Using *i** in Requirements Projects: Some Experiences and Lessons

Neil Maiden, Sara Jones, Cornelius Ncube and James Lockerbie

Centre for HCI Design
City University, London

Introduction

The *i** approach has been available in research communities for more than 10 years, but it has not been applied widely in industrial requirements projects. This is in spite of undoubted strengths, which include a simple but formal and stable semantics, a graphical modelling notation that is simple to use, models that are amenable to computational analysis, and applicability in both agent-oriented and goal-oriented requirements methods. Furthermore, *i**'s capabilities to bridge the gap between organisational, socio-technical and software systems, by modelling goal-based dependencies between organisation, work role and software actors, makes it increasingly important in a world in which we depend on computers in many aspects of our everyday lives. Based on these strengths, we sought to integrate the *i** approach into new requirements processes rolled out as part of our transfer of requirements knowledge from research to practice.

This chapter reports three major industrial projects in which we have applied the *i** approach to specify complex socio-technical systems. It outlines the rationale for using *i** in these projects, how *i** was integrated with other requirements modelling and specification techniques, and what happened as a result of applying *i**. The chapter ends with ten lessons that we learned about using *i** effectively in industrial projects, and the benefits that can be gained from its effective use. We believe that these lessons have broader implications for the uptake of requirements modelling techniques.

The remainder of the chapter is in 4 sections. As it assumes a basic knowledge of the *i** approach we do not describe the *i** approach and types of model that are reported in the chapter. Section 2 describes how we integrated *i** into our broader RESCUE requirements process. Section 3 describes 3 European air traffic management projects that *i** and RESCUE were applied to, the *i** models that were produced, and the benefits of these models to the requirements specifications produced in each project. Section 4 reports ten lessons learned that readers can apply to their own requirements processes and projects. The chapter ends with a description of how these lessons have influenced our own research agenda.

*i** in the RESCUE Process

The RESCUE process was developed in response to a request from Eurocontrol, the organisation overseeing European air space, to deliver new and more effective processes for discovering and specifying requirements for new air traffic management systems (Maiden et al. 2003). Such systems are socio-technical systems that introduce new technologies to support the redesign of human work – typically that of air traffic controllers. As such RESCUE needed to acquire, describe, model and analyse requirements on organisations, on work undertaken by individuals who fulfil roles in these organisations, and software systems that will bring about changes in the work. This need to model and specify socio-technical systems was the principal reason for selecting *i** to be an important component of RESCUE.

RESCUE supports a concurrent engineering process in which different modelling and analysis processes take place in parallel. The concurrent processes are structured into 4 streams shown in Figure 1.

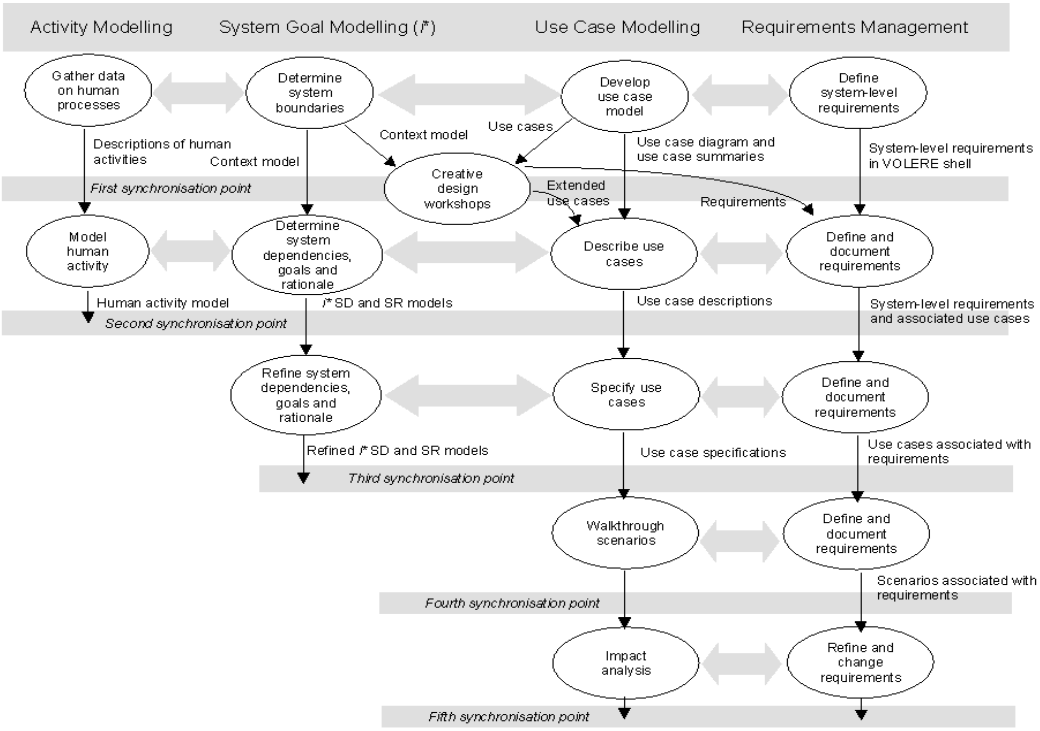

Figure 1. The RESCUE process structure – activity modeling ends after the synchronization stage at stage 2, system modeling after the synchronization stage at stage 3, and scenario-driven walkthroughs and modeling requirements after synchronization checks at stage 5. *i modeling is part of the system goal modeling stream**

Each stream has a unique and specific purpose in the specification of a socio-technical system:

1. Human activity modelling provides an understanding of how people work, in order to baseline possible changes to it (Vicente 1999);

2. System modelling enables the team to model the future system boundaries, actor dependencies and most important system goals using the *i** approach (Yu & Mylopoulos 1994);
3. Use case modelling and scenario-driven walkthroughs enable the team to communicate more effectively with stakeholders and acquire complete, precise and testable requirements from them (Maiden 2004);
4. Managing requirements enables the team to handle the outcomes of the other 3 streams effectively as well as impose quality checks on all aspects of the requirements document (Robertson & Robertson 1999).

Sub-processes during these 4 streams are co-ordinated using 5 synchronisation stages that provide the project team with different perspectives with which to analyse system boundaries, goals and scenarios. The 5 key synchronisation points are at the ends of RESCUE's 5 stages, implemented as one or more workshops with deliverables to be signed off by stakeholder representatives:

1. The boundaries point, where the team establishes first-cut system boundaries and undertakes creative thinking to investigate these boundaries;
2. The work allocation point, where the team allocate functions between actors according to boundaries, and describe interaction and dependencies between these actors;
3. The generation point, where required actor goals, tasks and resources are elaborated and modelled, and scenarios are generated;
4. The coverage point, where stakeholders have walked through scenarios discover and express all requirements so that they are testable;
5. The consequences point, where stakeholders undertake walkthroughs of the scenarios and system models to explore impacts of implementing the system as specified on its environment.

The synchronisation checks applied at these 5 points are designed using a RESCUE meta-model of human activity, use case and *i** modelling concepts constructed specifically to design the synchronisation checks. You can read more about these synchronization stages and their effectiveness in Maiden et al. (2004a).

The next section describes how we applied RESCUE and the *i** approach in 3 different air traffic management projects.
.

Three RESCUE Projects

This section reports 3 previous air traffic management projects in which *i** was applied to model and analyse requirements for new socio-technical systems – CORA-2, EASM and DMAN. Each project is described, the use of *i** modelling is reported, and example *i** models from each project are described and reviewed as a basis for the lessons learned in the remainder of the chapter.

*i** Modelling in the CORA-2 Project

In 2001 we worked with Eurocontrol to design and implement RESCUE to discover stakeholder requirements for CORA-2 (Conflict Resolution Assistant), a system to provide computerised assistance to air traffic controllers to resolve potential conflicts between en-route aircraft. The project team used 3 creativity workshops to discover new concepts and requirements for CORA-2 (Maiden et al. 2004b), ART-SCENE scenario walkthroughs to discover missing requirements for the system (Mavin & Maiden 2003), and *i** modelling to understand the relationships between the CORA-2 system and other systems and human actors in the air traffic control tower.

CORA-2 was the first project to which RESCUE was applied, and the 5 synchronization stages described in the previous section were not implemented. Instead the *i** models were used to discover important requirements on the entire CORA-2 system, and to review system boundaries. Both *i** Strategic Dependency (SD) and Strategic Rationale (SR) models were produced. One analyst produced both without explicit guidelines from RESCUE, using published material on *i** (e.g. Yu & Mylopoulos 1994) to develop each model.

The final SD model is shown in Figure 2. It specifies 8 actors and 24 dependencies between these 8 actors. It specifies the *CORA-2* software system (in the centre), adjacent software systems such as the *flight data processing system* and *conflict detector*, and human roles that are redesigned such as *controllers* who are using the *CORA-2* software. The model has some interesting properties. All dependencies are goal-type dependencies, such as *controller depends upon CORA-2 to implement chosen resolution*, and resource-type dependencies such as *CORA-2 depends on conflict detector to have detected conflicts*. During retrospective questioning the analyst revealed that the resource-type dependencies were derived from direct translation of data flows from an informal context diagram of the CORA-2 system into actor dependencies. The presence of goal dependencies was indicative of the strong focus on functional rather than non-functional requirements during that phase of CORA-2 requirements modelling.

The second feature of the SD model was its centralizing structure. All but 2 of the 24 dependencies include the new CORA-2 software system as either a depender or a dependee. No dependencies between other software-based systems upon which CORA-2 depended were modelled as we might have expected when specifying a socio-technical system. One reason for this was the narrow view of CORA-2 held by some members of the project team. CORA-2 had originally been conceived as a software component without consideration of the wider socio-technical redesign of controller work. The introduction of *i** modelling challenged this view in the project, but a change in this view did not manifest itself in the resulting models. The third feature of this SD model was the failure to separate out the different human roles of controllers who would work with the CORA-2 software into distinct and separate actors on the model. The analysts designed the CORA-2 software system to be used in different ways by pairs of controllers – planning controllers and tactical controllers – who work together to control the air traffic in a sector. Clearly these two controller roles depend upon on each other to resolve aircraft conflicts safely and efficiently, however these important dependencies were not

modelled explicitly in *i** for the CORA-2 system. Again, one reason for this was the narrow, software-oriented view of CORA-2 at the beginning of the project. *i** provided important constructs for directly challenging this view, but again changes to the view were not manifest in the models that resulted,

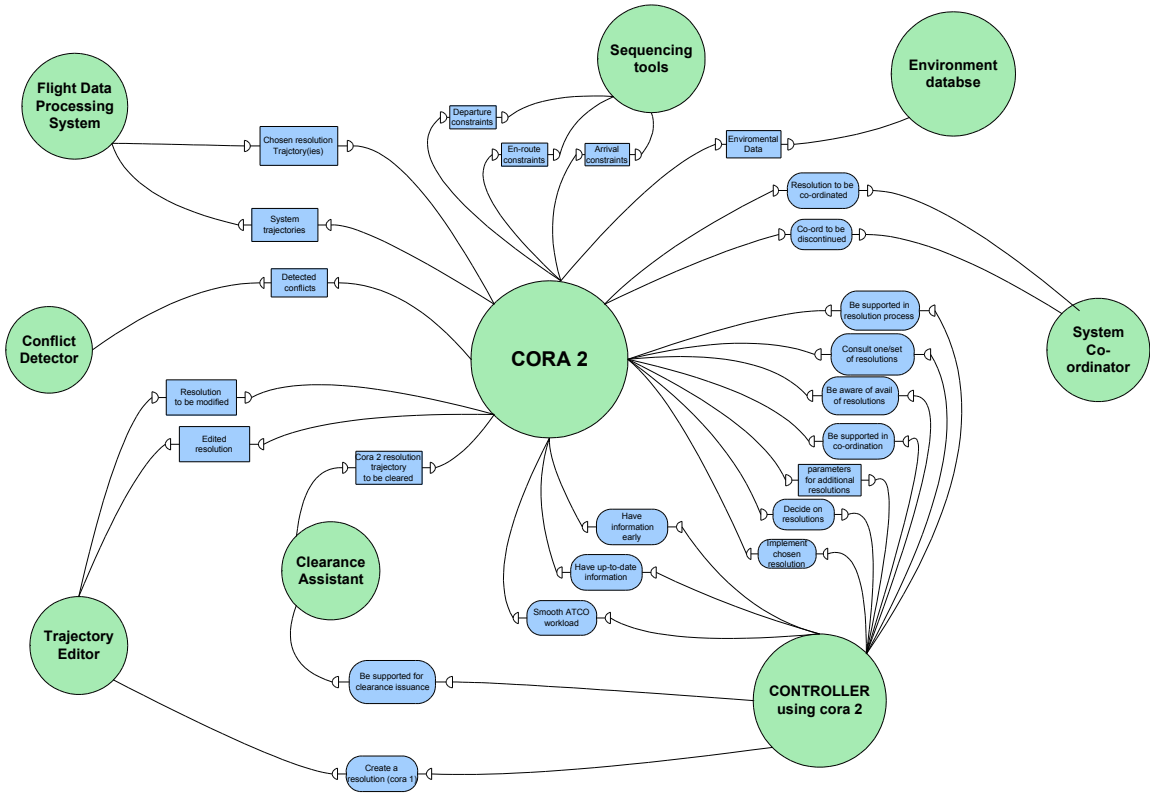

Figure 2. The CORA-2 SD model, showing the new *i software actor in the middle of the diagram, redesign controller actors in the bottom right of the diagram, and adjacent systems.**

Although the model was accepted and proved to be useful during the CORA-2 requirements process, it had properties that suggested potential extensions to RESCUE that improve *i** system modelling. These included guidance to separate out human roles into different actors in a system model, and model dependencies as soft goal and goal dependencies rather than resource dependencies that approximate to data flows between actors. These guidelines were included in subsequent versions of RESCUE and are explored in the lessons.

*i** Modelling in the EASM Project

In 2004 we worked with the UK's National Air Traffic Services (NATS) to apply the latest version of RESCUE to the specification of the UK's Enhanced Air Space Management system (EASM). EASM is a new socio-technical system that is, in essence, a scheduling system that will enable more effective, longer-term planning of UK airspace use. The project team again used one creativity workshop to discover new concepts and requirements for EASM, ART-SCENE scenario walkthroughs to discover missing requirements for the system, and context and *i** modelling of the new EASM system to understand the actors and boundaries of the EASM system. A later version of RESCUE

provided more concrete guidance for discovering actors, modeling goal and soft goal dependencies, and specifying dependencies more precisely.

One analyst produced the first version of the SD model shown in Figure 3 using RESCUE guidelines. The model was developed in two phases. In the first, a workshop was held with key stakeholders to discover actor dependencies that were listed in a dependency table described in Lesson 3. In the second phase, one analyst used the table to construct a graphical *i** SD model that was analysed and presented to stakeholders for validation and analysis. Analysis of this graphical model revealed to the team's surprise, for the first time, that EASM had been modelled as three separate systems that had no dependencies between them. The first system is associated to the *AMC actor* and is shown in the top right-hand side of Figure 3. These 5 inter-dependent actors had no strategic dependencies on other modelled actors. The second showed two dependencies between the *ATC* and *FDP system* actors, but again strategic dependencies on other EASM system actors. The third system, expressed in the rest of the model in Figure 3, related to dependencies based on use of the new *ASM support software* system.

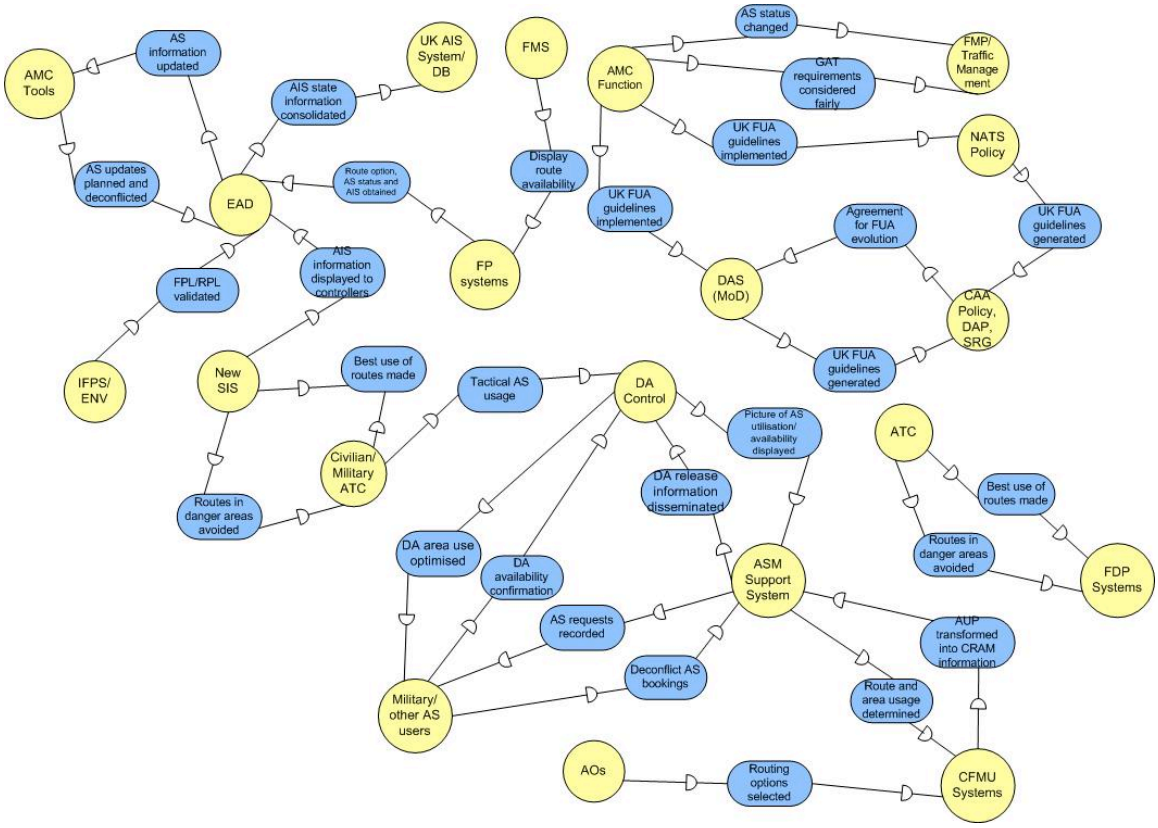

Figure 3. The first-cut EASM SD model, showing three separate systems with no strategic dependencies between them.

These emergent model features were explored through further analysis, in particular to discover whether EASM was indeed independent 3 separate strategic systems as modelled. The answer was no, and further development of the model took place. The final version of the EASM SD model incorporated several enhancements. In response to missing strategic dependencies, five new goal dependencies between *ASM function* (an

actor changed from the first version of the model) and *ASM support system* were added to reflect new relationships identified in other *EASM system models*. Similarly new dependencies were also added between the *FDP system*, *ASM Support System* and *EAD*, and between *CFMU systems* and the *ASM function*. Elsewhere in the model, actor roles were clarified, for example *ATC* evolved into *civil and military ATC*, which was important civil and military controllers had different goals and soft goals on the EASM system. The final model is shown in Figure 4.

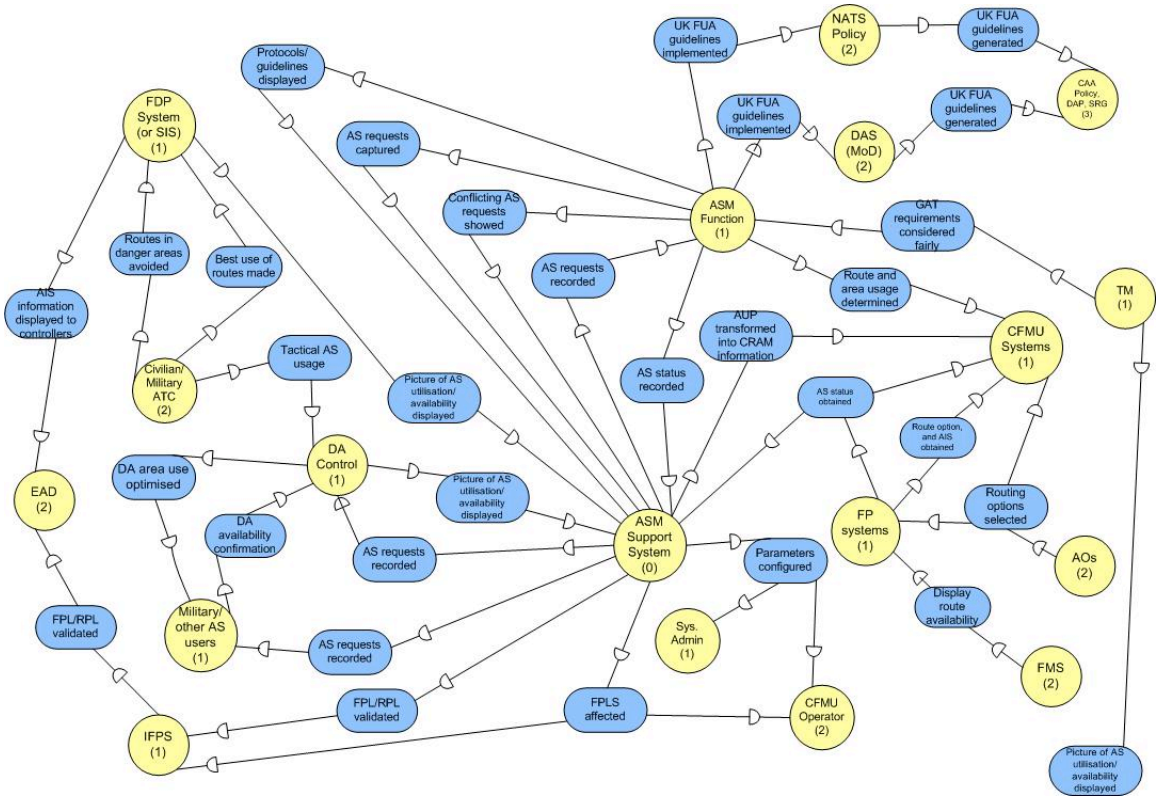

Figure 4. The revised EASM SD model, extended to model important but missing dependencies identified from graphical modelling of actor dependencies

To conclude, analysing and reviewing the first version of the EASM SD model identified incorrect and missing dependencies that were not detected from the initial dependency tables. The SD model was evolved in several iterations from the first to the final version. The simple *i** SD notation facilitated evolution and change of the model.

*i** Modelling in the DMAN Project

In 2003 we also worked with the NATS to apply a version of RESCUE to the specification of DMAN, a socio-technical system for scheduling and managing the departure of aircraft from major European airports such as Heathrow and Charles de Gaulle. A requirements team that included engineers from UK and French air traffic service providers modelled the DMAN system and requirements. The project team applied human activity modelling to understand the current system context, one creativity workshop to discover new concepts and requirements (Maiden et al. 2004b), ART-SCENE scenario walkthroughs to discover missing requirements for the system (Maiden

& Robertson 2005), and context and *i** modelling to understand the actors and boundaries of the DMAN system (Maiden et al. 2004a). Once again, a later version of RESCUE provided more concrete guidance for discovering actors, model goal and soft goal dependencies, and specifying the dependencies more precisely. The team developed one SD model and one complex SR model of the future DMAN system.

Figure 5 shows the SD model for the DMAN system. The SD model specified 15 actors with 46 dependencies between these 15 actors. It specifies other systems that either depend on or are depended on by DMAN (e.g. the *TACT* and *A-SMGCS* systems that are not elaborated on in this paper), and human roles that depend on DMAN to do their work (e.g. *Runway ATCO* and *Departure Clearance ATCO*). For example, the SD model specifies that *DMAN depends on TACT to achieve the goal CTOT and slot messages updated*, and *A-SMGCS depends on DMAN to undertake the task update taxi time estimates*. Likewise, *DMAN depends on the Tower Departure Sequencer ATCO to have the departure sequence manual update*, and the *Departure Clearance ATCO depends on DMAN to achieve the soft goal workload not increased.*

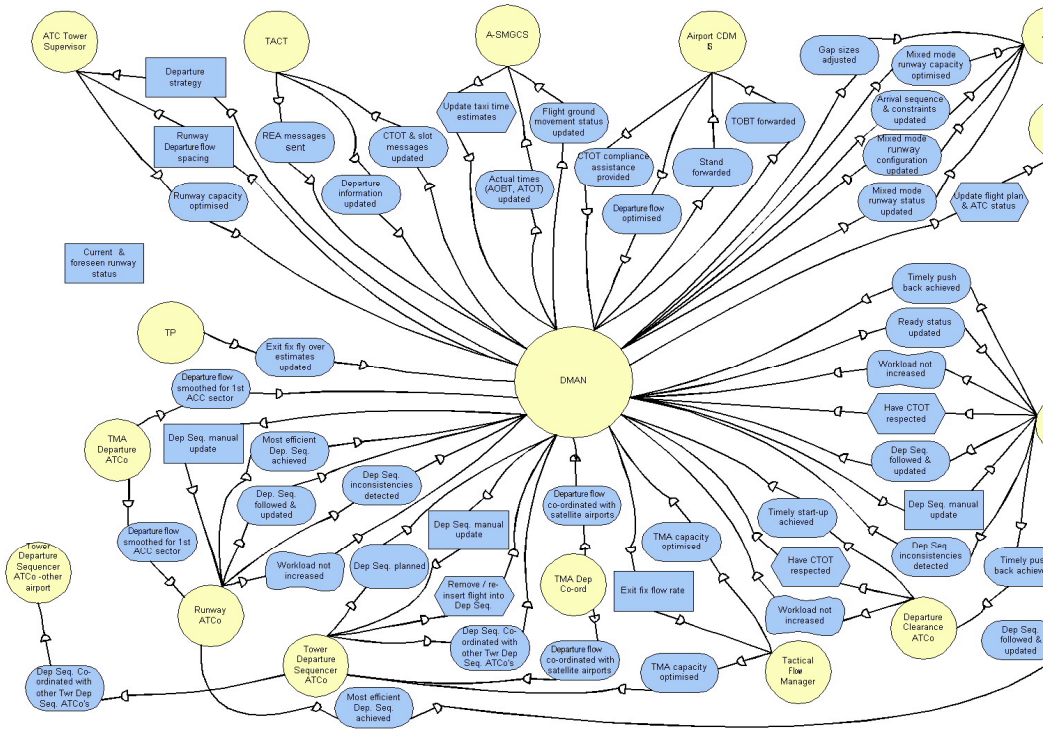

Figure 5. Part of the SD Model for DMAN, showing most DMAN actors and actor dependencies

The DMAN SD model warrants some comparison with the earlier CORA-2 SD model. Firstly, the DMAN SD model includes more goal and less resource dependencies, reflecting an increased goal modelling effort. Secondly, in spite of the RESCUE guidelines, the SD model only includes one dependent soft goal – *workload not increased*. This inclusion reflected a concern that new departure scheduling technologies should not increase workload overall. This is not to say that other soft goals, such as *performance* and *reliability,* were not important. However, the analysts did not consider these soft goals to be as important to decision-making about the design of DMAN as

increased workload, and hence these were not included in the model. Thirdly, the model reflects the specification of a socio-technical rather than software system, through the inclusion of more human actor roles than the CORA-2 system, and more work-related dependencies between these human actor roles.

Figure 6 shows part of the DMAN SR model for the *Runway ATCO* actor. This actor undertakes one major task – *control flight around the runway* – that is decomposed into other tasks such as *issue line-up clearance* and *issue take-off clearance*. The former task can be further decomposed into sub-tasks and sub-goals which, if undertaken and achieved, contribute negatively to the achievement of the most important soft goal – that *workload should not be increased*. Furthermore, to do the *issue line-up clearance* task, the Runway ATCO depends on the resource *flight information* from the electronic flight strip. This demonstrates how the SR model was used to refine one important soft goal – *workload not increased* – to explore in more detail contributions of different new work tasks to the achievement or otherwise of that soft goal.

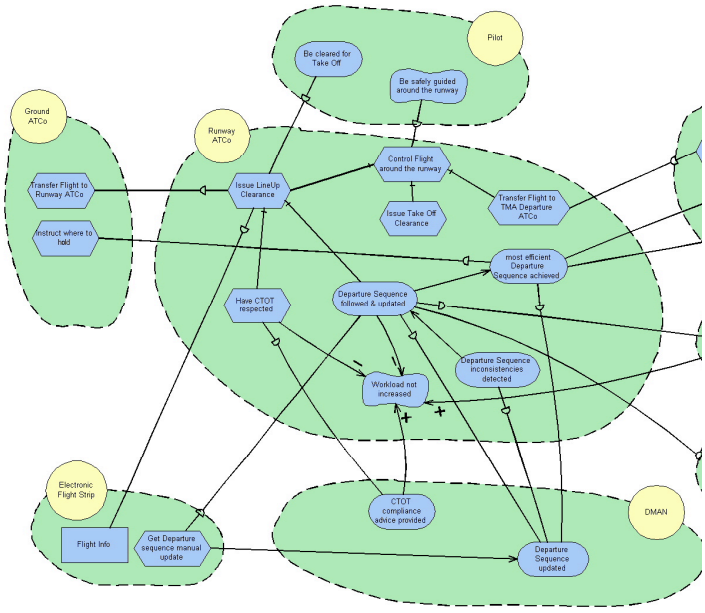

Figure 6. Part of a draft SR model for DMAN, showing elements for the Runway ATCO actor and its dependency link with other actors.

To conclude, *i** modelling of DMAN was instrumental in specification of DMAN as a socio-technical rather than software system, in contrast to earlier experiences with CORA-2. The SD model identified different types of actor and adjacent system upon which requirements were specified in the DMAN requirements specification. SR models were important for modelling actor behaviour and exploring important contributions to the satisfaction of important soft goals. Later in the project, the DMAN team used the DMAN SD model to generate candidate requirements statements directly, as we reported in one of following lessons.

*i** Modelling Experiences: Conclusions

*i** models were successfully developed and applied in the CORA-2, EASM and DMAN projects. We also successfully applied *i** in GOMOSCE, a project funded by Dstl, to explore how to make goal-based trade-offs about architectures for network-enabled capabilities in military domains. This and the air traffic management projects led us draw ten lessons that were applied successively in these projects. The remainder of this chapter summarizes these lessons and reports them to be learned and applied by others.

Lessons Learned

*i** modelling in the 3 reported air traffic management system projects led the RESCUE process team to draw important lessons that have led to revisions of RESCUE. We divide these lessons into 2 types – lessons that describe how to apply *i** modelling in real-world requirements projects, and lessons that report the benefits that can accrue from *i** system modelling. Firstly we report lessons about how to apply *i**, and in particular how to get started and produce some first-cut *i** models, which in our experience is more difficult than it might at first appear.

1. Kick-start *i** Modelling with Context Modelling

One problem that the published *i** approach does not address directly is where to start *i** modelling. System goal modelling can be undertaken in many different contexts, to explore incremental work change, to drive product innovations, or to inform architectural trade-offs. *i** models developed for each type of analysis will be different. Hence, it is difficult to say, a priori, where to start *i** modelling. Furthermore the focus on modelling rigor imposed by *i** semantics often makes it difficult to find the most important actors and dependencies from which to produce a first-cut SD model. Our solution is to produce context models beforehand, then to use this context model to guide the development of the first-cut SD model.

A context diagram is, in essence, a data flow diagram. In its simplest form the system that is to be designed and developed is represented by a circle, with the name of that system written in the circle. Actors, which can be human roles, other systems and organisations with which the new system will interact, are written as outside of the circle. Interaction is represented as a data flow, the flow of exchange of data, either between an actor and the new system, or between actors. The arrowhead indicates the direction of the flow of the information, and arrows are labelled to describe the information that is flowing.

In RESCUE we have extended the context diagram by defining different types of system boundary. The reason is simple. When you design a complex system, some things are clearly within your design remit and others are clearly outside of it. For example, in a simple automatic bank teller system, the design of the teller system is within the design remit of the banking IT department - that is it can redesign the teller system. It also has the remit to design how a bank teller, its employee, refills it every morning. Other behaviour is clearly beyond its design remit - for example the design of bank notes - that the bank must treat as a domain assumption during the redesign. However, in socio-technical systems there is a grey area of things that you cannot directly redesign, but you

can seek to influence the behaviour of using your design. The obvious example in this domain is the client. Whilst a bank cannot make a client behave in a certain way (at least not without breaking the law), they can seek to influence how a client uses an automatic bank telling machine by design of the teller machine, such as additional services, improved accessibility, and lower charge costs.

In RESCUE we explore this grey area by defining 4 types of system:

1. The technological systems, expressed in terms of software and hardware actors;
2. The redesigned work system, expressed primarily in terms of human actors;
3. Other hardware, software and people systems that are directly influenced by the redesign of the new system;
4. External systems beyond the boundaries of the socio-technical system.

What you get by producing the context diagram first is a clearer understanding of the system itself prior to producing the more complex SD model. In short, the context diagram is a prototype SD model.

Figure 7 shows a context model developed retrospectively for the CORA-2 system shown earlier in Figure 2. In the centre circle is the new *CORA-2 software system*. At the next level are 2 human roles - *planner controller* and *tactical controller* that the project team in CORA-2 was explicitly and deliberately redesigning - it was able to redesign the work of the controllers to keep it line with the new *CORA-2 software system*. At the third level, are other software systems that, although beyond the design remit of the CORA-2 team, the team sought to influence the design of. These systems included *CORA-1* (the system that detected en-route conflicts that the CORA-2 system resolved), and *TED* (the trajectory editor). The CORA-2 team realized that CORA-2 would be able to compute possible conflict resolutions if the *CORA-1 system* detected the conflicts in certain ways and provided certain information. Therefore, it sought to influence the design or redesign of CORA-1. It did this during the requirements process by generating and specifying requirements on CORA-1 upon which CORA-2 requirements were dependent. At the fourth level are the systems and other actors that are beyond the design remit and influence of the current project.

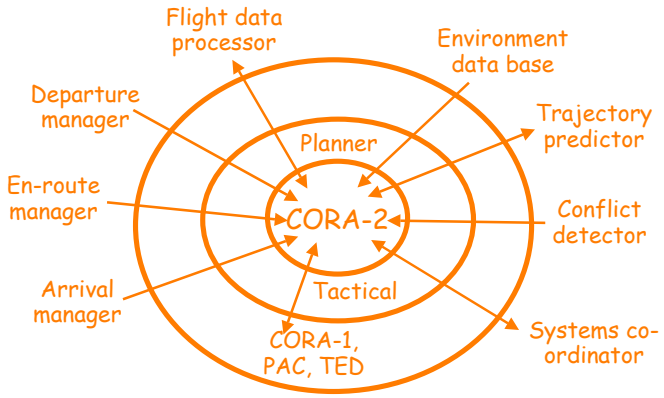

Figure 7. Context model developed retrospectively for the CORA-2 system, showing the new software system, actors whose work is redesigned as part of the project, actors whose behaviour is not within the remit of the project to redesign, but is influenced by the new software and redesigned work, and external systems.

2. Think About Adjacent Systems

Neither context diagramming nor the *i** approach offers strong advice for discovering actors. Therefore, to supplement the use of context diagrams for getting started, we draw on the notion of adjacent systems from Robertson & Robertson (1999). An adjacent system is a system that is related to the system that you are specifying, and upon which your system or product might be dependent, either for information or services. Adjacent systems are typically one class of dependent actors in an SD model. However, Robertson & Robertson (1999) go one step further and characterise different types of adjacent system that are useful for discovering and modelling actors in SD models. These 3 types are active, autonomous and co-operative adjacent systems, and each type has implications for *i** SD modelling. Each is described in turn.

An active adjacent system behaves dynamically, and interacts with or participates in the new system. Active adjacent systems are usually roles that are fulfilled by humans. They initiate events, and when they do, have some objective in mind. They can work with the product by exchanging data and other signals, until their objective is satisfied. We can predict the behaviour of an autonomous agent within reason, and you can expect it to respond to signals from your work. As long as there is some perceived benefit to the adjacent system, it will obey (more or less) instructions from the work. Furthermore an active adjacent system is likely to respond in a suitably short time, not to delay the transaction any more than necessary. This has implications for types of *i** dependencies that will exist between actors. Active adjacent actors in SD models often fulfil depender roles and have goal and soft goal dependencies on other actors.

An autonomous adjacent system is some external body, such as another company, a department, or a customer who is not directly interacting with your work. It acts independently of the work or system being studied. Autonomous adjacent systems communicate through one-way data flows. Again this has implications for types of *i** dependencies that will exist between actors. Given the relatively weak coupling with an autonomous adjacent system, actors will depend on such systems for resources that might contribute to the achievement of a goal or soft goal, but will not depend directly on such systems for goal and soft goal achievement. Hence, expect to model resource-type dependencies when an autonomous adjacent system is the dependee.

A co-operative adjacent system can be relied on to behave predictably when called upon. In other words, they co-operate with the product or system to bring about some desired outcome. This is almost always done by the means of a simple request-response dialogue. It is unlikely that you will need, or want, to change the interfaces with the co-operative system. As co-operative systems are black boxes, their services are stable, and there is rarely much to be gained from trying to change them. Again this informs the types of dependencies that might be modelled for a co-operative adjacent system. Expect such systems to fulfil dependee roles that enable depender actors to achieve soft goals and attain goals as well as undertake tasks and obtain resources.

We can return to the example CORA-2 system shown in Figures 2 and 7 to demonstrate different types of adjacent system. Active systems adjacent to the *CORA-2 software system* are the *planner* and *tactical controller* roles, undertaken by human beings and initiating behaviour to resolve conflicts through the use of the *CORA-2 software system*. The *flight data processing system* is an autonomous adjacent system. Its behaviour is independent of the *CORA-2 software system*, and dependencies between these two systems are expressed as resource dependencies, such as for information about *trajectories*. Finally the *CORA-1 software system* is a co-operative adjacent system, because the *CORA-2 software system* relies on it to behave predictably when called upon to detect potential conflicts. Our assumption when specifying the wider CORA-2 system was that the CORA-1 system would provide the CORA-2 system with a stable service.

3. Use Dependency Lists to Discover Dependencies

Another problem when developing an *i** SD model is how to establish all of the possible dependencies. Analysts in our projects found it difficult to know how to ensure that all possible dependencies had been considered. Our solution, implemented in RESCUE, is very simple. Before attempting to draw a first-cut graphical SD model, list out the possible dependencies between actor pairs in a tabular notation. We were surprised how effective this proved to be.

A SD model dependency link is a link between two actors and indicates that one actor depends on another for something that is essential to the former actor. The depending actor is called the depender, the actor who is depended upon the dependee and the process element around which the dependency. Although *i** provides a useful graphical notation, it is perhaps less effective for discovering and expressing actor dependencies in the first place. This is where the tabular notation comes in.

Prior to producing a SD model, RESCUE analysts run workshops with stakeholders that seek to complete the table showed in Table 1. An analyst encourages the stakeholders to consider each possible pair of actors, often identified with context diagrams reported in Lesson 1, to discover possible dependencies between them. The tabular representation affords discovery of dependencies in several ways. Firstly, the sequential order of the table encourages a more systematic approach, leading to a more complete specification of dependencies. Contrast this with difficulties that often arise when trying to graph-walk an *i** model to consider all possible dependencies. Secondly, the row-at-a-time structure of the list encourages all stakeholders to consider one dependency at a time, thus improving the accuracy of the specification and obtaining more agreement about the dependencies that are specified. Thirdly, and perhaps most importantly, the column structure of the table is, we believe, more cognitively natural to stakeholders when reasoning about and expressing dependencies. Whereas the graphical notation of an SD model places the dependency between the two actors, the RESCUE table places the dependency at the end, thus facilitating communication of and experimentation about dependencies. To see what we mean, try reading out load the first dependency from the table: *TACT depends on DMAN to have REA messages sent*. This format is easy to recall and reason about.

ID	Depender		Dependee		Dependency
1	TACT	depends on	DMAN	to	REA messages sent
2	TMA Dep. ATCo	depends on	Runway ATCo	to	Departure flow smoothed for 1st ACC sector
3	Airport CDMs	depends on	DMAN	to	Stand forwarded
4	….	depends on		to	
		depends on		to	
		depends on		to	

Table 1. A RESCUE dependency table, partially completed from dependencies shown in the DMAN SD model depicted in Figure 5.

In conclusion, our experiences have shown that one effective way to produce *i** SD models is to, well, hide the *i** SD model. Modelling syntax and semantics are often distractions for most stakeholders who have not been trained in system modelling approaches, even in highly technical domains such as air traffic management. The goal of SD modelling is to model strategic dependencies between strategic actors. Simple tables implemented in familiar technologies such as Word concentrate stakeholders on these dependencies and avoid possible distractions associated with graphical *i** models.

4. Get Your Training in First

This lesson might seem very obvious, but is important for some less obvious reasons. One of our principal findings from the reported projects is that it is easy to use *i** badly, but less so to use it effectively. There are 2 main reasons for this Firstly, the *i** approach is substantially different to most other modelling approaches that analysts and stakeholders have been exposed to. The focus in SD models on actors and dependencies in a socio-technical system contrasts with existing methods such as the UML and its simple notations such as use case and class diagrams to model a system that is primarily software-oriented. Whereas UML specifications describe what a system shall do, *i** models also specify why it shall do this with cross-references to goals and soft goals. Secondly, although *i** only has 5 process elements (actors, goals, soft goals, tasks and resources), these elements can be used in combination in many different ways in SD and SR models, and this gives rise to modelling problems that only emerge through practice with *i**. Whilst some of the other lessons in this chapter are in direct response to problems that we have observed, others need to be experienced by analysts and stakeholders to be understood and avoided in the future. In short, you need to let people learn from their *i** modelling mistakes.

To minimize the effect of these mistakes on real projects, allow a substantial period of training for relevant staff before a RESCUE or *i** project starts. On most RESCUE projects we timetable 3 full days of *i** training for all analysts, composed in part of short lectures with notes, but mostly of group modelling and critiquing exercises of examples related to the project domain.

5. Heuristics to Choose Process Elements

Choosing the process element to include in an SD actor dependency can be more difficult than it first appears. Should the analyst model a dependency as a resource that A depends on B for, as a task-type dependency describing the task that A needs the resource from B for, or as a goal-type or soft goal-type dependency to depict the goal that is attained or soft goal that is achieved by undertaking the task with the resource? To try and answer

these questions in RESCUE, we developed the following three heuristics that our analysts and stakeholders have applied successfully to avoid difficulties when choosing to model dependencies and the type of dependency to be modeled:

1. **Discover the real dependency**: it can be difficult to choose between several associated, on the surface valid dependencies. Often process elements in a model are associated. Resources are consumed in tasks that are completed to attain goals and achieve soft goals. So is the dependency a resource, task, goal or soft goal dependency? We apply a simple test based on the exclusivity of the dependency. If the depender actor depends on dependee only to obtain a resource, but can still undertake the associated task without the dependee actor (e.g. through other sources of resource), then the dependency is a resource-type dependency. If, however, the depender actor cannot undertake the task but still attain the associated goal or achieve the associated soft goal without the dependee actor (e.g. by undertaking task on own), then the dependency is a task-type dependency. Otherwise, only if the depender cannot attain the goal or achieve the soft goal without the dependee actor do we model a goal or soft goal-type dependency;
2. **Task dependencies**: task dependencies can be difficult to specify. Therefore assume that, in a task-type dependency, it is the depender who initiates the task. One option is to avoid task-type dependencies in SD models. If you have task-type dependencies, ask why the actors want to undertake these tasks, to turn them into goal-type and soft goal-type dependencies. Note that this will not always be possible, but be prepared to challenge dependencies;
3. **Naming dependencies**: it is very important to be careful when naming goals and soft goals, in order to ensure that the notion of the goal or soft goal will be understood by other people. Use the following guidelines. The description of a goal should describe a desirable state <desirable state>, for example *ticket purchased, car repaired.* The wording of a soft goal should describe some properties or constraints on that state <desirable state> <adjective | adverb>, for example *ticket purchased quickly, car repaired cheaply*. Tasks should be specified using active verbs describing how something is done <do task>, for example *purchase tickets online*. Finally, resources are described using a noun <resource>, for example *conflict information, 5 seconds, ticket.*

6. Cost-Effective Use of SR Modelling

So far our lessons have said very little about how to develop and apply *i** SR models. One reason for this is the considerable effort needed to produce a complete SR model for even a moderately complex system, as indicated by the scale of the SR models that were produced by one analyst for the DMAN project, see Figure 6. As well as the intellectual effort needed to develop each actor model, the resulting SR models were large and somewhat difficult to manage, especially given the rudimentary tool support available. Furthermore, the benefits obtained from SR modelling within RESCUE were limited, due primarily to the parallel development of use case specifications that represented concepts similar to those found in SR models. Our experiences suggest that SR modelling should be used more selectively in future projects.

One important role of SR models is to describe how actors will behave to achieve and attain their own and collective soft goals and goals respectively. This behaviour is normally expressed as tasks and sub-tasks that consume and produce resources, and are means to attain goals and achieve soft goals. However, these soft goal/goal, task and resource structures are also specified, if perhaps less explicitly, in use case normal courses. Moreover use cases expressed in structured English are often easier to produce, read and manage than SR models, even though they lack some of the expressive power of SR models. Therefore, in RESCUE, we recommend using use case specifications for modelling most actor behaviour.

But SR modelling can still fulfil the following roles in RESCUE projects. Firstly, it can be used prior to use case specification, alongside the use case diagram, to explore important system actors, goals and tasks that can be used to discover and structure use case specifications. The Rational Unified Process (RUP) still offers limited advice in the area of use case discovery, whilst UML use case diagramming notations are weak and open to misinterpretation. Focused use of SR models can complement use case diagrams, and discover goals that behaviour specified in use cases should attain, and coarse-grain tasks undertaken by actors to attain the goals.

Secondly SR modelling should be used to model important semantics that are not represented in use case specifications. In particular use cases have no explicit representation for task/action contributions to soft goals. Modelling such contributions is important if analysts are to understand how behaviour specified in use cases achieves different soft goals, to inform trade-off analysis and other forms of goal-related decision-making. Therefore, construct SR models that are related to single use cases to investigate goal attainment and soft goal achievement, and explore different possible use case specifications through different modelled tasks.

Thirdly, use SR models to explore dependencies that can exist between use cases. The behaviour specified in use cases, which are treated in UML as stove-piped partial behaviour specifications, can often depend on behaviour in other use cases, and SR models can be used to explore these task- and resource-type dependencies. Consider the following example that emerged from the DMAN project (Maiden et al. 2004). The *i** models specified that the *TMA Departure ATCO depends on the Runway ATCO to do the task control flight after takeoff* (referred to here as task T1), which in turn depends on the *Runway ATCO doing the task transfer flight to TMA Departure ATCO* (referred to as task T2). Task T1 maps to action-7 in UC7, and task T2 maps to action-6 in UC13. This reveals an implied dependency between UC7 and UC13, and that action-6 in UC13 shall happen before action-7 in UC7. From this and 5 other similar dependencies, we produced a simple model showing previously un-stated dependencies between DMAN use cases that have important implications for the timing and order of actor behaviour in the future DMAN system.

7. Provide simple-to-use tools

Our experiences reveal that *i** modelling can be challenging. Therefore even simple forms of support can make the difference between the success and failure of *i** in a

requirements project. One obvious area where we can provide assistance is with tool support for *i** graphical modelling. In RESCUE we have provided the REDEPEND modelling tool, which provides systems engineers with *i** modelling and analysis functions, coupled with additional functionality and the reliability of Microsoft's Visio product. REDEPEND is delivered as 2 simple plug-in files for the MS Visio 2003 application. All that an analyst needs to do to work with REDEPEND is to install these files in the correct Visio directory.

Once installed, REDEPEND provides a graphical palette from which systems engineers can drag-and-drop *i** concepts in order to develop SD and SR models. Part of the DMAN SD model, redrawn using the latest version 4.1 of REDEPEND for clarity in this chapter, is shown in Figure 8. REDEPEND provides 2 palettes on the left-hand side, the top one for SD modelling and the bottom one for SR modelling (it has more modelling elements). When an analyst wants to draw a SD or SR model, s/he drags and drops the relevant element from the palette onto the workspace. Associations, such as dependencies, task decompositions and means-ends links, are also dragged and then connected to process elements already on the diagram. Other diagramming advances in version 4.1 (the split palette is one) include colour coding of process elements to improve diagram readability, and a simple 'process element check' feature, which enables an analyst to indicate that a chosen element of a model has already been validated during a model review or walkthrough, so that it need not be returned to unnecessarily later in the review.

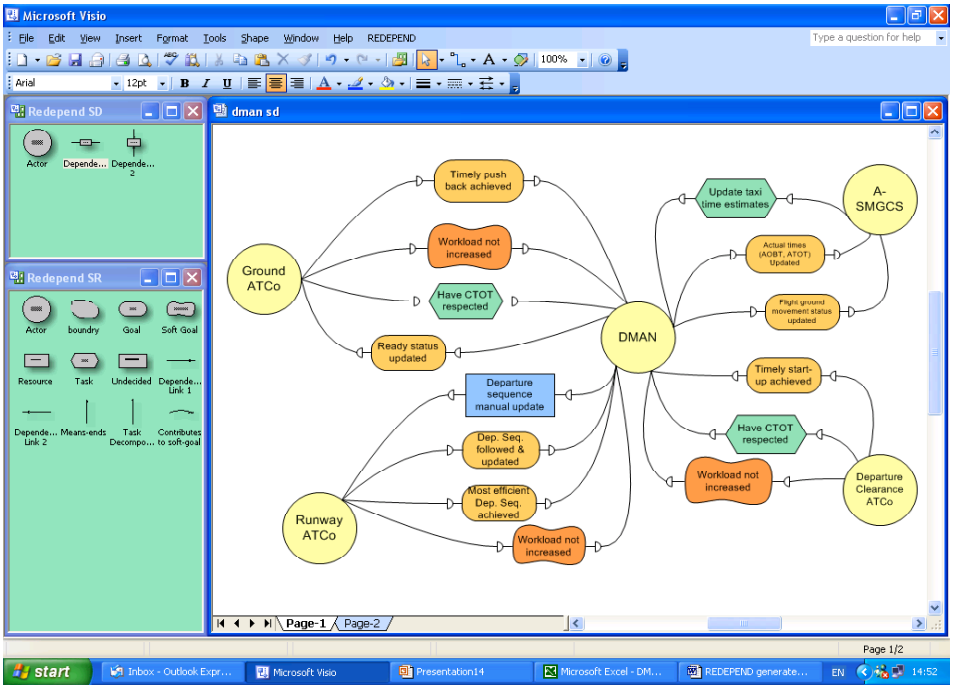

Figure 8. REDEPEND's standard graphical modelling view, showing part of the redrawn SD model from the DMAN project.

Figure 9 shows two other important features of REDEPEND's graphical modelling support. The left-hand side shows how simple right-hand mouse click menus enable the analyst to change dependency types in the same SD model at the click of a button. This is important to encourage exploration and revision of *i** models in the early stages of a

project. The right-hand side of Figure 9 shows a simple feature added to type each actor as within the system boundaries and a new software actor, within the boundaries and a redesigned stakeholder, or an adjacent system outside the system boundaries. This feature is used for automatic requirements generation in Lesson 9.

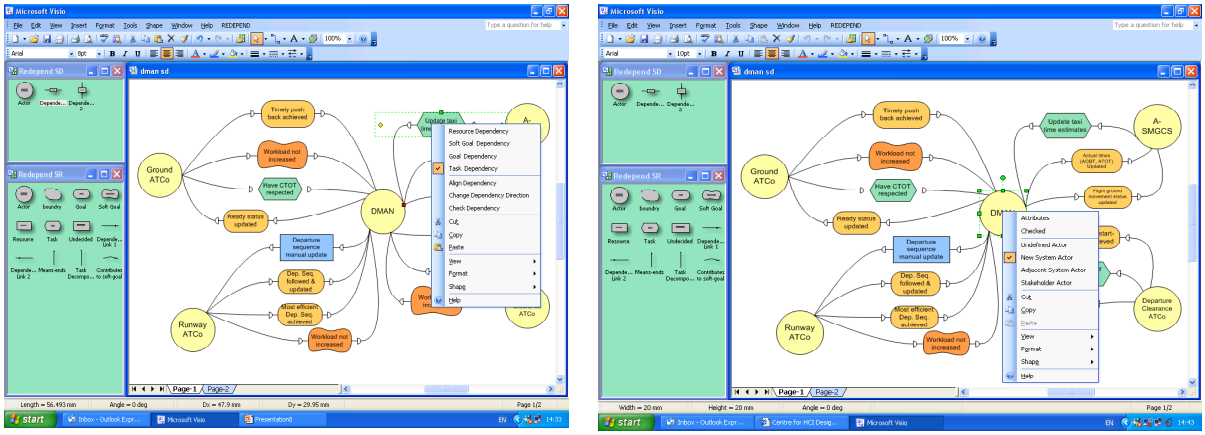

Figure 9. Other REDEPEND graphical modelling features, showing on the left-hand side how to change process element types in a dependency relationship, and on the right-and side more effective element alignment.

So far we have reported lessons that are intended to guide projects to kickstart and use *i** modelling in requirements projects. The remaining three lessons take a different perspective – what benefits can a project derive from using *i** models in requirements processes.

8. SD Models for Exploring System Boundaries

In our projects, SD models have proven to be very useful for thinking about and exploring system boundaries. Whilst context models described in Lesson 1 provide an important first view of a system from which to construct the *i** SD model, boundaries are expressed in terms of data flows between actors on either side of a boundary. In contrast, the SD models themselves refine these boundaries and allow analysts to explore them in terms of goals and soft goals.

In simple terms, a SD model replaces data flows expressed in the context model with goal and soft goal dependencies that actors want to achieve. This provides an important test of a system boundary. If the depender actor in a dependency relationship wants to attain a goal or achieve a soft goal, and the project will test to determine whether the depender actor attains the goal or achieves the soft goal, then the depender actor is part of your socio-technical system. Conversely, if the project is not interested if the depender actor attains the goal or achieves the soft goal, then the actor is not part of your system.

On the surface, this boundary test is stating the obvious to many requirements practitioners, but *i** provides practitioners with several important advantages. Firstly the focus of SD models, on the strategic dependencies between actors, means that the SD model identifies, by default, the actor goals and soft goals that form the boundary test. Secondly, the model expresses the tests in the right form – desirable properties that an actor is seeking to attain or achieve. Alternative modeling representations, from context

models to UML use case models and class models, do not provide this explicit first-order representation of goal and soft goal. Thirdly, the test is simple to apply, and can be applied recursively to actors in the model that are further and further from the new software system that is being introduced. Thus, the test provides a simple-to-use stopping point for *i** SD modeling. Try it – it really works!

9. *i** SD Models to Generate Textual Requirements Statements

There are many model-based specification and analysis approaches reported in the literature to specify requirements (e.g. De Landtsheer et al. 2005). In contrast, most organizations continue to represent requirements textually, both to enable requirements to be reviewed by stakeholders, and to deliver requirements documents that are legally binding on the contractor. Unfortunately, most modelling approaches have not been designed to support the derivation of requirements statements from models or to be used along side textual requirement descriptions. Therefore, in RESCUE, we have extended the REDEPEND software tool to generate candidate requirements automatically from *i** SD models using simple patterns. This approach is reported at length in Maiden et al. (2006).

In RESCUE we designed simple patterns – recurring syntactic and semantic structures in the *i** models – that are applied automatically to any SD model expressed in REDEPEND to generate textual requirement statements. Our patterns are not traditional in the design sense – a solution to a problem in context. Rather each pattern defines one or more desired properties (requirements) on the future system that must be satisfied for the SD model dependency to hold for the future system. As such, the SD model, which has been signed off as complete and correct, informs further discovery and specification of requirements statements.

REDEPEND v4.1 implements 19 different patterns, divided into 2 types. The first 16 are specific to the *i** model dependency, defined in terms of the dependency's process elements (goal, task, resource and soft goal) and the types of depender and dependee actors (new system, adjacent system, stakeholder) based on the different boundaries identified during context modelling. These 16 patterns can be divided into three broad categories – P1-P6 that link the software system actors; P7-P12 that link stakeholders to the new system; and P13-P16, which are independent of these actor types.

Consider the example of pattern P3. REDEPEND applies pattern P3 to generate candidate requirement statements related to each instance of an SD dependency in which a new software actor (e.g. the *CORA-2* or *DMAN software systems*) depends on an adjacent system actor to achieve a goal. The pattern specifies the need for functional requirement statements on the new system to attain a goal G, on the adjacent system to enable the new system to attain G, and 4 types of non-functional requirement statement on the new and adjacent system to provide resources that enable the attainment of G on time, reliably, accurately and with up-to-date resources. An important domain assumption underpins this pattern – that the goal dependency can only be achieved by the exchange of some resource – normally information – between the new and adjacent systems. Domain assumptions also underpin the types of non-functional requirement specified in

the pattern. Safety-critical air traffic management systems necessitate timely and up-to-date resources and goal satisfaction, whereas other non-functional requirements types such as usability and security are less important and not defined in the patterns implemented in the reported projects.

The remaining 3 patterns were specified to handle composite process elements in *i** model dependencies. Whereas RESCUE mandates atomic requirement statements that cannot be further decomposed, the systems engineers had developed the *i** SD model to include compound dependencies such as *DMAN depends on ATC tower supervisor for current and foreseen runway status*, primarily to simplify the development and management of the complex *i** models. Therefore we introduced 3 patterns to detect and decompose composite dependencies.

Pattern matching and automatic requirements generation were implemented in REDEPEND v4.1. The 19 patterns are represented in an MS Excel file, so that new patterns can be added with requiring any changes to REDEPEND software. Full details of REDEPEND v4.1 are available in Lockerbie (2005).

Figure 10 describes how REDEPEND generates requirements from an analyst's perspective. The left-hand side shows how an analyst accesses the requirements generation function, from the REDEPEND top-line pull-down menu. The right-hand side shows one possible representation of requirements generated from the *i** SD model. All requirements are automatically generated into an MS Word document, as this is the most common storage mechanism for requirements, even when using requirements management tools such as DOORS and Requisite Pro. Each requirement in the document is structured using and expressed with a partially complete VOLERE shell (Robertson & Robertson 1999). For each requirement, the shell specifies a unique identifier for the requirement in the generation run; the requirement type; the requirement description; a rationale of canned text describing how the requirement was generated; and the source dependency in the SD model from which the requirement was generated.

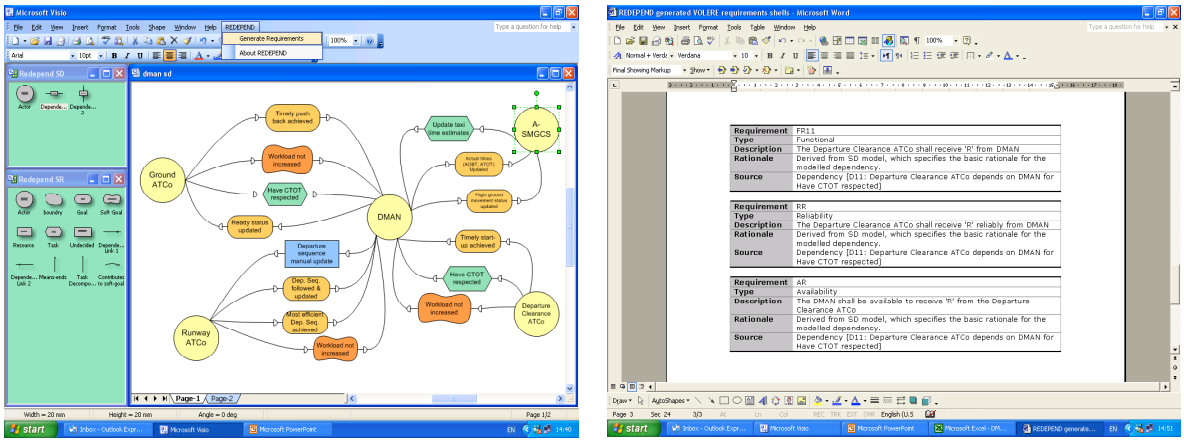

Figure 10. Automatic requirements generation in REDEPEND, showing how generated is called, and one outcome of the requirements generation process

In the original DMAN project we prototyped pattern-based requirements generation manually with the DMAN SD model containing 15 actors with 46 dependencies between these 15 actors. One systems engineer, an experienced member of the RESCUE team, assigned to the DMAN project, took a total of 3 working days to apply the 19 patterns to all 46 dependencies modeled in the SD model. The result was 214 new DMAN requirement statements – almost 25% of the total number of requirements statements in the final DMAN requirements specification. Given the DMAN requirements project duration – 10 months – this represents a major advance of the DMAN specification in a short period of time, notwithstanding the time spent to produce the SD model in the first place. The majority of these requirements statements were retained in the final DMAN specification (Maiden et al. 2006).

In contrast in a recent trial, REDEPEND version 4.1, running on a standard laptop PC, took 12 seconds to generate 287 requirements automatically from the same DMAN SD model. A larger number of requirements were generated due to refinements in the 19 patterns that led to more requirements of different types being generated. This result would suggest that automatic generation of requirements is potentially cost-effective, if the patterns generated as what analysts and stakeholders want. This question is explored in the last lesson.

10. Use *i** Models to Explore Candidate Requirements

REDEPEND provides new capabilities for generating candidate requirements statements from *i** models that, in turn, can change how we use *i** models in requirements. In particular, by automatically generating these candidate requirement statements, we aim to exploit evidence that people are better at identifying errors of commission rather than omission (Baddeley 1990), which is they are better at recognizing incorrect rather than missing requirements statements. We have already exploited this general trend in human cognition for recall to be weaker than recognition when designing the ART-SCENE scenario walkthrough tool (Maiden 2004). Generated requirement statements are delivered to systems engineers so that they can be modified or rejected easily using macros that we implement in the REDEPEND. Figure 11 shows how REDEPEND delivers candidate requirements that the pattern generation algorithm has generated to an analyst to select or reject prior to generated in the structured VOLERE shells in MS Word in the previous lesson. All of the shown requirements were generated from one dependency D11 in the SD model. The analyst can tick and un-tick selected requirement statements using the simple feature. The list also shows dependencies for which REDEPEND did not generate requirements statements.

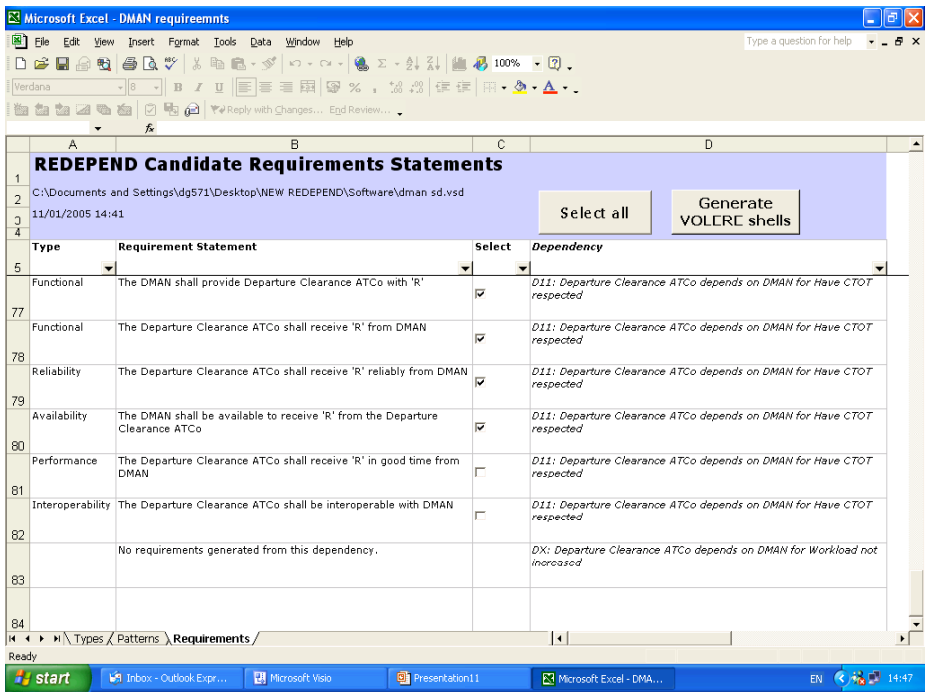

Figure 11. Requirements browsing in REDEPEND, showing lists of candidate generated requirements that an analyst can accept or reject using simple tick boxes

So how does REDEPEND do this? In simple terms, all generated requirements are output into tailored MS Excel sheets that provide an auto-filter for reviewing the requirements. This auto-filter provides the analyst with the capability to order all generated requirements in ascending and descending order of different requirement attributes, and to restrict the list to requirements of certain selected types such as *performance*, *reliability* and *availability*. More fine-grain selection of individual requirements is possible using the tick box filter.

Initial evaluation feedback from NATS analysts on this REDEPEND feature was positive, and we look to gather and report evaluation data in the future. With these new features, REDEPEND provides analysts with capabilities to construct *i** SD models during a workshop, automatically generate candidate requirement statements from this model in real time, then walkthrough these generated requirements to select and reject them. We believe that such tangible and immediate benefits from *i** will have implications for its future uptake.

Conclusions

This chapter reports our experiences in applying *i** in industrial requirements projects, and ten lessons that we learned and share with readers of this book. Some lessons are simple for readers to apply – for example getting the training in early. Others, in particular those that relate to REDEPEND, need our software to implement. Readers are encouraged to take inspiration from these lessons. Most have been implemented by us with relatively little resource and no external funding. The availability of adaptable graphical modelling technologies made this possible. So, if readers do not want to use

REDEPEND, we encourage you to develop and experiment with your own tools. You'll be surprised how quickly you will perceive benefits from *i**.

The reported work has informed part of our own requirements research agenda. At the time of writing, October 2005, we are exploring the following extensions to REDEPEND:

- Supporting the development and management of large-scale *i** models. In particular SR models are large, difficult to develop, and as a result hard to manage. New diagramming capabilities are needed to support development of integrated SR models;
- Developing context models and transforming them automatically into *i** SD models. Analysts can use REDEPEND version 4.1 to develop context models using standard Visio diagramming palettes, but REDEPEND cannot interpret these diagrams. REDEPEND will be extended with a new context diagramming palette, and new capabilities to generate first version *i** SD diagrams from these context diagrams;
- Related to this, dependency tables will be added to REDEPEND, to allow analysts to complete these tables and generate first version *i** SD diagrams from such a table;
- Working with NATS (the UK's National Air Traffic Services) to extend REDEPEND with new capabilities for safety analyses of systems modelled with *i**. In particular REDEPEND will be required to parse process elements such as goal, soft goal and task descriptions;
- Extend REDEPEND with scenario walkthrough capabilities to discover missing contributes-to soft goal links, based on scenario generation techniques from ART-SCENE (Maiden 2004). Scenarios are partial specifications of behaviour that contribute positively or negatively to soft goals modelled in *i** SR models. Walking through these scenarios can help analysts to discover all important contributes-to soft goal links, especially if we tailor ART-SCENE's scenario generation although to generate relevant prompts.

Finally, we will continue our transfer of requirements knowledge through RESCUE and *i**, in particular to evaluate the new versions of REDEPEND reported in the lessons, and future work outlined. We look forward to the challenge, and to reporting it in the future.

References

- Baddeley, A.D., 1990, 'Human memory: Theory and practice', Lawrence Erlbaum Associates, Hove.
- De Landtsheer R., Letier E. & van Laamsweerde A., 2003, 'Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models', Proceedings 11th IEEE International Conference on Requirements Engineering, IEEE Computer Society Press, 200-210.
- Lockerbie J., 2005, 'Automating the Pattern-Based Generation of Requirements for I* System Models', MSc Dissertation, School of Informatics, City University, November 2005.

- Maiden N.A.M., 'Systematic Scenario Walkthroughs with ART-SCENE', in 'Scenarios, Stories and Use Cases', Eds Alexander & Maiden, John Wiley, 166-178.
- Maiden N.A.M., Jones S.V. & Flynn M., 2003, 'Innovative Requirements Engineering Applied to ATM', Proceedings ATM (Air Traffic Management) 2003, Budapest, June 23-27 2003.
- Maiden N.A.M.,, Jones S.V., Manning S., Greenwood J. & Renou L., 2004a, 'Model-Driven Requirements Engineering: Synchronising Models in an Air Traffic Management Case Study', Proceedings CaiSE'2004, Springer-Verlag LNCS 3084, 368-383
- Maiden N.A.M., Manning S., Robertson S. & Greenwood J., 2004b, 'Integrating Creativity Workshops into Structured Requirements Processes', Proceedings DIS'2004, Cambridge Mass, ACM Press, 113-122
- Maiden N.A.M., Manning S., Jones S. & Greenwood J., 2006, 'Generating Requirements from Systems Models using Patterns: A Case Study', Requirements Engineering Journal.
- Maiden N.A.M. & Robertson S., 2005, 'Integrated Creativity into Requirements Processes: Experiences with an Air Traffic Management System', Proceedings 13th IEEE International Conference on Requirements Engineering, IEEE Computer Society Press, 105-114.
- Mavin A. & Maiden N.A.M., 2003, 'Determining Socio-Technical Systems Requirements: Experiences with Generating and Walking Through Scenarios', Proceedings 11th International Conference on Requirements Engineering, IEEE Computer Society Press, 213-222
- Robertson S. & Robertson J., 1999, 'Mastering the Requirements Process', Addison-Wesley.
- Vicente, K., Cognitive work analysis, Lawrence Erlbaum Associates, 1999.
- Yu E. & Mylopoulos J.M., 1994, 'Understanding "Why" in Software Process Modelling, Analysis and Design', Proceedings, 16th International Conference on Software Engineering, IEEE Computer Society Press, 159-168