R*i*SD: A Methodology for Building *i** Strategic Dependency Models

Gemma Grau, Xavier Franch, Enric Mayol, Claudia Ayala,
Carlos Cares, Mariela Haya, Fredy Navarrete, Pere Botella, Carme Quer
Universitat Politècnica de Catalunya (UPC), LSI - Campus Nord, Barcelona (Catalunya, Spain)
{ggrau, franch, mayol, cayala, ccares, mhaya, fjnavarrete, botella, cquer}@lsi.upc.edu

Abstract

Goal-oriented models have become a consolidated type of artefact in various software and knowledge engineering activities. Several languages exist for representing such type of models but there is a lack of associated methodologies for guiding their construction up to the necessary level of detail. In this paper we present RiSD, a methodology for building Strategic Dependency (SD) models in the i notation. RiSD is defined in a prescriptive way to reduce uncertainness when constructing the model. RiSD also tackles two fundamental issues: on the one hand, it tends to reduce the average size of the resulting models and, on the other hand, it allows including some traceability relationships in the resulting models. As a result, we may say that RiSD increases the understandability of goal-oriented models whilst improving all construction.*

1. Introduction

In the last years, the construction of *goal-oriented and agent-oriented models* has become an extended practice in fields such as requirements engineering and organizational process modeling [1, 2]. One of the most widespread goal-oriented languages is the *i** notation proposed by Eric Yu in the first half of the 90's [3, 4]. *i** allows for the clear and simple statement of goals that system actors have and dependencies among them.

In despite of its utility, the intensive use of *i** reveals some difficulties. In [5] we have tackled one of them, namely the diversity of *i** dialects and variations that may be disturbing when learning the notation. In this paper we deal with another two drawbacks that we have experimented: the absence of detailed methodologies for building the models and the complexity of the resulting models.

– **Absence of methodology**. Currently we can say that there is a lack of guidance for supporting the prescriptive construction of *i** models. There exists a consolidated methodology such as Tropos [6] but it is aimed mainly to the guidance of the whole software development process. In this sense, it supports the conception of a global solution for the problem at hand, but gives a high degree of freedom for the construction of the models themselves (i.e., which intentional elements exist). One could argue that this is precisely a property inherent to agent-oriented methodologies [4], but the flexibility of the *i** language means to have multiples choices when building a model (i.e. when to include an intentional element or not, which type of element is the most appropriate for a given situation, etc).

– **Complexity of the models**. Models for non-trivial systems grow very quickly and are plenty of intentional elements of many types without obvious relationships among them. There are two main types of hidden relationships. On the one hand, two intentional elements may depend one on another (e.g., one may imply the other). On the other hand, two intentional elements may be at different levels of detail, being one a refinement of another. For some types of relationships we may find constructs in the language but not for all, especially when referring to one of the two types of models offered by *i**, namely *Strategic Dependency* (SD) model.

In this paper, we propose R*i*SD, a methodology for building Reduced *i** SD models for software systems. R*i*SD is defined as a set of activities structured in two phases, one for constructing the social system (without software) and the other for constructing the socio-technical system (with software). Both phases may involve the partial or total construction of the other type of *i** models, namely *Strategic Rationale* (SR) models. R*i*SD includes precise questions and answers that guide the development process and provide cut criteria for choosing among different types of intentional elements when diverse options exist. The size of the resulting model is reduced due to these criteria. R*i*SD includes also some traceability constructs that show the relationships among intentional elements and enhances therefore understanding of the model.

2. The *i** language

The *i** language defined by Eric Yu [3, 4] proposes the use of two models, each one corresponding to a different abstraction level: a *Strategic Dependency* (SD) model represents the intentional level and the *Strategic*

Rationale (SR) model represents the rational level. We present in this section the *i** constructs needed in R*i*SD.

A SD model consists of a set of nodes that represent *actors* and a set of *dependencies* that represent relationships among them, expressing that an actor (*depender*) depends on some other (*dependee*) in order to obtain some objective (*dependum*). The *dependum* is an intentional element that can be a *resource*, *task*, *goal* or *softgoal* (see section 4 for a detailed description). Actors may be specialized through the *is-a* relationship.

A SR model allows visualizing the intentional elements into the boundary of an actor in order to refine the SD model to add reasoning ability. The dependencies of the SD model are linked to intentional elements inside the actor boundary. The elements in the SR model are decomposed accordingly to the links:

– *Means-end links* establish that one or more intentional elements are the means that contribute to the achievement of an end. When there is more than one means an OR relation is assumed, indicating the different ways to obtain the end

– *Task-decomposition links* state the decomposition of a task into different intentional elements. There is a relation AND when a task is decomposed.

Last, we mention that actors may enclose subactors in their SR decomposition. Subactors just define frontiers in a SR model that group closely-related intentional elements. Links that relate intentional elements belonging to different subactors are converted into dependencies; also, new dependencies may be identified.

The graphical notation is shown in figure 1 using an excerpt of a model for an academic tutoring of students. On the left-hand side, we show the SR model of a tutor and the hierarchical relationships among their internal intentional elements. On the right-hand side, we show the SD dependencies between a student and a tutor. Neither specializations nor subactors appear.

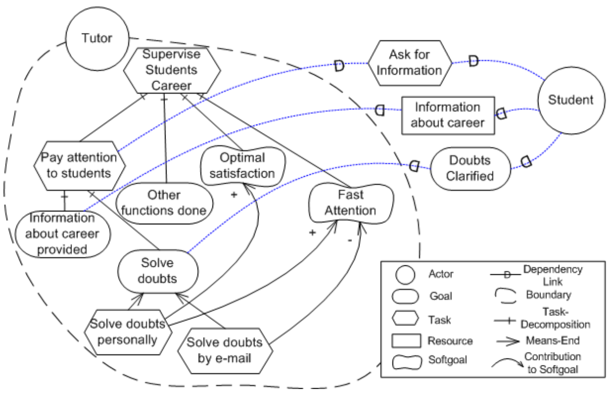


Figure 1. Excerpt of *i** model for an academic tutoring system.

3. A procedure for building *i** SD models

In this section, we present an overview of the R*i*SD methodology for building *i** SD models for software systems. R*i*SD guides the model development process by means of precise questions to the modeller. It has two clearly differentiated phases. The first one deals with the construction of a social system model. This model is characterized by the fact that it does not include the software system and therefore it focuses on the stakeholder needs. Afterwards, in the second phase, the software system is incorporated to obtain a socio-technical system, and the SD model is reconfigured around this new component. This development process is similar to the early requirements analysis and late requirement analysis proposed by the TROPOS methodology [7].

The social system model is constructed iteratively. It begins with the identification of an initial set of social actors involved in the addressed problem and their main goals. Then, strategic dependencies among actors are identified and classified by considering which is the most appropriate type of each dependum. To assist in this decision, R*i*SD provides a clear cut criteria by means of short, concise and focused questions. At this point, a first version of the social system model is obtained. To refine this model, existing dependencies are analyzed to identify if new actors or new dependencies should be incorporated to the model, in which case the process iterates.

The socio-technical system model construction is also iterative. It begins with the definition of the software system as a new actor (with its main goal) and its inclusion in the social system model. Next, considering this actor the existing dependencies are reassigned. The system may be decomposed into subsystems which are modeled as new actors (subactors) and, therefore, the existing dependencies are reassigned again. New subsystems may depend on each other and these dependencies must also be established. A refinement process, similar to that performed in the social model, may also be applied. Both phases can be iterated as it can be seen in figure 2.

Although R*i*SD focused on the construction of SD models, both phases may involve the partial or total construction of the other type of *i** models: SR models.

Throughout the paper we use for discussion an example about the specification of a software system for supporting information reliability in an organization.

4. Phase I: social system construction

We describe in this section the construction of the social system related to our example in several activities. Before

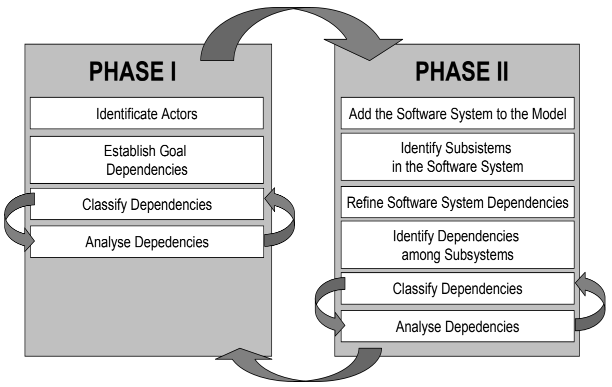

PHASE I
Identificate Actors
Establish Goal Dependencies
Classify Dependencies
Analyse Depedencies
PHASE II
Add the Software System to the Model
Identify Subsistems in the Software System
Refine Software System Dependencies
Identify Dependencies among Subsystems
Classify Dependencies
Analyse Depedencies

Figure 2. Diagram of the two phases of the RiSD methodology

beginning this process is advisable to carefully examine and describe the domain of interest. As one of the most endangering points when examining a domain is the lack of a standard terminology, the construction of a glossary of terms helps to avoid semantic problems when constructing the model. The use of auxiliary models, such as UML, can be also useful for understanding the different concepts involved in the domain. The effort invested in this preliminary domain study has to be proportional to the deep of knowledge implied in the model we want to build (superficial or very precise representation), as the knowledge might also be acquired during the process when needed.

Activity I.1. *Identify departing actors.*

The goal of this activity is to discover the main actors of the social system and their goals. The actors are required to have a clear strategic value for the modeled system; it is useful to use a metaphor to think about the system.

In our case, we use a client-server metaphor: an actor (the client) provides and consumes a resource (the information) that is under the control of an organization (the server). This and other metaphors could be organized in the form of a catalogue of *i** organizational patterns [8].

Thus the departing actors and their goals are: *User* with the goal *Digital Information Kept Reliable*, and *Organization*, with the goal *Digital Information Produced and Preserved* as we illustrate in figure 3.

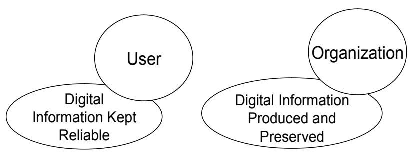

User
Digital Information Kept Reliable
Organization
Digital Information Produced and Preserved

Figure 3. Departing actors for the information reliability case.

Activity I.2. *Establish goal dependencies among actors.*

This activity aims at identifying dependencies among actors. By default, we classify them as goal dependencies, which are the most common type due to their strategic value. Activity 3 will confirm or change this classification.

The crucial point of this activity is to identify just those dependencies that are really needed. This criteria is obviously fuzzy and therefore the number of dependencies that will arise in this step is inevitably subjective, which in fact is a characteristic of goal-oriented modeling [4]. However, when using a catalogue of *i** organizational patterns such as [8], the dependencies already proposed in the patterns can be adapted to the social system we are modeling. Thus, haziness is reduced because the first set of dependencies among the actors is the one provided by the pattern. In our example, as we are considering a client-server metaphor, we try to reduce the uncertainness of the activity by means of the following 2-stage procedure:

– First, we shall respond to the question: which services does the user require to the organization for the social system providing some added value? For each service, we include a dependency from the user to the organization.

– Then, we shall respond to another question: which behavior does the organization require to the user for supporting (at least partially) the requested services?

If answers to these questions are not clear, we may build a first level of SR decomposition of some actor, which will show more explicitly which means can be undertaken by the actor itself and which others need the support of some other actor and therefore a dependency.

Figure 4 shows the result of the activity in our case. We identify two main services requested by the user, and one behavior that partially supports them. Dependencies shall be generic enough in order not to exclude important aspects. For instance, if we were directly talking about viruses or spam in our example, other aspects such as cryptography could be left out of the system.

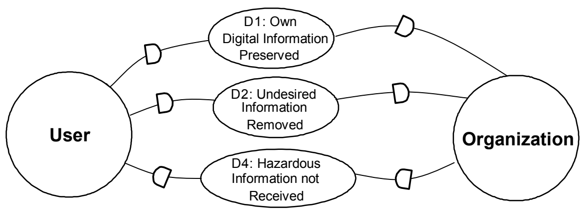

D1: Own Digital Information Preserved
D2: Undesired Information Removed
D4: Hazardous Information not Received
User
Organization

Figure 4. Main dependencies in the information reliability

Activity I.3. *Classify the added dependencies*

The goal of this activity is to define more precisely the dependencies identified in activity 2. In this activity, we definitively classify the dependums as a task, resource, softgoal or goal. Moreover, after identifying the type of dependum, we propose some syntactic patterns to name these dependencies more precisely.

To classify each dependency into a valid type of *i** we propose a set of questions to be answered following a predefined ordering as shown in the graph of figure 5. In nodes 1 to 4 a question must be answered; in nodes 5 to 8 a specific type of dependency has been identified; in nodes 9 to 11 some additional softgoal dependencies may be added to the model. In the graph, each type of dependum is identified by a capital letter: **R**esource, **T**ask, **G**oal and **S**oft**G**oal.

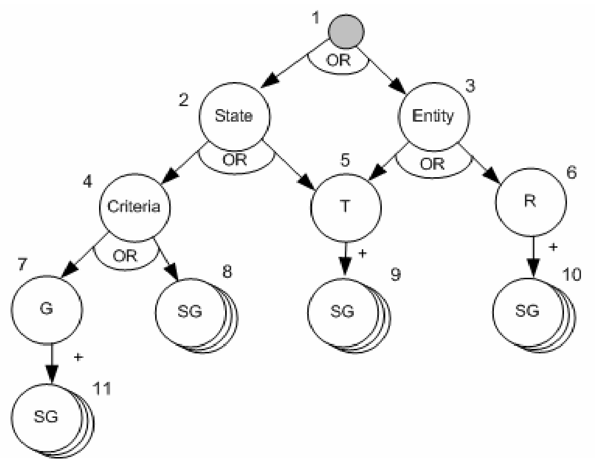


Figure 5. Graph to classify *i** dependencies

Starting at node 1, questions to answer at each node to classify the dependency D, from A to B, identified in activity I.2 are:

1. Does the depender depend on the dependee to achieve an entity or to attain a certain state? If entity, go to 3; else, go to 2.
2. Is the depender interested in attaining the state following a particular process? If so, classify D as task dependency and go to 5; else, go to 4.
3. Is the depender interested in obtaining the entity following a particular process? If so, classify D as task dependency and go to 5; else, classify D as resource dependency and go to 6.
4. Is there a clear cut criteria to determine the achievement of the state? If so, confirm the dependency D as goal dependency and go to 7; else, classify D as softgoal dependency.
5. Is there some additional restrictions on how to execute the task? If so, for each restriction, establish a new softgoal dependency from A to B.
6. Is there some additional properties that the resource must met to be acceptable? If so, for each property, establish a new softgoal dependency from A to B.
7. Is there some extra conditions that the achievement of the goal must satisfy? If so, for each condition, establish a new softgoal dependency from A to B.

In our example, the three departing dependencies are left as goal dependencies, since all of them correspond to states and their achievement (removed, received and preserved) can only be or not. Furthermore, there are not additional conditions for the achievement of the goal. Thus, figure 4 becomes also the result of activity I.3.

To improve the understandability of the new classified dependencies, the names assigned to their dependums shall be kept short and precise and be consistent throughout the model. There are some conventions issued by different authors and we adhere to that of Yu [4], summarized in Table 1 (parenthesis stand for optionality). Longer descriptions can be added to the documentation, especially if using tool support such as OME [9] or REDEPEND [10]. We remark the case of softgoal dependencies, in which we distinguish among dependencies that stand alone (node 8 in the graph of figure 5), whose pattern is Goal-Syntax + Complement; and dependencies that qualify another dependum of the model (nodes 9, 10 and 11), in which the qualifier is a Complement and (optionally) the dependum between brackets (as done in [4]). Note that using these syntactical patterns we will use short names that are specific to the semantics of the dependum, increasing in this way the comprehensibility of the model.

Depen-dum	Syntax	Example
Task	Verb + (Object) + (Complement)	Answer doubts by e-mail
Resource	(Adjective) + Object	Virus List
Goal	Object +Passive_Verb	Information kept preserved
Softgoal	− Goal syntax + Complement − (Object) + Complement ([Dependum])	− Information checked in a transparent manner − Timely[Virus List]

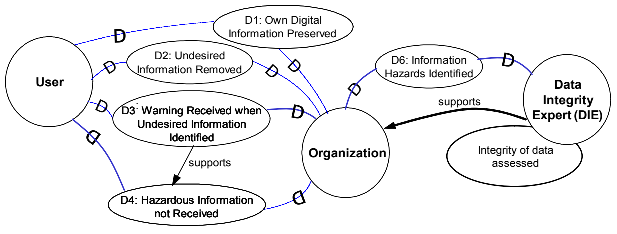
Table 1. Syntactic conventions for *i** dependums.

Activity I.4. *Analyze the consequences of the dependencies*.

For every dependency added in activity I.2 and classified in activity I.3, we must check if either the dependee is able to satisfy by itself the required dependum or if it needs some help from other actor, that may already exist, or not yet (in the last case, its goal must be declared first, as done in activity I.1). For deciding this, a question is raised whose concrete form depends on the type of dependum: does the depender need some support to attain the goal, or produce the resource, or execute the task, or accomplish the behavior? If the answer to this question is not obvious, it may be necessary to develop one or two levels of decomposition of the SR diagram of the dependee actor taking as root an intentional element equivalent to the involved dependum.

An important decision we have taken that applies to this activity: we have added a traceability construct to keep the path from actors and dependencies that have come into existence to support; we call this construct "supports". We can also use this construct to make even more explicit the binding among softgoals identified in activity I.3 as qualifiers of other dependums.

Figure 6 shows the SD model obtained after considering the dependencies appearing in figure 4. We have added a goal dependency supporting the dependency from the organization to the user. More remarkably, the organization needs a supporting actor to identify the hazards that endanger the managed information. This new actor, a data integrity expert, has a concise and clear goal in the context of the system.



D1: Own Digital Information Preserved
User
D2: Undesired Information Removed
D3: Warning Received when Undesired Information Identified
supports
D4: Hazardous Information not Received
Organization
D6: Information Hazards Identified
supports
Data Integrity Expert (DIE)
Integrity of data assessed

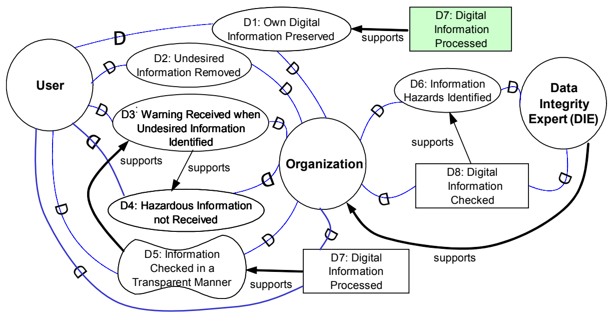
Figure 6. First refinement in the information security case

Iteration. *Refining the social system.*

To end this first stage of R*i*SD, we iterate activities I.3 and I.4 as required. Again the answers to the identified questions are crucial to progress towards the objective.

The termination condition is as usual somehow subjective. However, a useful rule that applies is the following: if the last iteration has identified just resource and task dependencies, then we can stop the process. Refinement of task and resource dependencies is usually too prescriptive at the SD level, just identifying steps of the tasks, or components of the resource.

Figure 7 shows the final result in our case. We have done just 2 more iterations. The final model consists of 3 actors (User, Organization and DIE) and 8 dependencies (D1 to D8), with 4 supporting relationships among them. We can check that the last dependencies added are resources. We also observe that the model just introduces a softgoal dependency, incorporating a fundamental



D1: Own Digital Information Preserved
supports
D7: Digital Information Processed
User
D2: Undesired Information Removed
D3: Warning Received when Undesired Information Identified
supports
supports
D4: Hazardous Information not Received
D5: Information Checked in a Transparent Manner
supports
D7: Digital Information Processed
Organization
D6: Information Hazards Identified
supports
D8: Digital Information Checked
Data Integrity Expert (DIE)
supports

Figure 7. Final social system in the information security case

behavior that shall be observed in the system. This model provides a highly strategic view of the social system, ready to be reconsidered once the software system is incorporated.

5. Phase II: socio-technical system construction

Phase II consists mainly of putting the software system at the heart of the social system model, reassigning the existing dependencies, and relating them to operational concepts coming mainly from the software marketplace.

Activity II.1. *Putting the software system in the social system model.*

The first activity defines a new actor for the software system, states its high-level goal and reassigns the existing dependencies as needed. The goal can be stated simply as providing assistance to the general pursued objective. Since in our example the main beneficiary is the user, the simplest way to state the goal is "Provide assistance for reliable information".

Dependency reassignment can be decided by answering the question "May the software system provides any assistance on attaining the goal / producing the resource / executing the task / achieving the property of the dependency?" It is very likely that more than one answer is possible, meaning that there is more than one way to assign responsibilities, in which case we have different alternatives to be analyzed.

Figure 8 shows the result in our example. We have taken the strategic decision of giving the software system as much responsibility as possible. With this rationale behind, we have been able to reassign all the dependencies to stem from, or point to, the software system. The new actor acts then as a mediator among all the involved parties.

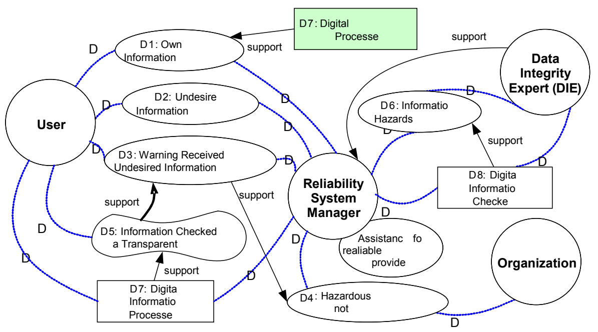

D7: Digital Processe
D1: Own Information
support
D
support
Data Integrity Expert (DIE)
D2: Undesire Information
D6: Informatio Hazards
User
D3: Warning Received Undesired Information
support
Reliability System Manager
D8: Digita Informatio Checke
support
D5: Information Checked a Transparent
Assistanc fo realiable provide
Organization
support
D7: Digita Informatio Processe
D4: Hazardous not

Figure 8. Putting the software system into the social

Activity II.2. *Identify subsystems in the software system and use them to drive a first level of SR*

Usual software systems are large enough to prevent the definition of a single, monolithic actor to cover their goal. We can use *i** actor decomposition facilities to split the single software actor into several subsystems, each of them with a well-defined goal. The combination of all the resulting goals must cover the main one.

This activity may be conducted through the combination of two strategies:

– *Dependency-driven*: the existing dependencies identify subsystems of interest. For each dependency, we can raise the question "is it identifying one or more goals in the software system?".

– *Market-driven*: knowledge of the marketplace makes some subsystems evident. The key question here is "which type of available software packages apply to the problem at hand?".

In our example, since there is a great deal of software packages dealing with information reliability, the second strategy predominates and therefore some widespread subsystems are identified: anti-virus, anti-spam, filters, and others; we show just the first two of them in figure 9. A goal is introduced for each one, which is defined as a means to attain system's goal in the corresponding SR diagram. Furthermore, It means that social actors are introduced accordingly in the SR (e.g, user in figure 9).

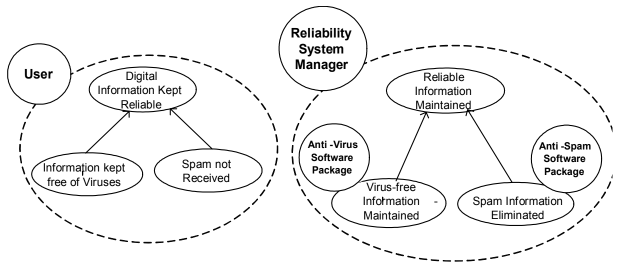

User
Digital Information Kept Reliable
Information kept free of Viruses
Spam not Received
Reliability System Manager
Reliable Information Maintained
Anti -Virus Software Package
Anti -Spam Software Package
Virus-free Information - Maintained
Spam Information Eliminated

Figure 9. Splitting the software system into subsystems

Activity II.3. *Refine software system dependencies into subsystem dependencies.*

Once the subsystems are identified, the main dependencies can be reassigned. For each dependency and subsystem, we use the following two questions:

– Does the dependency involve the subsystem? The answer is straightforward if activity II.2 has followed the dependency-driven strategy.

– If the answer is yes, then: how does the subsystem interpret the concepts involved by the dependency?

This activity raises the second type of traceability construct we introduce in our framework: the "refines" relationship. The dependencies stemming from/pointing to the subsystems refine (and substitute) the original ones that involved the software system.

We focus here in the anti-virus related part. In this case, the key correspondences among the abstract social system and concrete subsystem concepts are: the hazard is the virus, the information that flows among actors is a file, and reliable interchange means detecting and removing (whenever possible) viruses from the file. With these guidelines, we can obtain the model presented in figure 10.

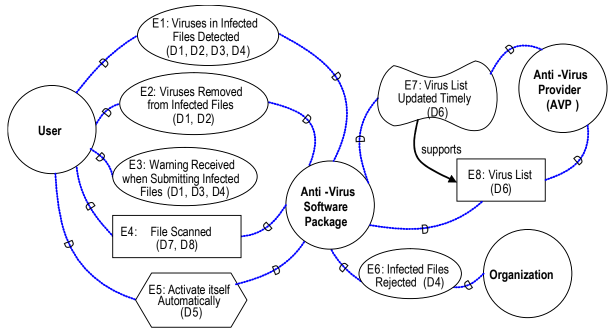

E1: Viruses in Infected Files Detected (D1, D2, D3, D4)
E2: Viruses Removed from Infected Files (D1, D2)
E7: Virus List Updated Timely (D6)
Anti -Virus Provider (AVP)
User
supports
E3: Warning Received when Submitting Infected Files (D1, D3, D4)
E8: Virus List (D6)
Anti -Virus Software Package
E4: File Scanned (D7, D8)
E6: Infected Files Rejected (D4)
Organization
E5: Activate itself Automatically (D5)

Figure 10. From software system to subsystem dependencies

For simplicity of the drawing, dependency refinement relationships are shown through the identifiers enclosed in parenthesis, which refer to dependencies that appear in figure 9. We remark that the refinement is many-to-many. We remark also that since we have a first level of decomposition in the social actor SR diagrams, we can reallocate the dependencies also in their side. Last, perhaps surprisingly, we have observed that dependers and dependees do not need to be kept strictly during the mapping. For instance, this is the case of dependency E3 that is declared as a refinement of dependency D4 (among others).

Activity II.4. *Identify subsystems dependencies.*

If subsystems coexist as part of the software system, it is very likely that they are related somehow. In particular, we will usually find that one subsystem may provide services needed by others. From the strategic point of view, some goals of one subsystem may depend on other subsystems and therefore we use again *i** dependencies to state them. In our example, one of the types of spam are messages containing viruses, thus we establish a goal dependency stating this (see figure 11).

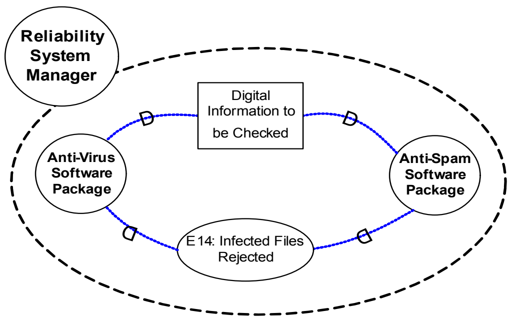

Reliability System Manager
Digital Information to be Checked
Anti-Virus Software Package
Anti-Spam Software Package
E14: Infected Files Rejected

Figure 11. Establishing subsystem dependencies

Activities II.5 and II.6. *Classify and Analyze dependencies.*

These activities are analogous to activities I.3 and I.4 (therefore they have the same name) which are also iterated. To sum up, in our example, focusing on the anti-virus part, we obtain, at the end, the actors and dependencies shown in figure 12. Remarkably, we have two new actors, an anti-virus administrator and the general concept of software package that may also acts as anti-virus user, therefore it is defined as a specialization of user. Then this part of the system is composed by 6 actors and 14 dependencies, which is a reasonable size for a concrete facet of information management as reliability is.

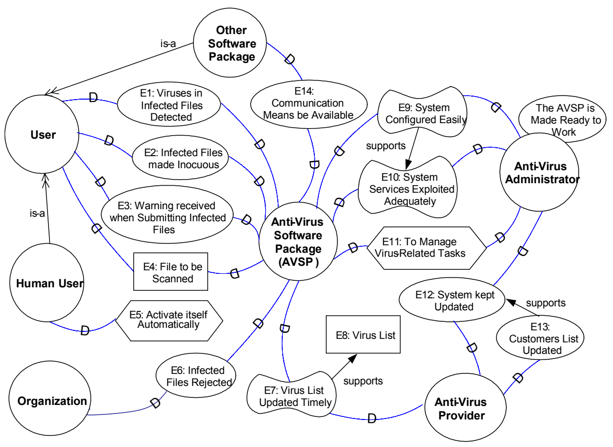

Other Software Package
is-a
User
E1: Viruses in Infected Files Detected
E14: Communication Means be Available
E9: System Configured Easily
The AVSP is Made Ready to Work
E2: Infected Files made Inocuous
supports
E10: System Services Exploited Adequately
Anti-Virus Administrator
is-a
E3: Warning received when Submitting Infected Files
Anti-Virus Software Package (AVSP)
E11: To Manage Virus-Related Tasks
Human User
E4: File to be Scanned
E12: System kept Updated
supports
E5: Activate itself Automatically
E8: Virus List
E13: Customers List Updated
Organization
E6: Infected Files Rejected
E7: Virus List Updated Timely
supports
Anti-Virus Provider

Figure 12. Part of the resulting socio-technical system.

6. Related Work

Even though there exist some goal-oriented methodologies based on the *i** language, as far as we know, most of them are not as precise as R*i*SD. This is mostly due to the purpose of the model: in our context, *i** SD models are later used for assessing different architectural solutions [11] and therefore it is important to have concrete guidelines about how the model is built. As a remarkable exception, we may find some approaches like [12] in which the intention is to generate UML models from a departing goal-oriented model, and therefore some precise model construction rules are also needed.

As mentioned in section 1, the most relevant work in this area is the Tropos methodology [6, 7]. Its main purpose is to define an *i**-based agent-oriented software development methodology. Tropos supports the whole software development cycle from requirements analysis to implementation proposing an *i** model at each development stage. Furthermore, in [13] some transformations are proposed to refine an early requirements *i** model into an implementation *i** model. However, these transformations do not really guide the SD model construction process itself. We can conclude that Tropos and R*i*SD are two complementary approaches, one focusing in the large-scale software development process and the other in the small-scale model development process.

Another related line of research is that of generation of *i** diagrams from other kind of models, in particular UML models. For example, in [11] it is shown how to create them from use cases. This approach requires therefore these departing UML models to exist, which is not our case.

7. Conclusions

The most relevant contributions of the R*i*SD methodology presented in this paper are in relation with the two drawbacks that we have identified in section 1:

– **Absence of methodology**. R*i*SD provides prescriptive guidance to the modeller reducing then the subjectivity that is inherent in goal-oriented modeling. It consists of two phases, which are decomposed into several activities. Each activity is supported by some rules, criteria, questions and patterns to be considered. Remarkably, we have given accurate hints for identifying and classifying dependencies. On the other hand, iteration and intertwining are recognized in the methodology providing then some necessary flexibility degree.

– **Complexity of the models**. As a result, we have obtained models that are more easily analyzed since there is a well-defined and consistent rationale behind. We have also incorporated traceability with two new constructs, “supports” and “refines”. In addition, we have recognized the need of having clear syntactic conventions to be used consistently. Due to the nature of R*i*SD, the resulting models are kept as small as possible, trying to cope with one of the most important problems in the use of *i**, namely scalability of the models.

With respect to our immediate future work, we aim at defining a similar methodology to guide the construction of SR models, to be integrated with R*i*SD.

Acknowledgements

This work has been done in the framework of the research project UPIC, ref. TIN2004-07461-C02-01, supported by the Spanish Ministerio de Ciencia y Tecnología. Some authors have grants that partially support their work: C. Ayala, by the Catalan government Generalitat de Catalunya; C. Cares, by the MECE-SUP FRO0105 Project of the of Chilean government; and G. Grau, by a UPC research scholarship.

8. References

[1] E. Yu, J. Mylopoulos. “Understanding "Why" in Software Process Modelling, Analysis, and Design”. *Proceedings of the 16th International Conference on Software Engineering, 16th International Conference on Software Engineering* (ICSE’94) Sorrento, Italy, 1994. pp. 159-168.

[2] A. van Lamsweerde. “Goal-Oriented Requirements Engineering: A Guided Tour”. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, (RE´01), Toronto, Canada, 2001. pp. 249.

[3] E. Yu. “Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering”. *Proceedings of the 3rd IEEE Int. Symposium on Requirements Engineering*, (RE'97), 1997, Washington, USA. pp. 226-235.

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

[5] C. Ayala, C. Cares, J.P. Carvallo, G. Grau, M. Haya, G. Salazar, X. Franch, E. Mayol, C. Quer. “A Comparative Analysis of *i**-Based Goal-Oriented Modeling Languages”. In *Procs. International Workshop on Agent-Oriented Software Development Methodology (AOSDM’2005) at the 7th International Conference on Software Engineering and Knowledge Engineering,* 2005.

[6] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, P. Traverso. “Specifying and analizing early requirements in Tropos”. *Requirements Engineering Journal*, 9 (2), 2004, pp. 132-150.

[7] Castro, J.; Kolp, M.; Mylopoulos, J. “A Requirements-Driven Development Methodology”. *Proceedings of the 13th International Conference on Advanced Information Systems Engineering* (CAiSE’01), Interlaken, Switzerland, 2001, pp. 108-123.

[8] M. Kolp, P. Giorgini, J. Mylopoulos. « Organizational Patterns for Early Requirements Analysis”. *Proceedings of the 15th International Conference on Advanced Information Systems Engineering* (CAiSE’03), Klagenfurt/Velden, Austria, pp. 617-632.

[9] OME3 page, http://www.cs.toronto.edu/km/ome, last accessed April 2005.

[10] N. Maiden, P. Pavan, A. Gizikis, O. Clause, H. Kim, X. Zhu. “Integrating Decision-Making Techniques into Requirements Engineering”. *Proceedings of the 8th International Workshop on Requirements Engineering: Foundation for Software Quality* (REFSQ’02), Essen, Germany.

[11] V. Santander, J. Castro. “Deriving Use Cases from Organizational Modeling”. In *Proceedings of the 10th International Conference on Requirements Engineering* (RE’02), Essen, Germany, 2002. pp. 32-42.

[12] H. Estrada, A. Martínez, O. Pastor. “Goal-based business modeling oriented towards late requirements generation”. *Proceedings of the 22nd International Conference on Conceptual Modelling* (ER), Chicago (USA), 2003. pp. 277-290.

[13] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. “Modelling early requirements in Tropos: a transformation based approach”. *Proceedings of the Agent-Oriented Software Engineering* (AOSE’01). LNCS 2222. Springer-Verlag, Montreal, Canada, May 2002, pp. 151–168.