our original view). Therefore, we need to scale it down further so as to make each sub-view, when visualized, more comprehensible; the approach we are taking now is to separate top-level non-functional intentional elements from the functional ones.

This separation appears natural when the internal rationale of a modeled actor gets extremely complex, featuring numerous internal elements and intertwined internal intentional links. When internal rationale becomes difficult, typically functional and non-functional parts are considered separately, at different times.

Functions of a system are normally considered first in order to verify the workability of certain system configurations. During this process, softgoals that do not serve as descendents of some given functionality appear irrelevant.

After these functions become relatively stable, and especially when alternative routines are available, we record their side-effects using a contribution (and correlation) network of softgoals. If necessary, an evaluation process can be employed to decide the level of satisficeability of the top-level softgoals when assuming each alternative. Resulting labels from different alternatives of the top-level softgoals can be compared. During this process, those functional elements that not contribute to any softgoals appear irrelevant.

However, redundancies are expected in the non-functional and functional views since most process elements cast certain effects to some softgoals, and since these elements may also be decomposed into softgoals. Thus, given the current level of tool support, we do not suggest excessive use of this separation—since any change to those overlapping elements requires synchronization to several other views.

Selection Rule

Formally, we obtain an Internal-Non-functional view from a Single-Actor-Internal view (for actor *a*) by applying the query **internalNonfunctionalRule**, and an Internal-Functional view from a Single-Actor-Internal view by applying the query **internalFunctionalRule**.

internalNonfunctionalRule(v_a:InternalViewClass)::=

§o:ObjectClass· o∈ v_a ∧o∈ {find_root_softgoals(a),

{find_all_descendants(sg) | sg ∈ find_root_softgoals(a) }}

Query45

find_root_elements(a:ActorElementClass)::=

§e:IntentionalElementClass· e.parent=a ∧ ¬(∃l:IntentionalLinkClass·l.from=e)

Query46

find_root_softgoals(a:ActorElementClass)::=

§sg:SoftgoalElementClass· sg∈ find_root_elements(a)

internalFunctionalRule(v_a:InternalViewClass)::=

§o:ObjectClass· o∈ v_a ∧o∈ {find_root_functionals(a),

{find_all_descendants(g) | g ∈ find_root_functionals(a) }}

Query47

find_root_functionals(a:ActorElementClass)::=

§fe:IntentionalElementClass·

(fe∈ find_root_elements(a)) ∧ ¬(fe in SoftgoalElementClass)

7.2.4 Single-Softgoal View

Informal Description

The Single-Softgoal view presents a selected actor, one of its top-level softgoal, and all the descendents of the softgoal.

Example

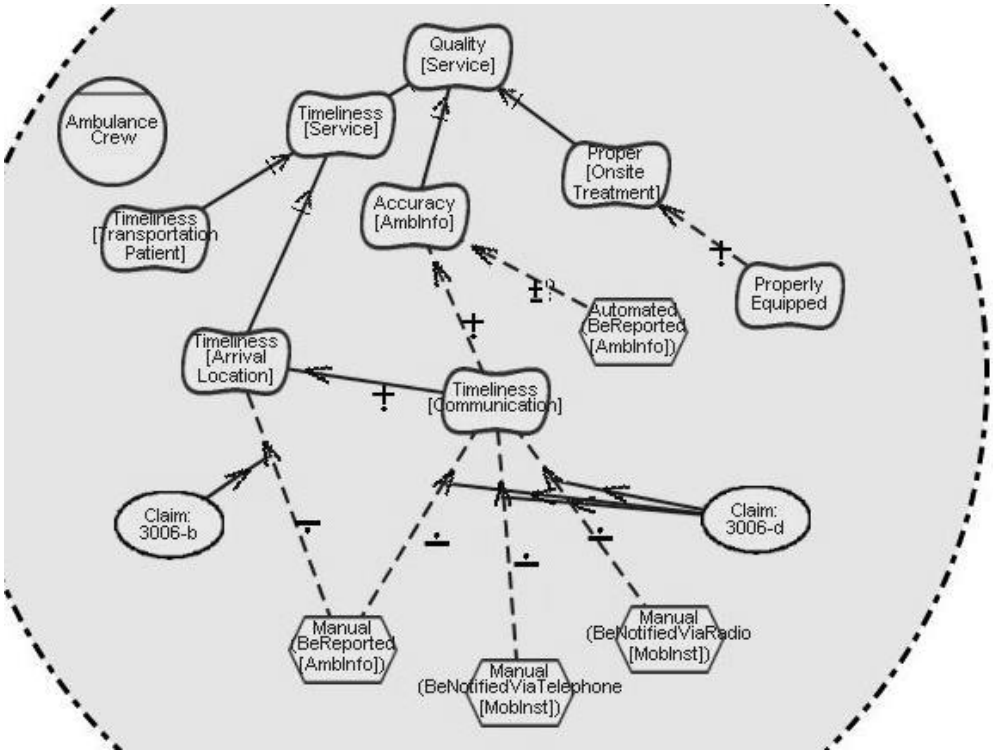


Figure 7.2-8 Single-Softgoal view derived from the Single-Actor-Internal Nonfunctional view presented in the previous section

The view in Figure 7.2-8 is actually the same as its parent Internal-Non-functional view shown in Figure 7.2-6. This is because our sample contains only one top-level softgoal **Quality [Service]**, and no further view decomposition is necessary. This fact reminds us that rules can be selectively applied to a given application, and that users should only apply those rules they consider necessary.

Justifications

In the non-functional view of a single actor, relationships towards different top-level softgoals can be intertwined, a fact that makes it difficult to study the process elements and the rationales behind these elements for a given softgoal.

Using a Single-Softgoal view, leaf-process elements that will affect the satisficeability of the given softgoal are distinguished. The rationale for selecting those leaf elements also becomes obvious. Thus, it appears natural to decompose

a Non-functional view into Single-Softgoal views when the former becomes barely comprehensible.

However, for reasons similar to those stated in Section 7.2.3, we do not suggest excessive use of this view.

Selection Rule

Formally, we obtain a Single-Softgoal view from an Internal-Non-functional view (for actor *a*) by applying the query **nonfunctionalSingleSoftgoalRule**. We pass the selected softgoal (*sg*) as an input argument to the query.

nonfunctionalSingleSoftgoalRule(v_a:NonFuntionalViewClas, sg:SoftGoalElementClass)::=

§o:ObjectClass· o∈ v_a ∧o∈ find_all_descendants(sg) %Query12

7.2.5 Single-Affected-Dependum or Actor View

Informal Description

A Single-Affected-Dependum view presents the selected actor and a selected dependum that the former affects. In this context, by *affect* we mean that elements from the actor exert contributions to the outgoing dependency link of the dependum. In this light, this view also includes the internal elements that exert the effects, the dependum, and the dependee of the dependum.

A Single-Affected-Actor view presents the selected actor and a selected other actor that the former affects. In this context, by *affect* we mean that elements from the actor exert contributions to the external links exerted from the other actor. In this light, this view also includes the internal elements that exert the effects, and the external links that these elements affect.

Example

Figure 7.2-9 shows a sample of a Single-Affected-Dependum view of role **Ambulance Crew as Impactor** from the LAS case study. Note that agent

Ambulance Crew connecting to the dependum should be omitted from the view. Due to limitations in tool support, we have to retain it to ensure the dependency link (from BeArrived [within 11 mins to Ambulance Crew] does not disappear. The two internal elements contribute negatively to the softgoal dependum **BeArrived [within 11 mins]** from agent LAS Management to agent Ambulance Crew.

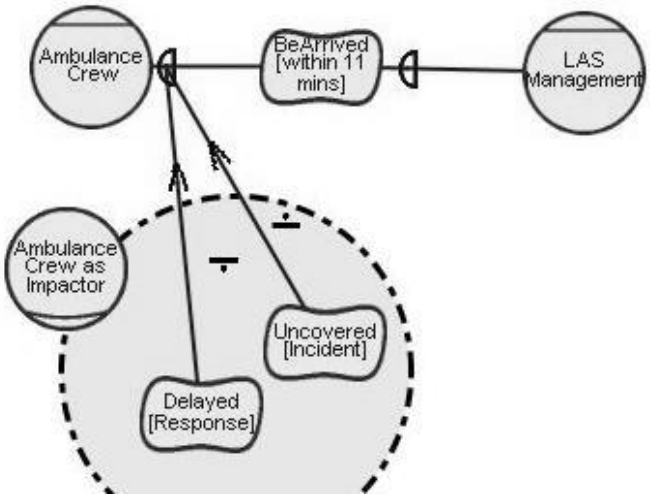


Figure 7.2-9 Sample Single-Affected-Dependum view showing one affected dependum BeArrived [within 11 mins] from the LAS case study

Figure 7.2-10 shows a sample of a Single-Affected-Actor view for agent **TCG** to affect role **Hacker/Malicious User**. This sample is taken from the Trusted Computing Group (TCG) case study (Horkoff 2004)—since we do not have such patterns in the LAS case study. From the sample, we see that internal elements of agent TCG (e.g., **Isolate Applications**) cast negative effects (e.g., a *Hurt* contribution) to the two external links (e.g., the *Break* contribution to softgoal dependum **Protect [Stored Data]**) exerted from role Hacker/Malicious User.

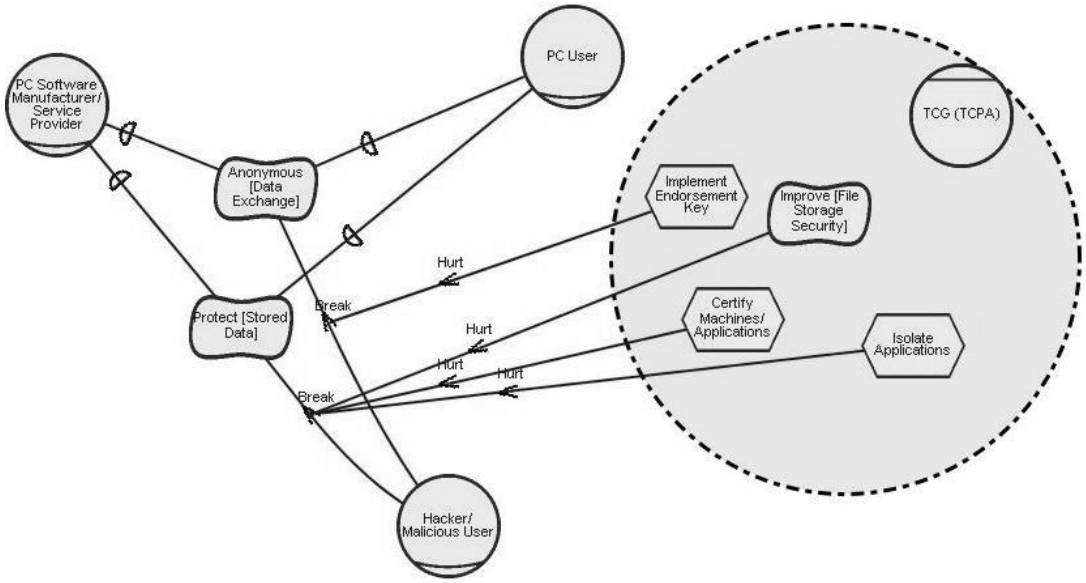


Figure 7.2-10 Sample Single-Affected-Actor view showing the effects to Hacker from agent TCG from the TCG case study

Justifications

In a Single-Actor-External view, multiple internal elements may contribute different effects to the same dependum (e.g., Protect [Stored Data]) or to the external links exerted from the same actor (e.g., Hacker/Malicious User). Sometimes these effects get complex, and thus we further decompose the External view to a set of Single-Affected-Dependum and Single-Affected-Actors views.

Under certain circumstances, users may want to study the external effects of a certain dependum on a certain actor individually. In this light, using a Single-Affected-Dependum or a Single-Affected-Actor view provides just sufficient information for users to understand which internal elements of an actor may contribute what effects to a selected subject. These types of views are normally quite simple, and users of them are not distracted by unnecessary information towards other external elements.

However, there may exist too many external dependums or external links that one actor can affect. Applying this type of view excessively could result in a

huge amount of fragmented views. Thus, we suggest using this view only when it is absolutely necessary –when the circumstances described in the above paragraph become totally fulfilled. Or a user may combine a few of these types of views so long as the complexity of the resulting visualization is acceptable.

Selection Rule

Formally, we obtain a Single-Affected-Dependum view from a Single-Actor-External view (for actor *a*) by applying the query **singleAffectedDependumRule.** We pass the selected dependum (*dl*) which gets affected as input arguments to the query.

singleAffectedDependumRule(v_a:ExternalViewClass, dl:DependencyLinkClass) ::=

§o:ObjectClass· o∈ v_a ∧o∈ {find_contribution_to_dependum(a, dl),
find_contributer_to_dependum(a, dl) }

Query48

find_contribution_to_dependum(a:ActorElementClass,dl:DependencyLinkClass)::=

§l:IntentionalLinkClass·(l.from.parent=a) ∧ (l.to=dl)

Query49

find_contributor_to_dependum(a:ActorElementClass,dl:DependencyLinkClass)::=

§e: ElementClass· ∃l:IntentionalLinkClass·
(l.from=e ∧ l∈ find_contribution_to_dependum(a,dl))

Formally, we obtain a Single-Affected-Actor view from a Single-Actor-External view (for actor *a*) by applying the query **singleAffectedActorRule**. We pass the selected actor (*a1*) who gets affected as input arguments to the query.

singleAffectedActorRule(v_a:ExternalViewClass, a1:ActorElementClass)::=

§o:ObjectClass· o∈ v_a ∧o∈ {find_contribution_to_actor(a,a1),
find_contributor_to_actor(a,a1)}

Query50

find_contribution_to_actor (a,a1:ActorElementClass)::=

§l:IntentionalLinkClass· (l.from.parent=a) ∧

(∃l1:IntentionalLinkClass· (l1.from.parent=a1) ∧ (l.to =l1))

Query51

find_contributor_to_actor (a,a1:ActorElementClass)::=

§e:ElementClass· ∃l:ContributionLinkClass·

(l.from=e ∧ l∈ find_contribution_to_actor(a,al))

7.3 Summary

In this chapter, we presented a hierarchy of partial SR views, and each of the views was explored in detail. These eight SR views were studied in this section: the Single-Actor-Focus SR view, the Single-Actor-Internal view, the Single-Actor-External view, the Internal-Non-functional view, the Internal-Functional View, the Single-Softgoal view, the Single-Affected-Dependum view, and the Single-Affected-Actor view.

The hierarchy of partial SR views was illustrated using a generalized view map. The Single-Actor-Focus SR view is placed as the top-level node in this hierarchy. We also presented a way of using the sub-SR views to work with the evaluation process and showed how to organize the set of resulting EVLR views.

The SR views are presented from both informal and formal aspects. An informal description gives the reader a basic idea of what kinds of elements are qualified for a specific partial view. The formal definition of the selection rule attached to each view class makes it possible to automate these views in an i* modeling tool. Some justifications for the each view are included.

Examples from the LAS case study was used to illustrate the idea of an original Single-Actor-Focus SR view and various types of sub-SR views it can derive, making it possible for the reader to compare the differences between the view types. One special example was cited from the TCG case study (Horkoff 2004) to demonstrate the Single-Affected-Actor view, since we did not have this modeling pattern in the LAS study.

8 Application—Re-presenting the Trusted Computing Group Case Study

The Trusted Computing Group (TCG) case study (Horkoff 2004) was first generated in summer 2003. The case study explored opposing viewpoints from two groups—proponents of TCG and opponents of TCG—and, accordingly, constructed two sets of diagrams. Each diagram is labeled as a "model" in the TCG case study. There are approximately 120 such models in the document, and more than half of them contain over 40 i* notations (elements and links) each. One extreme case contains 44 elements and around 100 links. With the volume of information trying to express in one diagram, text in each element turns out hardly readable, and links are so intertwined that it is hard for a reader to identify connections between the elements. The TCG case study documented in (Horkoff 2004), which we cite as TCGCS throughout this chapter, raised considerable scalability issues in the i* framework.

The complexity and size of TCGCS renders it a good example in validating our newly proposed view extension. Thus, we used the resulting diagrams from TCGCS to demonstrate that our proposed approach can simplify the representation of the huge models, yet serve the same purpose as those diagrams shown in the original document. In this chapter, we highlight some interesting parts from TCGCS that are considered sufficient to illustrate the use of our view extension. The rest of the original work can be organized following a similar manner.

Our rework and TCGCS differ in the use of terminologies and the organization of the diagrams.

Terminologies used in our proposed view extension differ from what was used in TCGCS. The two sets of diagrams produced in TCGCS are considered as two i* *baseline models*, representing the situations of TCG from two contrasting viewpoints. We name the one representing the viewpoint from TCG proponents

as “TCG.Pro,” and the one for the opponents as “TCG.Anti.” The term “model” (diagram) from TCGCS corresponds to the concept of *view* in our extension. Each view is a projection over the baseline model according to some predefined *selection rules* in the view extension. In our rework, we name each derived view following a consistent naming convention, prefixing it with the name of its corresponding baseline model.

The manner we followed in presenting the views also differs from TCGCS. Views (models) in TCGCS were created and documented as the need arose, without a predefined systematic method. This practice appears natural during the model acquisition process, yet model users may find it difficult to locate specific information from the 120 models. We partition the views obtained by using the view extension into four basic types (AC, SD, SR, and EVLR), and show the views in a sequence according to their types.

Our rework of TCGCS resulted in a total of 37 diagrams, showing the baseline model, 15 AC views, 8 SD views, and 13 SR views. Among these views, only 2 remain exactly the same as what was demonstrated in TCGCS, 17 of which are newly added ones, the other 18 being modified. In addition, four view maps (VM) for showing the relationship for basic views, AC views, SD views, and SR views, respectively, were also supplied to make attainable the relationship among views from the same group.

EVLR views are not presented in this chapter since we found it impractical to fit the evaluation diagrams from TCGCS into our EVLR views. A major reason for the difficulty is that the label propagation algorithm employed in TCGCS allows a label be propagated from a dependum to both its depender and dependee, while we feel it only natural to propagate a label to a depender. Any dependee, in i* semantic, should have the autonomy to decide its own label regardless of what was assigned to its dependum. Since this issue deserves further research, and since we cannot supply meaningful results unless this issue is properly resolved, we have decided to omit the EVLR views of TCG in this thesis. Nevertheless, as argued in Section 7.1, omitting the EVLR feature does not affect our view

extension, because the EVLR views are considered as SR views with only the evaluation will be different.

Section 8.1 presents an overview of the relationships between the baseline models and original views that will be used in the subsequent sections; Section 8.2 to 8.4 present the partial AC, SD, and SR views we obtained from the case study, respectively; and Section 8.5 summarizes results and contributions resulting from this reworking of TCGCS.

8.1 Overview

Figure 8.1-1 shows the VM of the basic views for our TCGCS rework. Each view is represented using a rectangle; the view name and view type are separated by a semi-colon; and the corresponding visualized diagram is included in the bracket. The views shown in a dashed rectangle do not appear in this section for we selectively apply our approach to interesting parts. Yet they do—or should—exist in TCGCS in order to maintain the completeness of TCGCS. The view shown in a dotted rectangle implies it does not necessarily exist even in the original TCGCS, and can be derived from other views. A detailed definition of the graphical notations for a VM can be found in Section 4.2.

From Figure 8.1-1(a), we see that the proponents baseline model **TCG.Pro** is decomposed into four basic views. The Basic AC view (**TCG.Pro.AC**) and the Basic SD basic view (**TCG.Pro.SD**) are visualized in both this chapter and TCGCS. We use these two basic views as our original view to derive a set of partial AC and SD views in the subsequent sections, respectively.

Figure 8.1-1(b) shows that **TCG.Anti**, the baseline model from the TCG opponents' viewpoint, was not presented explicitly in TCGCS. This may because TCG.Anti differs only in the rationales surrounding actor TCG from TCG.Pro. The SR view for actor TCG from TCG.Anti (**TCG.Anti.SR.SA-TCG**) is the extreme case which containing 44 elements and over 100 links in one diagram, so we choose it as our original view to derive a set of partial SR views.

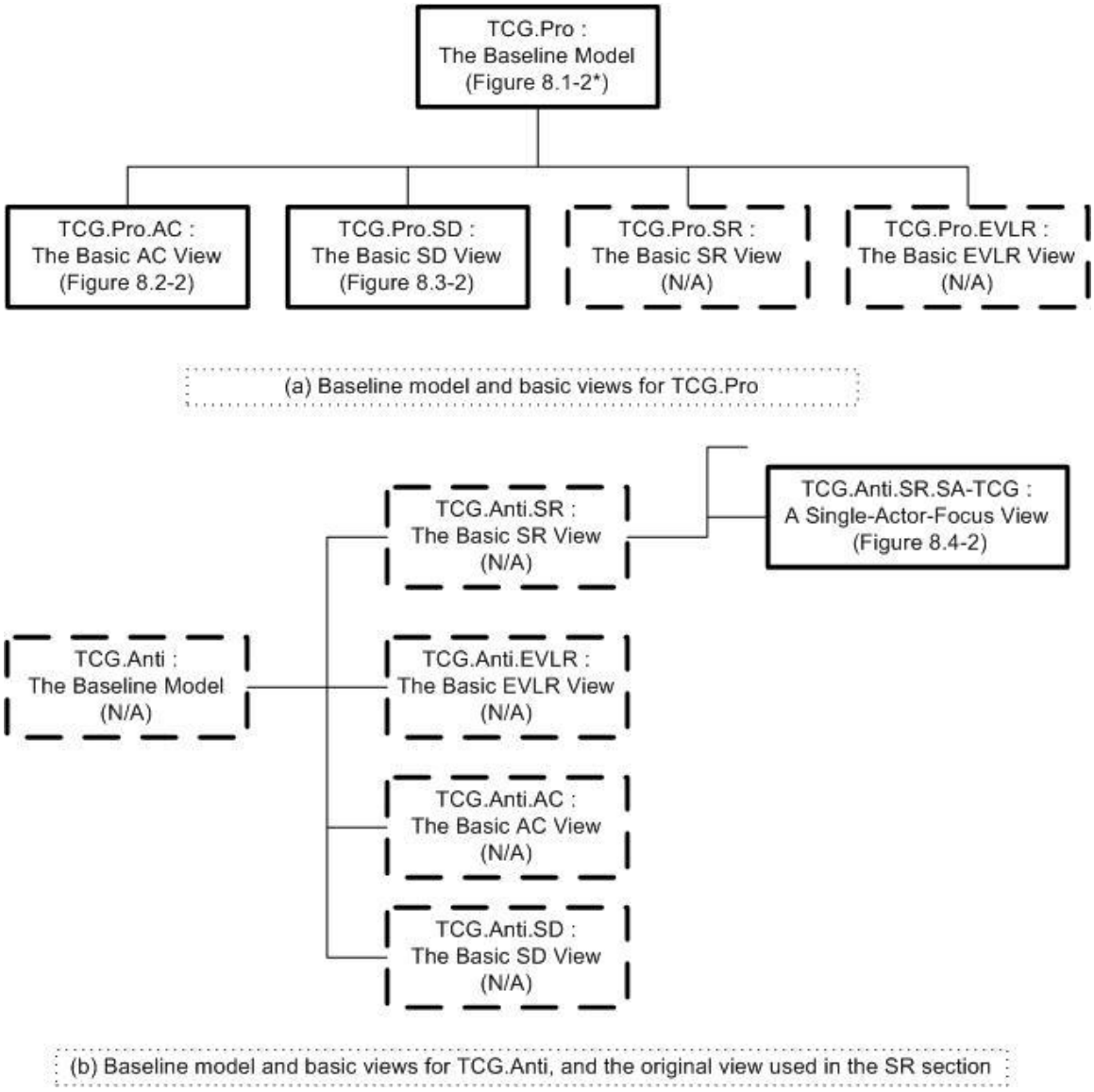


Figure 8.1-1 View map showing the relationships among the basic views from TCGCS

Figure 8.1-2 shows the revised Baseline Model that we constructed according to the diagrams presented for the TCG proponents' viewpoint in TCGCS. This model bears the name **TCG.Pro**. The original views used in the AC and SD sections are derived from this baseline model. However, this baseline model is a partial one, not showing the internal structures within each actor.

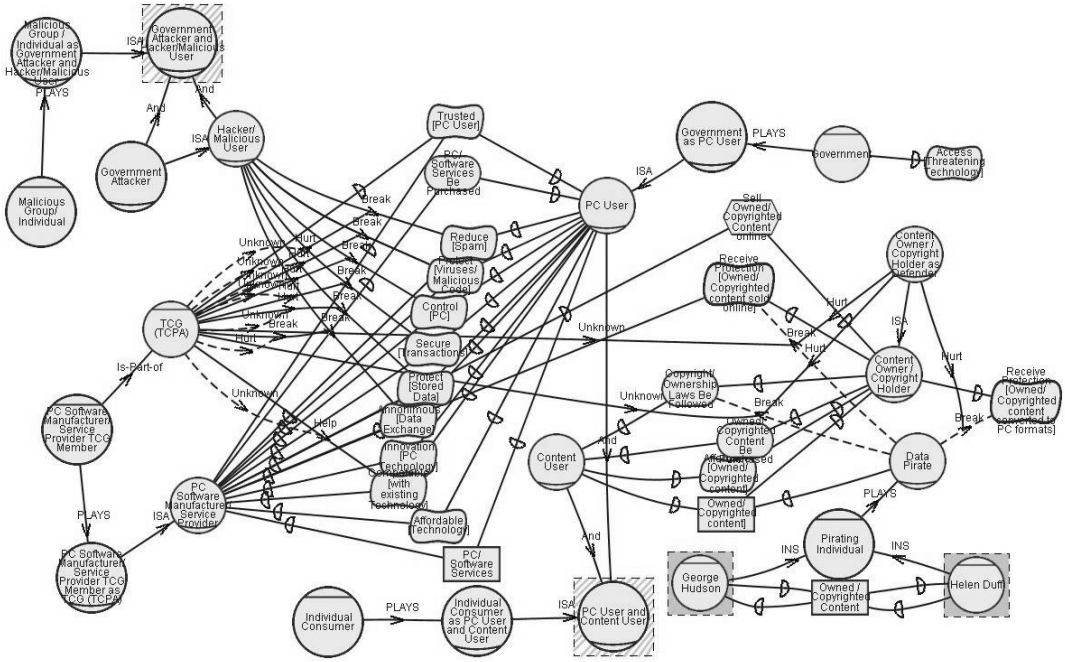


Figure 8.1-2 Revised Baseline Model representing the viewpoints from the proponents of TCG

8.2 Actor Class Views

In this section, the selected original view (**TCG.Pro.AC**) is scaled down into a set of related partial AC views; their relationship is shown in Figure 8.2-1. TCG.Pro.AC (or **AC**) is decomposed into two Single-Network views: one for producer (**AC.SN-Producer**) and one for consumer (**AC.SN-Consumer**). AC.SN-Consumer is further decomposed in two dimensions. One dimension is decomposed according to element types, resulting in a Plain-Actors-Only view (**AC.SN-Consumer.PlainActors**), an Abstract-Actors-Only view (**AC.SN-Consumer.AbstractActors**) and an Agents-Only view (**AC.SN-Consumer.Agents**). The other dimension is decomposed according to plain actors, resulting in four Single-Plain-Actor sub-views (e.g., **AC.SN-Consumer.SPA-PCUser**). A set of Direct-Replaceable views (e.g., **AC.SN-Consumer.DR-IndividualConsumer**) are also derived from AC.SN-Consumer.

We have shown the decomposition of AC.SN-Consumer in Figure 8.2-1, and there are others such as sub views of AC.SN-Producer which also belongs to this category. Their relationship would follow the same pattern as shown for AC.SN-Consumer in Figure 8.2-1, so we do not repeat them in this section.

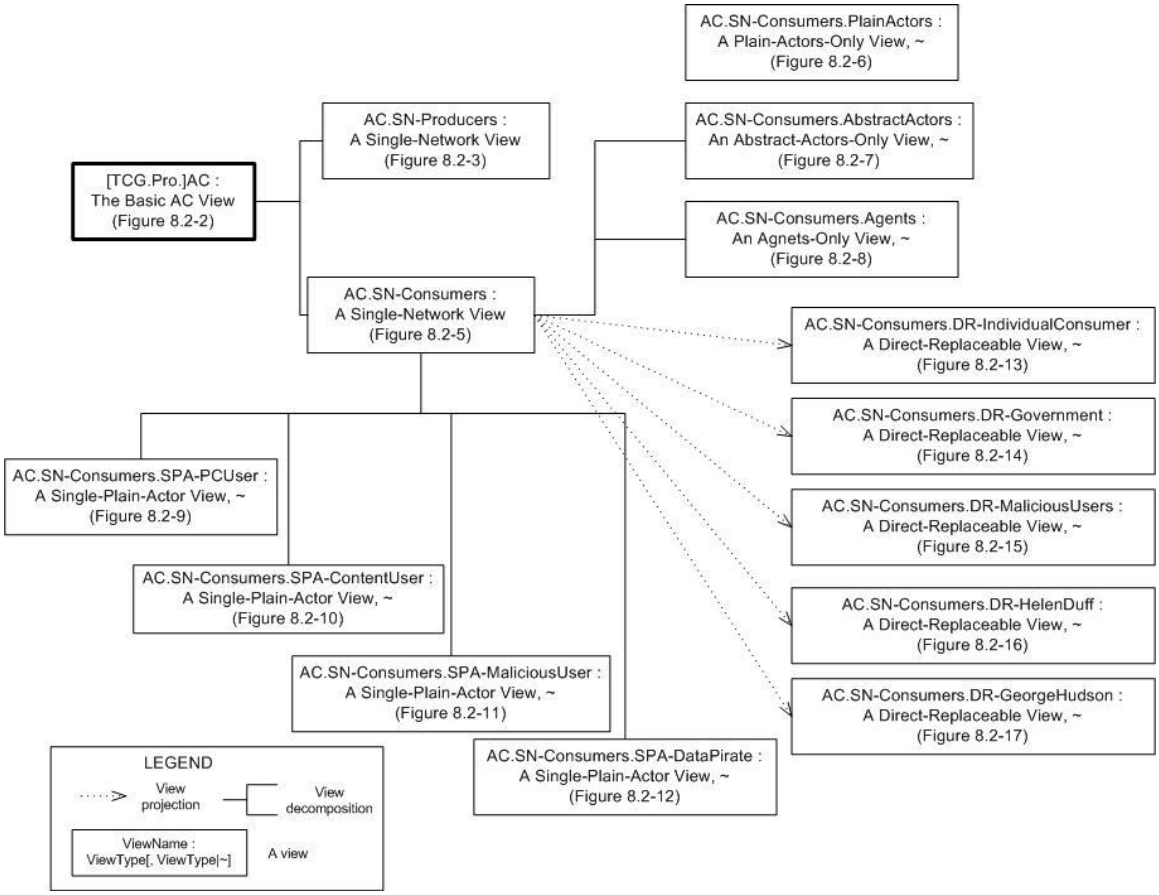


Figure 8.2-1 View map for some partial AC views

8.2.1 The Basic AC view

Figure 8.2-2 shows the Basic AC view from the TCG proponents' viewpoint, and we name it as **TCG.Pro.AC**. This is an example that a basic view, direct projection over a baseline model, still complex. In the diagram showing below, approximately 47 actors and 50 links are presented, making it almost unreadable.

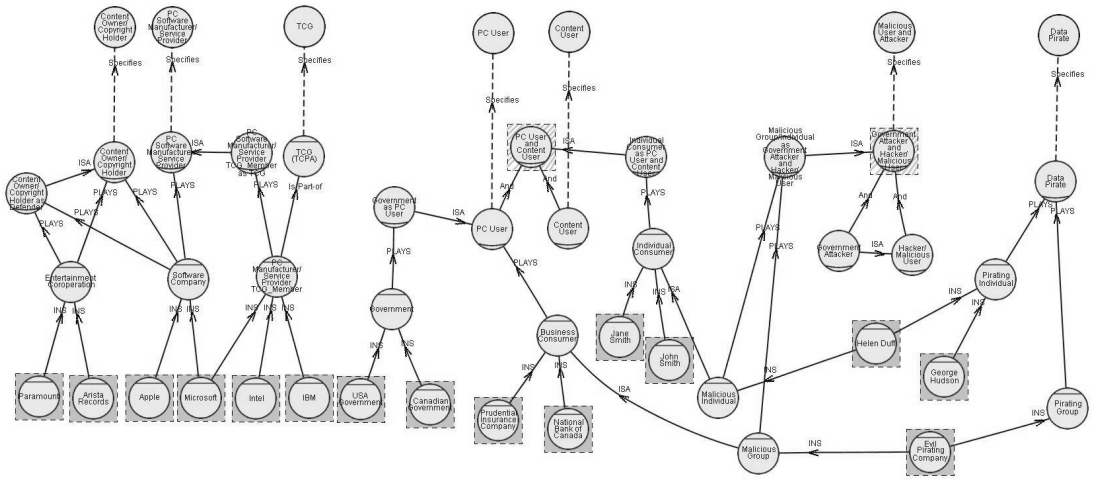


Figure 8.2-2 Basic Actor Class View

8.2.2 Single-Network views

Applying the single network rule (singleNetworkRule) to TCG.Pro.AC, we obtained two Single-Network views: **TCG.Pro.AC.SN-Producer** and **TCG.Pro.AC.SN-Consumer**.

TCG.Pro.AC.SN-Producer (Figure 8.2-3) exhibits shows the associations between three plain actors—**Content Owner/Copyright Holder**, **PC Software Manufacturer/Service Provider**, and **TCG**—and their specified forms. There is a corresponding diagram (model 4.3) to this view in TCGCS.

Since the notion of “specifies”, “complete composition”, and “agent instance” are newly introduced in our extension, our AC views embraces extra information (e.g., “agent TCG specifies plain actor TCG”) and distinguished agent instances (e.g., IBM) than their corresponding original models in TCGCS. All AC views presented in this section resemble these features, so we will not repeat this point again.

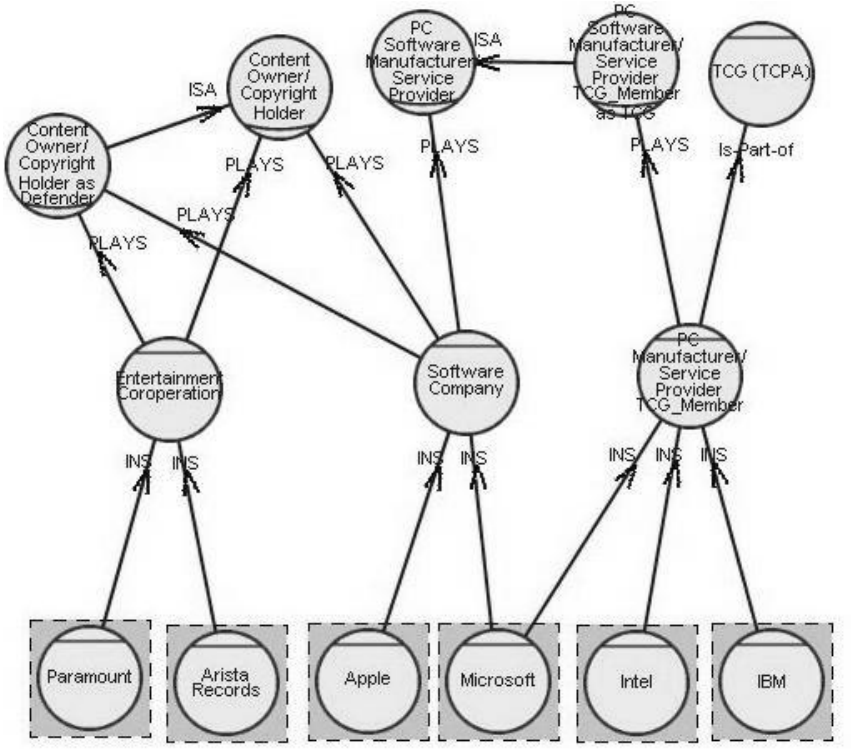


Figure 8.2-3 Single-Network view for producers from the TCG proponents' viewpoint

TCG.Pro.AC.SN-Producer (Figure 8.2-4) is constructed based on information collected from TCGCS (including model 4.6, model 2.6.3, model 2.6.4, and so on). Plain actors (e.g., actor **PC User**), specified actors (e.g., role **PC User and Content User**), and agent instances (e.g., **Helen Huff**) were also added. These new actor elements are added to fill the logic gaps between actors in TCGCS so that users can apply the *external relationship inheritance rule* to calculate indirect external dependencies. In this light, actor associations expressed in AC views can support automated substitution of actors in SD views (see Section 5.2.3 for detail).

However, with these enriched information to TCG.Pro.AC.SN-Producer, the view appears more complex than the original one; thus, further decomposition is required to improve its comprehensibility.

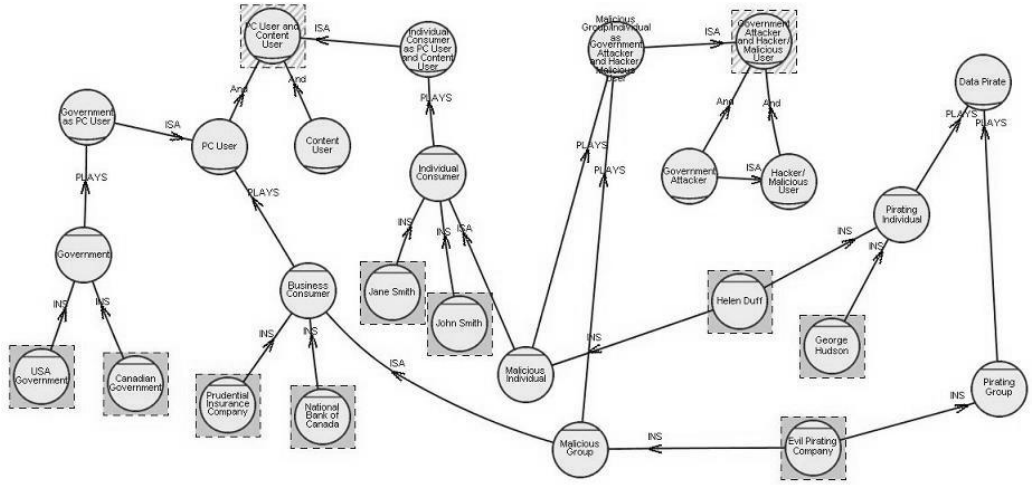


Figure 8.2-4 Single-Network view for consumers from the TCG proponents' viewpoint

8.2.3 Plain-Actors-Only, Abstract-Actors-Only and Agents-Only views

Figure 8.2-5 shows the Plain-Actors-Only view derived from TCG.Pro.AC.SN-Consumer by applying the plain actors rule (plainActorsOnlyRule). This view contains only plain actors and their direct specified forms, and it is named as **TCG.Pro.AC.SN-Consumer.PlainActors**. The view does not correspond to any diagram in TCGCS, nor does it appear immediately useful in this case, yet it might be in other cases. We show the view here to demonstrate a systematic approach in deriving various types of views.

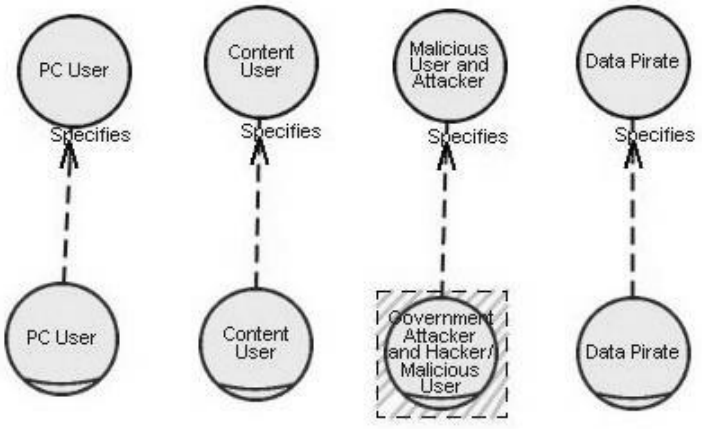


Figure 8.2-5 Plain-Actors-Only view for consumers group

Figure 8.2-6 shows the Abstract-Actors-Only view derived from TCG.Pro.AC.SN-Consumer by applying the abstract actors rule (abstractActorsOnlyRule). This view contains only abstract actors and the associations among them, and it is named as **TCG.Pro.AC.SN-Consumer.AbstractActors**. The view is a revised version of its correspondence in TCGCS (model 4.5), based on the modification we made in TCG.Pro.AC.SN-Consumer.

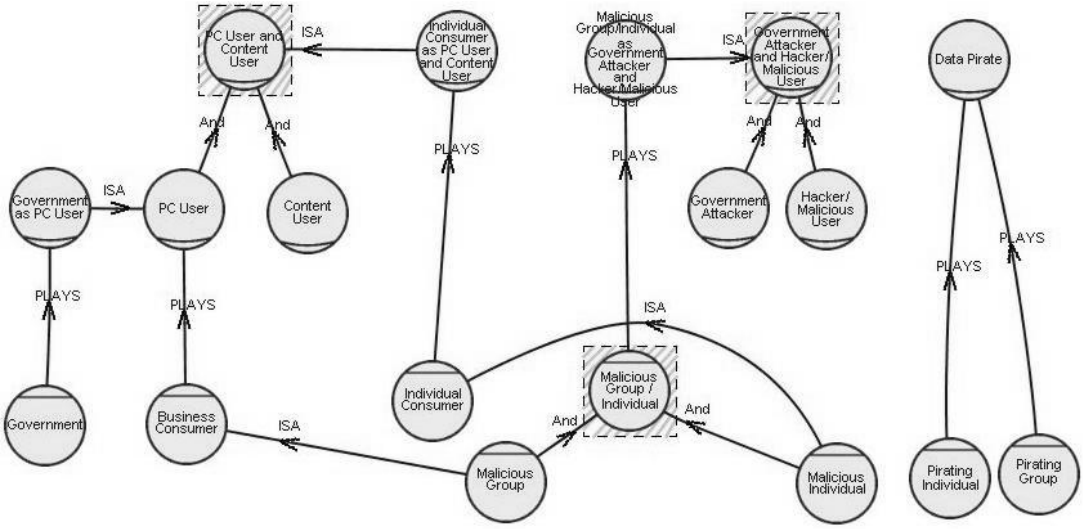


Figure 8.2-6 Abstract-Actors-Only view for consumer group

Figure 8.2-7 shows the Agents-Only view derived from TCG.Pro.AC.SN-Consumer by applying the agents rule (agentsOnlyRule). This view contains only agents and agent instances and the associations among them, and it is named as **TCG.Pro.AC.SN-Consumer.Agents**. Same as TCG.Pro.AC.SN-Consumer.PlainActors, we show this view here to demonstrate a systematic approach in deriving various types of views.

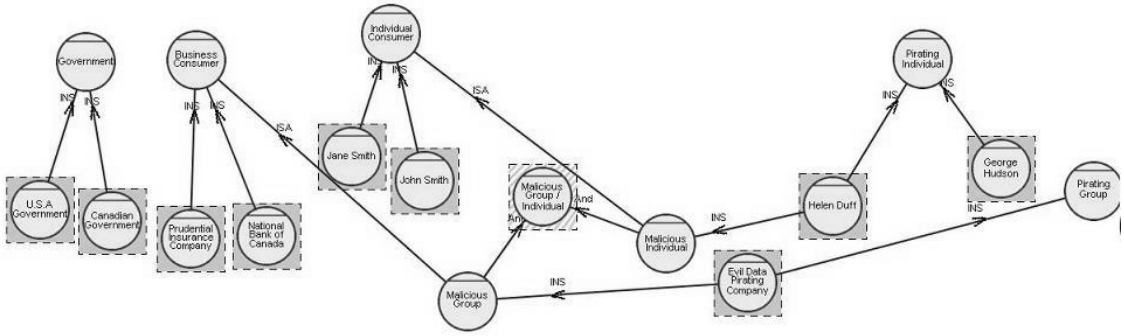


Figure 8.2-7 Agents-Only view for consumers group

8.2.4 Single-Plain-Actor views

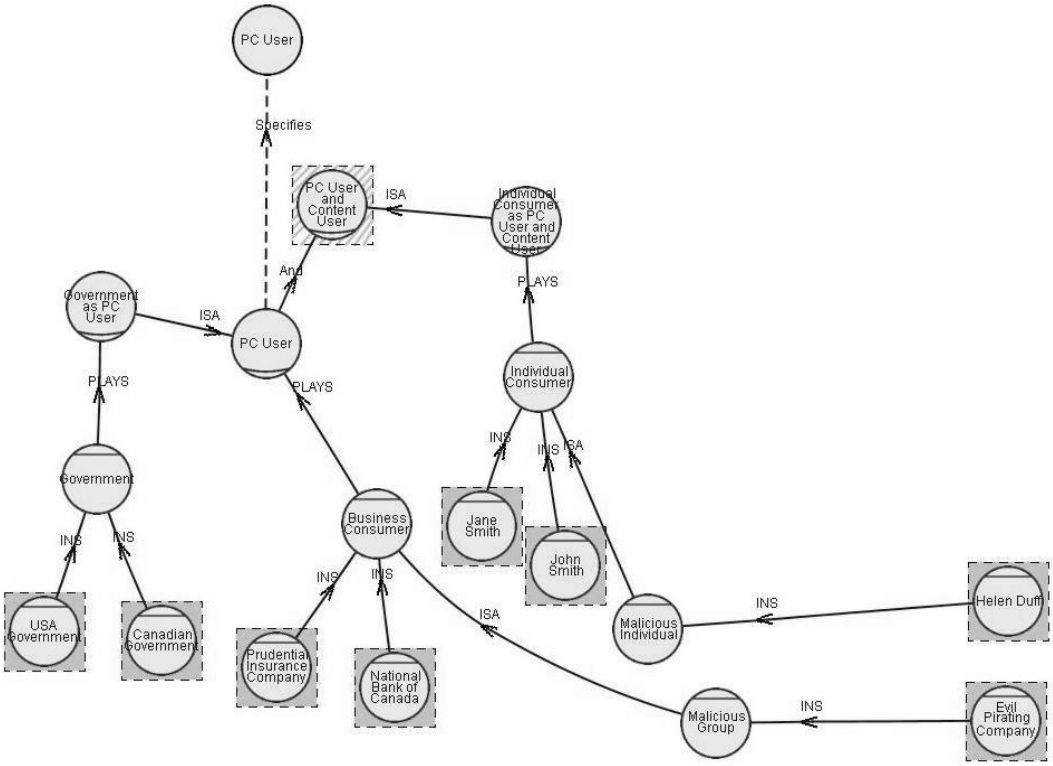


Figure 8.2-8 Single-Plain-Actor view for "PC User"

Figure 8.2-8 to Figure 8.2-11 show four Single-Plain-Actor views derived from TCG.Pro.AC.SN-Consumer by applying the single plain actor rule (singlePlainActorRule) for each of the four plain actors, one at a time. This type of view contains the selected plain actor and the specified forms that can inherit all of its external relationships. We name the four views **TCG.Pro.AC.SN-**

Consumer.SPA-PCUser, **TCG.Pro.AC.SN-Consumer.SPA-ContentUser**, **TCG.Pro.AC.SN-Consumer.SPA-MaliciousUser**, and **TCG.Pro.AC.SN-Consumer.SPA-DataPirates**. These views do not have a correspondence in TCGCS, but we show them to illustrate how a given AC view might be decomposed according to domain knowledge plain actors.

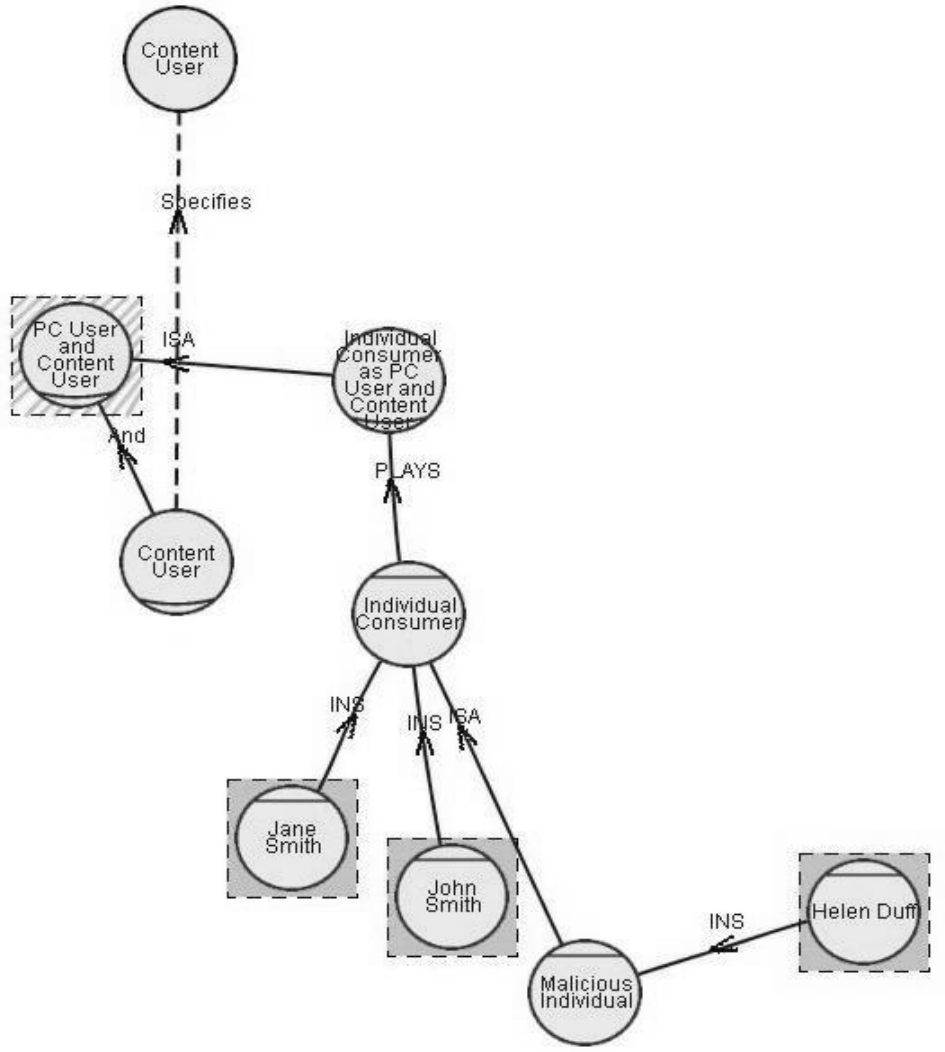


Figure 8.2-9 Single-Plain-Actor view of "Content User"

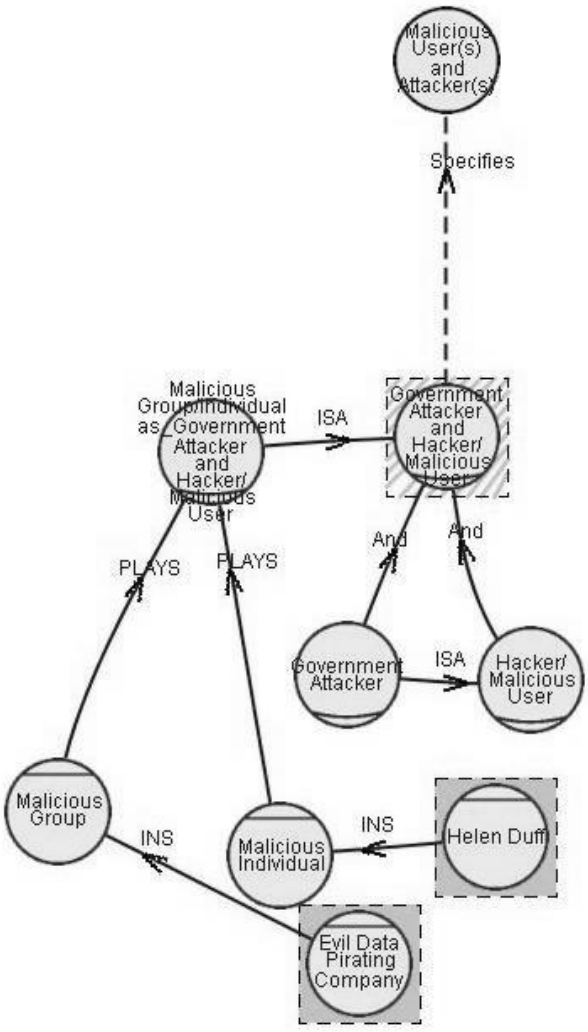


Figure 8.2-10 Single-Plain-Actor view of "Malicious User(s) and Attacker(s)"

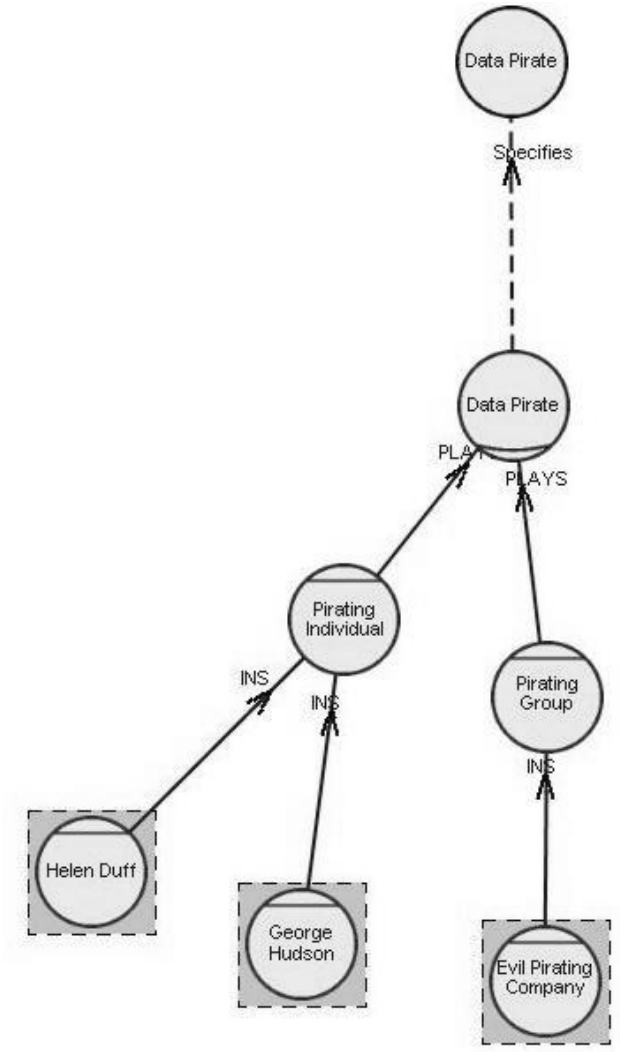


Figure 8.2-11 Single-Plain-Actor view of "Data Pirate"

8.2.5 Direct-Replaceable views

Figure 8.2-12 to Figure 8.2-15 show Direct-Replaceable views derived from TCG.Pro.AC.SN-Consumer by applying the direct replaceable rule (directReplaceableRule) for the selected actors, one at a time. We use these views to deduce inter-actor dependencies. The given actor can stand in for other actors shown in this view in any SD view containing the latter. We highlight the given actor using a solid rectangle. This type of substitution implies that the given actor has either the exact same external relationship as, or a larger set of external relationships than, the ones that are directly replaceable by it.

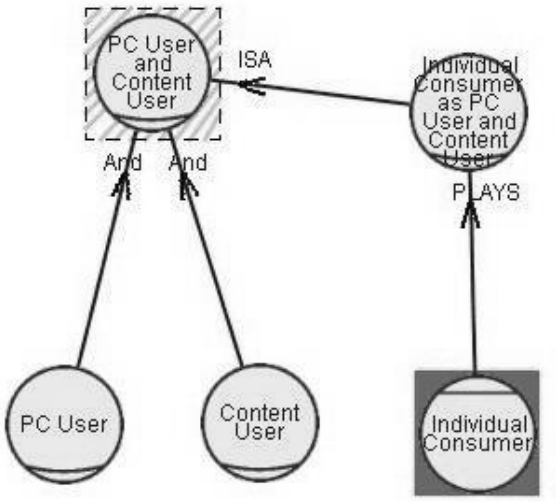
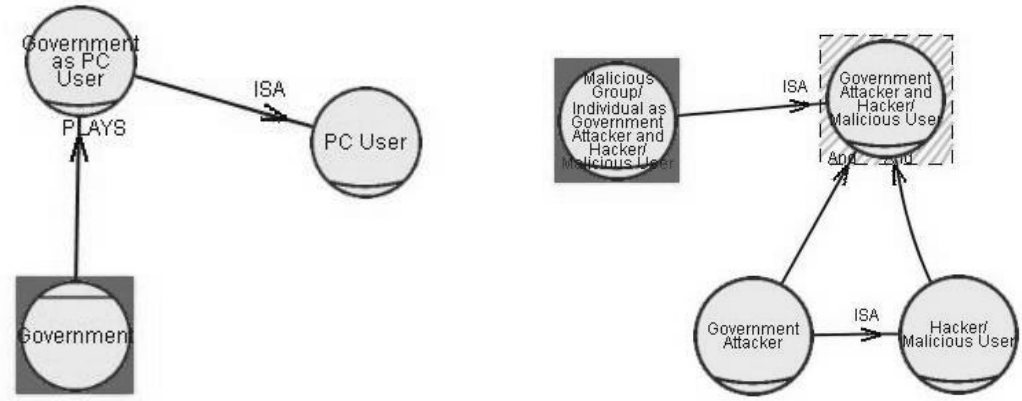


Figure 8.2-12 Direct-Replaceable actors view of agent Individual Consumer

For example, Figure 8.2-12 shows a given actor "agent **Individual Consumer**" and the Direct-Replaceable view of it. There are corresponding diagrams (model 2.6.3 and model 2.6.4) for this view in TCGCS. We name this view **TCG.Pro.AC.SN-Consumer.DR-IndividualConsumer**. From this view we learnt that the given actor inherits all external relationships from role **PC User**, role **Content User**, role **PC User and Content User**, or role **Individual Consumer as PC User and Content user**, and that therefore agent Individual Consumer can substitute any of these in an SD view.

Some other examples are the Direct-Replaceable views for agent **Government** (Figure 8.2-13(a)) and role **Malicious Group…Users** (Figure 8.2-13(b)), and we name them **TCG.Pro.AC.SN-Consumer.DR-Government** and **TCG.Pro.AC.SN-Consumer.DR-MaliciousUsers**, respectively. There are corresponding diagrams (model 2.17.1 and model 2.19.2, respectively) for these views in TCGCS,



(a) For agent Government **(b) For role Malicious Group…User**

Figure 8.2-13 Direct-Replaceable views for specified actors

In addition, we show some views that do not exist in TCGCS but will be used to derive SD views of agent instances such as **Helen Duff** and **George Hudson**. Figure 8.2-14 and Figure 8.2-15 show the Direct-Replaceable views for Helen Duff and George Hudson, and we name them **TCG.Pro.AC.SN-Consumer.DR-HelenDuff** and **TCG.Pro.AC.SN-Consumer.DR-GeorgeHudson**, respectively.

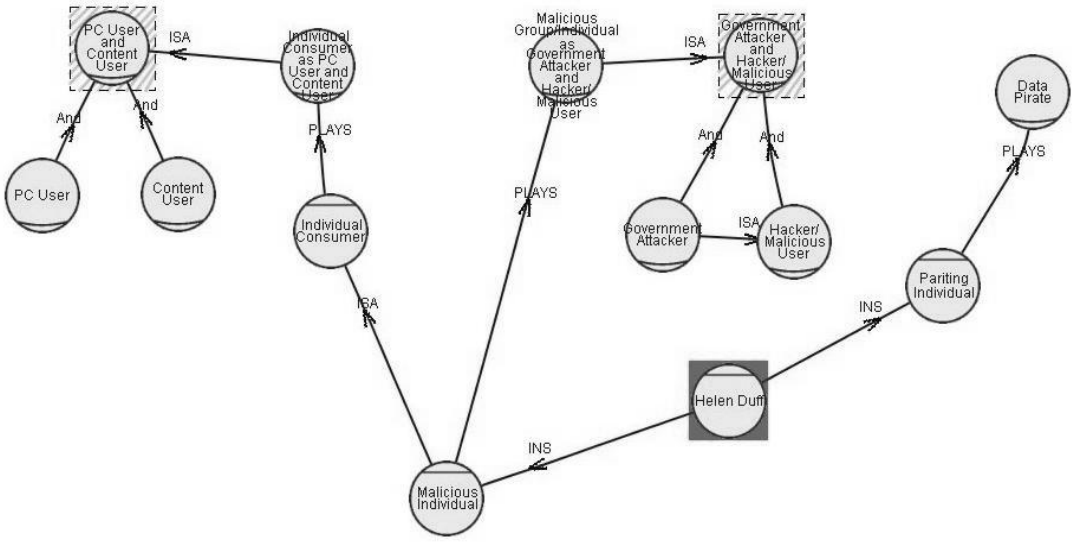


Figure 8.2-14 Direct-Replaceable view for agent instance Helen Duff

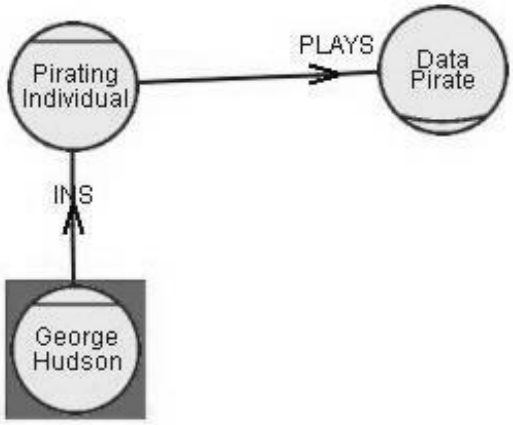
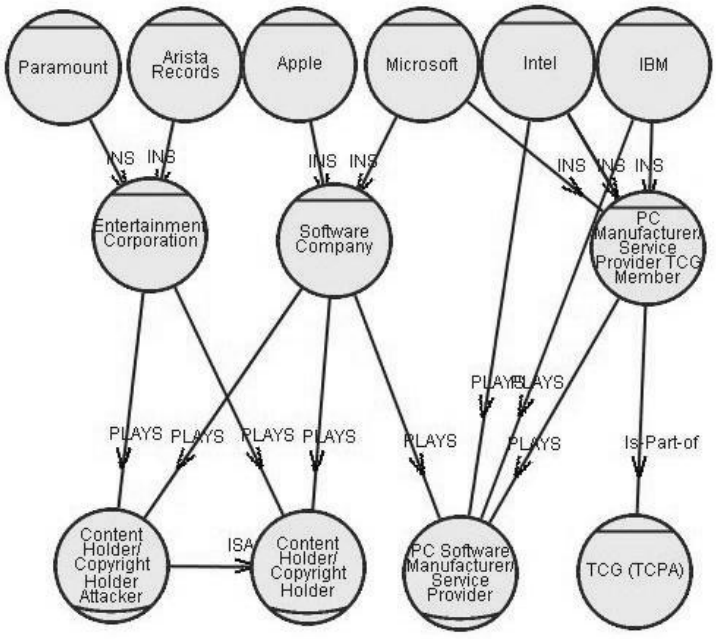


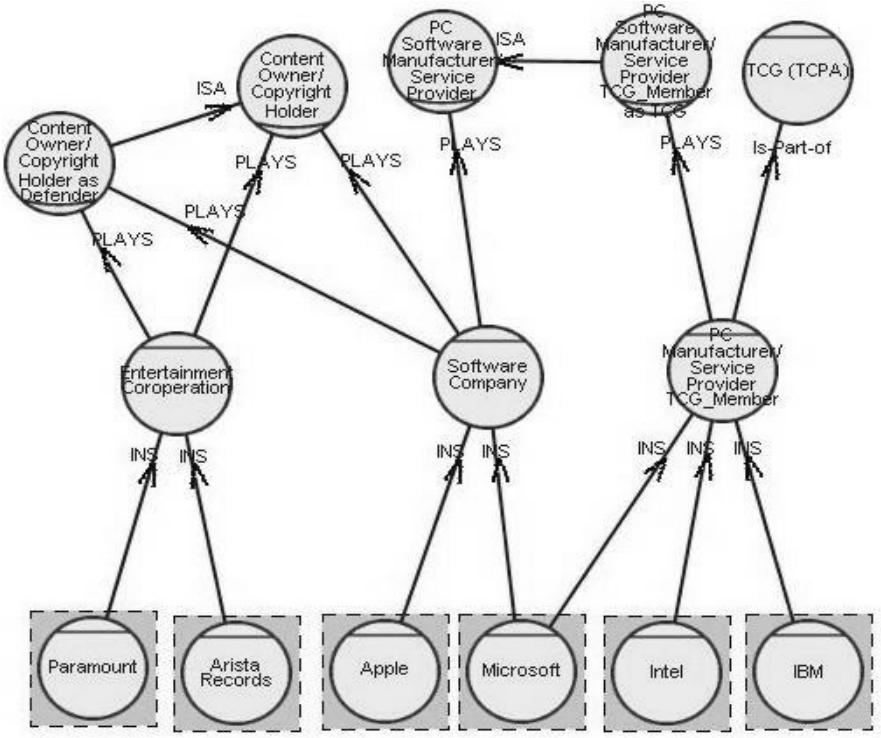
Figure 8.2-15 Direct-Replaceable view of agent instance George Hudson

8.2.6 Discussion

In this section, we demonstrate the process and results of the decomposition of the basic AC (TCG.Pro.AC) into various forms of partial views according to the selection rules. Relationships among these views were presented in a View Map, where each view (diagram) is modeled as a node in a tree-like structure. This view map helps increase the efficiency in accessing the distributed views across a document.



(a) Model 4.3 in the original work



(b) Single-Network view for consumers

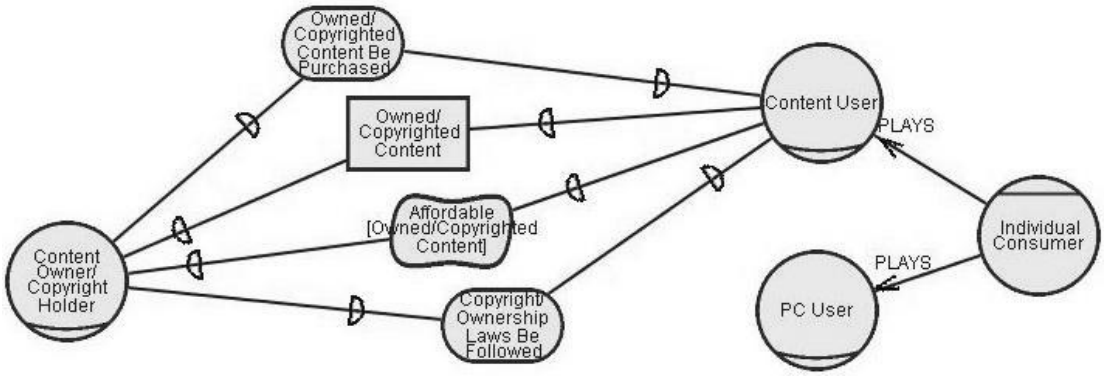
Figure 8.2-16 Comparison of Actor Views (diagram) showing redundancy identified

The original Basic Actor Class view from TCGCS was developed in an ad-hoc manner, and thus contains inconsistencies. Without a systematic method, it was difficult to identify these problem areas through its 120 diagrams. Our research enforced for the first time a tighter relationship between the AC and the SD views so that modeled elements are subject to a more rigorous consistency check within one model. Using this technique, we identified redundancy, logic gaps, and inconsistency from the original TCG case study.

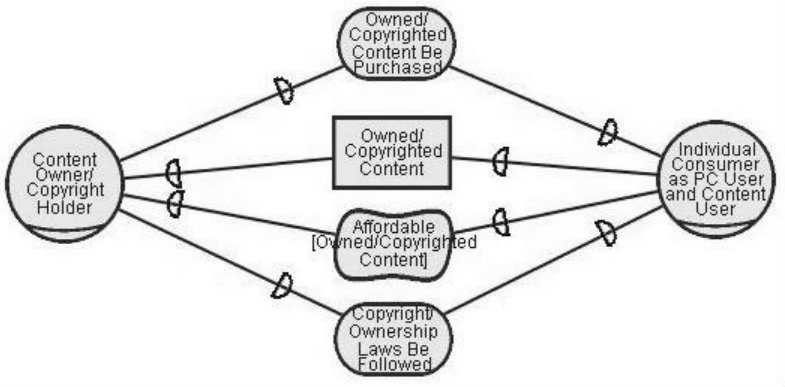
First, we identified redundancy in the original model. In Figure 8.2-16(a), there are two "plays" links to role **PC Software manufacturer/ Service Provider** that originated from agents **Intel** and **IBM**, respectively. During our revisit, we found that these links are redundant since each of them has been implied by the "INS" links from it (e.g., Intel) to agent **PC Manufacturer/ Service Provider TCG Member**, and then by the "plays" link from the latter to

role PC Software manufacturer/ Service Provider. In fact, Intel and IBM appear to be agent instances, and in our reformulated i* semantics, they should not relate to "plays" links. Therefore, in our modified version (Figure 8.2-16(b)), these redundancies are removed, and Intel and IBM are highlighted as agent instances to avoid confusion from the agent.

Next, we identified logical gaps in the way actors in the SD views are replaced. For example, in TCGCS, role **Content User** in Figure 8.2-17(a) was replaced by role **Individual Consumer as PC User and Content User** (ICPCUCU) in Figure 8.2-17(b), and the latter seems to share the same set of external relationships as the former. We inferred the human reasoning from this transition: First, since agent Individual Consumer "plays" role Content User and PC User, we introduce a new role ICPCUCU to cover all three actors; next, since role ICPCUCU covers Content User, it should support all the external dependencies of the latter. These rationales were not specified explicitly in the original model and this fact may have led to user confusion, while the replacement of actors cannot be automated.



(a) Model 2.6.3 in TCGCS



(b) Model 2.6.4 in TCGCS

Figure 8.2-17 Example of logic gaps in actor replacement

To make the transition from the model 2.6.3 to model 2.6.4 automatically obtainable from the Baseline model, we modified these associations among agent **Individual Consumer**, role **Content User**, role **PC User**, **and** role **Individual Consumer as PC User and Content User** (ICPCUCU). Figure 8.2-18 shows the result of our modification. We first separate this part of information from the SD view and fit them into the AC view. Then we introduce a new role **PC User and Content User** (PCUCU) as the *whole* for role PC User and role Content User. From the implication of the "complete composition" links in the AC view, we know that the new role (the whole) inherits all external relationships from its parts. We let role **ICPCUCU** be a specialized form of role **PCUCU** through the "ISA" link, and we know that the former inherits all external relationships from the latter. Thus, **ICPCUCU** indirectly inherits all external relationships from role Content User. The above analysis process reaches the same result as was expected in the transition shown in Figure 8.2-17, yet this process demonstrates a systematic approach and can be fully automated.

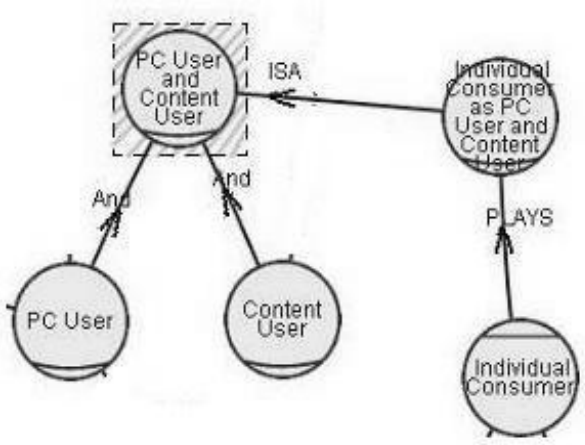
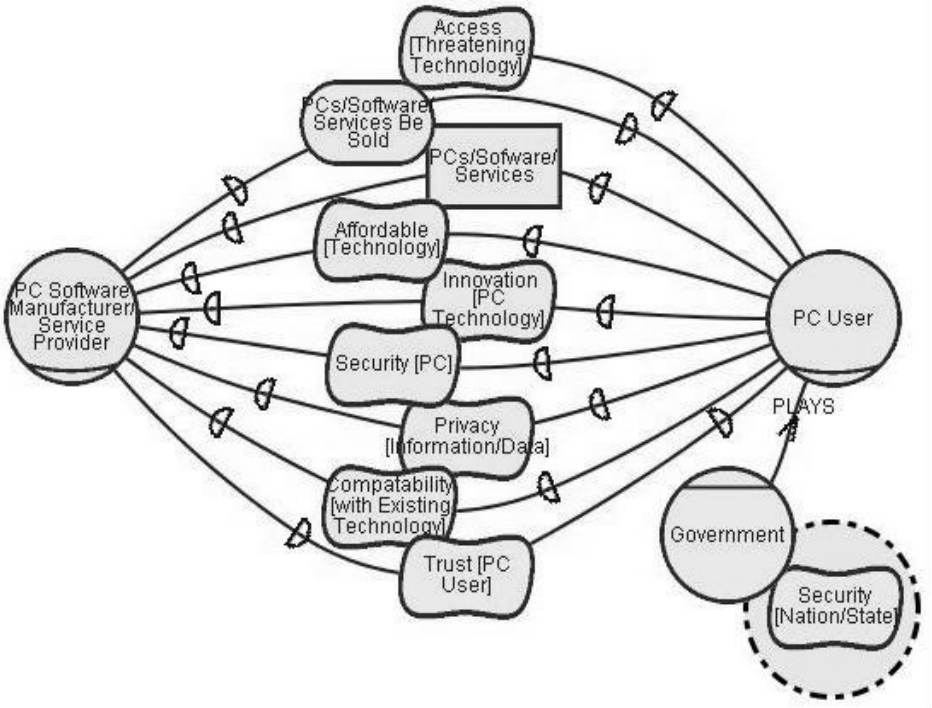
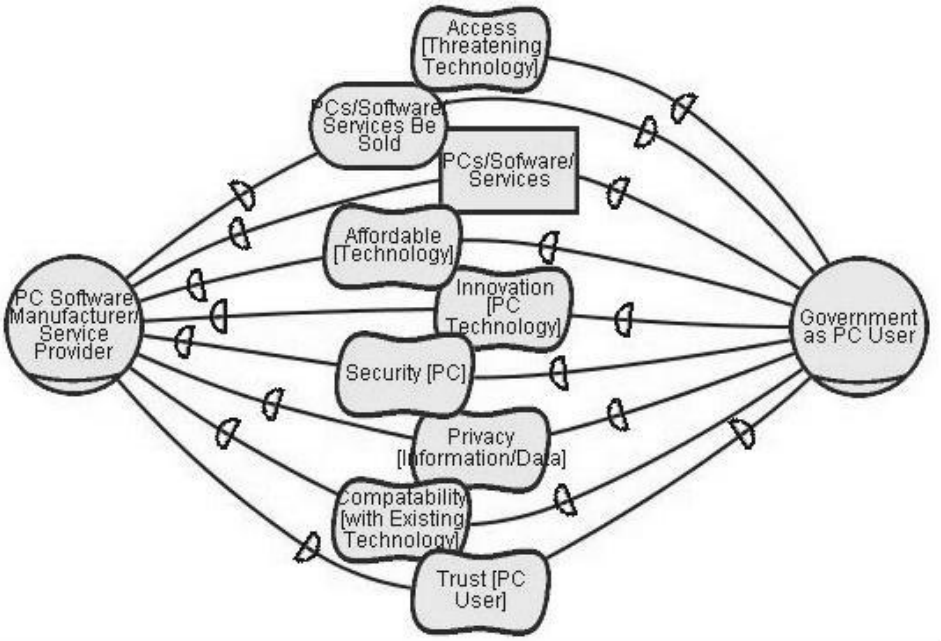


Figure 8.2-18 Modified representation to fill the logic gap

Similar adaptations have been made to the substitution of **PC User** (Figure 8.2-19) and **Hacker/Malicious User** (Figure 8.2-20). Our modified versions are shown in Figure 8.2-21(a) and (b), respectively.

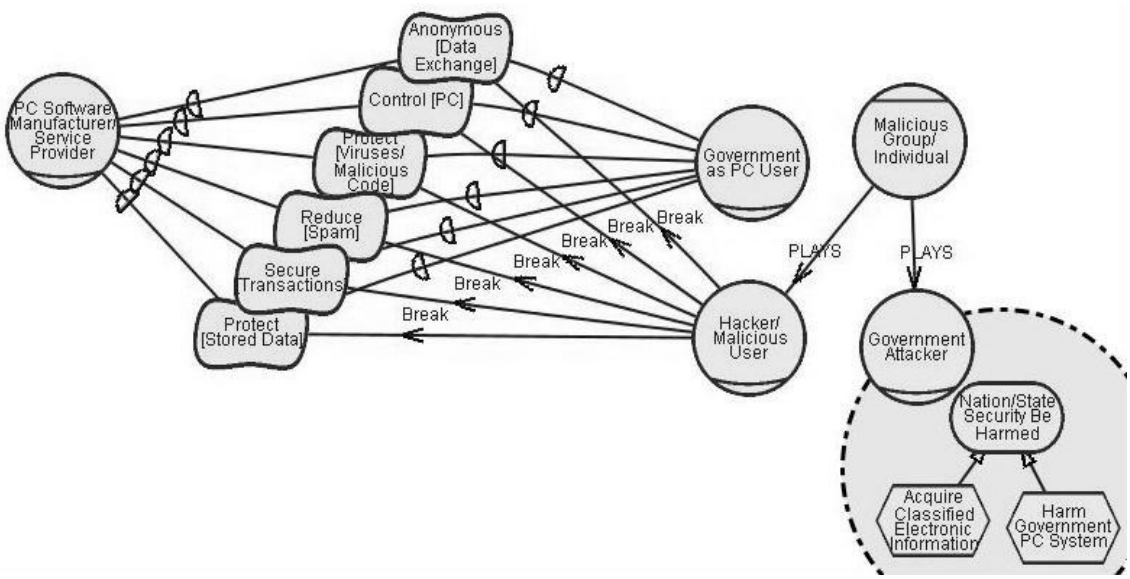


(a) Diagram showing actor association

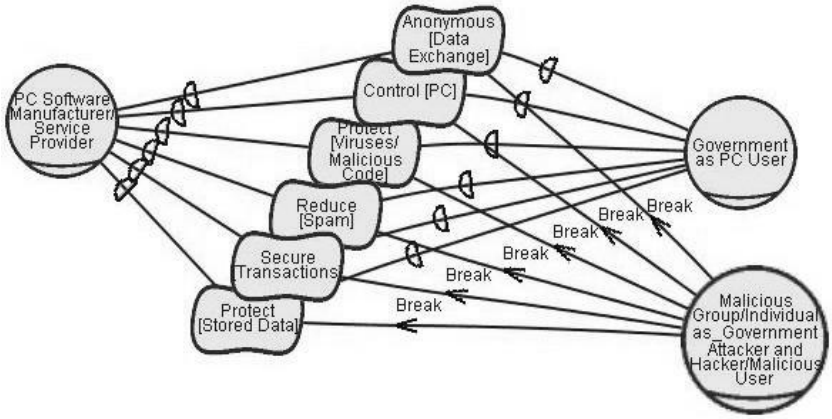


(b) Diagram showing the resulting substitution

Figure 8.2-19 Substitution of role PC User in TCGCS



(a) Diagram showing actor association



(b) Diagram showing the result of substitution

Figure 8.2-20 Substitution of Hacker/Malicious User in TCGCS

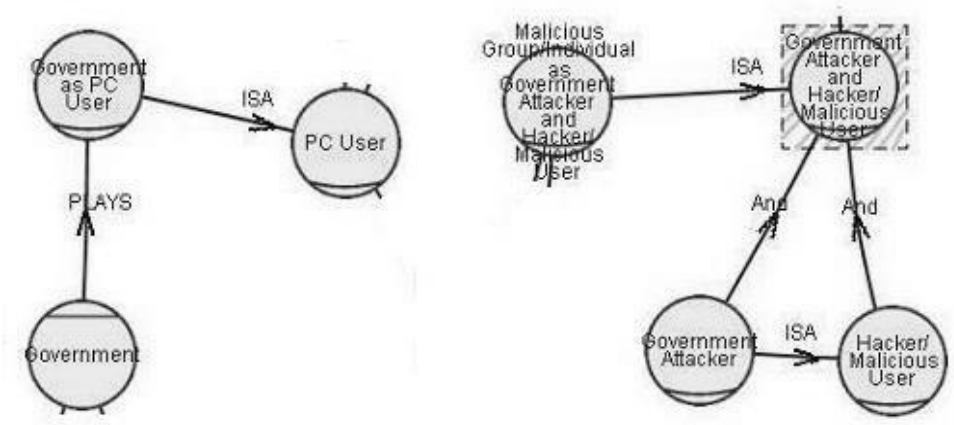


Figure 8.2-21 Our modified AC views in removing the logic gaps

Finally, we identified one inconsistency (or duplicate) in the actor-type assignment. From models 2.5.11 to model 2.5.13, agent **PC Manufacturer/Service Provider TCG Member as TCG (TCPA)** (PCMSPTCGMTCG) seems to have replaced role **PC Manufacturer/Service Provider** (PCMSP) in the SD diagram. If we follow the same tacit logic explained in the previous comment, the former (agent PCMSPTCGMTCG) has to "plays" the latter (role PCMSP) to make the replacement in the SD view consistent. On the other hand, agent PCMSPTCGMTCG seems related with agent TCG (TCPA) in some way. Since TCG is a group, most likely the former should be "is-Part-of" the latter. However, there already existed an agent **PC**

Manufacturer/Service Provider TCG Member that has exactly the same actor associations in the model. Thus, either agent PCMSPTCGMTCG is a duplicate or it introduces some inconsistency; it does not seem like a duplicate in that the author used three models to emphasize it. Based on the above assumptions, we modified PCMSPTCGMTCG into a role that is a specialized form of role PCSMSP and is played by agent PCMSPTCGM. It still can replace role PCSMSP in any SD view. The modified version is shown in Figure 8.2-22.

(?? The corresponding models in TCGCS are very big, do I show them here??)

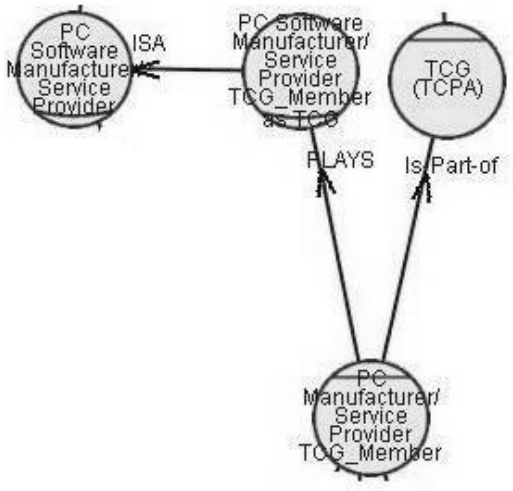


Figure 8.2-22 Modified version showing PCMSPTCGMTCG as a role

The AC view appears to be the weakest part in TCGCS, but this is a result of the lack of definitions, rules and guidelines in previous i* literature. With the clarification in our reformulated i* framework—and especially with the introduction of the *external relationship inheritance rule* along association links—redundancies, logical gaps and even inconsistencies that existed in the original model were revealed. Thus, our approach not only scales down complex AC views, but also helps verify the validity of large scale i* models.

8.3 Strategic Dependency Views

Pair-wise-Actors and Single-Actor-Focus SD views were extensively used in TCGCS. This intuitive approach matches exactly what we have proposed in

Chapter 6. We can stay with the Specified Actor Based SD view throughout our rework because the two baseline models documented in TCGCS only contain specified actors. Thus, the only problem is the lack of a reference structure for the SD diagrams in the original document.

In this section we present related SD views in a centralized manner and provide their relationships in a view map. Our purpose is to verify our proposed view extensions, so we choose just enough diagrams from TCGCS to test each type of view. There are other diagrams in TCGCS correspond to SD views, since they would follow the same pattern as the ones we discuss in this section, we do not show them here.

We choose the Basic SD view (**TCG.Pro.SD**) from TCG proponents' view point as the original view. Figure 8.3-1 shows the relationships between TCG.Pro.SD and the sub-views derived from it. TCG.Pro.SD (abbreviated as **SD**) is first decomposed into a set of Single-Actor-Focus SD views, and we select five of them in this section, as follows: role Government as PC User (**SD.SA-GovernmentPCU**), role Individual Consumer as PC User and Content User (**SD.SA-IndividualPCUCU**), role Malicious User, agent TCG (**SD.SA-TCG**), and agent instance George Hudson (**SD.SA-GeorgeHudson**). SD.SA-GovernmentPCU and SD.SA-TCG are further scaled down to Pair-wise-Actors views **SD.PW-GovernmentPCU-PCSMSP** and **SD.PW-TCG-HackerMU**, respectively.

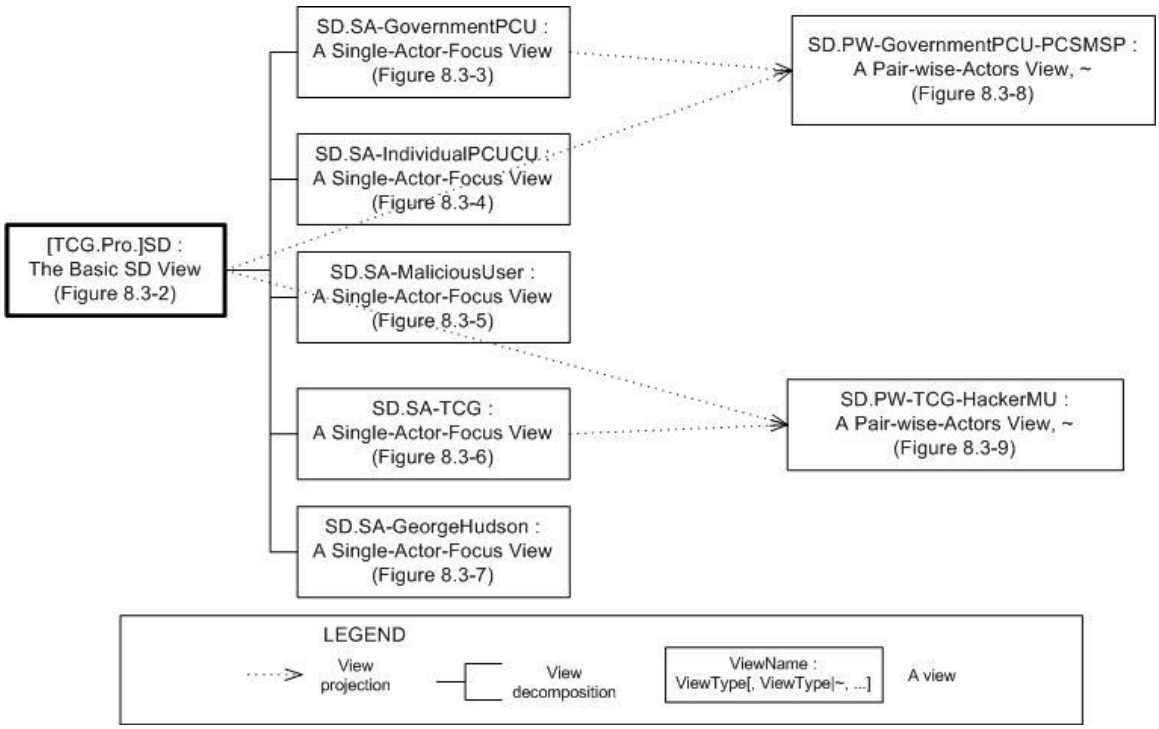


Figure 8.3-1 View map for partial SD views from the Pro TCG view point

8.3.1 The Basic SD view

Figure 8.3-2 shows the Basic SD view from the TCG proponents' viewpoint; we name it as **TCG.Pro.SD**. This view shows an extremely complex relationship among actor PC User, TCG (TCPA), PC Software Manufacturer/Service Provider, and Hacker/Malicious User. It appears quite difficult to read.

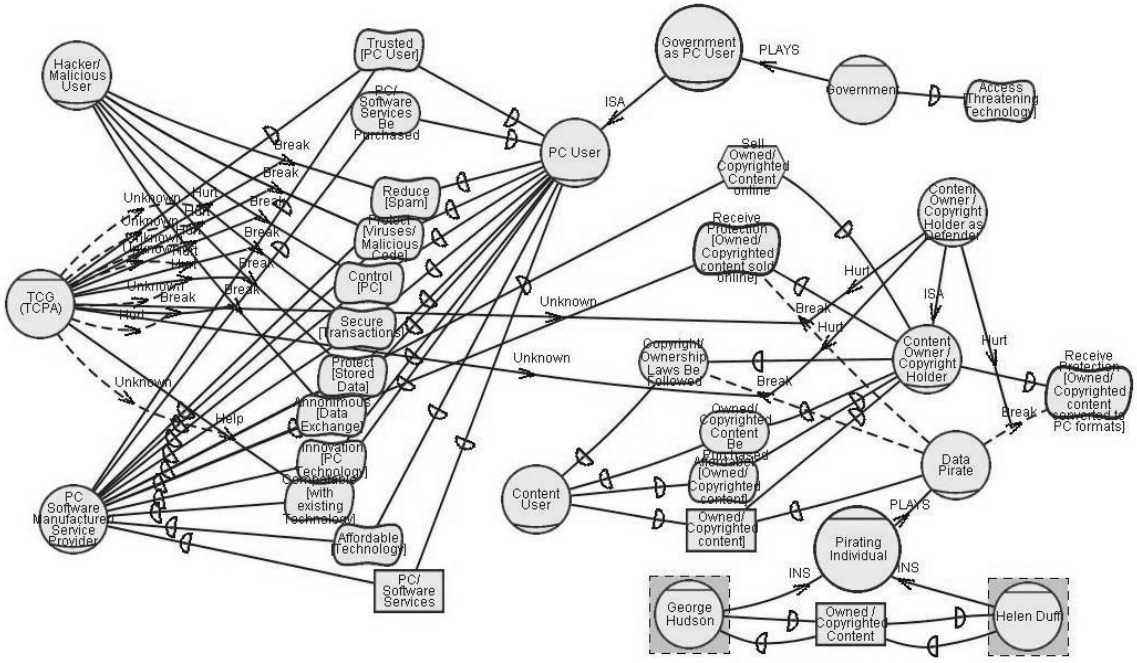


Figure 8.3-2 Basic SD view from the TCG proponents' viewpoint

8.3.2 Single-Actor-Focus SD views

We apply the single actor rule (singleActorFocusSDRule) to TCG.Pro.SD and obtain a set of Single-Actor-Focus SD views. Those for the following five actors are presented: role **Government as PC User**, role **Individual Consumer as PC User and Content User**, role **Malicious Group/Individual as Government Attacker and Hacker/Malicious User**, agent **TCG**, and agent instance **George Hudson**.

Figure 8.3-3 shows the Single-Actor-Focus SD view of role Government as PC User. We name it **TCG.Pro.SD.SA-GovernmentPCU**. There are two correspondence diagrams (models 2.20.1 and 2.18.1) to this view in TCGCS.

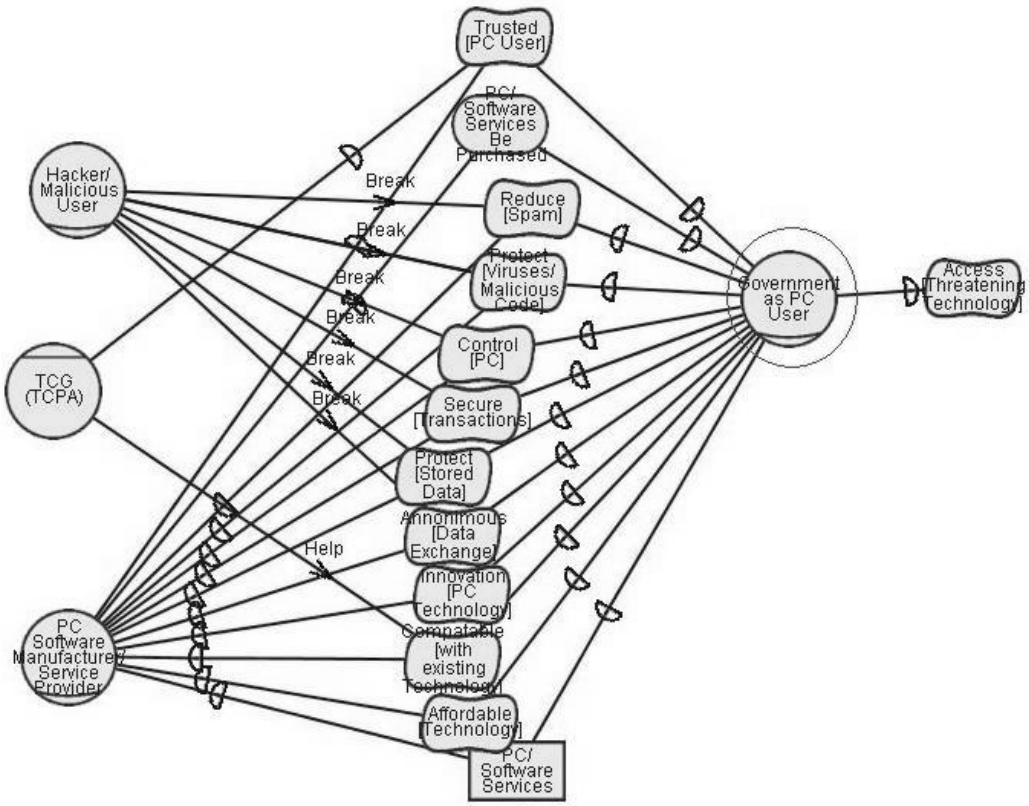


Figure 8.3-3 External relationships for role Government as PC User

Figure 8.3-4 shows the Single-Actor-Focus SD view of role Individual Consumer as PC User and Content User (ICPCUCU). We name it **TCG.Pro.SD.SA-IndividualPCUCU**. It summarizes the information contained in TCGCS (model 2.3.1 for PC Manufacturer, model 2.4.1 for Hacker, model 2.5.1 for TCG, and model 2.6.1 for Content Owner). Using actor associations (shown in Figure 8.2-12) and the external relationship inheritance rule, we know that role **ICPCUCU** shall inherit all external relationships for role **PC User** and role **Content User**. That is the method we used to calculate the external relationships for ICPCUCU.

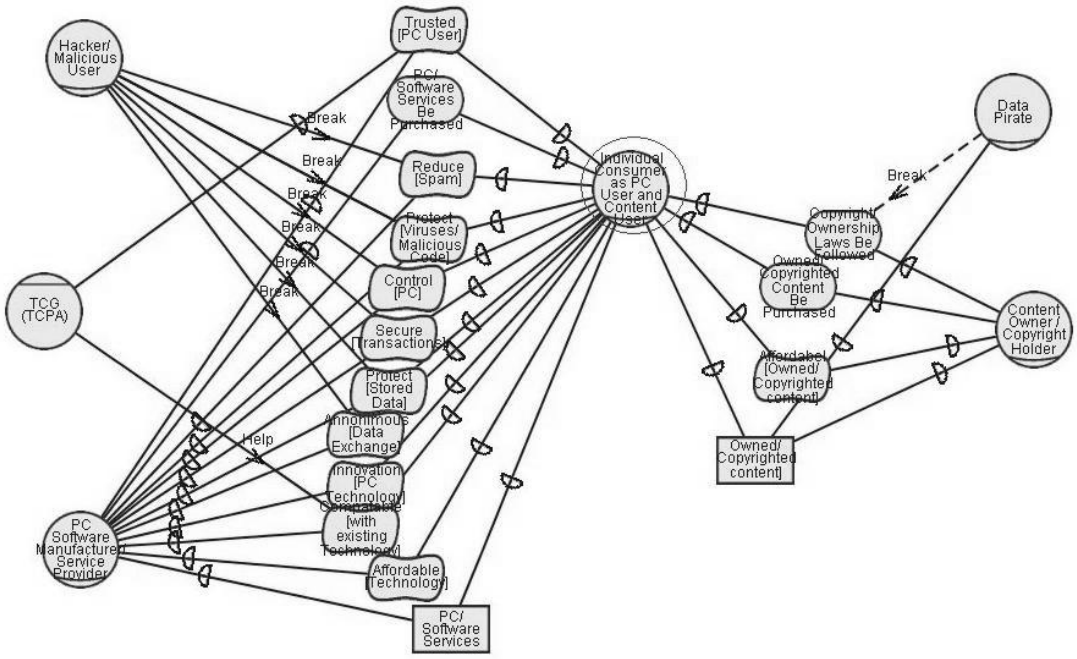


Figure 8.3-4 External relationships for role Individual Consumer as PC User and Content User

Figure 8.3-5 shows the Single-Actor-Focus SD view for role Malicious Group/Individual as Government Attacker and Hacker/Malicious User (MGIGAHM). We name it **TCG.Pro.SD.SA-MaliciousUser**. There is a correspondence in TCGCS (model 2.4.1) to this view. We replaced role **Hacker/Malicious User** (HMU) with role **MGIGAHM** in our version – because the former is a specified form (ISA) of role **Government Attacker and Hacker/Malicious User** (GAHMU), while GAHMU is the whole of HMU. According to the external relationship inheritance rule, MGIGAHM inherits all external relationships from HMU. Therefore, the replacement in this SD view is legal.

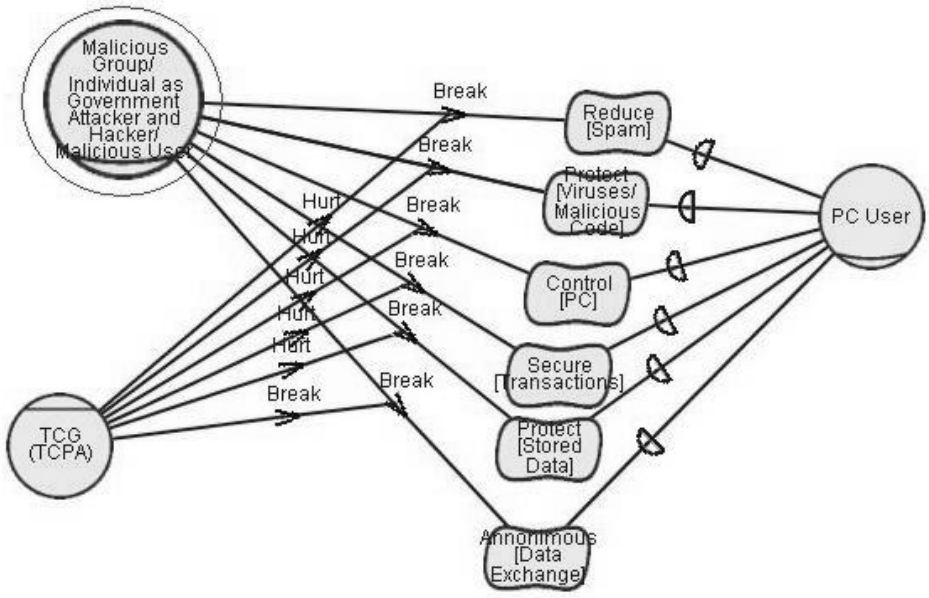


Figure 8.3-5 External relationships for Malicious Group/Individual as Government Attacker and Hacker/Malicious User

Figure 8.3-6 shows the external tasks and goals of agent TCG and illustrates how they affect the effects exerted by role Hacker and role Data Pirate. We name it **TCG.Pro.SD.SA-TCG**. Information shown in this view corresponds to three diagrams (models 2.5.1, 2.10.1, and 2.14.1) in TCGCS. Note that in the diagram shown below, we use the full name of each dependum to indicate its depender and dependee in the form "(depender, dependee)". For example, **Reduce [spam] (PCUser, PCMSPTCG)** denotes that **PC User** depends on **PC Software Manufacturer/ Service Provider as TCG Member** (**PCMSPTCG**) to reduce spam.

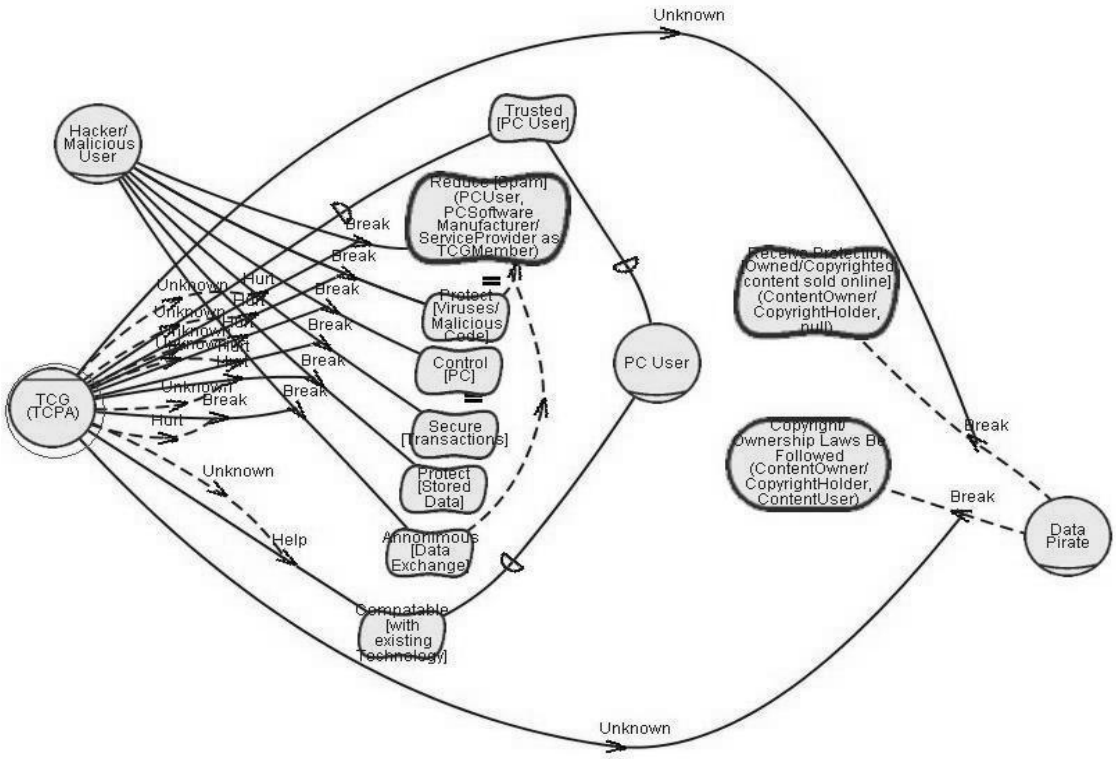


Figure 8.3-6 External relationships for agent TCG (TCPA)

Figure 8.3-7 was derived from the Single-Actor-Focus SD view for role **Data Pirate** and the Direct-Replaceable view of agent instance **George Hudson** (Figure 8.2-15). Since the agent instance "plays" role Data Pirate, it inherits all external relationships of that role. In addition, we know from TCG.Pro.SD (Figure 8.3-2) that this agent instance has extra dependencies to another agent instance **Helen Duff**. Therefore, we combined the above information and produced the Single-Actor-Focus SD view for George Hudson below. Part of our information is obtained two diagrams (models 2.6.8 and 2.6.6) in TCGCS. We name this view **TCG.Pro.SD.SA-GeorgeHudson**.

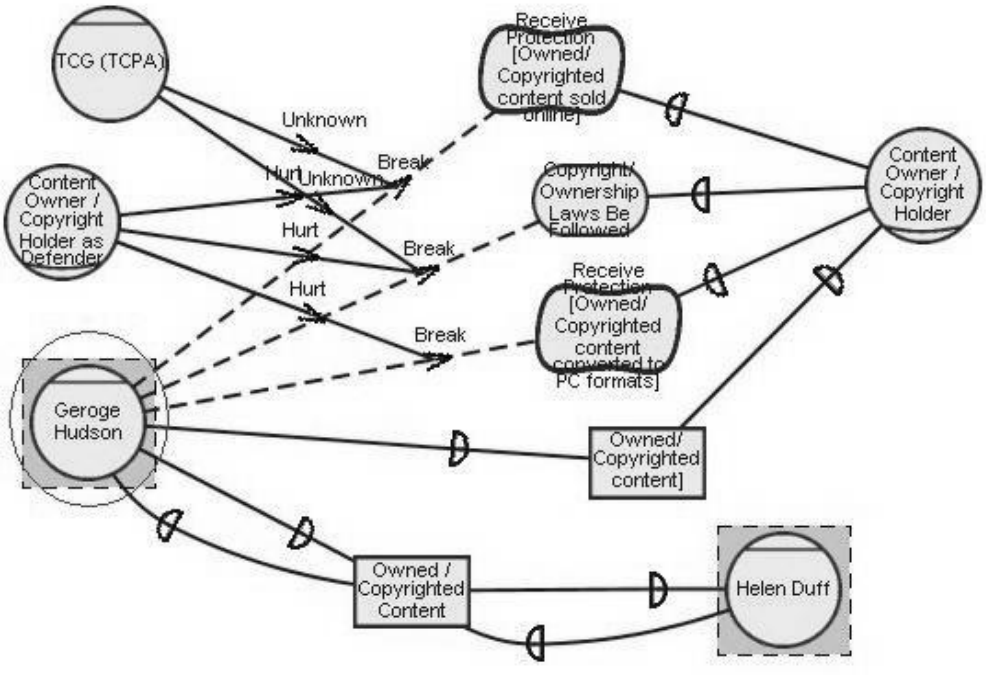


Figure 8.3-7 External relationships for agent instance George Hudson

8.3.3 Pair-wise-Actors SD views

Figure 8.3-8 shows the Pair-wise-Actors view for PC Software Manufacturer/ Service Provider and Government as PC User. We obtained it by applying the pair-wise rule (pairwiseActorsRule) over TCG.Pro.SD or TCG.Pro.SD.SA-GovernmentPCU. We name this view **TCG.Pro.SD.PW-GovernmentPCU-PCSMSP**. This view appears exactly the same a diagram (model 2.18.1) in TCGCS—except for the omission of the dangling dependum Access [Threatening Technology].

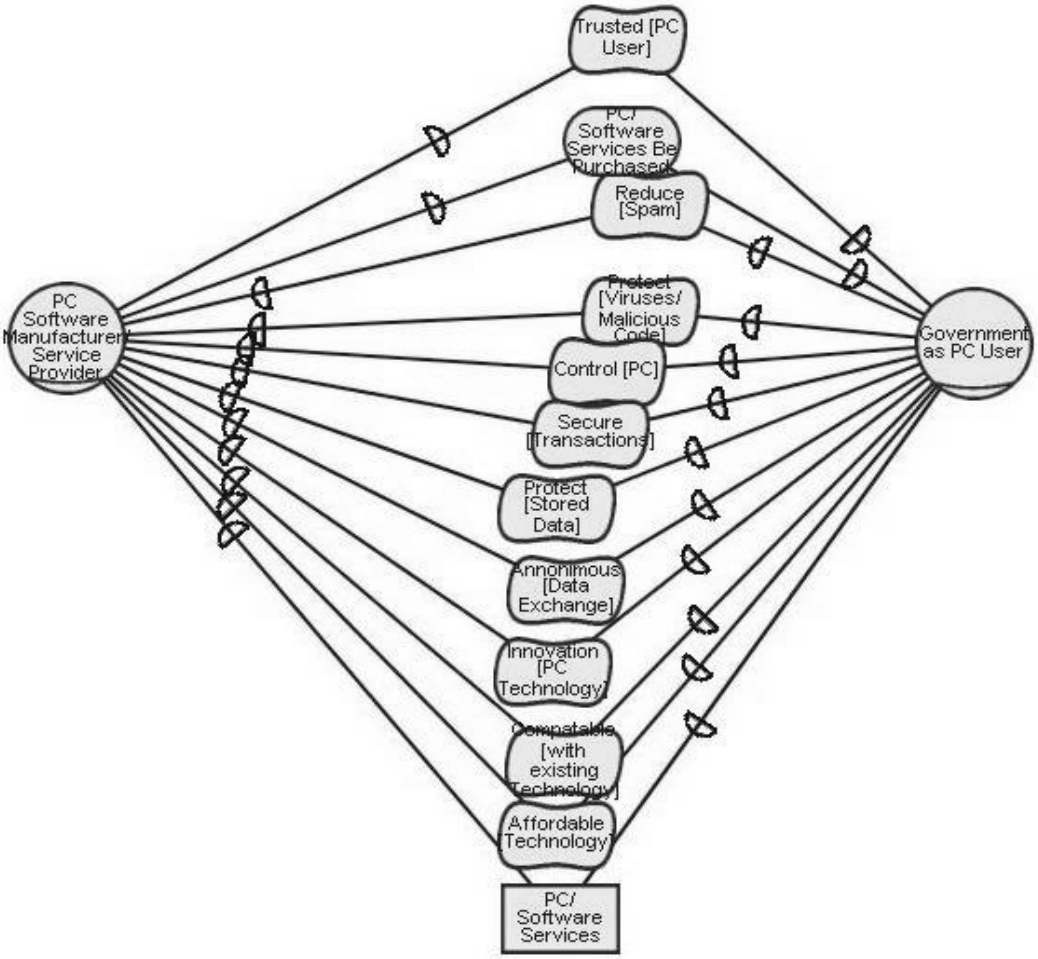


Figure 8.3-8 Pair-wise view for PC Software Manufacturer/Service Provider and Government as PC User

Figure 8.3-9 shows the Pair-wise-Actors view for TCG and Hacker/Malicious User. We obtained it by applying the pair-wise rule (pairwiseActorsRule) over TCG.Pro.SD or TCG.Pro.SD.SA-TCG. We name this view **TCG.Pro.SD.PW-TCG-HACKERMU**. This view conveys the same information as does its correspondence diagram (model 2.5.1) in TCGCS. Yet it appears much simpler and more comprehensible, with the omission of the depender (PC User) and dependee (PCMSPTCG) of the six dependums (e.g., Reduce [Spam]) and the 12 corresponding dependency links.

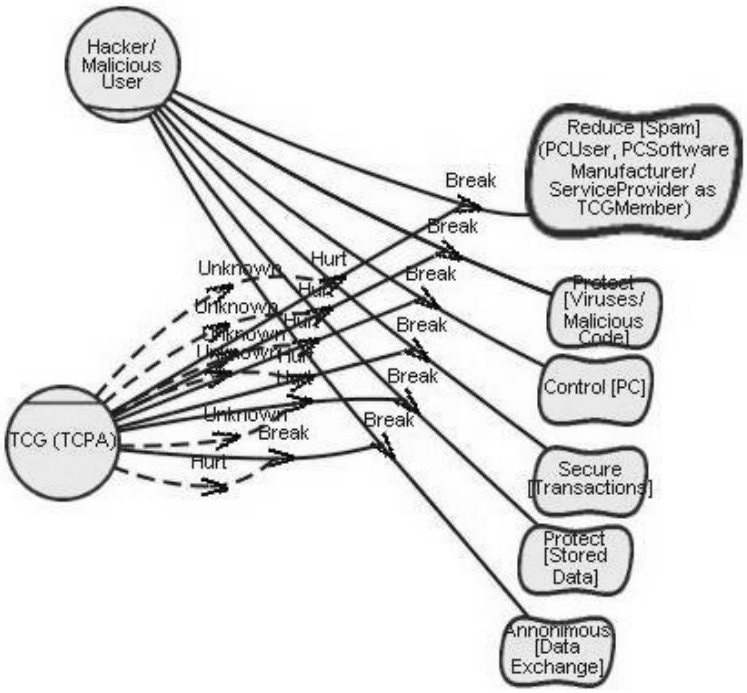


Figure 8.3-9 pair-wise view for TCG (TCPA) against Hacker/Malicious User

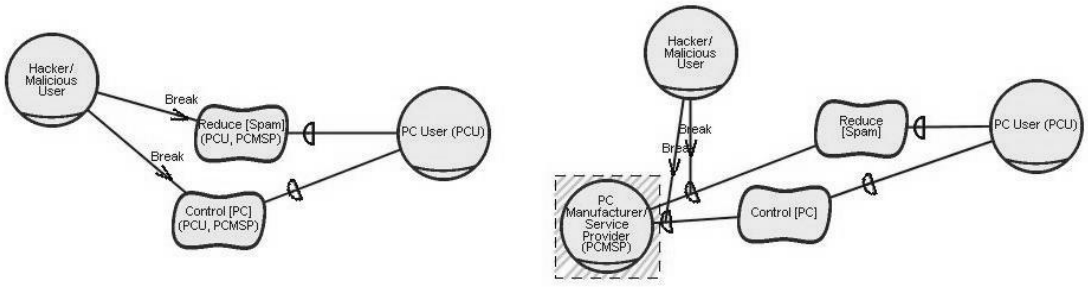
8.3.4 Discussion

In this section, we validate our approach in reducing a Basic SD (TCG.Pro.SD) into various forms of partial SD views so as to increase its comprehensibility. The reduction was performed manually according to the selection rules defined for each SD sub-view. Resulting partial views were presented in a top-down flavor—that is, from the complex and complete basic view to the simplified partial views. Relationships among these views are presented in a View Map.

In TCGCS, intuitive pair-wise views are used extensively. Consequently, the presentation makes it difficult to perform node analysis centering on a given actor. To study the vulnerability and opportunity of a given actor, model users need to study several diagrams, usually shown in separate chapters. During our rework of TCGCS, Single-Actor-Focus views were summarized according to all pair-wise SD views related across the original document to the following selected actors: role **Government as PC User**, role **Individual Consumer as PC User and Content User**, role **Malicious Group/Individual as Government**

Attacker and Hacker/Malicious User, agent **TCG**, and agent instance **George Hudson**.

A difference also exists in the way we should express external contribution from an actor to a dependum. For example, the external *break* contribution from role **Hacker/Malicious User** ends at softgoal dependum **Reduce [spam]** (Figure 8.3-10(a))in TCGCS, but according to our reformulated i* semantics it should end at the corresponding outgoing dependency link of the dependum (Figure 8.3-10(b)).



(a) Style used in this chapter **(b) Style proposed in Chapter 6**

Figure 8.3-10 Differences in expressing external contributions to dependums

In fact, the style applied in TCGCS appeared more concise in the graphical representation (no extra actor **PCMSP**, highlighted with dashed rectangle, shown in the left-side diagram), and easier for defining selection rules (since fewer elements need to be selected). We removed the TCGCS style from our proposal because this difference could have different implications in terms of i* semantics; we show our concern by way of the example shown in Figure 8.3-10. Breaking a dependum (e.g., **Reduce [Spam]**) directly suggests that this dependum will not stand, so the corresponding dependee's (e.g., **PCMSP**) internal rationale might be affected. This conforms to the label propagation algorithm employed by TCGCS, which propagates labels from a dependum along both directions of the dependency links, towards internal elements, to both its depender (e.g., **PCU**) and dependee (e.g., **PCMSP**). By breaking a dependum's outgoing dependency link, we restricted the break effect to only the depender (e.g., **PCU**). This style

of label propagation algorithm is employed in the LAS case study and other previous literatures (Yu and Liu 2000; Liu et al. 2003). However, this issue lays in the i* semantic itself, and its description is not an intent of this thesis. Furthermore, once a consistent semantic and graphical representation is selected, we can adjust the definition of the single actor focus selection rule (singleActorFocus[SD|SA]Rule) to make our view extension compatible.

We could have converted the style to our proposed one; however, we did not modify it, for the difference does not affect our reduction of views. For simplicity, we assume the different graphical notions are semantically equivalent.

The difference between TCGCS and the reformulated i* framework in presenting the external break contribution shown in Figure 8.3-10 incurs other differences in graphical representation. One is the extra actor **PC Software Manufacturer/ Service Provider** (**PCMSP**) shown in part (b) of the above diagram. For emphasis, we highlighted with a dashed rectangle, but there is no semantic meaning behind this graphical notation. Another is the naming of the dependums. Since we do not show the dependees the dependums depend on in part (a) of the above diagram, we use the full name of each dependum to indicate its depender and dependee in the form "(depender, dependee)". For example, **Reduce [spam] (PCUser, PCMSP)** denotes that **PC User** depends on **PC Software Manufacturer/ Service Provider** (**PCMSP**) to reduce spam.

Despite the differences existing in the SD diagrams, we consider that our approach can present what was modeled in the SD diagrams from TCGCS. The major contribution is that we offered overview information, rules to reduce complex SD views, and guidelines to present related SD views in a systematic manner.

8.4 Strategic Rationale Views

Given the complexity of the Basic SR view from TCG, we cannot conveniently show it in one diagram. The Basic SR view can be reduced to a set of Single-Actor-Focus views, one for each actor, following a similar single actor

focus rule as described in the SD view. These sets of views can be further reduced in a similar manner following the set of partial SR view selection rules. Therefore, we can use the Single-Actor-Focus view of one actor to validate the effectiveness of the SR part of our view extension.

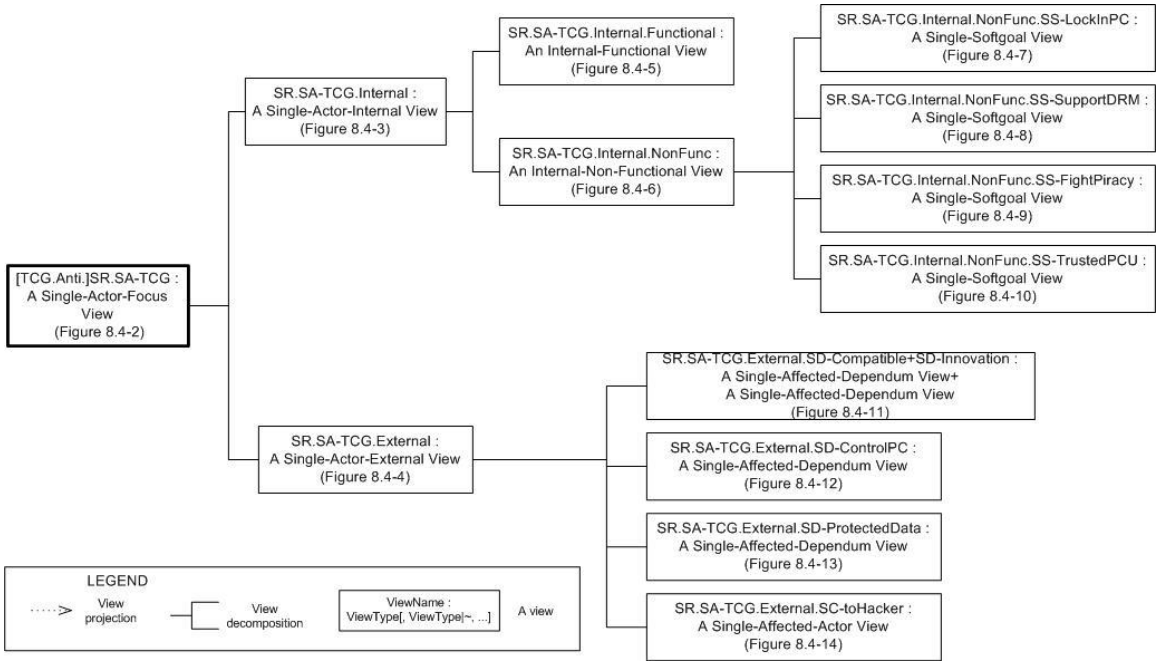


Figure 8.4-1 View map for partial SR views from the Anti-TCG viewpoint

In this section, we choose the Single-Actor-Focus view for agent TCG (the extreme complex case) from the opponents' viewpoint (**TCG.Anti.SR.SA-TCG**) as our original view. This original view was scaled down into a set of related partial SR views; their relationships are shown in Figure 8.4-1. TCG.Anti.SR.SA-TCG (abbreviated as **SR.SA-TCG**) is first decomposed into a Single-Actor-Internal view (**SR.SA-TCG.Internal**) and a Single-Actor-External view (**SR.SA-TCG.External**). The Internal view is further decomposed into an Internal-Functional view (**SR.SA-TCG.Internal.Functional**) and an Internal-Non-functional view (**SR.SA-TCG.Internal.NonFunc**). The Non-functional view is further decomposed into four Single-Softgoal views (e.g., **SR.SA-TCG.Internal.NonFunc.SS-LockinPC**). The External view can be further decomposed into four Single-Affected-Dependum views (e.g., **SR.SA-TCG.External.SAD-ControlPC**) and one Single-Affected-Actor view (**SR.SA-**

TCG.External.SAA-toHacker). Two Single-Affected-Dependum views, **SR.SA-TCG.External.SAD-Compatible** and **SR.SA-TCG.External.SAD-Innovation**, are shown in one diagram, and we name the combined one **TCG.External.SAD-Compatible+SAD-Innovation**.

8.4.1 The Single-Actor-Focus SR View for agent TCG

Figure 8.4-2 shows the Single-Actor-Focus SR view for agent **TCG**, derived by applying the single actor focus rule (singleActorFocusSRRule) over TCG.Anti.SR. It corresponds to the SR model for agent TCG (model 3.2.5) in TCGCS, and we name it **TCG.Anti.SR.SA-TCG**. This view is used as our original view from which other partial views presented throughout this section will be derived.

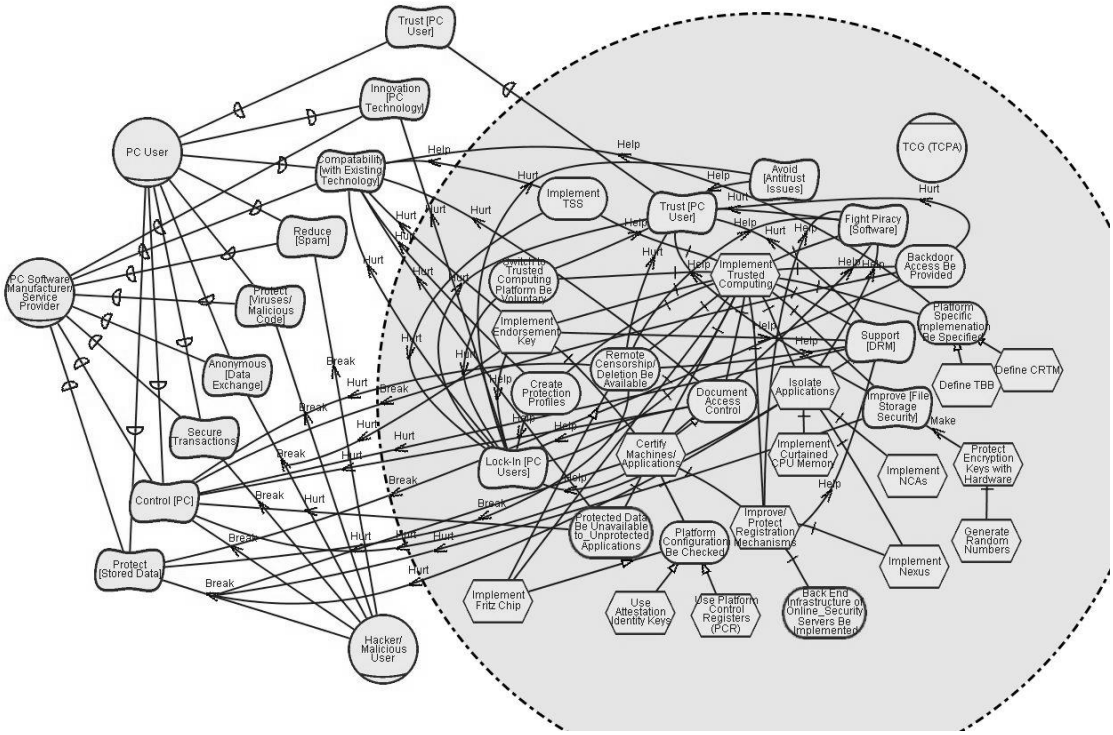


Figure 8.4-2 Single-Actor-Focus SR view from the Anti-TCG viewpoint of TCGCS

8.4.2 Single-Actor-Internal and External views

Figure 8.4-3 shows the Single-Actor-Internal view for agent **TCG**, derived by applying the internal rule (singleActorInternalRule) over TCG.Anti.SR.SA-TCG. It corresponds to the same diagram (model 3.2.5) in TCGCS, and we name it **TCG.Anti.SR.SA-TCG.Internal**.

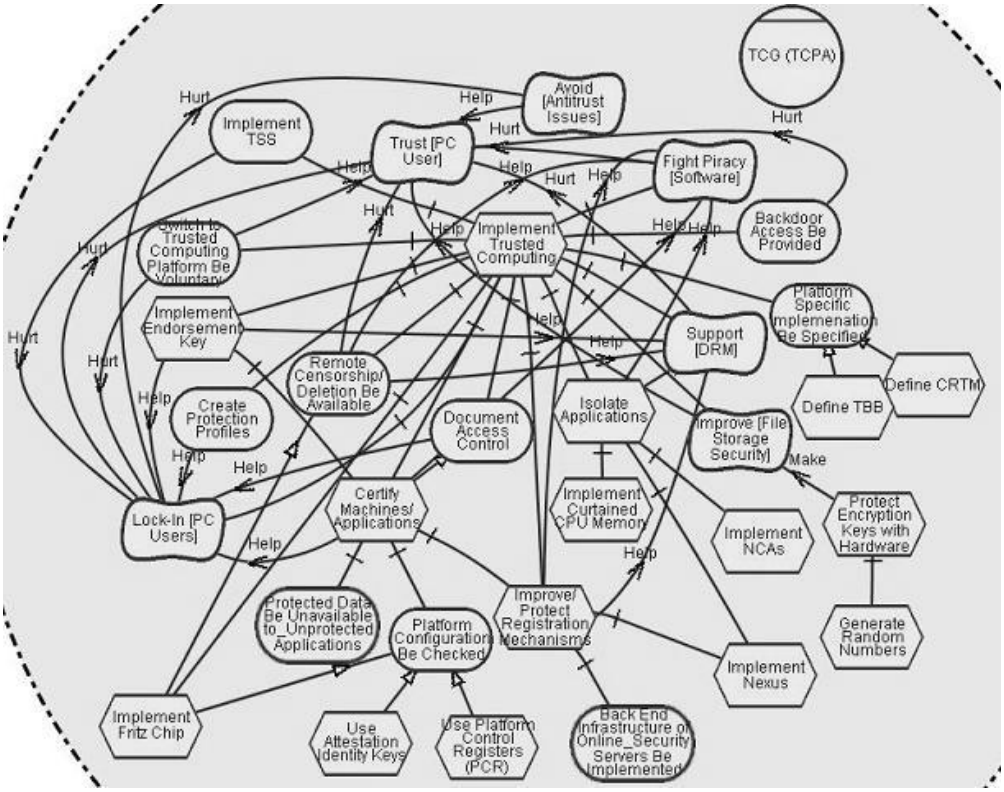


Figure 8.4-3 Single-Actor-Internal view for agent TCG

Figure 8.4-4 shows the Single-Actor-External SR view for agent TCG for agent **TCG**, derived by applying the external rule (singleActorExternalRule) over TCG.Anti.SR.SA-TCG. This view does not appear in TCGCS. We introduced it as an intermediate model summarizing all external relationships of TCG. We name this view **TCG.Anti.SR.SA-TCG.External.**

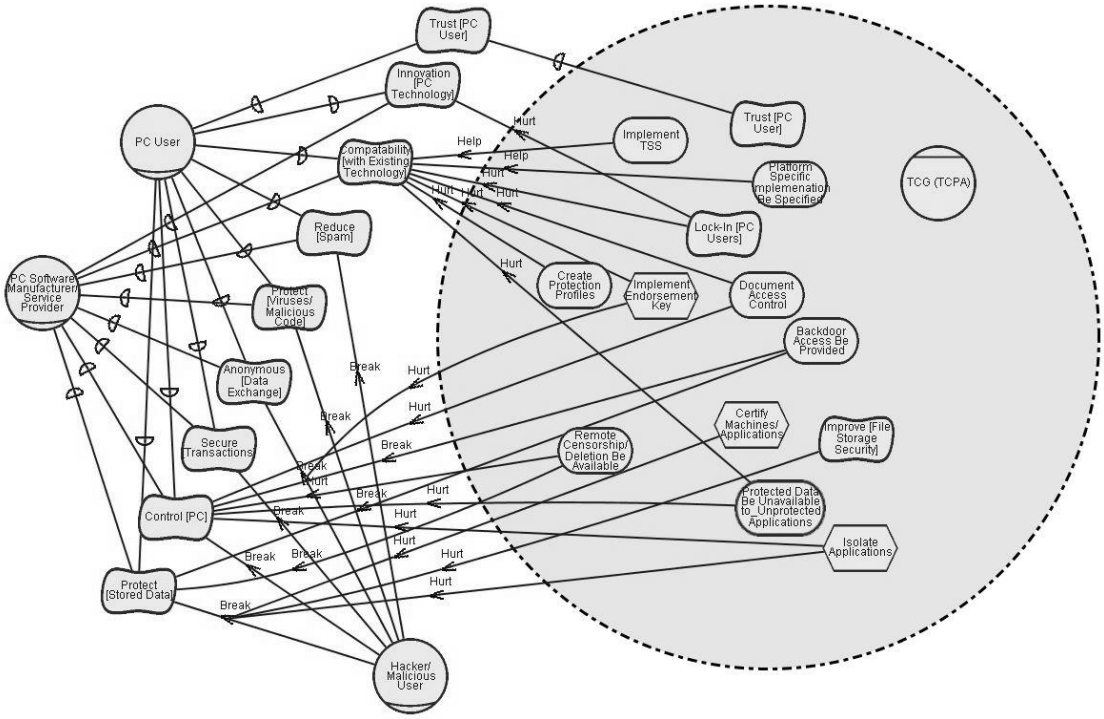


Figure 8.4-4 Single-Actor-External view for agent TCG

8.4.3 Internal-Functional and Non-functional views

Figure 8.4-5 shows the Internal-Functional view for agent **TCG**, derived by applying the functional rule (internalFunctionalRule) over TCG.Anti.SR.SA-TCG.Internal. There is a correspondence to this view in TCGCS (model 3.1.1), and we name it **TCG.Anti.SR.SA-TCG.Internal.Functional**.

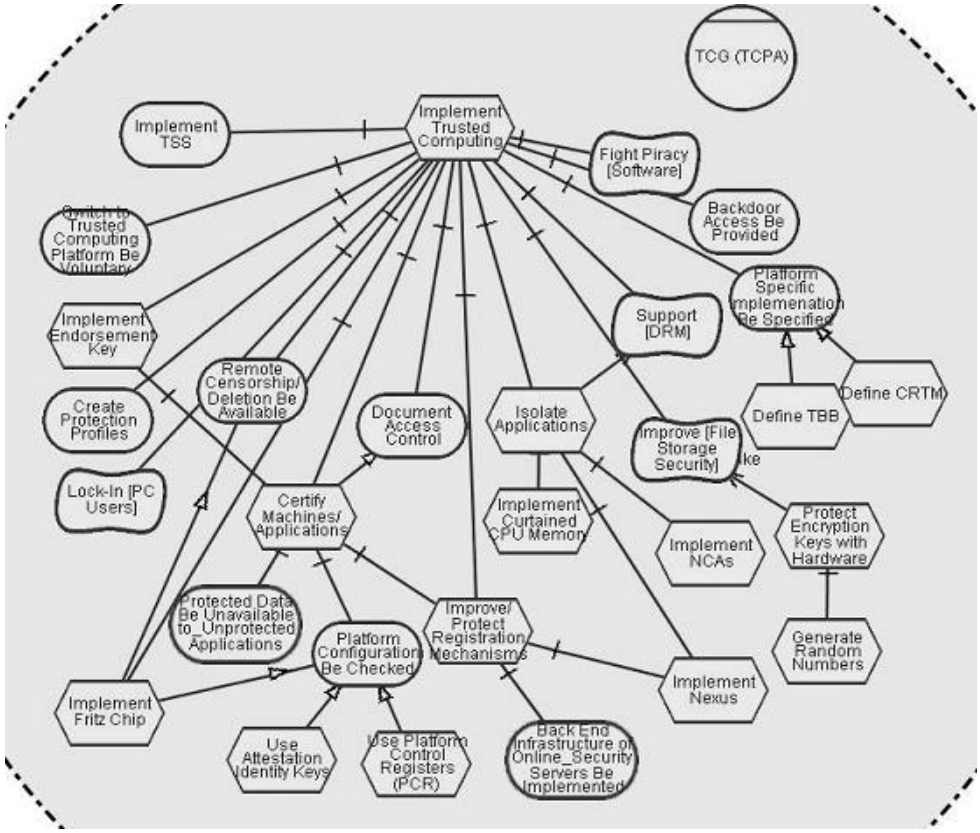


Figure 8.4-5 Single-Actor-Internal-Functional view for agent TCG

Figure 8.4-6 shows the Single-Actor-Internal Non-Functional view for agent **TCG**, derived by applying the non-functional rule (internalNonfunctionalRule) over TCG.Anti.SR.SA-TCG.Internal. There is a correspondence to this view in TCGCS (model 3.1.6), and we name it **TCG.Anti.SR.SA-TCG.Internal.NonFunc**.

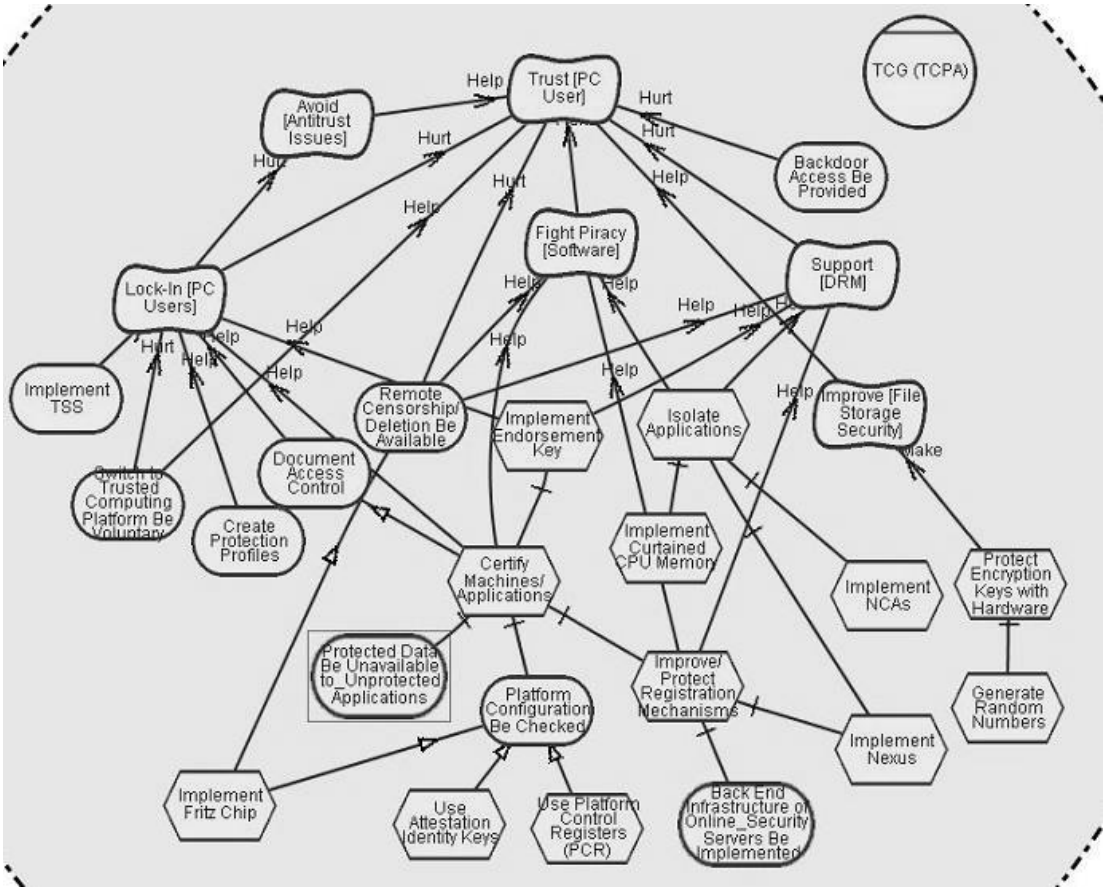


Figure 8.4-6 Single-Actor-Internal Non-functional view for agent TCG

8.4.4 Single-Softgoal views

Figure 8.4-7 shows the Single-Actor-Internal Single-Softgoal view of softgoal **Lock-in PC Users** internal to agent **TCG**, derived by applying the single softgoal rule (nonfunctionalSingleSoftgoalRule) over TCG.Anti.SR.SA-TCG.Internal.NonFunc. There is a correspondence to this view in TCGCS (model 3.1.2), and we name it **TCG.Anti.SR.SA-TCG.Internal.SS-LockinPCU**.

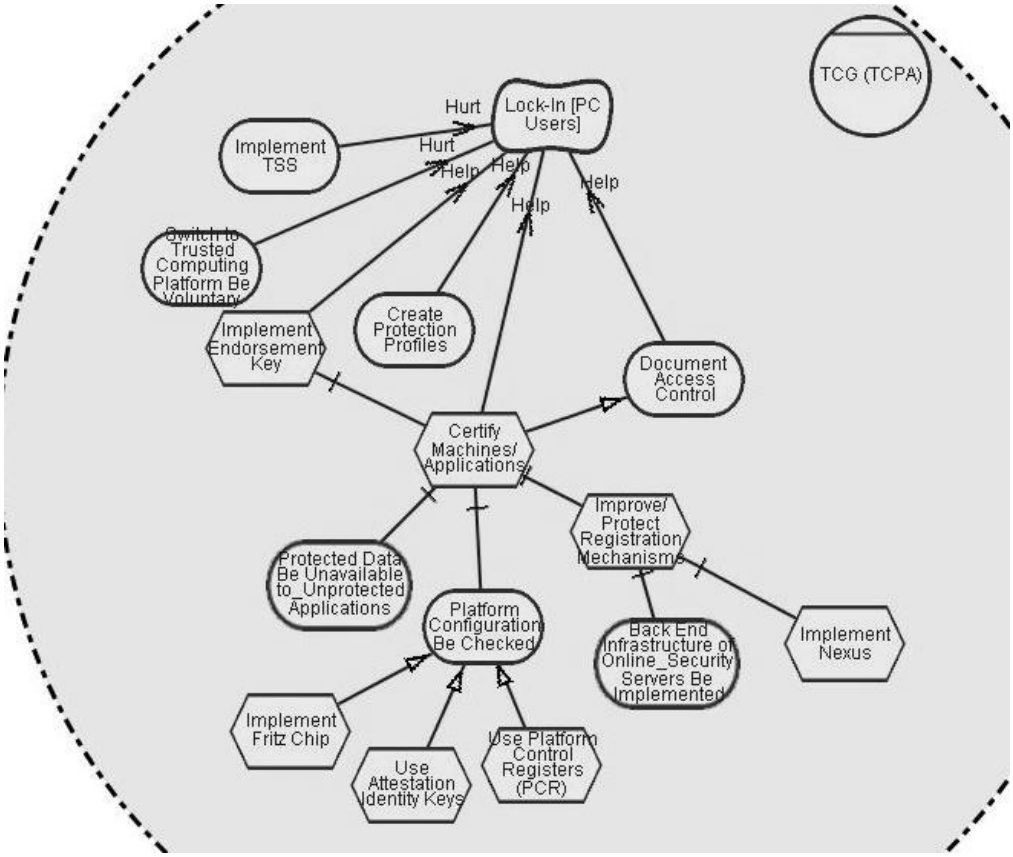


Figure 8.4-7 Internal Single-Softgoal view for softgoal Lock-in PC Users

Similarly, Figure 8.4-8 to Figure 8.4-10 show the Single-Actor-Internal Single-Softgoal view of softgoal **Support [DRM]**, **Fight Piracy [Software]**, and **Trusted [PC User]**, and there correspondences to these views in TCGCS (models 3.1.5, 3.1.4, and 3.1.3, respectively). We name them as **TCG.Anti.SR.SA-TCG.Internal.SS-SupportDRM**, **TCG.Anti.SR.SA-TCG.Internal.SS-FightPiracy**, and **TCG.Anti.SR.SA-TCG.Internal.SS-TrustedPCU**, respectively.

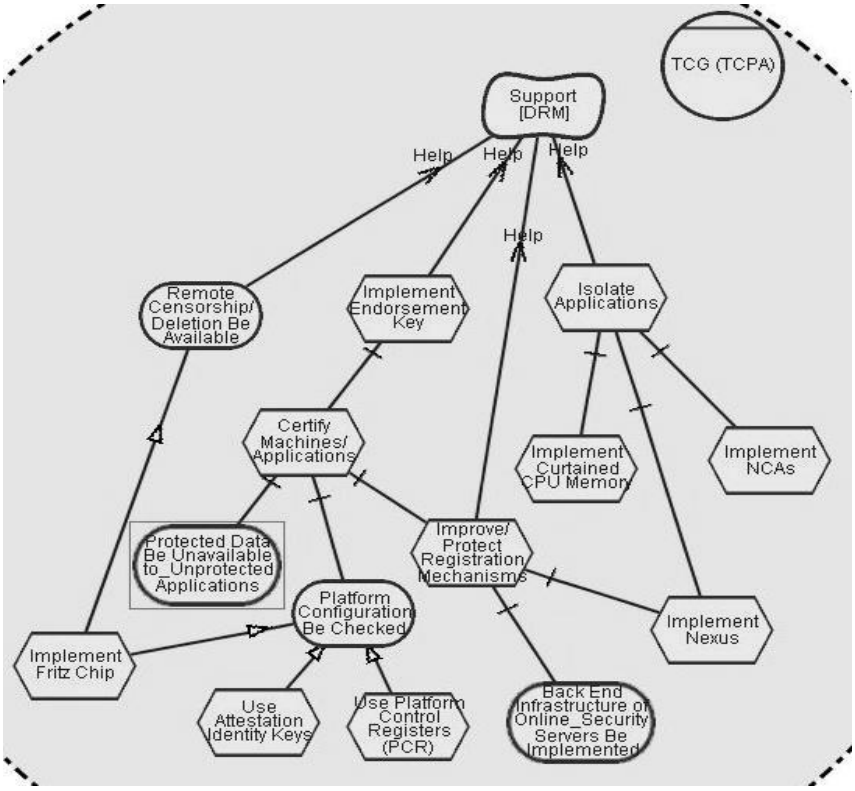


Figure 8.4-8 Internal Single-Softgoal view for softgoal Support [DRM]

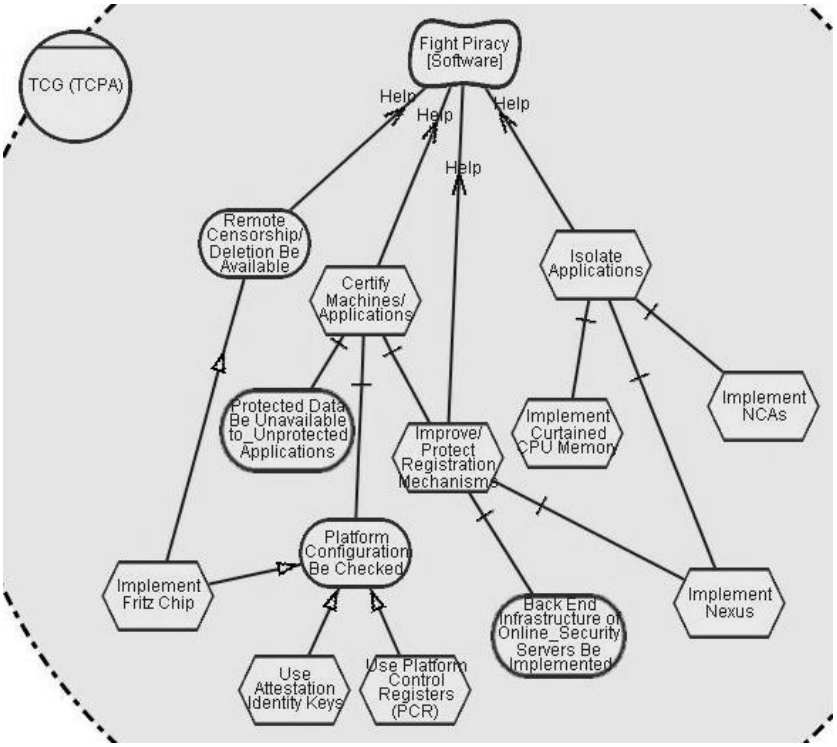


Figure 8.4-9 Internal Single-Softgoal view for softgoal Fight Piracy [Software]

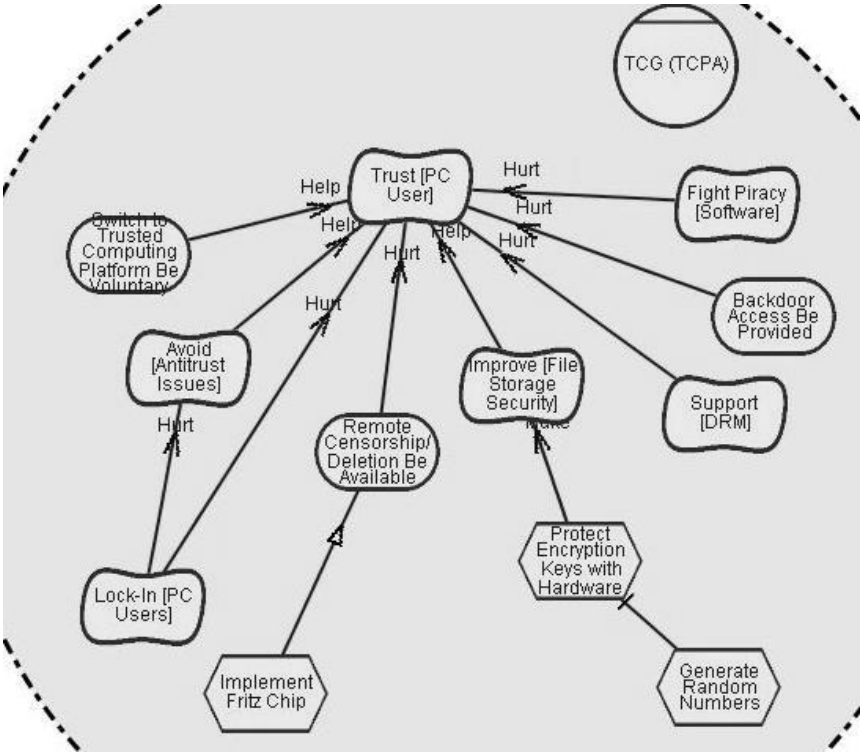


Figure 8.4-10 Internal Single-Softgoal view for softgoal Trusted [PC User]

8.4.5 Single-Affected-Dependum or Actor views

Single-Affected-Dependum views presented in this section are all derived by applying the single affected dependum rule (singleAffectedDependumRule) over TCG.Anti.SR.SA-TCG.External.

Figure 8.4-11 shows the Single-Affected-Dependum views for softgoal dependum **Compatibility [with existing Technology]** and **Innovation [PC Technology]**. There is a correspondence to this view in TCGCS (model 3.2.2). We name it **TCG.Anti.SR.SA-TCG.External.SAD-Compatible+SAD-Innovation**. TCGCS shows these two external dependums in one diagram since their relationship to each other is simple and it would be a waste of space to use two diagrams. However, this is human decision; the step we recommend in applying our view extension is for users to obtain single dependum views first and then combine into a multiple-dependum view the ones they consider related.

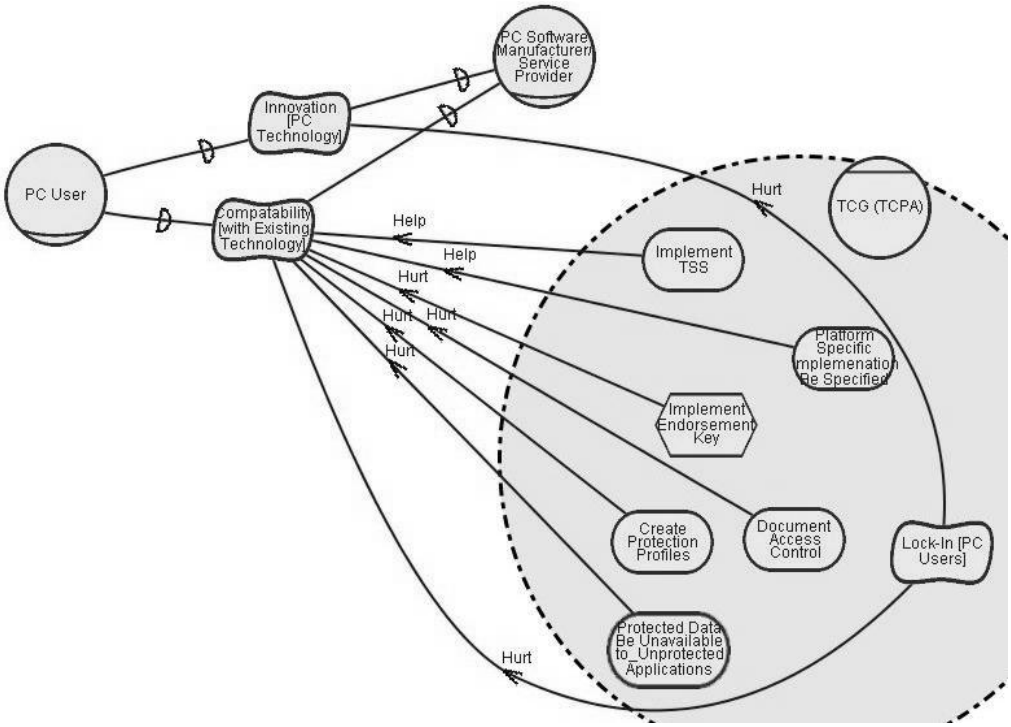


Figure 8.4-11 External Affected Multiple-Dependums view for dependums Compatibility and Innovation

We omitted the internal rationale for TCG in Figure 8.4-11—because we are concerned only with the external effects of TCG. We consider it sufficient to show only the elements that contribute to external objects in answering questions such as "How would the application of the Trusted Computing Group affect the control of PC to each PC user?" (See Section 7.2.2 for detailed justifications).

Figure 8.4-12 and Figure 8.4-13 show the Single-Affected-Dependum views for softgoal dependums **Control [PC]** (model 3.2.3) and **Protect [Stored Data]** (model 3.2.4), respectively. We name the former **TCG.Anti.SR.SA-TCG.External.SAD-ControlPC** and the latter **TCG.Anti.SR.SA-TCG.External.SAD-ProtectSD**.

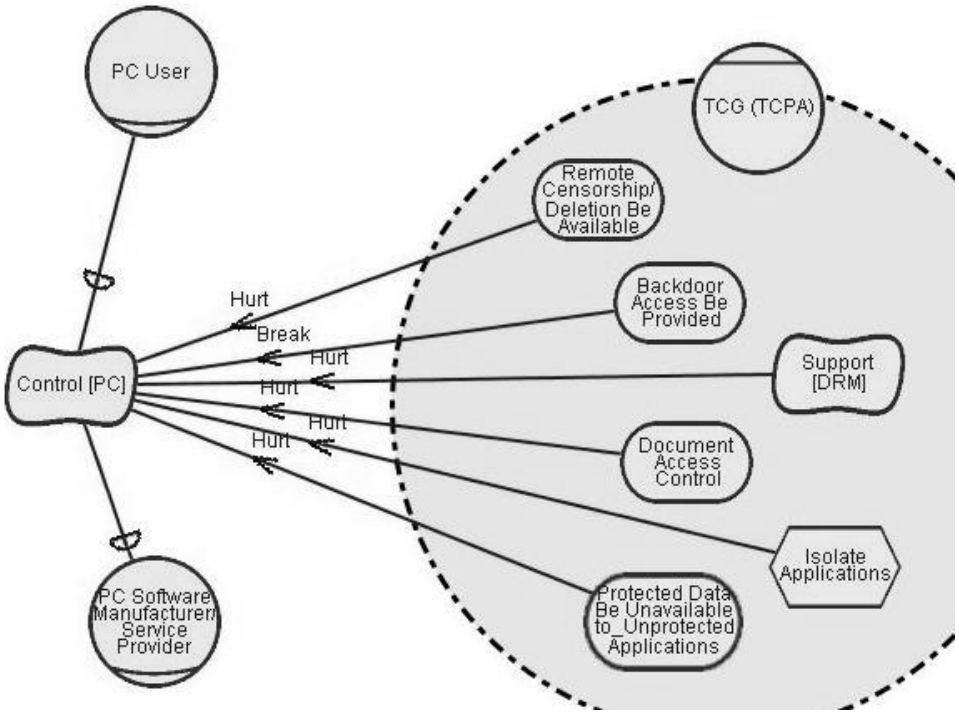


Figure 8.4-12 External Single-Affected-Dependum view for Control [PC]

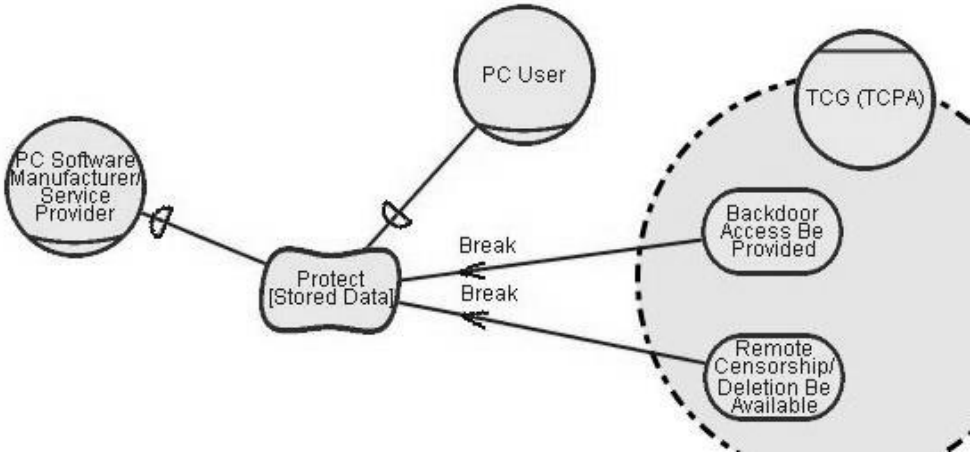


Figure 8.4-13 External Single-Affected-Dependum view for Protect [Stored Data]

Figure 8.4-14 shows the Single-Affected-Actor view to role **Hacker/Malicious User**, derived by applying the single affected actor rule (singleAffectedActorRule) over TCG.Anti.SR.SA-TCG. There is a correspondence to this view in TCGCS (model 3.2.1), and we name it **TCG.Anti.SR.SA-TCG.External.SAA-toHackerMU**.

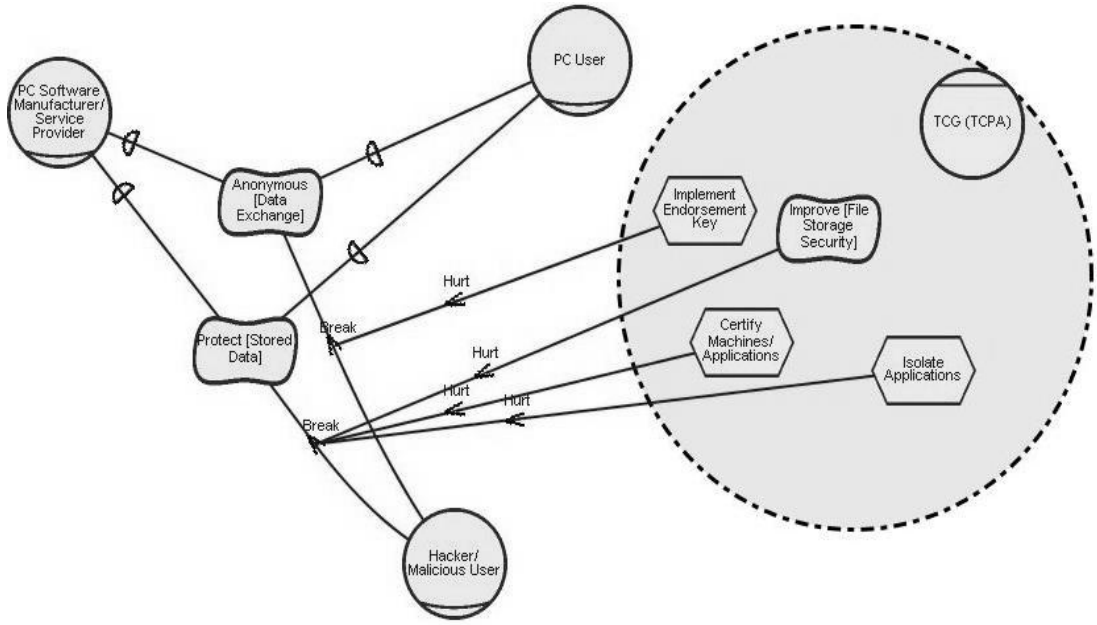


Figure 8.4-14 External Single-Affected-Actor view to Hacker/Malicious User

8.4.6 Discussion

In this section, we demonstrate the process and results of dividing the Single-Actor-Focus SR view for TCG (TCG.Anti.SR.SA-TCG). A hierarchy of sub-views of TCG.Anti.SR.SA-TCG was derived following the guidelines in our view extension, and their relationships were presented in a View Map.

The intuitive approach taken in TCGCS to break down the complex SR views conforms to what we propose in this thesis. Therefore, there exists a one-to-one mapping between the set of partial views and the original “models” in TCGCS.

Our work has enhanced the current state of the art by, first, producing a view map showing the layout and connection among the views. Another improvement is the reduced complexity in each view, a reduction attributed to the formally defined selection rules associated with each type of view. Unnecessary elements are removed in the views—especially the external ones. Compared with their corresponding original models shown in TCGCS, the new views appear concise and more comprehensible.

However, during real applications, some of the views could appear over-simplified, and displaying them separately would be a waste of space. In this sense, views can be combined as long as they remain comprehensible. For example, the two Single-Affected-Dependum views **SR.SA-TCG.External.SAD-Compatible** and **SR.SA-TCG.External.SAD-Innovation** are shown in one diagram (Figure 8.2-12). This action is subject to human decision. We recommend users apply our view extension to obtain single dependum views first, and then combine in a single diagram the ones they consider closely related.

The perfect matching between the SR views presented in this section and the original ones from TCGCS demonstrate the ability of our approach in conveying the same amount of information to i* model users. Our major contribution is that we offer overview information and clear-cut rules.

8.5 Contributions and Results

We tested the validity of our proposed view extension against TCGCS, and we outline our result in this chapter. This process resulted in a total of 37 diagrams, showing the baseline model, 15 AC views, 8 SD views, and 13 SR views. Among these views, only 2 remain exactly the same as what was demonstrated in TCGCS, 17 of which are newly added ones, the other 18 being modified. In addition, 4 View Maps for showing the relationship for basic views, AC views, SD views, and SR views were also supplied to make attainable the relationship among views from the same group.

Our approach is NOT to redo the case study. Therefore, the name or type of any modeled elements remains intact from their original forms. Even the greatly enhanced AC views only experienced changes in some association links and the addition of some extra (or intermediate) actor elements. No actor that existed in the original models was removed from our views.

However, our approach is to reorganize the diagrams designed for representing the same model in a systematic manner. Consequently, the sequence in which we present the views in this chapter differs from that in TCGCS; this is

because the two approaches emphasize on different processes: TCGCS focuses on the model generation, but our approach targets on model representation. Accordingly, the organization of views also differs between the two approaches: We organize views according to their types (meta-concept driven approach), while TCGCS organizes according to the actor pairs each view presented (application-domain knowledge driven approach).

Our approach offers a method to enhance the consistency, clarity, and accessibility of the two models in TCGCS. These benefits are achieved by applying the concepts streamlined in the reformulated i* framework, by the reference structure offered by the view extension, and by the formal definition of selection rules associated with each type of view.

The reformulated i* framework enforces the bonds among the basic views. From the discussion of applying the AC views (Section 8.2.6), we learnt that the enforced bond between the AC and SD views helped identify inconsistencies and correct logical gaps out of the original model. Therefore, the reformulated i* framework helps increase the consistency of i* models.

The reformulated i* framework also formulates an *external relationship inheritance rule* over actor associations. This rule can help remove duplicated dependency links in the SD view. For example, Figure 8.5-1 shows the original SD level baseline model summarized from diagrams in TCGCS. In our revised version (Figure 8.1-2), the redundant external relationships surrounding role **PC Software Manufacturer/Service Provider TCG Member as TCG** (**PCSMSPTCGMTCG**) is removed. Since following the "ISA" link to role **PC Software Manufacturer/Service Provider,** we know that **PCSMSPTCGMTCG** can inherit all external relationship from **PCSMSP**. Thus, we can safely remove all 12 incoming dependency links towards the former actor without losing any modeled information. With less intertwined links in our revised presentation, clarity of relationships among modeled elements increases.

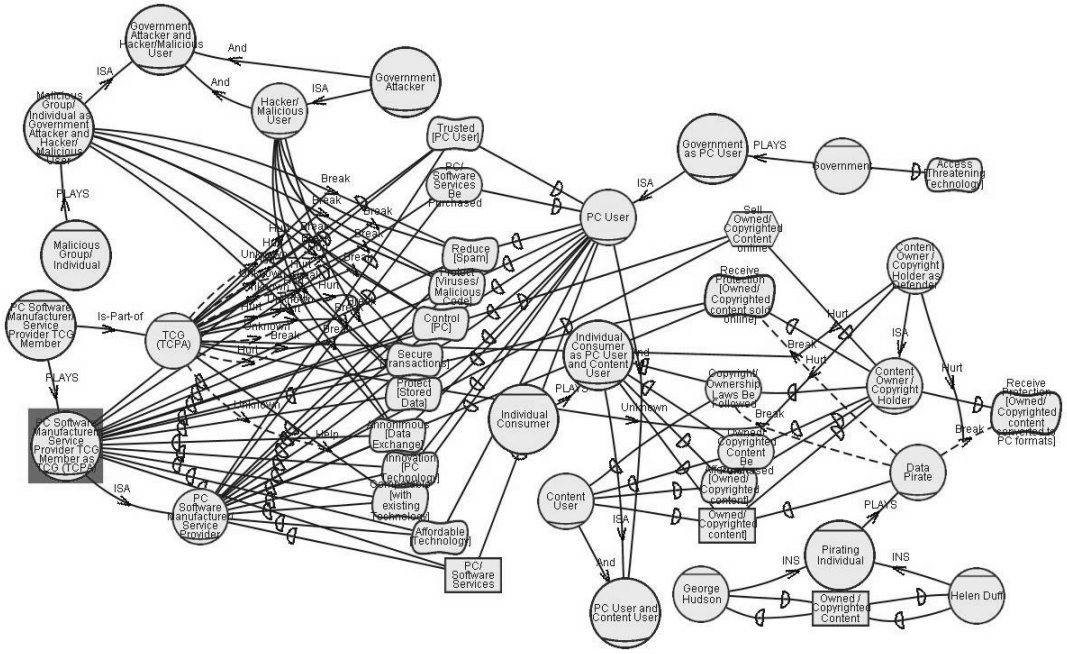


Figure 8.5-1 The original model from the Pro TCG viewpoint

The introduction of the reference structure helps organize views (or diagrams) derived from the same baseline model in a systematic manner. The reference structure is realized by 4 view maps during our revision of TCGCS. For each view presented in a view map, users can identify its type, parent view(s), and child view(s) (if applicable). Its corresponding caption (e.g., Figure 8.1-2) in this documentation is also displayed. It is convenient to locate any view and switch to any of its relatives—even inside a huge document. This action appeared time-consuming if one had browsed through the 200-page TCGCS. Apparently, applying the reference structure improves accessibility of views designed to represent the same model, and we presume the efficiency of this structure increases proportional to the size of the model.

The formally defined selection rules associated with each type of view help remove irrelevant elements from a view. By removing unrelated information from some complex views (or diagrams), we make them more concise and comprehensible. A lack of tool support for automated view synchronization

would also increases the maintainability of an i* model. We have demonstrated that views derived following the proposed selection rules can serve the same objectives as those in TCGCS. With reduced complexity in each view, information that is to communicate with model users becomes obvious. If every view from a model appears concise, then the clarity of the entire model certainly increases.

Applying the view extension to revise TCGCS made the presentation of models in TCG case study clear, consistent, and accessible.

9 Conclusions

9.1 Summary of Results

The main result of this research is a view extension compatible with the original i* framework presented by Yu (Yu 1994). The extension offers a set of guidelines and rules on decomposing or segmenting a large-scale i* model to multiple views. Each view has a type, and the view type decides the type of i* elements that view should allow. Information contained in each view, when visualized, should be readily comprehensible to humans using the model. The extension also provides a reference structure so that the views are organized in a systematic manner and are easy to access. The reference structure is visualized using View Map, a built-in type of diagram supported by the view extension. Notations used in the View Map are also formalized—graphically—in the view extension.

A secondary result of this research is the reformulating of the i* framework. The reformulated framework distinguishes and formalizes a notion of *view*, categorizes meta-level i* concepts into four basic views, and enforces the implicit bonds among the meta-concepts in the basic views. The four types of basic views are the Actor Class (AC) view, the Strategic Dependency (SD) view, the Strategic Rationale (SR) view, and the Evaluation Results (EVLR) view.

Representation constructs of meta-level concepts from the original and the reformulated i* framework are embedded in Telos (Koubarakis et al. 1989). Telos is the conceptual modeling language chosen by Yu to embed the original i* framework (Yu 1994). However, the formal constructs shown in Yu's original thesis and the Organization Modelling Environment (OME) tool differ in style, and we base our formal constructs on those that are used in the OME tool. Concepts introduced in the view extension such as model, view (basic and

partial), and selection rule are also embedded in Telos. These concepts are embedded in Telos following the same style as concepts in i*. For example, the concept model is represented by a meta-level *model class*, and each type of view is represented by a meta-level *view class*. An i* model or a physical view is represented as an instance of the corresponding model class or view class, respectively.

While basic view types are defined in the reformulated i* framework, partial view types are defined in the view extension. Partial view types further differentiate each basic view type, resulting in four groups of Telos view classes. In this thesis we discuss three of them in detail—AC, SD and SR, each in a separate chapter. In these detailed discussions, each type of view is illustrated in terms of

- An informal description of what type of meta-level object should be included in the specific view type.
- A simplified example of the use of the type of view in the London Ambulance Service (LAS) case study.
- Justifications of the applicability of the partial view type and the consequences of using it.
- A formal definition of the selection rule that is attached to the corresponding Telos view class of the given view type. The selection rule is presented in the form of First Order Logic (FOL) using meta-level classes embedded in Telos.

The validity of the view extension was examined against the Trusted Computing Group case study which was originally documented by Horkoff (Horkoff 2004). Comparisons with the diagrams (called models by Horkoff) presented in (Horkoff 2004) were made for each of the three types (AC, SD, SR) of views. The view extension demonstrated a more organized approach in presenting the set of diagrams designed for the same i* model. A diagram is the visualized form of a view. Three View Maps for AC, SD and SR views,

respectively were also supplied to make attainable the relationship among views from the same group.

9.2 Contributions

This work offers a systematic approach to presenting large scale i* models. The foundation of this approach lies in the notion of view and the meta-level concepts of the i* framework. By defining views, this approach splits a baseline i* model into a set of self-containing views that can address some specific application domain-related questions.

This work advances i* into a more practical and ready-to-use stage:

- It streamlines into a unified style graphical i* notations scattered through previous literature and appearing sometimes in different forms.
- It enforces the bonds among the basic views. Each SD view is considered an abstraction of its corresponding SR view. Each EVLR view has an SR view on which it is based. Actor associations expressed in the AC view can be used to facilitate the replacement of actors in SD views.
- It enhances communication by breaking down the complexity and size of the baseline model and converting it into readable-size views.
- It embeds both meta-concepts of i* and meta-concepts in the view extension into Telos, the selected conceptual modeling language in (Yu 1994). This formalization makes it possible to automate the selection rules defined for each view in any commercial tool. Moreover, the formalization ensures consistency in applying our proposed approach across different applications.
- It reformulates the formal representation of meta-concepts of the i* framework into the Organization Modeling Tool (OME) style, resulting in filling the gap between the theoretical i* model and its actual implementation.
- It transfers ideas from database systems to the knowledge-base-oriented i* framework—treating the modeling concepts as meta-model (schema), a set of

modeled application-domain knowledge as the baseline model (data table), and the projection of the modeled knowledge as a view (data view).

- It borrows from IDEF0 (IDEF 1993) the technique of presenting a reference structure of diagrams (views) designed for one model. Each diagram is treated as a node in a node tree (a visualization of the reference structure) in IDEF0. Similarly, we denote each view as a node in a connected graph which we call View Map.

Furthermore, this work provides an alternative way to communicate the information from the original TCG case study (Horkoff 2004). For the AC views, missing bonds among actors were added following the implications of actor associations reinforced in the reformulated i* framework. For the SD views, Single-Actor-Focus views were emphasized so as to allow an overview of the situation of an actor within TCG. For the SR views, most of the diagrams were simplified by eliminating half of the elements, yet they retain the ability to address the same issue as its corresponding diagram shown in the original TCG case study (Horkoff 2004).

Overall, this work offers a better means to represent an existing i* model. With a formally reformulated i* framework and the view extension, large-scale i* models can be displayed in an organized manner. Relationships among different parts of a large model can be rendered easy to observe, helping i* users to perform model checking. The handful of guidelines and live examples offered in this work, along with the definition of the view types, make the i* framework ready to put in practice. Therefore, even though the work does not address all scalability issues, we consider it has prepared and readied i* for quite a broad range of applications.

9.3 Future Directions

This work represents an important first step forward in addressing the scalability issues in the i* framework. Further research at the forefront of knowledge in this area is required to provide i* users a complete package of

rules and guidelines to handle large-scale applications. Other meta-concepts or domain-based patterns are available to help design new types of views. The guidelines for constructing an i* model—not just representing it—in a systematic manner are yet to be synthesized. This work is subject to validation in broader applications.

9.3.1 Meta-model related future work

Other meta-concepts from the i* framework can be employed in designing new selection rules. Associating these rules with view class can define new view types, and thus extend the view extension as follows:.

- The concept of *routine* "is a sub-graph in the visualized SR view with a single link to a 'means' node from each 'end' node" (Yu 1994). In other words, a routine refers to a particular alternative to achieve some goal that is considered a *decision point*. A decision point is a goal that has multiple means-ends linked to it, originating from different tasks (see Section 3.2.3 for more details). A new view type that presents a single routine can be designed.

- Yu (Yu 1994) provides for "three degrees of *dependency strength*: open (uncommitted), committed, and critical." New view types could be designed so that only dependencies at a certain degree are to be presented.

- The *direction* of a dependency link can also be exploited to derived views including only incoming or outgoing dependencies.

Moreover, in this document we have not discussed in detail the Evaluation Results (EVLR) view and naming conventions; these issues require follow-up investigation to complete this work.

9.3.2 Use generic knowledge-base driven techniques

Given the rich set of meta-concepts defined in i*, meta-concept-based scalability controls already result in considerable scale-downs. In other words,

by partitioning elements in the model according to their *types* alone, we can reduce the size of the basic views proportionally.

However, domain knowledge may contribute to generic guidelines from another dimension.

Applications from similar application domains may possess similar characteristics that can be generalized and reused. For example, security-related applications tend to categorize actors by normal actors, attackers, and defenders (Yu and Liu 2000; Liu et al. 2003) TCGCS. In reliability-critical applications, actors can be categorized into normal actors, abusers, and mitigators (Alexander 2003; You 2003). These patterns might be used to design new types of views (e.g., a view presents only normal actors).

Organizations may demonstrate similar organization structures, which follow a "headquarter—division—sub-division—sub-sub-division…" hierarchy. Actors can be partitioned according to their division or sub-division (e.g., a view presents only actors from the same division). An intermediate abstraction level actor, such as "a division," may also be introduced to the extension to allow a view to show relationships among divisions.

This line of future direction is considered important in that the distributed nature of the i* framework is quite appropriate for modeling open-ended applications which are richer in domain knowledge. Actors may be categorized into several groups. However, criteria in organizing an object in an i* model according to this line of reasoning require further investigation.

9.3.3 Guidelines for the modeling process

Guidelines in addressing scalability issues during the modeling process are most critical. When an application reaches a certain size, the resulting work should be distributed among different modelers; the model should be constructed over a period, and be refined continually as domain knowledge is accumulated during the modeling process. Without general guidelines in breaking down the

workload and the methods for maintaining model-wise consistency, either the modelers must spend extra time defining application-specific rules, or the integrity and correctness of the targeted model will be jeopardized.

However, the forward engineering (modeling) process of i* requires intensive human interaction and decision; this is because the modeling process embeds deeply into each specific application domain, and significant features vary drastically from one application to another. For example, the LAS project, as a close-end application, is required to analyze what mistakes each participant makes during a normal operation; on the other hand, TCGCS, as an open-ended environment, is required to analyze what impacts TCG should be dealt with from a third-party stakeholder. It is thus more difficult to generalize the rule in the modeling process.

As a result of the foregoing, even though this work has demonstrated the strength of the view extension in presenting large-scale i* models, to what extent it can help the modeling process remains unclear. Nonetheless, we believe that the manner in which we present the view can help modelers plan their procedures in constructing and analyzing the models. Further in-depth study is required to provide direct and useful guidelines on this issue.

9.3.4 Broader applications

Over the past 10 years, the application area of i* has changed continually. From 1996 to 1997, the i* research group explored intensively Business Process Reengineering, and conducted organization impacts analyses—mostly by studying the graphical models (which we call views) along various links. From 1997 to 1999, the strength of i* in Requirements Engineering (RE) and System Architecture were presented from various perspectives. From 2000 until now, focus has shifted onto internet-related non-functional requirements, including trust (Yu and Liu 2000), privacy (Yu and Cysneiros 2002; Liu et al. 2003), security (Liu et al. 2002; Liu et al. 2003), and protection of Intellectual Property

(Yu et al. 2001). The utility of i* shifted from a more internal process reengineering to an open-ended distributed agent-oriented approach.

The view extension is validated against one medium-size application, but more applications may be used to further validate the concept. Due to the richness of the i* concepts and the uncertainty in open-ended agent-oriented application areas, we anticipate variations in i* utility. As a result, we believe that the current defined views are likely insufficient to present an i* model from other discipline.

To explore and implement the full potential of this research, a broader scope of applications than now available is recommended to validate this work. A clear advantage is that the design of the view extension is extensible, and new types of views can be added to the current one following the Telos syntax as long as a selection rule is provided.

Appendix

A. Transformation of FOL Formula

To verify the correctness of the formula encoded in the First Order Logic (FOL) form across this thesis, we prototyped them using ConceptBase. ConceptBase is a "prototype deductive object base [manager] supporting the Telos data model" (Jarke et al. 1995). O-Telos is a variant of Telos that is implemented in ConceptBase (Jarke et al. 1995). In this section, we illustrate the method to transform a FOL formula to an O-Telos class.

This thesis presents two means in defining concepts introduced in the view extension. The first one is to define new meta-classes by restraining an existing one with a deduction rule. The other method is to define queries that include instances of only a certain type. The two methods can both defining concepts and can appear equivalent, when instantiated, in constructing an i* baseline model. Both means use some FOL expression as the criterion for selecting qualified elements.

Section A.1 discusses the transformation of the definition of meta-classes; Section A.2 presents the transformation of the definition of queries. Section A.3 presents the transformation of the expressions.

A.1 Transform definition of meta-classes

The FOL format for defining a meta-level class takes the following pattern. The class name is bolded. Texts in brackets <> denote variables appeared in the formulae.

<**class_name**>::=<var>:<base_class_name> with "<rule_name>_rule"
<rule_name>_rule::= <expression (FOL style)>

The corresponding O-Telos format is as follows:

```
Individual <class_name>
in Class, MetaClass
```

```
isA <base_class_name>
with
     rule
     <rule_name>rule:
          $ for all <var>/<base_class_nam>
          <expression (O-Telos style)> =>
          (<var> in <class_name>) $
end
```

For example, the definition of external link takes the following format, where the assignment of the variables in the formulae is shown in Table 9-1.

Table 9-1 Variable assignment for defining meta-class "external link"

Variable	Value
<class_name>	ExternalLinkClass
<var>	l
<base_class_name>	IntentionalLinkClass
<rule_name>	external
<expression (FOL style)>	(l in find_all_external_links())

The resulting definition of class ExternalLinkClass is represented as follows:

ExternalLinkClass::=l:IntentionalLinkClass with "external_rule"
external_rule::= (l ∈ find_all_external_links())

The corresponding O-Telos form is as follows:

```
Individual ExternalLinkClass
in Class, MetaClass
isA IntentionalLinkClass
with
    rule
     externalrule: $ forall l/IntentionalLinkClass
       (l in find_all_external_links())
          ==> (l in ExternalLinkClass)
     $
end
```

A.2 Transform queries

To make the view extension mountable to the i* framework, most of the new concepts are defined using query classes. The symbol "§" denotes *for all those* in the FOL pattern. The definition of a query in FOL takes the following format:

<query_name>([<arglist>])::=
§<return_var>:<return_var_type > · <expression (FOL style)>

Where <arglist> is defined as follows:

<arglist>::=<arg>[,<arglist>]
<arg>::=<input_var>:<input_var_type>

Queries without any input variable are mapped to QueryClass, while those with input variables are transformed to GenericQueryClass.

```
Individual <query_name> in QueryClass isA <return_var_type>
with
  attribute, retrieved_attribute
      <attributelist>
  attribute, constraint
       c: $ <expression>$
end

Individual <query_name> in QueryClass isA <return_var_type>
with
  attribute, retrieved_attribute
      <attributelist>
  attribute, parameter
      <arglist_o>
  attribute, constraint
       c: $ <expression (in O-Telos style)>$
end
```

We use <attributelist> to denote the set of attributes that are defined in the <return_var_type>, and <arglist_o> to denote the set of input variables in O-Telos format. Where <arglist_o> and <attributelist> are formally defined as follows:

<attributelist>::= <attribute>[;<CR><LF> <attributelist>]
<attribute>::=<attr_var>:<attr_var_type>

<arglist_o>::=<arg>[;<CR><LF> <arglist_o>]

For example, the definition of query find_internal_connectors takes the following format, where the assignment of the variables in the formulae is shown in Table 9-2.

Table 9-2 Variable assignment for defining query "find_internal_connectors"

Variable	Value
<query_name>	find_internal_connectors
<arg>	a:ActorElementClass
<return_var>	"e"
<return_var_type>	IntentionalElementClass
<expression>	∃ l: LinkClass…∨ (l in ExternalLinkClass)

The resulting definition of query "find_internal_connectors" is represented in FOL as follows:

find_internal_connectors(a:ActorElementClass)::=
§e:IntentionalElementClass·
∃ l: LinkClass· e.parent=a ∧ (l.from=e ∨ l.to=e)
∧ (l in DependencyLinkClass) ∨ (l in ExternalLinkClass)

The corresponding O-Telos GenericQueryClass is as follows:

```
Individual find_internal_connectors
in GenericQueryClass
isA IntentionalElementClass
with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $
     (exists l/LinkClass
     (this parent ~a) and ((l from this) or (l to this))
     and (l in DependencyLinkClass) or
      (l in ExternalLinkClass) )
     $
end
```

A.3 Transform expressions

Each expression serves as either the deductive rule (for a meta-class) or the integrity constraint (for a query class) is translated from the FOL style to O-Telos. Table 9-3 shows the mapping of operators between FOL and O-Telos. See (Jarke et al. 2003; ConceptBase Team 2003) for detail definitions of the O-Telos language.

Table 9-3 Mapping of expressions and logical operators from FOL to O-Telos

FOL	ConceptBase	Remark
<input_var>	~<input_var>	
<return_var>	this	
<var>.<label> =<target>	(<var> <label> <target>)	
∃	exists	
∀	forall	
¬	not	
∧	and	
∨	or	
∈	in	Instance of

In query find_internal_connectors, the FOL style expression is as follows, where *a* is the <input_var> and *e* is the <return_var>.

∃ l: LinkClass· e.parent=a
∧ (l.from=e ∨ l.to=e)
∧ (l in DependencyLinkClass) ∨ (l in ExternalLinkClass)

The corresponding O-Telos translation is:

```
exists l/LinkClass (this parent ~a)
 and ((l from this) or (l to this))
 and (l in DependencyLinkClass) or (l in ExternalLinkClass)
```

B. Queries in O-Telos Style

Each of the queries defined in First Order Logic (FOL) in this thesis is translated in its corresponding O-Telos style, and tested using ConceptBase. To perform this test, we first constructed O-Telos representation of the reformulated i* framework and loaded it into ConceptBase. Then we designed sample domain-level i* models and loaded them into ConceptBase. Last, we ran each query, supplying the required input arguments, and checked the correctness of the results (the set of returning objects).

Hereafter are list of all the queries defined in this thesis. Queries and definitions are numbered according to their sequence of appearance in this thesis. We organize them into four .sml files. “DefinitionQueries.sml” contains definitions and queries defined in Section 4.3; “ACViews_Queries.sml” contains queries defined in Section 5.2; “SDViews_Queries.sml” contains queries defined in Section 6.2; and “SRViews_Queries.sml” contains queries defined in Section 7.2.

```
{
*     File : DefinitionQueries.sml
*     Purpose : Definitions of concepts and related query classes
*     created : 09/01/04 Jane You
*     last change:
*     Content: Def1~3, Query1~14
}

{# Definition of extra model related types #}

{# Def1: DependumElementClass #}
Individual DependumElementClass in Class, MetaClass isA SubElementClass
with
    rule
      dependum_rule: $ forall e/SubElementClass
        not(exists a/ActorElementClass (e parent a)) ==> (e in
DependumElementClass)
      $
end

{# Def2: InternalElementClass #}
Individual InternalElementClass in Class, MetaClass isA
IntentionalElementClass with
    rule
      internal_rule: $ forall e/IntentionalElementClass
```

```
         (exists a/ActorElementClass (e parent a)) ==> (e in
InternalElementClass)
      $
end

{ this definition is not formalized in the thesis }
Individual DecisionPointElementClass in Class, MetaClass isA
GoalElementClass with
    rule
      dpointrule : $ forall e/GoalElementClass
       (exists l1,l2/IntentionalLinkClass (l1<>l2) and (l1 to e) and (l2
to e))
            ==>(e in DecisionPointElementClass) $
end

{# Query1: find_parent(e:IntentionalElementClass) #}
Individual find_parent in GenericQueryClass isA ActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     e : IntentionalElementClass
  attribute,constraint
     c : $ (~e parent this) $
end

{# Query2: find_internal_elements(a:ActorElementClass) #}
Individual find_internal_elements in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $ (~a children this) $
end

{# Query3: find_incoming_dependencies_to_actor(a:ActorElementClass) #}
{# Comments: find dependency links that targets at "a" #}
Individual find_incoming_dependencies_to_actor in GenericQueryClass isA
DependencyLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (this to ~a) or (exists e/IntentionalElementClass (e parent
~a) and (this to e)) $
end

{# Query4: find_outgoing_dependencies_from_actor(a:ActorElementClass) #}
{# Comments: find dependency links that starts from "a" #}
Individual find_outgoing_dependencies_from_actor in GenericQueryClass
isA DependencyLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
```

```
  attribute,constraint
     c : $ (this from ~a) or (exists e/IntentionalElementClass (e parent
~a) and (this from e)) $
end

{# Query5: find_depender_actor(de:DependumElementClass) #}
Individual find_depender_actor in GenericQueryClass isA
ActorElementClass with
  attribute,parameter
     de : DependumElementClass
  attribute,constraint
     c : $ exists l/DependencyLinkClass
            ((exists e/IntentionalElementClass (e parent this) and (l
from e)) or (l from this)) and (l to ~de)$
end

{# Query6: find_depender_element(de:DependumElementClass) #}
Individual find_depender_element in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     de : DependumElementClass
  attribute,constraint
     c : $ exists l/DependencyLinkClass (l from this) and (l to ~de)$
end

{# Query7: find_dependee_actor(de:DependumElementClass) #}
Individual find_dependee_actor in GenericQueryClass isA
ActorElementClass with
  attribute,parameter
     de : DependumElementClass
  attribute,constraint
     c : $ exists l/DependencyLinkClass
            ((exists e/IntentionalElementClass (e parent this) and (l to
e)) or (l to this)) and (l from ~de)$
end

{# Query8: find_dependee_element(de:DependumElementClass) #}
Individual find_dependee_element in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     de : DependumElementClass
  attribute,constraint
     c : $ exists l/DependencyLinkClass (l to this) and (l from ~de)$
end

{# Query9: find_direct_external_link #}
{# Comment: this definition is a walk around due to problems in
implementing recursion #}
{#          we have two auxilary queries suffixed by -1 to help define
this query      #}
Individual find_direct_external_links in GenericQueryClass isA
IntentionalLinkClass with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists a/ActorElementClass dl/DependencyLinkClass
e/IntentionalElementClass
```

```
            (this from e) and (e parent a) and (this to dl) $
end

{# Query10: find_all_external_links(l:LinkClass) #}
{# Comment: this definition is a walk around due to problems in
implementing recursion #}
{#          we have two auxilary queries suffixed by -1 to help define
this query      #}
Individual find_direct_external_links1 in GenericQueryClass isA
IntentionalLinkClass with
  attribute,parameter
     l : LinkClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (this to ~l) $
end

Individual find_all_external_links1 in GenericQueryClass isA
IntentionalLinkClass with
  attribute,parameter
     l : LinkClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (this in find_direct_external_links1[~l/l]) or
            (exists l2/IntentionalLinkClass (this in
find_direct_external_links1[l2/l])
            and (l2 in find_all_external_links1[~l/l]) ) $
end

Individual find_all_external_links in GenericQueryClass isA
IntentionalLinkClass with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (exists dl/DependencyLinkClass (this in
find_all_external_links1[dl/l])) $
end

{# Query11: find_direct_descendants(ie:IntentionalElementClass) #}
Individual find_direct_descendants in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     ie : IntentionalElementClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass a/ActorElementClass
            (l to ~ie) and (l from this) and (~ie parent a) and (this
parent a) $
end

{# Query12: find_all_descendants(ie:IntentionalElementClass) #}
Individual find_all_descendants in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     ie : IntentionalElementClass
  attribute,constraint
```

```
     c : $ (this in find_direct_descendants[~ie/ie]) or
           (exists d/IntentionalElementClass a/ActorElementClass
                  (d parent a) and (this parent a) and
                  (d in find_all_descendants[~ie/ie]) and
                  (this in find_direct_descendants[d/ie]) ) $
end

{# Query13: find_direct_ancestors(ie:IntentionalElementClass) #}
Individual find_direct_ancestors in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     ie : IntentionalElementClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass a/ActorElementClass
              (l from ~ie) and (l to this) and (~ie parent a) and (this
parent a)$
end

{# Query14: find_all_ancestors(ie:IntentionalElementClass) #}
Individual find_all_ancestors in GenericQueryClass isA
IntentionalElementClass with
  attribute,parameter
     ie : IntentionalElementClass
  attribute,constraint
     c : $ (this in find_direct_ancestors[~ie/ie]) or
           (exists d/IntentionalElementClass a/ActorElementClass
                  (d parent a) and (this parent a) and
                  (d in find_all_ancestors[~ie/ie]) and
                  (this in find_direct_ancestors[d/ie]) )   $
end

{# Def3: ExternalLinkClass #}
Individual ExternalLinkClass in Class, MetaClass isA
IntentionalLinkClass with
    rule
      external_rule: $ forall l/IntentionalLinkClass
        (l in find_all_external_links ) ==> (l in ExternalLinkClass)
      $
end

{
*     File : ACViews_Queries.sml
*     Purpose : Define the query classes for the AC views
*     created : 08/04/04 Jane You
*     last change : 09/01/04 Jane You
*     Contents: Query15~26
}

{# Query15: theBasicActorClassView(m:BaselineModelClass) #}
{# Comments: load m into a ConceptBase server before running this query,
m becomes the default view #}
{#          following queries follow the same convention, running over a
default view #}
Individual the_basic_AC_view in QueryClass isA ObjectClass with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
```

```
     c : $ (this in ActorElementClass) or (this in AssociationLinkClass)
$
end

{# Query16: find_all_links(pv:ViewClass, cv:ViewClass) #}
{# Default view: pv #}
{# Input parameters: cv #}
Individual find_all_links in GenericQueryClass isA LinkClass with
  attribute,parameter
     cv : QueryClass
  attribute,constraint
     c : $ exists e1/ElementClass e2/ElementClass
            (e1 in ~cv) and (e2 in ~cv)
            and   (this from e1) and (this to e2) $
end

{# Query17: find_direct_associated_actors(a:SpecifiedActorElementClass)
#}
{# Default view: v from the singleNetworkRule #}
{# Input parameters: a #}
Individual find_direct_associated_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ exists l/AssociationLinkClass
            (l from this) and (l to ~a) or (l from ~a) and (l to this) $
end

{# Query18: find_all_associated_actors(a:SpecifiedActorElementClass) #}
{# Default view: v from the singleNetworkRule #}
{# Input parameters: a #}
Individual find_all_associated_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ (this in find_direct_associated_actors[~a/a]) or
           (exists a2/SpecifiedActorElementClass
                  (a2 in find_all_associated_actors[~a/a]) and
                  (this in find_direct_associated_actors[a2/a]) ) $
end

{# Query19: find_direct_specified_actors(a:PlainActorElementClass) #}
{# Default view: v from the singlePlainActorRule #}
{# Input parameters: a #}
Individual find_direct_specified_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : PlainActorElementClass
  attribute,constraint
```

```
     c : $ exists l/SpecifiesLinkClass
            (l from this) and (l to ~a) $
end

{# Query20: find_direct_replacing_actors(a:SpecifiedActorElementClass) #}
{# Default view: v from the singlePlainActorRule #}
{# Input parameters: a #}
Individual find_direct_replacing_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ exists l/AssociationLinkClass
            ((l in PartsLinkClass) or (l in
CompleteCompositionLinkClass)) and (l from ~a) and (l to this)  or
            ((l in ISALinkClass) or (l in INSLinkClass) or (l in
CoversLinkClass) or (l in PlaysLinkClass) or
             (l in OccupiesLinkClass)) and (l from this) and (l to ~a)
      $
end

{# Query21: find_all_replacing_actors(a:SpecifiedActorElementClass) #}
{# Default view: v from the singlePlainActorRule #}
{# Input parameters: a #}
Individual find_all_replacing_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ (this in find_direct_replacing_actors[~a/a]) or
           (exists a2/SpecifiedActorElementClass
                  (a2 in find_all_replacing_actors[~a/a]) and
                  (this in find_direct_replacing_actors[a2/a]) )  $
end

{# Query22: find_all_abstract_actors() #}
Individual find_all_abstract_actors in QueryClass isA
AbstractActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ this in AbstractActorElementClass $
end

{# Query23: find_all_plain_actors() #}
Individual find_all_plain_actors in QueryClass isA
PlainActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ this in PlainActorElementClass $
end
```

```
{# Query24: find_all_agents() #}
Individual find_all_agents in QueryClass isA SpecifiedActorElementClass
with
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (this in AgentElementClass) or (this in
AgentInstanceElementClass) $
end

{# Query25: find_direct_replaceable_actors(a:SpecifiedActorElementClass)
#}
{# Default view: v from the directReplaceableRule #}
{# Input parameters: a #}
Individual find_direct_replaceable_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ exists l/AssociationLinkClass
            ((l in PartsLinkClass) or (l in
CompleteCompositionLinkClass)) and (l from this) and (l to ~a)  or
            ((l in ISALinkClass) or (l in INSLinkClass) or (l in
CoversLinkClass) or (l in PlaysLinkClass) or
             (l in OccupiesLinkClass)) and (l from ~a) and (l to this)
      $
end

{# Query26: find_all_replaceable_actors(a:SpecifiedActorElementClass) #}
{# Default view: v from the directReplaceableRule #}
{# Input parameters: a #}
Individual find_all_replaceable_actors in GenericQueryClass isA
SpecifiedActorElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : SpecifiedActorElementClass
  attribute,constraint
     c : $ (this in find_direct_replaceable_actors[~a/a]) or
           (exists a2/SpecifiedActorElementClass
                  (a2 in find_all_replaceable_actors[~a/a]) and
                  (this in find_direct_replaceable_actors[a2/a]) )
      $
end

{
*     File : SDViews_Queries.sml
*     Purpose : Define the query classes for the SD views
*     created : 08/04/04 Jane You
*     last change : 09/01/04 Jane You
*     Contents: Query27~43
}

{# Query27: find_inter_dependums(A=(a1,...,am):ActorElementClass) #}
```

```
{# Comments: this query find the dependums among the selected set of
actors #}
Individual find_inter_dependums in GenericQueryClass isA
DependumElementClass with
  attribute,parameter
     A : QueryClass
  attribute,retrieved_attribute
     name : String;
     links : LinkClass
  attribute,constraint
     c : $ exists l1,l2/DependencyLinkClass a1,a2/ActorElementClass
           (a1 in ~A) and (l1 from this) and (a2 in ~A) and (l2 to this)
and
            ( (l1 to a1) or (exists e1/a1.children (l1 to e1)) ) and
            ( (l2 from a2) or (exists e2/a2.children (l2 from e2)) )
         $
end

{# Query28: find_inter_dependencies(A=(a1,...,am):ActorElementClass) #}
{# Comments: this query does not allow pending dependencies #}
Individual find_inter_dependencies in GenericQueryClass isA
DependencyLinkClass with
  attribute,parameter
     A : QueryClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists a/ActorElementClass b/DependumElementClass
            (a in ~A) and (b in find_inter_dependums[~A/A]) and
            ((this in find_outgoing_dependencies_from_actor[a/a]) and
(this to b) or
             (this from b) and (this in
find_incoming_dependencies_to_actor[a/a]))
         $
end

{# Query29:
find_direct_inter_external_links(A=(a1,...,am):ActorElementClass) #}
Individual find_direct_inter_external_links in GenericQueryClass isA
IntentionalLinkClass with
  attribute,parameter
     A : QueryClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists dl/DependencyLinkClass (dl in
find_inter_dependencies[~A/A]) and
            (exists a/ActorElementClass e/a.children (a in ~A) and (this
from e) and (this to dl) )
       $
end

{# Query30:
find_all_inter_external_links(A=(a1,...,am):ActorElementClass) #}
{# Comments: this query generates parser error with the line
      }
```

```
{     (exists a/ActorElementClass e/a.children (a in ~A) and (this from
e)) #}
Individual find_all_inter_external_links in GenericQueryClass isA
IntentionalLinkClass with
  attribute,parameter
     A : QueryClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ (exists a/ActorElementClass e/a.children (a in ~A) and (this
from e)) and
            ( (this in find_direct_inter_external_links[~A/A]) or
              (exists l2/IntentionalLinkClass (l2 in
find_all_inter_external_links[~A/A]) and (this to l2)) ) $
end

{# Query31: find_incoming_dependums_to_actor(a:ActorElementClass) #}
{# Comments: find dependum element that depends on "a" #}
Individual find_incoming_dependums_to_actor in GenericQueryClass isA
DependumElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists l/DependencyLinkClass (l from this) and (l in
find_incoming_dependencies_to_actor[~a/a]) $
end

{# Query32:
find_indirect_incoming_dependencies_to_actor(a:ActorElementClass) #}
{# Comments: find dependency links that ends at the incoming dependums
of actor "a" #}
Individual find_indirect_incoming_dependencies_to_actor in
GenericQueryClass isA DependencyLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists de/DependumElementClass (this to de) and (de in
find_incoming_dependums_to_actor[~a/a]) $
end

{# Query33: find_dependers_to_actor(a1:ActorElementClass) #}
{# Comments: find actors depends on "a" via a dependum #}
Individual find_dependers_to_actor in GenericQueryClass isA
ActorElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists d/DependumElementClass l/DependencyLinkClass
              (d in find_incoming_dependums_to_actor[~a/a]) and
              (l in find_outgoing_dependencies_from_actor[this/a]) and
              (l to d)
```

```
        $
end

{# Query34: find_outgoing_dependums_to_actor(a:ActorElementClass) #}
{# Comments: find dependum elements that "a" depends on #}
Individual find_outgoing_dependums_from_actor in GenericQueryClass isA
DependumElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists l/DependencyLinkClass (l to this) and (l in
find_outgoing_dependencies_from_actor[~a/a]) $
end

{# Query35:
find_indirect_outgoing_dependencies_from_actor(a:ActorElementClass) #}
{# Comments: find dependency links that ends at the outgoing dependums
of actor "a" #}
Individual find_indirect_outgoing_dependencies_from_actor in
GenericQueryClass isA DependencyLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists de/DependumElementClass (this from de) and (de in
find_outgoing_dependums_from_actor[~a/a]) $
end

{# Query36: find_dependees_from_actor(a:ActorElementClass) #}
{# Comments: find actors who "a" depends on via a dependum #}
Individual find_dependees_from_actor in GenericQueryClass isA
ActorElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists d/DependumElementClass l/DependencyLinkClass
              (d in find_outgoing_dependums_from_actor[~a/a]) and
              (l in find_incoming_dependencies_to_actor[this/a]) and
              (l from d)
         $
end

{# Query37:
find_externallinks_to_incoming_dependency(a:ActorElementClass) #}
{# Input parameters: a #}
Individual find_externallinks_to_incoming_dependency in
GenericQueryClass isA IntentionalLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
```

```
     c : $ exists dl/DependencyLinkClass
            (dl in find_incoming_dependencies_to_actor[~a/a]) and (this
to dl) $
end

{# Query38:
find_externallinks_originator_to_incoming_dependency(a:ActorElementClass)
#}
{# Comments: find the actor that has an external link ends at "a"'s
incoming dependency link #}
Individual find_externallinks_originator_to_incoming_dependency in
GenericQueryClass isA ActorElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists l/IntentionalLinkClass
            (l in find_externallinks_to_incoming_dependency[~a/a]) and
            (exists e/this.children (l from e)) $
end

{# Query39:
find_externallinks_to_indirect_outgoing_dependency(a:ActorElementClass)
#}
Individual find_externallinks_to_indirect_outgoing_dependency in
GenericQueryClass isA IntentionalLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists de/DependumElementClass dl/de.links
            (dl in find_indirect_outgoing_dependencies_from_actor[~a/a])
and (this to dl) $
end

{# Query40:
find_externallinks_originator_to_indirect_outgoing_dependency(a:ActorEle
mentClass) #}
Individual find_externallinks_originator_to_indirect_outgoing_dependency
in GenericQueryClass isA ActorElementClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String
  attribute,constraint
     c : $ exists l/IntentionalLinkClass
            (l in
find_externallinks_to_indirect_outgoing_dependency[~a/a]) and
            (exists e/this.children (l from e)) $
end

{# Query41: find_externallinks_from_actor(a:ActorElementClass) #}
{# Comments: find the external links that originated from actor "a" #}
Individual find_externallinks_from_actor in GenericQueryClass isA
IntentionalLinkClass with
```

```
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String;
     from : ElementClass;
     to : ObjectClass
  attribute,constraint
     c : $ (exists e/IntentionalElementClass (this from e) and (e parent
~a)) and
              (this in find_all_external_links)
       $
end

{# Query42:
find_externallinks_to_externallinks_from_actor(a:ActorElementClass) #}
{# Comments: find the external links that affect the external links
originated from actor "a" #}
Individual find_externallinks_to_externallinks_from_actor in
GenericQueryClass isA IntentionalLinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String;
     from : ElementClass;
     to : ObjectClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass (l in
find_externallinks_from_actor[~a/a])
            and (this to l) $
end

{# Query43: find_externallinks_target_from_actor(a:ActorElementClass) #}
{# Comments: find the links that the external links originated from
actor "a" ends at #}
Individual find_externallinks_target_from_actor in GenericQueryClass isA
LinkClass with
  attribute,parameter
     a : ActorElementClass
  attribute,retrieved_attribute
     name : String;
     from : ElementClass;
     to : ObjectClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass (l in
find_externallinks_from_actor[~a/a])
            and (l to this) $
end

{
*     File : SRViews_Queries.sml
*     Purpose : Define the query classes for the SR views
*     created : 08/05/04 Jane You
*     last change : 09/01/04 Jane You
*     Contents: Query44~51
}

{# Query44: find_internal_connectors(a:ActorElementClass) #}
```

```
{# Comments: find the internal elements that has an external link
connected #}
Individual find_internal_connectors in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $ (this parent ~a) and
            (exists l1/DependencyLinkClass (l1 from this) or (l1 to
this)) or
            (exists l2/IntentionalLinkClass (l2 in
find_externallinks_from_actor[~a/a]) and (l2 from this))
      $
end

{# Query45: find_root_elements(a:ActorElementClass) #}
Individual find_root_elements in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $ (this parent ~a) and
              (not (exists l/IntentionalLinkClass (l from this)) )$
end

{# Query46: find_root_softgoals(a:ActorElementClass) #}
Individual find_root_softgoals in GenericQueryClass isA
SoftgoalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $ (this in find_root_elements[~a/a]) $
end

{# Query47: find_root_functionals(a:ActorElementClass) #}
Individual find_root_functionals in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass
  attribute,constraint
     c : $ (this in find_root_elements[~a/a]) and not (this in
SoftgoalElementClass) $
end

{# Query48: find_contribution_to_dependum(a:ActorElementClass,
dl:DependencyLinkClass) #}
Individual find_contribution_to_dependum in GenericQueryClass isA
IntentionalLinkClass with
  attribute,retrieved_attribute
```

```
     name : String
  attribute,parameter
     a : ActorElementClass;
     dl: DependencyLinkClass
  attribute,constraint
     c : $ (this to ~dl) and
           (exists e/IntentionalElementClass  (e parent ~a) and (this
from e) )$
end

{# Query49: find_contributor_to_dependum(a:ActorElementClass,
dl:DependencyLinkClass) #}
Individual find_contributor_to_dependum in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a : ActorElementClass;
     dl: DependencyLinkClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass
            (l in find_contribution_to_dependum[~a/a,~dl/dl]) and (l
from this)  $
end

{# Query50: find_contribution_to_actor(a, a1:ActorElementClass) #}
{# Input argument: "a1" is the affected actor #}
Individual find_contribution_to_actor in GenericQueryClass isA
IntentionalLinkClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a0: ActorElementClass;
     a1: ActorElementClass
  attribute,constraint
     c : $ (exists e0/IntentionalElementClass (e0 parent ~a0) and (this
from e0) )and
           (exists l1/IntentionalLinkClass e1/IntentionalElementClass
                  (l1 from e1) and (e1 parent ~a1) and (this to l1)) $
end

{# Query51: find_contributor_to_actor(a, a1:ActorElementClass) #}
{# Input argument: "a1" is the affected actor #}
Individual find_contributor_to_actor in GenericQueryClass isA
IntentionalElementClass with
  attribute,retrieved_attribute
     name : String
  attribute,parameter
     a0: ActorElementClass;
     a1: ActorElementClass
  attribute,constraint
     c : $ exists l/IntentionalLinkClass
            (l in find_contribution_to_actor[~a0/a0,~a1/a1]) and (l from
this)  $
end
```

C. Facts about the London Ambulance Service Computer Aided Despatch System

We cite in this section the source of information on which we based for our London Ambulance Service (LAS) case study. All paragraphs appear in this section are items stated in the "Report of the Inquiry into the London Ambulance Service" (LAS-Report 1993). We select the part the describes the manual process, the constructs of the Computer Aided Despatch (CAD) system, and the system requirements for performance.

The manual system operates as follows:

Call Taking

3002 When a 999 or urgent call is received in Central Ambulance Control the Control Assistant (CA) writes down the call details on a pre0printed form (AS1 or AS2). The incident location is identified from a map book, together with the map reference co0ordinates. On completion of the call the incident form is placed into a conveyor belt system with other forms from fellow CA's. The conveyor belt then transports the forms to a central collection point within CAC.

Resource Identification

3003 Another CAC staff member collects the forms from the central collection point and, through reviewing the details on the form, decides which resource allocator should deal with it (based on the three London Divisions—North East, North West, and South). At this point potential duplicated calls are also identified. The resource allocator then examines the forms for his/her sector and, using status and location information provided through the radio operator and noted on forms maintained in the "activation box" for each vehicle, decides which resource should be mobilized. This resource is then also recorded on the form which is passed to a despatcher.

Resource Mobilisation

3004 The despatcher will telephone the relevant ambulance station (if that is where the resource is) or will pass mobilisation instructions to the radio operator if the ambulance is already

3005 According to the ORCON standards this whole process should take no more than 3 minutes.

The System Structure:

3119 The complete CAD system had a number of different elements including:

a) CAD software;

b) CAD hardware;

c) RIFS Communication Interface;

d) radio system;

e) Datarak Sub System;

f) Gazekeer and Mapping Software;

g) Mobile Data Terminals.

System Performance Requirements:

6082 We recommend that LAS makes available to interested parties such as Community Health Councils, purchasers of the service and London MPs its performance levels in respect of:

a) 999 telephone answering times;

b) activation percentage within three minutes;

c) response percentage within 8 minutes;

d) response percentage within 14 minutes.

Bibliography

Alexander I. 2003. “Misuse Cases: User Cases with Hostile Intent,” IEEE Software, 20(1), Jan.-Feb. 2003: 58-66.

Breitman KK, Leite JC, and Finkelstein A. 1999. “The World’s a Stage: a Survey on Requirements Engineering Using a Real-life Case Study”, Journal of the Brazilian Computer Society, 6.1, Campinas, July 1999.

Bubenko JJ, Persson A, Stirna J. 2001 Oct. User Guide of the Knowledge management Approach Using Enterprise Knowledge Patterns. Stockholm (Sweden): Department of Computer and Systems Science, Royal Institute of Technology. 52 p.

Carlson CR, Ji W, Arora AK. 1990. Elsevier Science Publishers B.V. In F.H. Lochovsky, editor. “The Nested Entity-Relationship Model,” Entity-Relationship Approach to Database Design and Querying, North-Holland, 1990: 221-236.

Campbell LJ, Halpin TA, Proper HA. 1996. “Conceptual Schemas with Abstractions—Making Flat Conceptual Schemas More Comprehensible,” Data & Knowledge Engineering, 20.1 (1996): 39-85.

Chung L, Nixon B, Yu E. 1997. “Dealing with Change: An Approach Using Non-Functional Requirements,” Requirement Engineering, Springer-Verlag, 1.4 (1997): 238-260.

Chung L, Gross D, Yu E. 1999. Kluwer Academic Publishers. In: Patrick Donohue, editor. “Architectural Design to Meet Stakeholder Requirements,” Software Architecture, 1999: 545-564.

Chung L, Nixon BA, Yu E, Mylopoulos J. 2000. Kluwer Academic Publishers. Non-Functional Requirements in Software Engineering. 472 p. ISBN 0-7923-8666-3.

ConceptBase Team. 2003. ConceptBase Tutorial. Aachen(Germany): Informatik V., RWTH Aachen. 10 p.

Castano S, DE ANTONELLIS V, FUGINI MG, PERNICI B. 1998. “Conceptual Schema Analysis: Techniques and Applications,” ACM Transactions on Database Systems, 23.3 (Sep 1998): 286-333.

Damm W, Harel D. 2001. Klumer Academic Publishers. “LSCs: Breathing Life into Message Sequence Charts,” Formal Methods in System Design, 19 (2001): 45-80.

Douglass BP. 2003. “UML 2.0 Incrementally Improves Scalability and Architecture.” Available: http://www.elecdesign.com/articles/print.cfm?articleID=5881 (Oct. 2003).

Dubois E, Yu E, Petit M. 1998. IEEE Computer Society. “From Early to Late Formal Requirements: a Process Control Case Study,” Proceedings of the 9th International Workshop on Software Specification and Design, Ise-Shima, Japan, Apr. 1998: 34-42.

Feldman P, Miller D. 1986. “Entity Model Clustering: Structuring a Data Model by Abstraction,” Computer Journal, 29.4 (Aug. 1986): 348-360.

Ghandi M, Robertson EL, Gucht DV. 1992. Springer-Verlag. In P. Loucopoulos, editor. “Leveled Entity Relationship Model,” Proceedings of the Fourth International Conference CAiSE’92 on Advanced Information Systems Engineers, volume 593 of Lecture Notes in Computer Science, May1992; Manchester, United Kingdom. p 456-473.

Gross D, Yu E. 2001. “Evolving System Architecture to Meet Changing Business Goals: an Agent and Goal-Oriented Approach,” ICSE-2001 Workshop: From Software Requirements to Architectures (STRAW 2001), Toronto, Canada, May 2001: 13-21.

GRL. 2003. "URN — Goal-oriented Requirement Language (GRL)," Recommendation Z.150: User Requirements Notation (URN) – Language requirements and framework, Sep. 2003.Available: http://www.usecasemaps.org/urn/z_151-ver3_0.zip . Last view Aug. 2004.

Harel D. 1988. "On Visual Formalisms," Communications of the ACM, 31.5 (May 1988): 514-530.

Horkoff J. 2004. "A Study of Trusted Computing Using the i* Framework." Working Paper, Knowledge Management Lab, Bell University Labs, University of Toronto. 135 p. Available: Last view Aug. 2004.

IDEF0. 1993. IDEF Family of Methods, Knowledge Based Systems, Inc. (KBSI). Available: http://www.idef.com/idef0.html. Last view Aug. 2004.

Jarke M, Jeusfeld MA, Quix C. 2003. ConceptBase V6.1 User Manual. Aachen(Germany): Informatik V., RWTH Aachen. 98 p.

Jarke M, Gallersdörfer R, Jeusfeld MA, Staudt M, Eherer S. 1995. "ConceptBase - A Deductive Object Base for Meta Data," Journal on Intelligent Information Systems, 2.4 (Mar 1995): 167-192.

Koubarakis M, Mylopoulos J, Stanley M, Borgida A. Feb. 1989. Telos: Features and Formalization. Toronto (ON): Department of Computer science, University of Toronto. Report nr KRR-TR-89-4. 84 p.

Kramer J, Wolf A. 1996. ACM SIGSOFT. "Succeedings of the 8th International Workshop on Software Specification and Design," Software Engineering Notes, 21.5, Sep. 1996: 21-35.

Lamsweerde AV. 2003. "Goal-Oriented Requirements Engineering: from System Objectives to UML Models to Precise Software Specifications," ICSE'03 Tutorial, Portland, May 2003. 159 p.

LAS-Report. 1993. Report of the Inquiry into the London Ambulance Service, electronic version prepared by prof. A. Finkelstein, available at http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html with permission from the communications directorate, South West Thames Regional Health Authority, original ISBN: 0 905133 70 6, 1993

Letier E. 2001. Reasoning about Agents in Goal-oriented Requirements Engineering [dissertation]. Belgium: Department of Computing Science and Engineering, Université catholique de Louvain. 283 p.

Liu L, Yu E. 2001. "From Requirements to Architectural Design - Using Goals and Scenarios," ICSE-2001 Workshop: From Software Requirements to Architectures (STRAW 2001), Toronto, Canada, May 2001: 22-30.

Liu L, Yu E, Mylopoulos J. 2002. "Analyzing Security Requirements as Relationships Among Strategic Actors," 2nd Symposium on Requirements Engineering for Information Security (SREIS'02), Raleigh, North Carolina, Oct. 2002.

Liu L, Yu E, Mylopoulos J. 2003. "Security and Privacy Requirements Analysis within a Social Setting," 11th IEEE International Conference on Requirements Engineering (RE'03), Monterey, California, Sep. 2003: 151-161.

OME. 2003. Organization Modelling Environment (OME) [Tool]. Knowledge Management Lab, Bell University Labs, University of Toronto. Available: http://www.cs.toronto.edu/km/ome/. Last view Aug. 2004.

You Z. 2003. "Applying the GRL Framework to the LAS-CAD Case Study." Working Paper, Knowledge Management Lab, Bell University Labs, University of Toronto. 65 p. Available: http://www.cs.toronto.edu/~janeyou/avs/csc2150Project.doc (Aug. 2003). Last view Aug. 2004.

Yu E. 1994. Modelling Strategic Relationships for Processing Reengineering [dissertation]. Toronto (ON): Department of Computer science, University of Toronto. 124 p.

Yu E. 1997 Jan. “Towards Modeling ad Reasoning Support for Early-Phase Requirements Engineering,” Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, Washington D.C., USA, Jan. 1997: 226-235.

Yu E. 1997 Jun. Presses Universitaires de Namur. In: E. Dubois, A.L. Opdahl, K. Pohl, editors. “Why Agent-Oriented Requirements Engineering,” Proceedings of 3rd International Workshop on Requirements Engineering: Foundations for Software Quality, Barcelona, Catalonia, June 1997.

Yu E, Liu L. 2000. “Modelling Trust in the *i** Strategic Actors Framework,” Proceedings of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies, Barcelona, Catalonia, Spain, June 2000.

Yu E, Liu L, Li Y. 2001. Spring Verlag. “Modelling Strategic Actor Relationships to Support Intellectual Property Management,” 20th International Conference on Conceptual Modeling (ER-2001), Yokohama, Japan, Nov. 2001: 164-178. LNCS 2224

Yu E, Cysneiros LM. 2002. “Designing for Privacy and Other Competing Requirements,” 2nd Symposium on Requirements Engineering for Information Security (SREIS’02), Raleigh, North Carolina, Oct. 2002.