CREWS Validation Frames: Patterns for Validating Systems Requirements[1]

N.A.M. Maiden, M. Cisse, H. Perez & D. Manuel

Centre for HCI Design,
School of Informatics,
City University,
Northampton Square,
London EC1V 0HB
United Kingdom
Tel: +44-171-477 8412
Fax: +44-171-477 8859
e-mail: n.a.m.maiden@city.ac.uk

Abstract

This paper proposes a pattern language for socio-technical system design to inform validation of system requirements. The development of this language takes inspiration from Alexander's pattern language for building design in architecture. It identifies different types of patterns which fulfil different roles in the requirements engineering process. This pattern-based validation approach has been operationalised in the CREWS-SAVRE software prototype. CREWS-SAVRE applies patterns to both scenarios and requirements documents to detect missing and incorrect system requirements, as well as to recommend new requirements which can improve the design of the socio-technical system.

1 Patterns in Requirements Engineering

Patterns are a novel alternative technique to help us better acquire, model and validate system requirements. In simple terms, patterns enable people to reuse knowledge about old solutions to solve similar new problems. However, there is little reported research into patterns for requirements engineering, in spite of the considerable current interest in software patterns for system design and implementation (e.g. Gamma et al. 1995). Indeed, each existing requirements engineering research initiative still tends to have a singular focus on process, domain or language. In contrast, patterns for requirements engineering, as we shall see, cuts across the divisions between process, domain and language. After all, experienced engineers do not separate them when acquiring, modelling and validating system requirements.

Patterns which describe the common elements of complex structures were first documented in the field of building architecture. Christopher Alexander, in his book "The Timeless Way of Building" (Alexander 1979), argues that "Beyond its elements, each building is defined by certain patterns of relationships amongst its elements." Patterns can be used to abstract away from the details of particular buildings and

[1] This research has been funded by the European Commission ESPRIT 21903 'CREWS' (Co-operative Requirements Engineering With Scenarios) long-term research project.

capture something essential to the design, for example the principles underlying the building, and the reason why the elements of the building are successful or unsuccessful (Kelly & McDermid 1997).

In this paper, we advocate the use of patterns that capture something essential to the design of socio-technical systems which include at least one significant software sub-system. The patterns can then be reused to guide the acquisition, modelling and validation of requirements for both socio-technical and software system design. This paper presents examples of patterns which capture the essential elements of socio-technical system design, introduces a software prototype which operationalises these patterns, and proposes an agenda for future development of patterns to aid requirements engineers.

The next section summarises some previous work on patterns in software engineering. Section 3 describes the use of design patterns to validate system requirements as part of the ESPRIT 21093 'CREWS' basic research project. Section 4 presents the prototype CREWS-SAVRE software tool. The paper ends with a discussion of patterns in requirements engineering, and future directions which the authors believe warrant further research and development.

2 Current Patterns for Software Engineering

Influenced by Alexander's work, there has been a recent increase in interest from software designers in patterns and pattern languages (e.g. Gamma et al. 1995). Although clearly useful to software designers, most current patterns and pattern languages are small in size and narrow in focus. For example, the 'Singleton' pattern (Gamma et al. 1995) enables a programmer to ensure that a class has one instance with a single, global point of access during low-level system design. Likewise, object-oriented analysis patterns such as the 'Participant-Transaction' pattern (Coad et al. 1995) enables an analyst to reuse two or three classes and their associations during the development of an object model. Because of this narrow focus, these patterns often fail to capture the essentials of a design because the design represented in the pattern is incomplete, that is it does not contain sufficient contextual and causal information to interpret the essential characteristics of the design.

However, new patterns for software engineering which do capture the essence of Alexander's original patterns are beginning to appear. One example is a language of safety case patterns (Kelly & McDermid 1997). Safety cases present arguments that a specific system is safe to operate. Across specific safety cases, patterns of argument emerge through, for example, common approaches to addressing a standard requirement or class of requirements, for example the ALARP (As-Low-As-Reasonably-Practicable) pattern. The patterns, Kelly & McDermid claim, can be simple and efficient solutions to general problems for the construction of a particular safety argument. Other recent examples of these "Alexander-style" patterns have been developed to inform design of human-computer interface guidelines, processes and organisations, and workplaces to understand the possible impacts of new technologies (Bayle et al. 1998).

From a "research and best practice" point of view, this move towards socio-technical system design patterns gives rise to at least two important questions: (i) what form of socio-technical system design pattern should we develop, and: (ii) how can such patterns support the requirements engineering process? We attempt to answer these questions by referring to Alexander's original patterns to inspire and inform the development of CREWS's socio-technical system design patterns.

3 CREWS Patterns and Validation Frames

Alexander's original patterns focus on the interactions between the physical form of the built environment and how this form inhibits or facilitates various sorts of individual and social behaviour in it. Bayle et al. (1998) report that this facilitation is more subtle than the simple detection of certain properties which afford actions. Rather, the emphasis is on the characteristics of the environment which might facilitate or inhibit action. For example, in his original 'Beer Gardens' pattern, Alexander suggests that local pubs should have activities around the edges and large tables in the middle to encourage people to cross through the centre, sit at tables and converse with their neighbours. As such, the physical form, or design, of the pub facilitates desirable behaviour in the customers. Furthermore, a pattern also captures the essentials of a 'good design', in that it maximises those characteristics which facilitate desirable actions over those that inhibit these actions.

If these pattern "characteristics" are applied to socio-technical system design patterns, a good pattern must capture the essential elements of the software system, and how the form of this system facilitates and inhibits desirable individual or social behaviour and because of the system. The form of the design can include that of the software system and the physical and social environments of the system's use. Indeed, the pattern can sometimes "design" individual and social behaviour in the context of the software system. Furthermore, because the pattern captures the notion of a 'good' design rather than a 'bad' design in the context of past experiences (e.g. prototypical design examples), a requirements engineering team will have more confidence that the design of the software and socio-technical systems will facilitate desirable actions.

3.1 Requirements, Scenarios and Patterns

As a starting point for designing socio-technical system patterns, we map the key elements of Alexander's patterns to CREWS system requirements, scenarios and patterns:

- the form of the software system is expressed as functional and non-functional requirements statements in a requirements document;
- desirable individual and social behaviour in the environment is expressed as scenarios which are sequences of events and actions which describe desirable future system use in the environment;
- 'good' socio-technical system design is expressed in patterns which capture elements of the software system form (i.e. the requirements) which facilitate or inhibit desirable behaviour (i.e. the scenarios).

Thus validation is achieved by matching the desirable behaviour in the environment (scenarios) to the form of the software system (requirements) using models of good design (patterns). We can view each pattern as being 'superimposed' on different parts of the scenario and requirements document to detect requirements which are missing or which inhibit desirable behaviour. This solution has been implemented in the CREWS-SAVRE prototype software tool developed as part of the European Union-funded ESPRIT 21903 'CREWS' (Co-operative Requirements Engineering With Scenarios) long-term research project. Before describing this software tool, let us first demonstrate the nature of CREWS's socio-technical system design patterns using 3 prototypical examples.

The MACHINE-FUNCTION Pattern: this first pattern captures something essential to the design of the requirements document rather than to the system itself. In broad terms, it states that a good requirements document shall include at least one functional requirement statement for each action which the software system is involved in. Implementing the functional requirement will thus ensure that the system undertakes the action described in the scenario. For example, consider a dealer who uses a financial foreign currencies trading system to record information about a currencies deal:

ACTION: a dealer enters data about a transaction into the software system;
REQUIREMENT: the system shall enable a user to enter information about a financial transaction into the dealing system.

This pattern fulfils the prerequisites for a socio-technical system design pattern, although at first glance it might not appear to do so. The form of the design (in this case the requirements document) has characteristics which facilitate the desirable behaviour of its users (i.e. the systems developers) to produce a complete and correct systems design. We envisage that this pattern will be used primarily for validating an existing requirements document using a scenario which acts as a 'test-script' for the document.

The COLLECT-FIRST-OBJECTIVE-LAST pattern: the second pattern is more true to Alexander's notion of a pattern, in that it captures something essential to the good design of a mechanical device with which a person interacts using one or more personal items to achieve an objective. The objective is that the person should not leave the personal items behind at the device at the end of a transaction with it. To ensure this, the device imposes a prescriptive sequence so that the person must first collect all items to achieve the objective. Consider a passenger who uses a travel ticket to pass through automatic gates at the entrance to a London Underground station:

OBJECTIVE: to pass through the automatic gates and enter the station;
PROBLEM: passengers sometimes forget to take their valid ticket with them;
SOLUTION: a passenger must collect the ticket from the machine for the gates to open.

This pattern can be seen in the design of other automatic gate machines, for example at entrances to stations on the Paris Metro and Tyne & Wear Metro networks. It can also be seen in the design of ticket collection points where a customer uses their credit card to collect pre-purchased cinema tickets. It also fulfils the prerequisites for a socio-technical system design pattern. The form of the device design has characteristics which facilitate desirable behaviour, that is the user shall not leave their personal items with the device. The pattern includes the rationale as well as successful and unsuccessful elements of the design.

The INSECURE-SECURE-TRANSACTION pattern: the third pattern is also true to Alexander's notion of a pattern, but this time it captures something essential to the good design of a wider socio-technical system. It is called the INSECURE-SECURE-TRANSACTION pattern. Central to this pattern is the objective that a secure transaction between a person (or people) and the software system shall not become insecure due to negligence from the person/people. To ensure that this objective is met, the pattern imposes one or more requirements and/or constraints on the design of the socio-technical system. Let us return to our dealer who is using a financial foreign currencies trading system to record information about a currencies deal:

OBJECTIVE: to ensure that the deal-recording transaction system remains secure;
PROBLEM: dealers are often interrupted when entering a transaction, thus potentially making the system insecure;
SOLUTION: to ensure that the system remains secure. Solution options include: (i) warning the dealer that the system is insecure through a visual or audio signal, so that the dealer might return to the transaction; (ii) closing the transaction so that no person, including the original dealer, can access it without going through the normal transaction access routines; (iii) stopping the dealer from being interrupted during the transaction, through either physical barriers or rules which control the socio-technical system; (iv) using physical barriers or socio-technical system rules to stop other people other than the original dealer using the transaction.

Clearly, in the trading system domain, the last two options are not applicable because of the open and interactive nature of financial dealing room floors.

3.2 Socio-technical System Design Pattern Types

The three examples of patterns presented so far in this paper reveal subtle differences in their nature and role in the requirements engineering process. The development of a comprehensive pattern language also requires the pattern authors to better understand these differences. To this end, CREWS uses NATURE's distinctions between the usage, system, development and subject models, known as worlds, from requirements engineering research to categorise its patterns (NATURE 1996). It proposes four pattern categories:

- SYSTEM DESIGN patterns which capture elements of good design of a socio-technical system. The form of the socio-technical system facilitates desirable agent actions in the environment. The focus is on NATURE's usage world which describes how the system is used to achieve work in the organisation (NATURE 1996). The INSECURE-SECURE-TRANSACTION pattern is an example of a system design pattern;

- DEVICE DESIGN patterns which capture elements of the design of devices with which an agent interacts to achieve desired behaviour. These patterns also relate to the usage world, but their focus is often on a single agent who interact directly with the software system and/or device in which the software system is embedded. The COLLECT-FIRST-OBJECTIVE-LAST pattern is a good example of a device design pattern;
- SOFTWARE DESIGN patterns which capture elements of the design of the device and/or system hardware to facilitate the required emergent behaviour of the software system. The focus here is on NATURE's system world (NATURE 1996) which contains information about the design and implementation of the software;
- SPECIFICATION DESIGN patterns which capture elements of the requirements document to facilitate or inhibit desirable behaviour of the system designers. These patterns relate to NATURE's development world which contains information about the processes and agents which lead to development of the socio-technical system. The MACHINE-FUNCTION pattern is a good example of a specification design pattern.

It is interesting to note that CREWS's pattern types differ from software and specification patterns advocated by other authors. For example, the object modelling patterns in Coad et al. (1995) belong to NATURE's subject world which models information about the real-world problem domain that the software system maintains information about (NATURE 1996). This distinction underlines the different focus of the CREWS patterns on requirements scoping, acquisition and validation rather than system modelling processes.

The remainder of this paper describes how CREWS uses system, device and specification design patterns to inform the scenario-based validation of system requirements. It puts particular emphasis on its operationalisation of patterns in the form of validation frames in the CREWS-SAVRE software prototype.

4 CREWS-SAVRE's Validation Frames: Implementing Patterns

CREWS-SAVRE supports the two approaches to scenario-based requirements validation shown in Figure 1. The first is guided walkthrough of scenarios by a team of requirements engineers and other stakeholders. The team walk through the sequence of events in the normal and alternative courses of the scenario to detect incomplete and incorrect requirements. Although it improves on the current ad hoc use of scenarios reported by Weidenhaupt and his colleagues (1998), the large number and complex nature of both the scenarios and the requirements means that the team are still likely not to find all of the missing and incorrect requirements in the requirements document.

In the second approach, CREWS-SAVRE delivers each pattern as one or more validation frames to validate system requirements in a requirements document using an automatic checker. The output is an agenda of issues to be addressed by the requirements engineering team. Each issue identifies a possible missing or incorrect requirement in the requirements document, a missing or incorrect event/action in the scenario, and advice often in the form of new requirements to resolves these errors and omissions. The team can use these issues to change the requirements document, the scenario or both, then to recheck the requirements document again with the automatic checker. It continues this iterative change-and-check process until there are either no remaining issues for that scenario and requirements document, or the remaining issues have been recorded and can be tolerated by the requirements engineering team. The algorithm which is used to validate one or more system requirements with a scenario is a simple, two-pass algorithm to check: (i) each event/action against all requirements, and: (ii) each requirement against all event/actions. The 'intelligence' of the approach, however, is in the validation frames which encapsulate 'good' design practice in the CREWS patterns. Let us look at these validation frames in more detail.

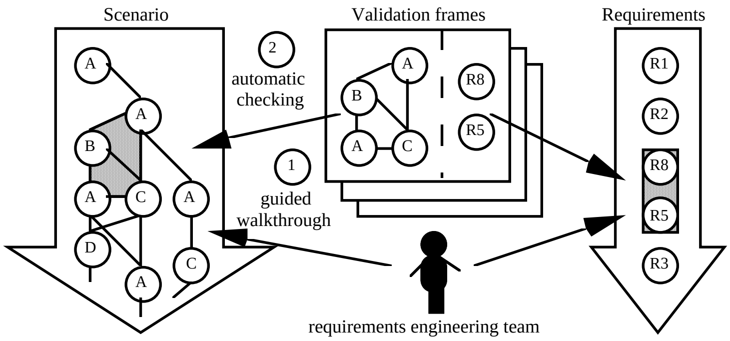

Figure 1 An overview of CREWS-SAVRE's requirements validation approach using validation frames.

4.1 CREWS Validation Frames

Each validation frame has five parts:

- a unique identifier for the frame;
- the CREWS pattern(s) which the frame operationalises;
- the situation which specifies the desirable behaviour in the environment;
- the requirement(s) which specifies the form of the designed socio-technical system;
- the consequences of detecting or failing to detect requirement(s) for the desirable behaviour.

Each time the validation algorithm applies a frame, the algorithm searches the scenario for a unique combination of interconnected events, actions, agents and objects of predefined types which is the signature of the frame. This combination defines the situation part of the frame. When the algorithm detects this combination, it looks for one or more requirements of a predefined type and content which, according to the frame, should be present in the document to facilitate the desirable behaviour. This form is the requirement part of the frame.

One of the major strengths of our validation frames is that CREWS-SAVRE does not require a domain-specific lexicon or model to apply, although a lexicon can be added to enhance validation. Rather, it combines simple pattern matching with a strong type model of requirements, events, actions, agents and objects. To type system requirements, we combine the VOLERE method and PS055 standard (Mazza et al.

1994) to type each requirement as a functional, behavioural, physical, performance, usability, interface, operational, timing, resource, verification, acceptance testing, documentation, security, portability, quality, reliability, maintainability or safety requirement. Each requirement in a document must have one and only one type.

Likewise each event, action, agent and object in the scenario has must have one of a predefined set of types specified in the CREWS-SAVRE use case/scenario meta-model (Maiden et al. 1998). Each action is either cognitive, physical, system-driven, communicative or complex. Each agent is either a human agent, machine agent or composite agent. Each object does or does not undergo a state change as a result of an action. Each agent-action involvement relation is specialisable to performs, initiates, ends, etc., and each action uses one or more objects. Furthermore, events can be linked in the scenario in a temporal sequence. Although these types, on their own, appear simple, their use when combined with a pattern matcher is very powerful, as we demonstrate through the validation frames for three CREWS patterns presented earlier.

Consider the validation frame F1, the operationalisation of the MACHINE-FUNCTION pattern. The situation-part is simple. It states that the frame shall be fired if and only if the scenario under analysis contains an event which starts an action, and this action involves an agent which a machine agent which is the software system under analysis. For each scenario event for which this situation exists, the automatic checker searches for one or more requirements which are defined in the requirement-part of the frame. For frame F1, these requirements are functional requirements which have a similar semantic content to the action which is started by the event in the scenario. To determine whether an event/action and a requirement are semantically equivalent, CREWS-SAVRE uses either a simple keyword checker or CREWS's natural language parser (Achour & Rolland 1997) to parse the natural language descriptions and determine a degree of equivalence which is either above or below a predefined threshold. If no such requirements can be detected, the consequence-part of the frame is accessed to produce recommendations which appear in CREWS-SAVRE's agenda list.

Validation-frame
Frame Identifier: F1
Design Pattern: MACHINE-FUNCTION
Situation:
    event(Ev) starts action(Ac) and
    action(Ac) involves agent(Ag) and
    agent(Ag) is-type = "Machine" and
Requirement:
    action(Ac) word-matches requirement(R) and
    requirement(R) is-type = "Functional requirement"
Consequence
    If found: link requirement(R) to event(Ev) in trace table;
    If not found: write to agenda: requirement type message = "Missing mandatory functional requirement".
End validation-frame

The second validation frame, F13, has a similar structure but a more complex situation which reflects the more complex nature of the COLLECT-FIRST-OBJECTIVE-LAST pattern. The situation-part states that the frame shall be fired if the scenario under analysis contains one event which ends one physical action which involves an agent which is a machine and uses an object, and a second, later event which starts a second physical action which involves a different human agent who uses the same object. Each time the situation occurs, the frame recommends design advice to ensure user collects object before completion of task so that item is not forgotten.

Validation-frame
Frame Identifier: F13
Design Pattern: COLLECT-FIRST-OBJECTIVE-LAST
Situation:
    event(evA) ends action(acA) and
    action(acA) is-type = "physical" and
    action(acA) involves agent(agX) and
    agent(agX) is-type = "machine" and
    action(acA) uses object(obA) and
    event(evB) starts action(acB) and
    action(acB) is-type = "physical" and
    action(acB) involves agent(agY) and
    agent(agY) is-type = "human" and
    action(acB) uses object(obB) and
    object(obA) = object(obB) and
    event(evA) before event(evB)
Requirement:
    no condition, fire for all cases
Consequence:
    Write to agenda: requirement type message = "Device design advice: ensure user collects object before completion of task so that item is not forgotten".
End validation-frame

The third validation frame, F10, also has a more complex situation to reflect the nature of the INSECURE-SECURE-TRANSACTION pattern. The situation-part states that the frame shall be fired in the scenario under analysis contains two events which involve the same two agents, one of those agents is a machine, and there are a significant number of events not involving the machine agent which occur in-between the two original events. For each scenario event for which this situation occurs, the automatic checker searches for semantically-equivalent functional requirements in the document. If no such requirements are found, the relevant issue is written to CREWS-SAVRE's agenda list.

Validation-frame
Frame Identifier: F10
Design Pattern: INSECURE-SECURE-TRANSACTION
Situation:
    event(evA) starts action(acA) and
    event(evB) starts action(acB) and

event(evA)≠event(evB) and
action(acA) involves agent(agA) and
action(acB) involves agent(agB) and
agent(agA)=agent(agB) and
agent(agA) is-type = "machine" and
not consecutive(evA,evB) (events between evA and evB > 2)

Requirement:
action(Ac) word-matches requirement(R) and
requirement(R) is-type = "Functional requirement"

Consequence:
If found: link requirement(R) to event(Ev) in trace table
If not found: write to agenda: requirement type message = "Missing functional requirement: the system shall warn or autologout the user after period of system inactivity".

End validation-frame

These three validation frames demonstrate the different forms of advice that can be offered to the requirements engineering team. The first frame, F1, recommends the inclusion of new requirements statements of the given type. The second, F13, gives more general advice about the design of the device, with subsequent implications for the requirements document. The third, F10, recommends generic requirements statements which the team can specialise and adapt for inclusion in the requirements document.

4.2 Implementing the Validation Frames

Most reported software patterns have not been implemented computationally, that is are not retrieved and exploited using software tools during systems development (e.g. Gamma et al. 1995). In contrast, CREWS-SAVRE exploits the precise specification of the validation frames to implement them in the software tool. As well as speeding up the requirements validation process, it ensures validation because CREWS-SAVRE uses each pattern to cross-check each scenario and scenario event with each requirement in the requirements document.

5 The CREWS Requirements Validator

CREWS-SAVRE is a prototype software tool which has been designed to guide systematic scenario-based requirements engineering. It has been developed on a Windows-NT platform using Microsoft Visual C++, VisualBasic5 and Access, thus making it compatible for loose integration with commercial requirements management and computer-aided software engineering software tools. It has been designed to be integrated with Rational's RequisitePro requirements management tool. This tool handles system requirements, thus enabling the development team to focus on novel implementations such as use case modelling, scenario generation and scenario-based requirements validation. The requirements validator component of CREWS-SAVRE supports dialogue with the requirements engineering team to select the requirements, scenarios and frames for validation, a validation algorithm which applies validation frames to detect all scenario and requirement omissions and errors, a natural language checker to detect possible semantic equivalence between one requirement statement and one scenario event/action, and validation frame administrator to enable a user to add, change and remove validation frames from the component.

Let us demonstrate the requirements validator with a simple banking ATM example. A requirements engineer called Roderick has already used CREWS-SAVRE's domain and use case modeller components to generate a number of scenarios to validate the high-level requirement "the system shall enable a customer to withdraw cash from it". One of the generated scenarios which describes the most frequent sequence of events is shown in Figure 2. The bottom left-hand corner shows the normal course of the generated scenario as a sequence of events which start or end actions in use cases. The type of each action is also shown prior to the action description. Roderick uses the "Back" and "Next" buttons to navigate through the scenario and select events. On the right-hand side are alternative courses generated automatically from Roderick's selection of generic, permutation and problem exception classes, see Maiden et al. (1998) for details. For each selected event, the tool presents alternative courses linked by the tool to that event, and advises Roger to decide whether each alternative course is (a) relevant, and (b) handled in the current requirements specification. Roderick has also generated a document in RequisitePro containing a small number of behavioural, functional and physical system requirements for the ATM. These requirements are also shown in Figure 2. These requirements and Roderick's scenario are the starting point for requirements validation.

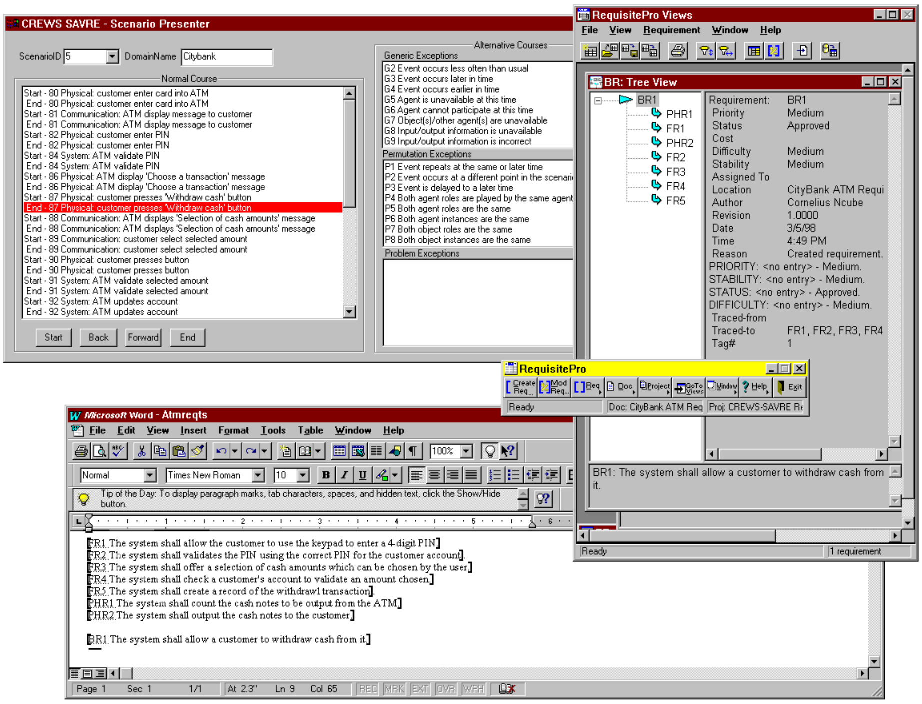

Figure 2 The starting point for requirements validation. The scenario is shown in CREWS-SAVRE's scenario presenter component. The system requirements are stored in a RequisitePro document.

Roderick uses CREWS-SAVRE's requirements validator component to validate the system's requirements. To enable validation, he must select a number of mandatory parameters including the scenario to use to validate the requirements, the requirements to validate, and the types of requirements which are to be checked. When validation is complete, Roderick can click on the 'See agenda table' to see results of the validation in the form of a list of issues for him to address. Each issue identifies one event/action in the scenario, one requirement in the requirements document and a message which identifies possible omissions and/or errors, and solutions to overcome these problems. Part of the agenda list generated from the scenario and requirements document in Figure 2 is shown in Figure 3. This part of the list shows missing functional requirements identified using validation frame F1 which implements the MACHINE-FUNCTION pattern. Such issues are not surprising given the richness of the scenario and small number of requirement statements in the requirements document. Elsewhere in the agenda list is advice given from firing frames F10 and F13 which implement the COLLECT-FIRST-OBJECTIVE-LAST and INSECURE-SECURE-TRANSACTION pattern. Roderick uses the issues in the agenda list to change system requirements, elaborate the scenarios and revalidate requirements using CREWS-SAVRE until validation of the document is complete.

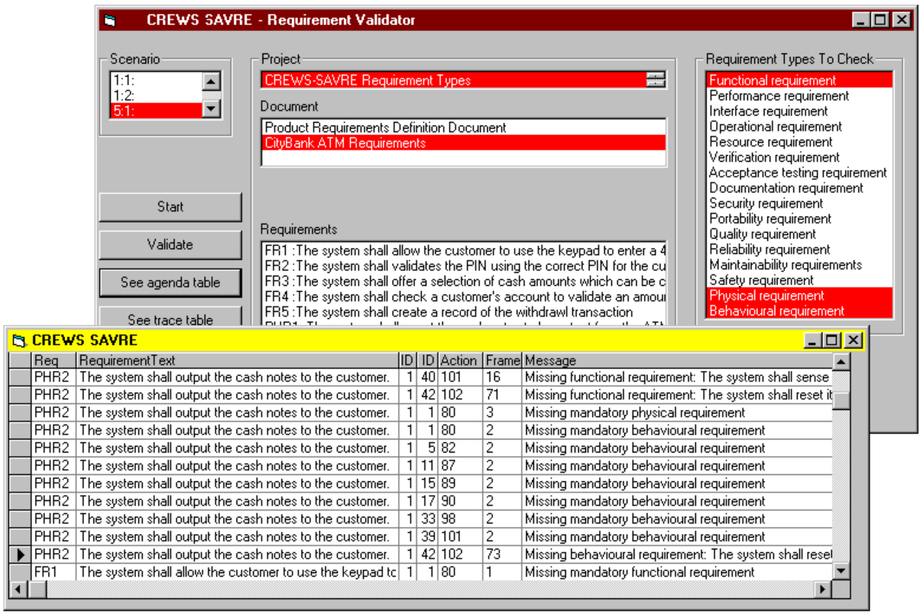

Figure 3 Part of the requirement validator's agenda list showing issues to resolve for the scenario and requirements document shown in Figure 2.

6 An Agenda for Patterns Development in Requirements Engineering

One of the main conclusions from the work reported in this paper is the effective synthesis of scenarios and patterns to validate system requirements. Indeed, scenarios and patterns share several important traits. First, both enable us to model both general-level and instance-level information in a single model. As the CREWS pattern language is extended, we envisage inclusion of domain-specific properties of agents, objects and actions, as well as domain-specific requirements and indeed software solutions to complement our generic definitions of situations and requirements. Second, both a single scenario and a single pattern can model information at different levels of precision and formalism (Rolland et al. 1997). Such flexible models are important in the requirements engineering process. Third, and most importantly in our view, scenarios often model system/user interaction (e.g. Jacobson et al. 1992, Graham 1996) central to Alexander's definition of a good pattern. Thus, scenarios are essential for the application of socio-technical system design patterns. We have only just began to explore the overlaps and relationships between scenarios and patterns.

As reported in the introduction, there has been little reported research and development into patterns in requirements engineering. To encourage and direct this work, we believe that a research and development agenda is needed. This paper presents our initial thoughts. The first agenda item is to develop a better understanding of the type, coverage and content of socio-technical system design patterns to scope and inform research and development activities. The tentative classification of patterns reported in this paper suggests that we can distinguish between patterns according to the nature of interaction between design form and desirable actions, with implications for different roles of patterns in requirements engineering. On the other hand, determining pattern coverage is more difficult because it often depends on technological advances. For example, the COLLECT-FIRST-OBJECTIVE-LAST pattern would not have been a common device interaction pattern two decades ago. The final point of this first item, determining the content, and in particular the situation-part of patterns, forces pattern authors to think about the semantics of scenarios and requirements. The CREWS meta-schema's model of requirement, event, action and object types is sufficient to generate a first-draft pattern language. However, in the longer term, patterns will we believe become more problem domain-specific to capture the richness of pattern situations and solutions.

A second, perhaps more important agenda item, is how to develop a pattern language for socio-technical system designs. As well as observing good designs in existing systems, we propose to use knowledge elicitation techniques such as laddering and card sorts to elicit design patterns from highly-experienced requirements engineers, consultants and system designers. Such people recall and reuse mental abstractions when specifying new requirements. Rosch (1983) argues that peoples' mental representations are accounted for by the structure of their environment, so these mental categorisations should provide an accurate categorisation of problems and hence good design patterns. Direct elicitation of such expertise is a cost-effective approach which, if handled carefully, can provide important information about design patterns.

The third agenda item is to determine the possible uses of socio-technical systems design patterns in the requirements engineering and wider systems engineering processes. Although originally intended to guide requirements validation, the patterns reported in this paper also have clear roles for deriving a high-level system design, bounding that design and acquiring requirements for software systems and devices to be developed as part of that design. One possible consequence is the development of a requirements patterns book to record and make these designs available to inform high-level systems engineering processes. Indeed, we see patterns as a central mechanism for making knowledge of different types reusable during the requirements and systems engineering processes. Such large-scale reuse provides an exciting vision of the future of the systems development process.

Acknowledgements

The authors wish to thank everybody who have contributed to the development of the patterns and software tool. They also thank the other members of the CREWS project for comments and advice. This research has been funded by the European Commission ESPRIT 21903 'CREWS' (Co-operative Requirements Engineering With Scenarios) long-term research project.

References

Achour C.B. & Rolland C., 1997, 'Introducing Genericity and Modularity of Textual Scenario Interpretation in the Context of Requirements Engineering', *CREWS Technical Report*, Centre de Recherche en Informatique, Universite Paris 1, France.

Alexander C., 1979, *'The Timeless Way of Building'*, NY: Oxford University Press.

Bayle E., Bellamy R., Casaday G., Erickson T., Fincher S., Grinter B., Gross B., Lehder D., Marmolin H., Moore B., Potts C., Skousen G. & Thomas J., 1998, 'Putting It All Together: Towards a pattern language for interaction design: A *CHI97 Workshop'*, *SIGCHI Bulletin*, 30(1), 17-23.

Coad P., North D. & Mayfield M., 1995, *'Object Models: Strategies, Patterns and Applications'*, Englewood Cliffs, Prentice-Hall.

Gamma E., Helm R., Johnson R. & Vlssides J., 1995, *'Design Patterns: Elements of Reusable Object-Oriented Software'*, Addison-Wesley.

Guindon R., 1990, 'Designing the Design Process: Exploiting Opportunistic Thoughts', *Human-Computer Interaction* 5, 305-344.

Kelly T.P. & McDermid J.A., 1997, 'Safety Case Construction and Reuse Using Patterns', *Proceeding SAFECOMP97*, Springer Verlag.

Maiden N.A.M., Minocha S., Manning K. & Ryan M., 1998, 'CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements', *Proceedings 4th International Conference on Requirements Engineering (ICRE98),* IEEE Computer Society Press.

Mazza C., Fairclough J., Melton B., De Pablo D., Scheffer A. & Stevens R., 1994, *'Software Engineering Standards'*, Prentice Hall.

NATURE, 1996, 'Defining Visions in Context: Models, Processes and Tools for Requirements Engineering', *Journal of Information Systems*, 21(6), 515-547.

Weidenhaupt K., Pohl K., Jarke M., Haumer P., Maiden N.A.M. et al., 1998, 'Scenario Usage in Systems Development: A Report on Current Practice', *IEEE Software*, March 1998.