Software Systems Modelling Group,
School of Design, Engineering & Computing,
Bournemouth University

Requirements Validation with Enactable Descriptions of Use Cases

John Mathenge Kanyaru

A Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy of Bournemouth University

August, 2006

To my parents, Romano Kanyaru & Dominica Kanyaru, for taking me to school despite their having never seen a classroom door…

Acknowledgement

Quite simply, this thesis could not have been written without the supervision, guidance and support of Keith Phalp and Martin Shepperd.

As my supervisor, Keith provided me with the inspiration, encouragement and freedom to pursue my occasionally bizarre ideas. As my friend, he has been unfailingly supportive, patient and generous. He has always been ready to listen and advise me on my endless technical and social dilemmas - and for that I am deeply grateful.

For over three years, Martin consistently provided me with constructive and insightful feedback on my work. He meticulously reviewed endless reports, papers and drafts of this thesis, and effectively treated my bouts of verbal diarrhoea!

Some of the ideas behind the work presented in this thesis originated during Karl Cox's PhD research at our research group between the years 1998 and 2003. His work on use case writing guidelines highlighted, at least, the need for rigorous scrutiny of use case specifications, and hence the need for tool support for that matter. Since then, Karl has been occasionally providing feedback on my work- thank you Karl.

The *EducatorTool*, described in chapter 4, was an evolutionary prototype. The feedback from audiences who sat through my early demos was invaluable.

Last but not least, the support of my family has sustained me throughout the last two years of my PhD. My very best friend and my wife, Shelly Ann, and my daughters, Damali and Muthoni, have all, in different ways, contributed to this thesis.

My parents have never understood why I would want to engage in another three years of study, but encouraged me to do so all the same. Their love and good wishes must have played a part in sustaining me through it all.

Financial Support

The work presented in this thesis was funded by the School of Design, Engineering and Computing (DEC). I am grateful for this support, without which I could not have made it on my own.

The views and opinions expressed in this thesis however, are not necessarily those held by DEC or its individual members, and any errors are of course, entirely the responsibility of the author.

List of publications

1) Kanyaru, J., K. Phalp, and M. Shepperd. *Enhancing use case descriptions with states: validation via enaction*; PREP2004 conference, University of Hertfordshire; April, 2004.

2) Kanyaru, J. and K. Phalp. *A lightweight state machine for validating use case descriptions*; ESERG technical report; May, 2004.

3) Kanyaru, J. and K. Phalp. *Requirements validation with enactable models of state based use cases*. in *Empirical Assessment in Software Engineering conference (EASE'05)*. Keele University, UK; April, 2005.

4) Kanyaru, J. and K. Phalp. *Supporting the Consideration of Dependencies in Use Case Specifications*. in *Requirements Engineering: Foundations for Software Quality (REFSQ'05)*; Porto, Portugal; June, 2005

5) Kanyaru, J. and K. Phalp. *Aligning business process models with specifications using enactable use case tools*. in *Requirements for business need workshop (at RE'05)*; Paris, France; August, 2005.

Table of Contents

List of Figures

List of tables

Abstract

The validation of stakeholder requirements for a software system is a pivotal activity for any non-trivial software development project. Often, differences in knowledge regarding development issues, and knowledge regarding the problem domain, impede the elaboration of requirements amongst developers and stakeholders. A description technique that provides a user perspective of the system behaviour is likely to enhance shared understanding between the developers and stakeholders. The Unified Modelling Language (UML) use case is such a notation. Use cases describe the behaviour of a system (using natural language) in terms of interactions between the external users and the system.

Since the standardisation of the UML by the Object Management Group in 1997, much research has been devoted to use cases. Some researchers have focussed on the provision of writing guidelines for use case specifications whereas others have focussed on the application of formal techniques. This thesis investigates the adequacy of the use case description for the specification and validation of software behaviour. In particular, the thesis argues that whereas the user-system interaction scheme underpins the essence of the use case notation, the UML specification of the use case does not provide a mechanism by which use cases can describe dependencies amongst constituent interaction steps. Clarifying these issues is crucial for validating the adequacy of the specification against stakeholder expectations.

This thesis proposes a state-based approach (the Educator approach) to use case specification where constituent events are augmented with pre and post states to express both intra-use case and inter-use case dependencies. Use case events are enacted to visualise implied behaviour, thereby enhancing shared understanding among users and developers. Moreover, enaction provides an early "feel" of the behaviour that would result from the implementation of the specification. The Educator approach and the enaction of descriptions are supported by a prototype environment, the EducatorTool, developed to demonstrate the efficacy and novelty of the approach.

To validate the work presented in this thesis an industrial study, involving the specification of real-time control software, is reported. The study involves the analysis of use case specifications of the subsystems prior to the application of the proposed approach, and the analysis of the specification where the approach and tool support are applied. This way, it is possible to determine the efficacy of the Educator approach within an industrial setting.

1. Introduction

The difficulty of matching software systems to the needs (requirements) of the stakeholders of the system is well recognised (e.g., [1], [2], [3], [4]). One way to determine the extent to which these needs will be met by the system is to involve the stakeholders in the validation of the system specification ([5], [6]). That is, given that a specification is a description of the proposed system's behaviour, validation of the specification is one of the appropriate ways in which to clarify the extent to which the specification matches the stakeholder requirements. This thesis presents a state-based approach for system specification with UML use cases, and the subsequent expression of dependencies amongst constituent use case events as a means to validation.

1.1 Software requirements

Prior to constructing any non-trivial software system, it is crucial that the purpose of the system is established. For the purpose to be established clearly, development participants often have to obtain knowledge about the problem domain. The problem domain constitutes the part of the world where the problem exists ([7], [8]). Software requirements can be regarded as the effects that stakeholders wish to be brought about in the problem domain ([7], [9]).

Determining software requirements is not the end of the requirements process. Development participants (software engineers and other stakeholders) must establish what the appropriate behaviour of the intended system should be in order to produce the desired effects during its use. This constitutes the software specification. Jackson [10] points out that a specification is a restricted form of software requirements. That is, the specification provides enough information for a developer to build the software system without further knowledge about the problem domain ([10], [11]). This view is also expressed by ([7], [12], and [13]) who observe that the input to a specification task is the requirements document. Hence, specification in itself is the invention and definition of behaviour of a software system such that the system will produce the required effects in the problem domain [7].

Several researchers (e.g., [14], [15], and [16]) observe that inadequate requirements determination, or flawed specification, inevitably leads to development of a system that does not meet its intended purpose. This observation is evident from the many software projects that are abandoned at huge financial cost, and often after many years of development effort. For instance, the London Ambulance Service dispatch system was scrapped after two days of its operation in 1992. The

problems cited for its failure were largely due to project management issues and an inadequate requirements process (see [17] and [18]). In 1993, the London Stock Exchange abandoned the development of its Taurus paperless share settlement system after more than 10 years development effort. It was estimated that, when the project was abandoned, it had cost the city of London over £480 million, whereas the original budget was £6 million [19]. The key problems cited for its demise were failure to reconcile conflicting requirements. A few years ago, the UK government set out to provide an electronic patient records' system in England, but the computer system has been reported (see [20]) to be two and half years late and over budget (£20bn rather than the initial £6.2bn). The main problem cited for its delay is that doctors have not been able to agree on exactly what information should be recorded for each patient. In other words, there is disagreement as to how much information should be included without getting patients' consent, which also means that the patients are not in fact consulted during the development of the system. These examples (and many others) indicate that some of the main issues contributing to the failure of software projects are related to inadequate requirements and specification.

One way to address these requirements and specification issues is to involve concerned stakeholders [21] in the elicitation of requirements and the validation of specifications. UML use cases have recently gained wide uptake as a requirements and specification notation, mainly due to their ease of construction and comprehension. Most crucially, use cases describe software behaviour from the point of view of external users, thereby depicting stakeholders' views of the system functions.

1.2 Use cases – an overview

The use case concept was originally proposed in the Objectory method [22] but has also been integrated in a number of other approaches including the Fusion method [23] and the UML([24],[25]). The basic elements of a use case specification include actors and use cases. Actors are external roles or user types that have an interest in the use cases ([26], [27]). A use case is a description of system usage from the viewpoint of an actor (or actors) ([22], [28]). There are two main ways of constructing use cases, either in the form of a use case diagram (e.g., as discussed in [29] and [30]) or textual use cases (e.g., as suggested in [31] and [32]). Due to the lack of detail regarding system behaviour depicted in use case diagrams, [33] argues that the diagram should not be used on its own for software specification. A term that has been used alongside (even interchangeably with) the use case notation is *scenario*. Scenarios are textual descriptions of use cases showing a specific path of behaviour through a use case. Thus, a use

case has a main path (or main scenario), and any alternative paths (scenarios) showing different pathways through the use case. Many authors (e.g., [34], [35], [36], and [32]) favour the textual use case (or the use case description). Use case descriptions delineate the constituent steps (events) of the use case, hence offering more detail regarding the interactions between actors and the system.

During specification tasks for non-trivial systems, many use cases may be constructed which in turn may have many constituent events. This thesis argues that inter-relating use cases for the same system, or indeed, inter-relating the constituent events of a single use case is crucial to the validation of the specification. Clarifying intra-use case dependencies is vital to determining the implications of interactions amongst constituent events of a use case; additionally, clarifying inter-use case dependencies is vital to determining the implications of interactions amongst use cases of the same or different systems. Research efforts so far have mainly focussed on the provision of authoring guidelines for use cases and where researchers have attempted to provide automated analysis of use case specifications, formal approaches have been used for the production of such specifications.

1.3 Problems with use cases

Much work has been devoted to UML use cases since the standardisation of the UML by the Object Management Group in 1997. Given the importance of the comprehensibility of descriptions, some researchers have focussed on the provision of authoring guidelines for use cases (e.g., [32], [33], and [37]). The main motivation for these efforts is that the UML specification of the use case (see [25]) does not state any guidelines to be followed when writing use case specifications. Hence, different writing styles can result in ambiguous specifications. Other researchers (e.g., [38] and [36]) have attempted to adopt formal specification techniques to produce precise use case specifications. The main argument by proponents of the use of formal techniques is that there is no precise syntax or semantics for UML use cases. However, the application of formal techniques in requirements and specifications is often a drawback (see [39], [40]) to the validation tasks which inevitably involve non-technical stakeholders. Hence, this thesis suggests that a formal approach is inappropriate for many development participants.

The core philosophy of the UML use case is to describe software behaviour as interactions between system users and the system; yet the UML offers no explicit mechanism for describing dependencies amongst constituent interaction steps of the use case. The UML specification (see

[25]) states that every use case should express a sequence of interactions that are independent of any other use case. This implies that use cases specifying the same system must not communicate or have associations with one another. A contradiction is that the UML however, describes some types of relationships between use cases (e.g., <<include>> and <<extend>>). Both <<include>> and <<extend>> imply the existence of use cases describing functions which are not necessarily complete and do require communication between the base use case and the included/extending use cases. Included use cases can be used to handle exceptions that might result in unrealistic computations. On the other hand, <<extend>> means that the extending use case is inserted at a designated extension point if a particular condition is true. Intra-use case dependencies such as "event ***e*** requires that event ***q*** has been previously executed" cannot be expressed in UML. More worryingly, inter-use case dependencies such as "use case A requires that use case X has been previously executed" cannot be formally expressed in UML. More often though, applications in the real world exhibit subtle interactions that are far from obvious and are difficult to describe without considering how they interact ([11], [41]). Furthermore, in reality, use cases and use case elements do interact. Hence, the property of independence of use cases cannot hold where decomposition of a system is crucial to its understanding. Indeed, other authors have noted that the independence rule is often flouted in practice ([42], [43]). It is argued in this thesis that as part of system specification and validation, it is crucial that interactions and dependency issues are explicitly described and clarified.

1.4 Approach to solving the problem

The traditional way of constructing models where interaction and dependency issues are of concern is the use of a process algebra suited to the task (e.g., [44] and [45]). Within business process modelling, graphically based models (mainly RADs) are deployed in constructing diagrams of partial processes which are then composed to depict a whole system ([46], [47]). The verification of the correctness of such models is sometimes conducted using formal descriptions of the process within an automated environment such as that described in ([48], [49], [50] and [51]).

In most commonly used process algebras (e.g., [45]), especially within business process modelling, it is well known that the flow of logic within processes is not always sequential. Processes are largely inter-dependent, both between themselves and within their constituent process steps. Hence, there are points of synchronisation due to processes invoking each other, or indeed waiting periods involving roles completing tasks in a context-dependent way [46].

Similarly, use cases are not always self-contained, transitive entities because often there are a number of actors needed to fulfil a system function and a number of interacting steps necessary to reach a desired computation result. The UML specification of the use case does not allow for the description of state-based relationships among use case elements. Hence, this thesis borrows from the process modelling community the concept of state-based specification where interactions between roles depend on the states of the roles. This approach enhances the level of detail in use case descriptions with state-based information, whereby each constituent event is augmented with a pre and post state. The 'pre' and 'post' states for each event are used as a mechanism for facilitating the consideration of dependencies amongst constituent events of the use case. The thesis proposes an automated environment for authoring and animating descriptions written using this state-based approach as a means of clarifying interactions and dependencies within use case specifications.

1.5 Research objectives

This research addresses the weaknesses of the use case description as a software specification notation; in particular, the research seeks an enhanced structure of the use case notation whereby the resulting specification is amenable to analysis of interaction and dependency issues. It is the position of this thesis that a state-based approach where use case events are augmented with named 'pre' and 'post' states provides insight about dependencies (and interaction issues) that cannot be gained from standard use cases.

Hence, there are three main objectives of this research:

1) To provide an enhanced structure of the use case description where dependencies and interaction issues can be delineated as a means to requirements validation. The enhanced structure provides a means for augmenting use case events with state-based information.

2) To provide automated support whereby the enhanced use case description (from objective one) can be authored and enacted (animated) to reveal the behaviour that would result from implementing a software system based on the specification. This enaction provides behavioural prototypes that depict the implications of the specification to the stakeholders.

3) To validate the efficacy of the enhanced use case structure and tool support via student workshops, seminar presentations, examples from literature and an industrial study. The aim of this validation is to determine the extent to which the enhanced structure and tool help in producing specifications that match the expectations of the system stakeholders as compared to the standard use cases.

Many state-based approaches and associated tools adopt a formally-based specification notation (e.g., [49], [36]) for the description of system behaviour. The construction of such specifications (and their readability) poses learning overheads to most non-technical participants [52]. This thesis provides a means to obviate any need for formal specification or translation of use cases. Thus, the adopted approach follows the controlled natural language guidelines of [37] (similar to those suggested in [33]). This way, the construction of specifications based on the proposed approach does not require the learning of any intermediate formal specification language. This ensures that the simplicity of the use case description is preserved. A number of student workshops, more formal workshops with colleagues, and an industrial project are reported to evaluate the proposed approach and tool support.

1.6 Overview of thesis contribution

The work presented in this thesis seeks to address a requirements engineering issue pertaining to the validation of software requirements and specifications. UML use cases have wide industry uptake as a specification notation, but offer an inadequate means of validating the specifications against stakeholder requirements. In particular, this thesis argues that use cases have no means for elaborating intra-use case dependencies or inter-use case dependencies. This thesis proposes a state-based approach, the Educator approach for authoring use case specifications whereby dependency issues are made explicit. This is the central contribution of the thesis. Additionally, the thesis presents a proof of concept tool, the EducatorTool, which acts as vehicle for illustrating Educator-based use cases. The Educator approach affords the capability of automated simulation of use case specifications within the EducatorTool, and this is considered an important aspect of validation.

1.7 Thesis roadmap

The rest of the thesis is structured as follows:

Chapter 2 provides an analysis of use case literature, and an analytical review of literature regarding the software engineering issues that are within the scope of this thesis. A review on software requirements and specifications is made; the chapter highlights the significance of rigorous validation of specifications, including the difficulties experienced in specification and validation activities. A discussion of the requirements engineering discipline is made including

suggested ways of surmounting the difficulties experienced. The chapter reviews the objectives of the thesis within the context of industrial strength CASE tools and other less holistic tools.

Chapter 3 discusses the Educator approach, including the extent to which it is informed by some process modelling approaches. The chapter discusses the Educator approach's dependency analysis mechanisms, and the potential for enacting Educator-based specifications due their state-based nature.

Chapter 4 discusses the EducatorTool, the motivation for its development, the architecture of the tool and the different types of enactable capabilities provided. The chapter also demonstrates the usage of the tool, including the adopted syntax for authoring descriptions.

Chapter 5 presents the process and the iterations that resulted in the development of the tool. The chapter outlines the various workshops and seminars that were conducted to obtain feedback in the refinement of the initial EducatorTool prototypes. The refinements were aimed at improving enaction functionality and the user interface of the tool.

Chapter 6 provides a presentation of an industrial study conducted as a proof of concept for the Educator approach and tool. The chapter presents a description of the company where the study was conducted; the nature of software systems developed by the company; the purpose (i.e., to provide a specification of real-time monitoring systems on the one hand, and to conduct an empirical application of this research on the other) of the industrial project is also discussed. The chapter outlines some of the data gathered, including a presentation of the data analysis and discussions on the findings from the study. The chapter discusses some of the limitations of the work presented in this thesis, including indication of directions for further work.

Chapter 7 offers some conclusions; the chapter reviews the objectives, and outlines the main contributions of the thesis; a summary and conclusion are also provided.

2. Analytical review

2.1 Introduction

Many difficulties relating to software requirements and specifications are often associated with inadequate communication ([53], [54]) between stakeholders and developers. Other difficulties emanate from a poorly managed requirements process ([55], [56], [57]) whereby inaccurate assumptions are made regarding the problem domain ([58], [59]). Further issues curtailing the requirements and specification process are to do with inadequate conflicts and inconsistencies handling techniques (see [18] and [60]).

In order to address some of the above issues (e.g., communication difficulties) and elaborate the effects that a system would have on the problem domain, the specification process must be able to expose any tacit issues or assumptions early during the specification process. Hence, a coherent requirements process must address many subtle issues allowing for the interrogation of the specification as a means to clarify requirements (e.g., [61]). This chapter discusses these requirements issues and the existing attempts (and their limitations) to address them.

2.2 Requirements Engineering (RE)

Zave [62] offers a definition of RE that is widely accepted by other researchers (e.g., [63] and [43]). In [62], RE is described as the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families.

Simply put, RE is the process by which software requirements are determined and the specification constructed. There are many inherent and subtle difficulties in the RE process.

2.2.1 RE Difficulties

One of the key products of a RE process is the specification which details the behaviour expected of the system under development. Specification of software behaviour is a difficult task ([64], [65]). It is a type of problem that has been termed uncertain ([66], [67]). The reasons for difficulties in constructing specifications include the fact that the specification activities are ill-structured and open-ended, and the knowledge available is incomplete [68]. More worryingly,

there is no notion of a finished specification, and the only criterion for stopping is some form of satisfaction ([69], [70]) on the part of the participants.

Four major areas of difficulty can be identified (see [56] and [71]) as contributing to the problems associated with the determination of requirements and specification:

1 The knowledge sources are diverse and vary greatly in the quality of information and exposition.
2 The actual knowledge takes many different forms that are often inadequately described.
3 Often, the contents of the specification need to be negotiated amongst competing resources or conflicting stakeholders.
4 The requirements may be difficult to comprehend and the specification difficult to construct due to unfamiliarity with the problem domain.

2.2.2 Surmounting RE difficulties

In order to produce software that meets its intended purpose, software engineers must conduct some form of validation. One such way is the writing of a specification in a formal specification language that ensures issues such as consistency are exposed or detected. The other way is to construct models of the specification that are amenable to automated analysis such that stakeholders can scrutinise the validity of the specification against their expectations. These two ways are discussed in sections 2.2.2.1 and 2.2.2.2.

2.2.2.1 Formal specification

Hunter ([18], [60]) proposes the rigorous translation of a specification using weakened classical logic as a means of testing the internal consistency of the specification. The use of weakened classical logic is meant to allow useful reasoning in the presence of inconsistencies within the specification. As a demonstration, Hunter conducts an analysis of various specification fragments using the proposed logic. The use of mathematical logics does not lend itself to ease of analysis by non-mathematical stakeholders who inevitably need to comprehend the specification during the validation process [39]. Furthermore, mathematical clarity does not always equate to a useful specification as often stakeholder views may still have been missed in a mathematically rigorous specification ([72], [73]).

Others have attempted to formalise notations such as process modelling notations (e.g., [51]) or some parts of UML. An example is the translation of UML use cases with the Z (e.g., [36]) and LOTOS (e.g., [44]) formal specification languages. In most of these efforts, the aim is to produce a precise specification that can be used within an automated environment for validation or testing. For example, in [36] a use case is first structured around a model based on temporal logic that allows the use of operators such as choice, repetition and interruption on specification parts during execution of tests. The temporal logic model and the Z language offer some powerful specification features that are not afforded by the UML specification of the use case. However, the argument that formal translation of use cases is necessary in order to conduct any automation (for testing) is not true since the work presented in this thesis indicates otherwise. For instance, chapter 6 shows that a system can be specified and validated using an enactable scheme that does not require the adoption of formal techniques.

2.2.2.2 Prototyping specifications

The traditional software engineering concept of prototyping is becoming increasingly popular during specification and validation (e.g., [74], [75]). Prototyping generally involves developing mock-ups of the system with an aim to initiate further scrutiny of the requirements and specification. Two main ways of system prototyping are throw-away and evolutionary prototyping ([68], [76], [77]). Throw-away prototypes are system mock-ups that are not meant to be kept after their use in further elaboration of the system's capabilities. Evolutionary prototypes are the mock-ups built with an aim to be incrementally developed and evolved into a fully working system. Sommerville [68] argues that since the objective of evolutionary prototyping is to "quickly" deliver a working system to end-users, the development starts with those requirements which are best understood. On the other hand, the objective of throw-away prototyping is to validate or derive the system requirements and the prototyping process starts with those requirements which are poorly understood [68].

In either case, the developers produce a partially working system that is either thrown away or changed. For large systems, the time taken to produce executable prototypes can be long and development effort may be wasted due to extensive changes (or throwing away) of the prototype [78]. This thesis argues for a prototyping scheme based on the specification rather than any implementation of the specification. That is, instead of producing executable mock-ups of the

system, it is considered far cheaper to prototype the specification (e.g., [79], [80], [81]) as a means of validating the implied behaviour before any further development.

2.3 UML support for Requirements Engineering

The UML is a non-proprietary modelling language widely used in (object oriented) software development. UML includes graphical notations that are used to create abstract models of a system – UML models. The dominant notation for requirements and specifications within the UML is the use case notation.

2.3.1 Use cases

A use case is a sequence of transactions in a system whose task is to yield a result of measurable value to an actor of the system ([22, 82]). Hence, a use case is more than a scenario as it comprises of the set of possible scenarios that indicate the different ways a system can be used to accomplish an external user's goal [83].

The use case notation is generally considered appropriate for describing and validating functional requirements ([84], [85]). The reason for this is that the use case notation describes system behaviour in terms of user-system interactions, hence making it possible for system stakeholders to identify the extent to which those interactions reflect their expectations of the system.

The standard elements of a use case are stated as follows (see [33], [86], and [30]):

1 Use case name (describes the actor's goal)
2 Actors (user types that interact directly with the system)
3 Trigger (what starts the use case)
4 Pre- and post-conditions (system states before and after the use case is performed).

A use case can be viewed as a vehicle for describing scenarios since use cases describe general system use whilst scenarios depict instances of this use. Hence, a use case approach is a means of collating and organising scenarios ([87], [54]). It is surprising that despite these potentially useful functions (e.g., organising scenarios) of use cases, there are no elaborate mechanisms for organising use cases with regard to their validation beyond the <<include>> and <<extend>> relationships. Indeed, the <<include>> and <<extend>> relationships are essentially aimed at

organising commonly accessed behaviours rather than organising constituent events or distinct use cases to determine the consequences of their execution.

Schneider and Winters [28] introduce the concept of primary and secondary scenarios to use cases. The primary scenario describes a most typical success scenario without any alternatives, referred to as a "happy day scenario". Secondary scenarios are the alternatives not addressed in the primary scenario. Taken together, the primary and secondary scenarios combine to make the complete use case. Fowler and Scott [30] describe a use case as a collection of scenarios that share a "common user goal". Cockburn [33] concurs with these definitions by observing that use case descriptions are an appropriate specification medium because they offer a means to step through the logic of the description from start to finish. There is no indication regarding what the consequence of executing the third, or say fifth step before the first would be in any given system use circumstance.

From the discussion above, it is clear that many proponents of use cases (e.g., [28], [88], [89]), assume the sequential execution of use cases, their constituent scenarios and actions. This assumption disregards the implications of executing certain actions prior to others, or certain scenarios prior to other scenarios, or indeed, particular use cases before (or after) other use cases. This thesis investigates these issues with an aim to provide a coherent mechanism by which dependencies and interaction issues within use case actions can be scrutinised and clarified.

2.3.2 Scenario-based techniques

Emphasis is laid on the term "scenario-based" techniques in this section because there has been a growing trend in several application areas for system behaviour to be modelled using a variety of notations for depicting scenarios. Several authors (e.g., [90], [91]) also deploy state machines to enhance scenarios.

Just like scenarios within the use case discourse, these other techniques recognise scenarios as partial stories which, when combined with other scenarios provide a more complete system description. This makes scenarios particularly well suited for incremental specification of software behaviour ([92], [93]) because stakeholders may develop descriptions independently, contributing their own view of the system to those of other stakeholders. Given that several scenarios can be written to describe different usage circumstances, composition algorithms are

often designed (see [38]) to enable the integration of the behaviour of different scenarios into a model that is amenable to behaviour analysis of the whole system.

Usually, scenario-based techniques are used in the early specification of a system (e.g., [94]), while state machines are used for dynamic modelling (e.g., [95]). The rigour afforded by some dynamic modelling techniques (e.g., state machines) should not be limited to software design alone. Behavioural modelling is essential also for understanding existing systems.

A widely used notation for scenarios (e.g., within the telecommunications industry) is the Message Sequence Charts (MSCs) [96]. MSCs are variations of the UML sequence diagrams [29]. Whereas MSCs and sequence diagrams have intuitive diagrammatic structures they are geared to constructing specifications based on one scenario per diagram. Clearly, one scenario conveys relatively little information, and often many scenarios are needed to provide a significant system description. In such circumstances, the combination of a number of scenarios into a coherent whole becomes a significant issue. Hence, how do researchers who focus on MSCs relate their partial specifications? There are two common ways of tackling this issue:

1) By inferring the relations between scenarios (using synthesis algorithms e.g., in [38] and [97]).

2) By requiring the relationships to be explicitly stated by stakeholders (e.g., [85] and [65]).

In the latter case, abstractions should be provided to specify these relationships. Several different answers are proposed to address this issue. For instance, [96] introduces a graph-like notation that shows how the system evolves from one scenario to another. The underlying notion used in [96] is that of scenario composition. The composition involves a process whereby new scenarios can be defined in terms of other scenarios by composing with sequential, choice, and iteration operators.

Kruger [98] suggests a different approach to that of scenario composition. This involves the identification of state conditions instead of composing behavioural fragments in MSCs. State conditions identify common states throughout different scenarios. Thus, two state conditions equally labelled on different scenarios indicate that the scenarios have a common point in the interactions they describe [98]. This allows the system to switch scenario when it reaches the common state.

There are a number of strengths and weaknesses regarding MSCs composition and state labelling approaches.

Some weaknesses of scenario composition mechanisms include:

1 Composition approaches may contend with a situation where a large number of very short scenarios must be composed in complex ways to describe the system's overall behaviour. This is an inherent weakness of MSCs (and sequence diagrams) [99].

2 Some composition algorithms may be too complex for non-technical stakeholders to comprehend and use in composition of scenarios ([100], [101]).

Some strengths of composition mechanisms include:

1 Promoting scenario reuse as they are composed to develop more complex system behaviours.

2 There is no requirement on the part of the stakeholder to specify scenarios with some kind of state machine model in mind.

On the other hand, state identification approaches are a convenient way of introducing complex behaviours without having to split scenarios into parts ([102] and [103]). A main system run can be described in one scenario and then accorded its global states in order to relate it with other scenarios. In addition, state identification can be used to introduce varied information about the system and may provide means for progressively moving into a more detailed over-all system description. On the negative side, requiring explicit identification of states requires consistency from stakeholders when constructing scenarios and forces them to reason about their system in terms of state machine models rather than sequences of actions. This requirement to be consistent on the part of the stakeholder is not the case when deploying the standard approach. Furthermore, lack of bespoke tools for supporting state identification approaches adds to the problems experienced in adopting the approach. The state identification approach of [98] is simplistic since it only endeavours to identify global states rather than states for constituent scenario steps. In this thesis, a more elaborate state identification approach is adopted where states for constituent use case actions are considered as a means to analysing intra-use case and inter-use case dependencies.

2.4 Automated use case environments

There have been recent attempts to provide automated environments where use case specifications can be authored and analysed. This section discusses some of these attempts, their strengths and shortcomings.

2.4.1 The UCEd approach

The UCEd approach is described in [84]. The approach considers use case descriptions of software behaviour and aims to produce animations of textual use cases. UCEd is similar to the Educator approach in two important ways. First, UCEd intends to retain the simple nature of the use case description by obviating any need for formal specification techniques. Second, the adoption of an animation scheme in order to provide a visual artefact to which stakeholders can react and validate.

A UCEd use case is either a normal use case or an extension use case. This categorisation is similar to the UML <<extend>> relationship (see [25]). A UCEd use case description comprises a global precondition and a global post condition. Again, these global conditions are part of the UML specification of the use case. UCEd offers a limited elaboration of conditions that may be associated with the use case steps. These conditions are:

1 Added conditions - the conditions added to a system after a use case step executes.
2 Withdrawn conditions - the conditions withdrawn from the system after a use case step executes.

Both added and withdrawn conditions are postconditions and it is not clear the need to declare two postconditions for a use case action.

The UCEd approach requires the modeller to construct both the use case description and a domain description. A domain description consists of all possible actors (termed concepts in the UCEd approach); the operations (or actions) of the concepts and their corresponding added and withdrawn conditions. A further UCEd description is a state machine description which is automatically generated by UCEd prior to undertaking the animation process. A state machine description itself comprises of two parts. The first part is a set of states (extracted from the domain description), and the other part comprises actions that are responsible for state transitions. During description simulation, a UCEd user should select a start state, and the action that would

trigger a transition from that state to a chosen end state. There are a number of problems with this approach:

1 There is laboured production of multiple artefacts surrounding the use case specification. The initial aim to preserve the simplicity of the use case is compromised by requiring users to construct and use domain models and state machine descriptions in a simulation.

2 The UCEd use case is based on Cockburn's use case template (see [33]). Cockburn's template recognises the existence of both primary actors and secondary actors for use case steps. The UCEd approach only proposes a primary actor for a use case step. There is no mention of secondary actors that may be involved in executing use case steps alongside primary actors.

3 There is little value to be associated with a UCEd simulation; tool users are allowed to make their own choice of their desired transitions; rather, it is far more useful to produce an animation based on the written description so that users scrutinise the extent to which the existing specification reflects their expectations.

4 The animation itself does not work as described in [84]; there is no functionality in UCEd to produce the simulations argued for.

5 The comparison made in [84] with Harel's Play-In/Play-Out approach [34] is erroneous. UCEd does not automatically generate a Graphical User Interface nor does it offer any animation functionality as suggested by the author.

2.4.2 The Play-In/Play-Out approach

The Play-In/Play-Out approach is a scenario-based specification mechanism [34]. The system's behaviour is captured (played-in) as scenarios using a Graphical User Interface (built using say, Visual Basic). A play-engine automatically generates a formal version of the played scenarios in the language of Live Sequence Charts (LSCs). LSCs are a visual formalism for specifying the scenarios of a system. LSCs allow for representation of scenarios that are mandatory, those that are allowed but not mandatory, and those that are forbidden. LSCs are an extension of Message Sequence Charts (MSCs), which do not make such distinctions.

Play-in is a mechanism for capturing use cases using a graphical user interface of the target system. The output of this process is a LSC specification. Play-Out is the process of testing the use cases by executing the LSCs. Hence, the input to the Play-Out process is a formal LSC specification. This formal specification (translation of LSCs) is done using a dialect of PI calculus.

A similar approach to Play-In/Play-Out is the Labelled Transition System Analyser (LTSA). The LTSA approach also accepts scenario specifications of behaviour regarding interactions amongst software components. The scenarios are written using the Finite State Process (FSP) algebra. An execution of FSP scenarios provides a graphical view of state transitions depicting inter-component communication. A problem with both Play-In and Play-Out and LTSA is that both approaches require learning effort on the part of non-technical modellers. For instance, a client who is unfamiliar with FSP or PI calculus may not be able to alter the specification even if the animations reveal incorrect behaviour.

2.4.3 The ArtScene approach

Maiden [104] describes ArtScene, a web-enabled tool that is claimed to enable organisations to generate and walk through scenarios, and thus discover the complete and correct requirements for new computer system. In this way, ArtScene is viewed as an enhancement to the use case notation. An issue with the ArtScene approach is the extent to which a modeller can generate scenarios as suggested in [104]. That is, it is not clear whether the claim that ArtScene is useful in scenario generation is practical since scenarios are descriptions of system behaviour rather than some processed data from a computer system.

In [104], it is argued that a project team using ArtScene writes use case specifications using structured templates embedded within the tool. The specification is then parameterised and parsed to enable ArtScene's two-step scenario generation algorithm to generate one or more scenarios. In the first step, the algorithm ought to generate a normal course scenario from action ordering rules and generation parameters in the use case specification. This is not demonstrated at all, nor does this author's analysis of the ArtScene approach provide any of the suggested scenario generation mechanism. Whereas ArtScene [104] observes that each different possible ordering of normal course events is a different scenario, ArtScene does not demonstrate the way in which ordering of events is handled or managed to generate new scenarios. It is suggested that in the second step, the scenario generation algorithm produces candidate alternative courses, which are expressed as "what-if" questions for each normal course event. To achieve this, [104] argues that a database containing over fifty different classes of abnormal behaviour and states in socio-technical systems is queried. These classes are based on error taxonomies in the cognitive science, human-computer interaction and safety-critical disciplines. The algorithm is not presented in its original form, nor is it demonstrated within ArtScene, hence, it is not likely to argue that ArtScene provides any more than an editor of use case descriptions based on HTML hyperlinks. Further issues with

ArtScene are summarised as follows. For the specification to be parsed, it has to be written according to some ordering rules. Those rules are not elaborated anywhere. Moreover, there is no indication to the specification authors what the useful parameters would be for ArtScene to conduct the first step. Additionally, the generation of scenarios from an initial specification seems a far fetched idea. It is not clear how different types of systems can have their scenarios automatically generated based on some underlying algorithms. Different problem domains have different roles and interaction issues ([59], [69], [105]). The ArtScene approach does not indicate how a main scenario (or alternative scenarios) is determined automatically for any possible use case of the system under consideration.

2.4.4 Executable Use Cases (EUCs) approach

The EUCs approach is presented in [106]. Example applications of the EUCs are outlined within specifications for a medical system [106] and banking system [89]. EUCs have three tiers, namely, the informal tier for expressing system behaviour using standard textual use case descriptions; the formal tier for providing the formal, executable translation of the use cases (using coloured Petri-nets), and the animation tier which provides graphical animation of the formal tier. The central argument in this three-tiered approach is that construction of visual prototypes requires the presence of a formal, interpretable product (hence the use of Petri-nets). Moreover, such a formal tier is considered necessary for any automated checking of the specification. The emphasis is on the iterative development of a specification by moving through the 3 tiers, and providing graphical animations to check the validity of the formal tier. The work presented in this thesis shows that it is possible to construct specifications that are amenable to automated analysis without delving into any formal specification techniques. The main concern is whether the development of the formal tier requires skills that many stakeholders might not have, and whether the efforts for developing such a middle-ground product is useful in comprehending the expectations of the stakeholders. Furthermore, similar work (see [51]) had been done close to a decade before the efforts on EUCs. The finding of [51] was that the creation of a bespoke interface for each specification task was too much effort, especially for business stakeholders.

2.5 Chapter summary

The focus on formal translation of use cases is mainly driven by the lack of precise syntax and semantics of the use case notation. The drawback of adopting formally based techniques is that non-technical users are often put off by such specifications.

Additionally, much focus has been on the management of use case descriptions using templates around which descriptions are written. Related to this idea is the research on guidelines and rules for authoring textual use cases. For much of these efforts, a lacking ingredient is the focus on the first principles of the use case notation. That is, the specification of software behaviour from the user's perspective and the elaboration of that specification to validate its adequacy against the user's initial expectations of the system.

Whereas there are well established notations (e.g., MSCs and sequence diagrams) for delineating use case instances (or scenarios), most of these notations are limited in the amount of detail they can reveal. For example, MSCs are used alongside formal specification languages to enhance their lack of detail and informality.

In view of these shortcomings in the use case specification, this thesis identifies a need to enhance the use case notation in order to make it amenable to dependency analysis. Inevitably, consideration of dependencies subsumes the interrogation of interaction issues. The thesis proposes the use of controlled natural language based on existing guidelines (see chapter 4, section 4.3). The aim is to keep the use case description simple while offering rigour of scrutiny via automated support where enaction can be applied to interrogate dependency issues.

3. The Educator approach

3.1 Introduction and motivation

There are various shortcomings of the specification approaches discussed in chapter 2 (section 2.4). For instance, most of the approaches adopt a formal specification technique (e.g., process algebra), which raises the learning overhead for the non-technical user. An additional issue with most of these approaches is that they require the production of multiple artefacts with a degree of completeness that prohibits further progress when requirements and domain information are not readily available. This thesis outlines general characteristics of a specification model that addresses some of these shortcomings, whilst focussing on dependency analysis.

Rather than adopt a rigid specification process (that also requires learning of formal languages), the analyst needs a model that can guide his or her expertise [56]. This model should support the creative input and interpretive skills of the analyst. The work reported in [51] suggests that such a model should be:

1. Flexible – the model should allow partial specification with incomplete information (e.g., with missing states or actors). For example, the approach outlined in this chapter allows analysis of use case descriptions for dependency information, and the subsequent revision of such descriptions upon finding new information about the description.
2. Co-operative – as many participants may be involved in the specification process, a consistent view of the specification should be providing for optimal co-operative specification and validation. That is, the specification model should enable co-operation among specification participants such that written information is commonly understood.
3. Amenable to automated analysis – the model should allow specifications to be prototyped during validation tasks.

However, in order to facilitate some automation, a degree of structuring must be introduced. An approach that supports an incremental, iterative process is needed in order to allow for revision and addition of newly elicited information. The individual steps that build the descriptions arise out of dialogues among participants, and a model that is overly prescriptive will severely limit the scope of these dialogues, possibly causing vital steps to be missed. Therefore, any automated support or formalization must account for, and indeed encourage, dialogue in the exploration of the current specification.

In order to allow the participants to control the process, the model should allow any order of discussion. In other words, it cannot be guaranteed that needed information will be provided immediately. [107] discusses the informality of human communication, listing abbreviation, ambiguity, poor ordering, incompleteness, and contradiction as key features. These features represent an essential part of the human thought process, as a means of dealing with complexity. People present ideas in the order they occur, not in an order which is convenient to the hearer. In particular, the human mind is adept at ignoring inconvenient consequences of particular statements with the intention of clarifying them later.

Finally, the model should allow the participants to delay the resolution of issues involving relationships among stated facts and the making of decisions. Requirements engineering is primarily an exploratory process [108], involving the gathering and formulation of knowledge. It is vital, therefore, that it does not become overly restricted by premature decisions [67]. A model for the specification process should encourage participants to gather all the relevant knowledge and explore all the issues regarding the problem before making a decision.

3.2 Clarification of terminology

The Educator approach draws upon work within business process modelling, especially, the use of enaction to construct business process descriptions that are amenable to automated prototyping. In describing the terminology used within this thesis (and particularly, this chapter), consideration is made of the meaning of such terminology within those other areas that inform this thesis.

A term that may be considered suitable for an activity (here termed the event) within a use case description (in the *subject verb object* structure), is the term *action*. Within business process modelling using Role Activity Diagrams (RADs), the term action has the definite meaning of an activity that is undertaken solely by the containing role (or actor in use cases) without interacting with any other role (or actor). The intention of the Educator approach is to be able to depict and model interactions among actors (within a use case and across distinct use cases), hence the term action seems inappropriate.

Whereas the term *event* was chosen for this purpose, the author is aware that other communities have different interpretations of the term event. For instance, events are usually considered to be external stimuli that may trigger the occurrence of activities (e.g., in UML activity diagrams) or state transitions (e.g., in UML statechart models).

3.2.1 Steps and Events

Within use case modelling, each use case line is often referred to as a step. This thesis observes that there is a strong relationship between event and step since the occurrence of an event amounts to completion of the step which contains it. It has to be noted however, that, these are not really the same thing. That is, it is only by adhering to use case guidelines (discussed in section 4.3) that each step has a single event, rather than this being imposed by the notation itself.

3.2.2 Events and States

A further subtle distinction concerns the idea of events (activities) being controlled by pre and post states. In realising that each event has an associated pre and post state (in the simplified state model) it is tempting to think of these as pre and post states of the event. However, in reality they are not states of the event at all, since events do not have state. For role based models, the roles have state, and for the Educator approach, the state is that of the actor. Hence, an event (activity) can only occur when it has been invoked by the associated primary actor.

3.2.3 Invoking Events

We reiterate that, within traditional state machine based formalisms, events are understood to be external stimuli for state transition or occurrence of an activity. Hence, the concept of 'invoking an event' may seem counter-intuitive. Even within the concrete example of the enactable model, the act of clicking on a choice (of activity), is often thought of as being an event (indeed, one might write a program which deals with mouse click events, and so on).

In taking these issues into consideration the following terminology is adopted; illustrations are provided with reference to lines from a simplified use case description:

1 Lecturer gives pen to Student

In the above description, the *Lecturer* is termed the primary actor, and the activity, *gives pen* is referred to as an event. A further realisation is therefore, that, events are associated with actors. For example, *gives pen*, is an event of the *Lecturer* actor. Hence, the enactable model would place the event *gives pen* within the actor window for *Lecturer*. *Student* (the object here and another actor) is referred to as the secondary actor.

The entire line is referred to as a use case step.

When pre and post *states* are added, the complete Educator-based description appears as follows:

		Lecturer		Student	
		pre state	post state	pre state	post state
Lecturer	gives pen to Student	has pen	no pen	no pen	has pen

For the *gives pen* event to occur, the *Lecturer* (primary actor) must be in the state *has pen*, and the *Student* (secondary actor) must be in the state *no pen*. Hence these states act, in effect as guards, or simplified preconditions on the event (the activity of giving pen) taking place. (Note that they are not quite the same as preconditions, as will be described below). However, these are actually states of the actors, not of the event. Similarly, the occurrence of the event causes the actors involved in this entire step (an interaction) to move to their post states.

This simple mechanism (as is explained in further examples) of augmenting a description with pre and post states enables modellers to control interactions among actors. That is, *Lecturer* and *Student* cannot interact, by way of the pen exchange (which involves invocation of the event *gives pen* of the *Lecturer* actor) without either actor being in the correct pre-state. Hence, the dependencies among events are also controlled (that is, the events that must occur in order for actors to change state such that other events may occur).

The relationship between event and actor has been described, so far, as an association. For the description to be enacted, the idea that events belong to actors allows each event to be visible within a given actor window.

3.2.4 The Educator state model

The state model adopted here is adapted from the RolEnact process modelling language, which in itself, is an enactable representation of Role Activity Diagrams.

One way in which this model differs from other state-based approaches is that the role (or, within Educator approach, the actor) has only one state variable, and thus, can only be in one state at any given time. Given the intention to be able to consider multiple dependencies, such as event x is dependent on completion of events y and z, this may seem to be an important limitation of the

model. However, the mechanism to deal with this issue, which entails creating individual roles for each parallel thread, each containing a state variable, also has some advantages for process and procedural modelling. A key advantage is the rigour afforded to the modeller in the analysis of interactions among actors (system users) within the domain of interest. A further advantage is that the model is much smaller and easier to comprehend, since instead of having multiple concepts (e.g., states, triggers, preconditions, etc) there are only states of actors to consider. For the purpose of making annotations to use case descriptions this is an important consideration given that only one further concept need be added.

In the following sections, simple RAD models are provided to help further clarification of the terminology used within the Educator model.

3.2.4.1 Pre and post (states and conditions) – case refinement

As an example, consider two independent events, *get apples*, and *get oranges*, of some actor (or role) each which result in the post states, *has apples* and *has oranges*. One might consider a third event, *make juice*, which can occur when either apples or oranges have been obtained. Traditionally, a guard on such an event might be a precondition such as *has fruit*.

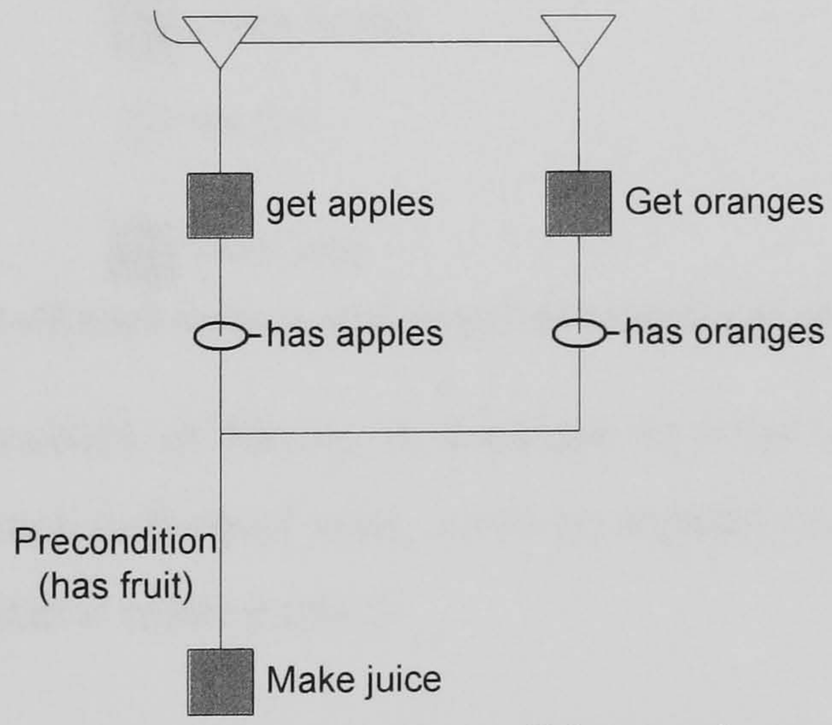


Figure 3.1: Traditional precondition used as a guard

Whereas the RAD model above is simple and elegant, it requires further understanding (semantic load) on the part of the non-technical reader. Furthermore, the model relies on the implicit understanding of the domain, thereby 'hiding' some issues that may prove crucial to validation. For example, one needs to know, or assume, that apples and oranges are both fruits. In implementation terms one might verify that they are objects of fruit type, but again this relies upon such understanding being clear and present in the first instance.

The Educator state model differs from the model shown above in that within Educator, the pre-state for make juice has to be an exact match and this requires an extra step. The extra step brings together two threads (independent behaviours) into a single state, *has fruit*. That is, one can still arrive at the state *has fruit*, as a result of either thread. Importantly, at any given time there is still only one state for the role (or actor), and hence a further simplification, for both understanding and implementation, is preserved. However, since state change requires an action, this means that there is a need for a further (often artificial) action in order for the actor to be in some more general state (e.g., *has fruit*). This need for an extra action may be seen as a weakness since it diverges from the way in which we imagine the problem domain to behave (analysis), and is also an extra effort for the modeller.

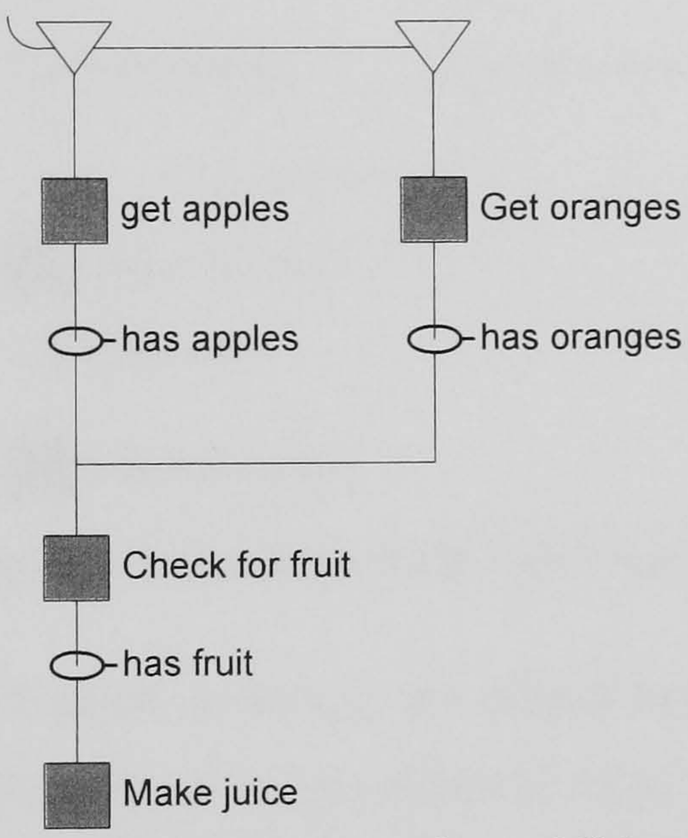


Figure 3.2: RolEnact state model requires additional action and state

In spite of the apparent drawback of having to consider an extra action, this latter mechanism facilitates consideration of each individual state, again an arguably positive benefit for validation, and the check that one has fruit is made explicit.

However, the use of these pre and post states is unusual and, hence, is differentiated, here, from the more flexible (but arguably more complex) notion of a precondition. (Note, however, that many validating such models would not come with such expectations or notational baggage).

A similar but more complex situation occurs when instead of having either one state or the other, an event is dependent on both. That is, instead of choice there is parallel behaviour, and the subsequent action requires that both conditions are satisfied.

3.2.4.2 Pre and post (states and conditions) – part refinement

Suppose our event is now *Make smoothie*, which requires that when we have fruit, we actually have both apples and oranges. For a use case we would be required to choose that the gaining of apples and oranges occurs in some arbitrary sequence. That is:

1 Fruit Finder get apples
2 Fruit Finder get oranges

However, in reality one might gather these fruits independently and in any, often unknown order. Process modelling notations such as RADs represent such behaviours as parallel threads:

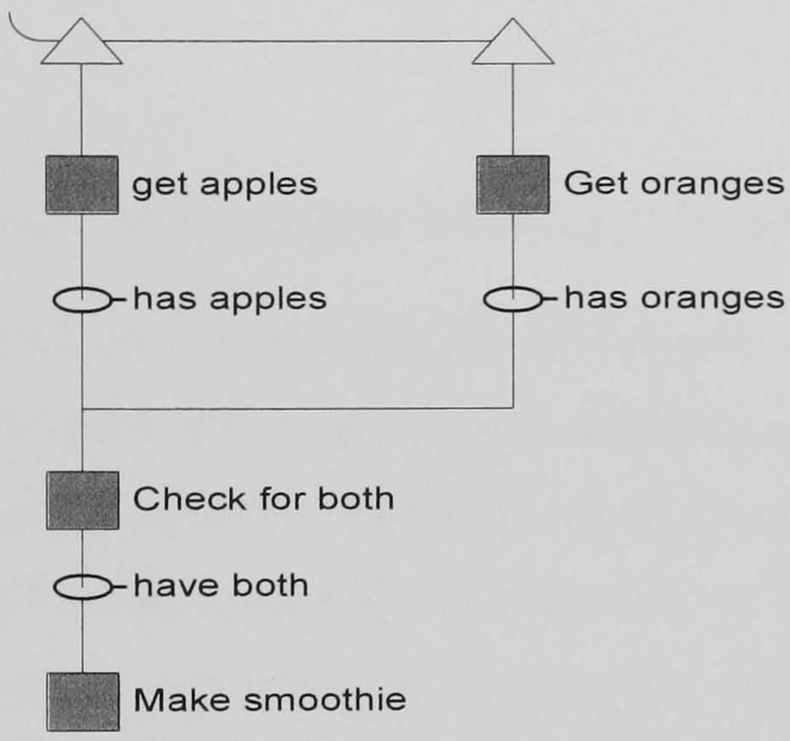


Figure 3.3: Standard RAD view of parallel

In RolEnact, and, hence, in Educator, however, we cannot have such parallel threads, and we employ the mechanism of splitting the role into different roles, each of which carries one of the state variables (the having apples or having oranges states). Below is a RAD representation of role of *Fruit Receipt* (left) and the separate roles *Apple Receipt* and *Orange Receipt* (right):

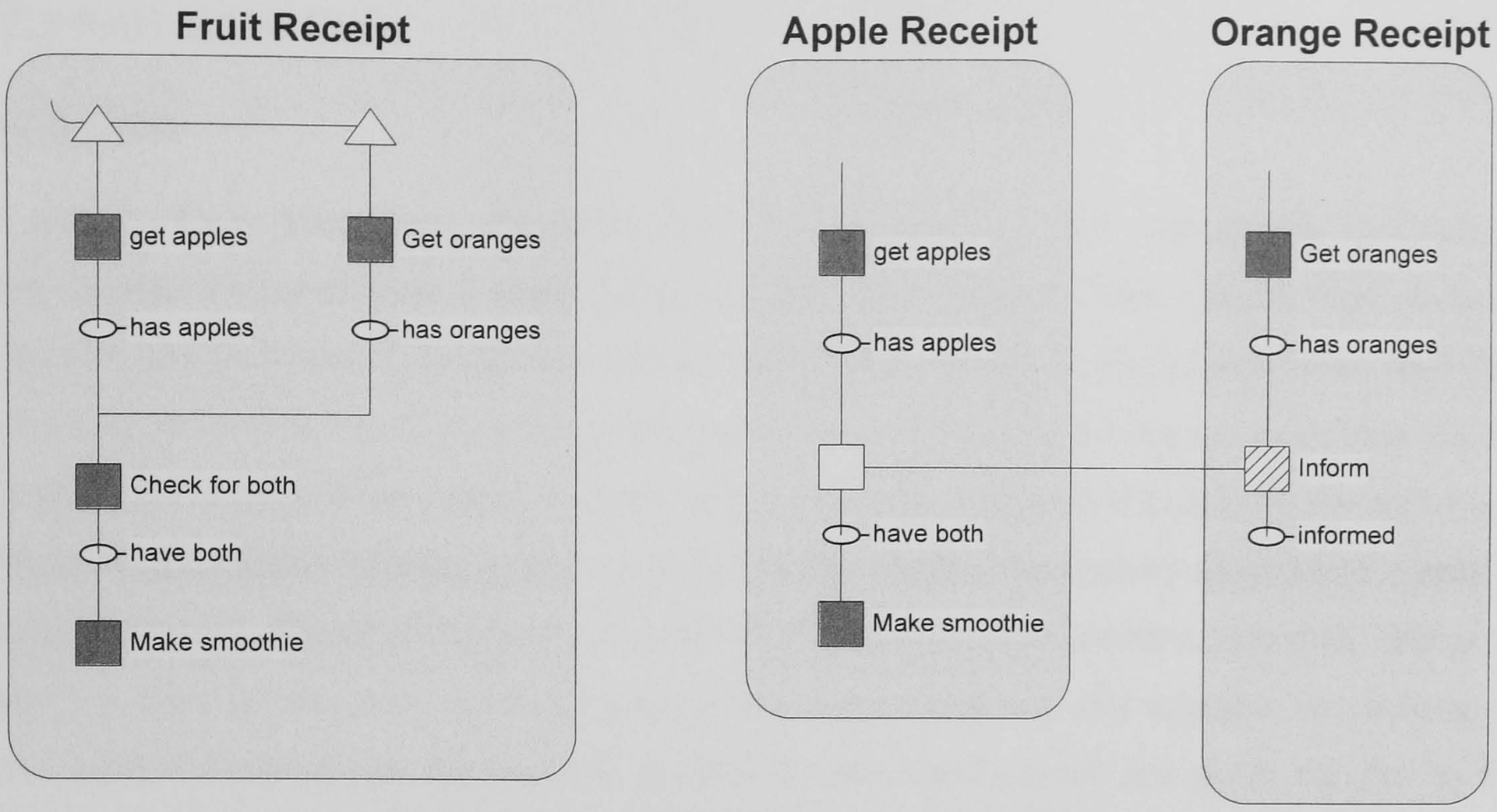


Figure 3.4: Separating roles in a process (RAD)

Since the threads (*Apple receipt* and *Orange receipt*) are separate, (as with choice) a mechanism is employed to join them, this time via an interaction. Since the interaction can only take place when we have both apples and oranges, the interaction replaces (rather than adds to) the action of checking. At first this may still seem particularly cumbersome, as we must have unique identifiers, different names for the sub-roles. However, separation into different roles is useful in other respects, both from a process perspective and a use case perspective.

Often, a user would consider the model as a vehicle for investigating the efficiency of current behaviour. For example, a common issue found with parallel behaviours in roles, is where one role appears burdened by many such tasks, and may often become a bottleneck. Consideration of parallel tasks as belonging to separate roles highlights the possibility that these sub-roles could be carried out by other resources, and further enforces the distinction between the role and the person (or resource). For example, in the fruit gathering it might seem reasonable to divide into apple and orange gathering roles, since typically these fruits may come from different suppliers, and there is no need for them to be received at the same time, or by the same person.

In use case descriptions, such situations tend to occur across distinct use cases. For the Educator state model, these are termed inter-use case dependencies. Consider the fruits example discussed earlier. It might be that one can have separate use cases for receiving apples, receiving oranges and making smoothie. Hence, a simple naming rule, such as *UCname.actor* will give unique identifiers to the actor in each use case, e.g., *receivingoranges.fruitperson.*

3.2.5 RAD as Use Cases

Interactions

In order to show interactions, the notion of a driving (primary) actor and passive (secondary) actor is also borrowed from a similar concept (driving and active roles) within Role Activity Diagrams and RolEnact. Consider the interaction between a *Lecturer* and a *Student* (as described previously in section 3.2.3). Here the primary actor is considered to initiate the interaction (or for Educator, the event). When Educator enacts this interaction, the event will only be shown (it will belong to) the window of the *Lecturer* (primary) actor. That is, the primary actor alone is able to initiate the event. The secondary actor is considered passive in the enaction, and their change of state is a result of the event (activity) *gives pen*, being carried out, (for example by clicking on *gives pen*). Of course, for the example described, once the *Lecturer* has given the pen to the *Student*, the *Student* may do likewise, that is they may act as the primary actor to give pen to the *Lecturer*. That is, the event *gives pen*, would then become available within the actor window for the *Student*. This also illustrates that an actor, may at different times, that is, for different use case steps, be either a primary or a secondary actor. In addition, Educator adopts the convention that use case descriptions will be written such that the primary actor is always the driving actor, and is always the subject of the step.

3.2.6 Discussion of advantages and disadvantages of the state model

Some of the advantages and disadvantages associated with the model adopted are summarised as follows:

- A model needs to spawn a role where there are multiple state variables. This can be seen as both an added complication and an issue for drawing space and layout.
- Additional roles are, however, often viewed as advantageous. For example where there is parallel behaviour one may wish to highlight the fact that this other (independent processing) could be carried out by another resource, or may even be another role.
- A model needs additional actions (or interactions) to join threads or to combine states. That is, both x & y, or x or y, would then lead to another state.
- Classic precondition hides states or implies behaviour. Making states explicit facilitates consideration of the states of the process.
- The precondition also requires some understanding on the part of the reader (semantic load), which may not be obvious for unfamiliar models.

- A significant consideration is that the model is intended to be accessible by business users, or typical use case writers, who may not be familiar with state models (and indeed, will not carry that baggage). Hence, only one additional concept is required.

3.2.7 Summary of terminology issues

In this section on clarification of terminology, illustrative fragments of process models have been used:

- To demonstrate how RADs represent business processes using states.
- To discuss the extent to which RolEnact provides an automated environment for constructing and enacting process descriptions that are equivalent to a RAD model.
- To discuss differences between RAD and RolEnact models.
- To discuss Educator model and how it is informed by the RolEnact model.
- To discuss terminology differences in both RAD and RolEnact modelling on the one hand, and Educator modelling on the other.

The choice of this model was not merely pragmatic (for implementation reasons). The Educator model is motivated by the need to provide an accessible notation, rather than focussing on modelling power and flexibility (as is evident in RolEnact and similar modelling environments). It is reiterated here that the idea behind the approach was to have a simple addition to use cases descriptions which would not be onerous for the modeller, either in terms of learning or effort.

3.3 Issues addressed

3.3.1 Validation

The user-system description of a system (as provided by use cases) is often considered a benefit for validation ([94], [83]) since users can identify the types of behaviours they expect from the system when they use it. In this thesis, validation is supported by considering dependencies among events in use case descriptions. In particular, the thesis argues that a use case specification approach that makes such issues explicit is needed. Furthermore, such an approach should retain the simple nature of the use case specification, in order to provide an accessible means for validation. That is, considering states for actors in a use case description helps clarify interactions among those actors. Additionally, the use of enaction on such descriptions helps modellers to step through the logic of the description, thereby clarifying the extent to which the description matches their expectations.

3.3.2 Dependencies and interaction issues

UML use cases have been widely taken up by industry, but use cases do not offer support for the purposes of rigorous specification and validation of users' expectations of system usage. Some of the issues outstanding are discussed below.

The UML specification of the use case (see [25]) argues that two or more use cases specifying the same system should not be associated, since each of them individually describes a complete usage of the system. Furthermore, to state anything about the internal behaviour of the actor, apart from its communications with the system, is also prohibited. However, this prohibition is limiting in practice since system users often act in context dependent ways during their use of the system [109]. The actions of one user often determine what other users are able to do with the system. Hence, use cases must be able to represent contextual behaviours that reveal interactions among constituent use case (activities) events and system users (or actors). The divide and conquer technique is an old software engineering (and mathematics [110]) adage. Jackson [11] observes that the conquered and solved parts must then be integrated to form a system-wide solution. The divide and conquer technique can be deployed in use case modelling to establish relationships that help integrate partial use cases of a system after decomposition. Such application of divide and conquer and use case integration would help to improve understanding of the overall system behaviour.

The UML specification ([25]) of the use case also places some constraints on the way use cases should be associated. For example, the UML requires that use cases can only be involved in binary associations (*include* or *extend*). In effect, this forbids a multiple-call by one use case to many use cases from which the calling use case might need some computed result. Other modelling approaches (e.g., role activity diagrams, which are used within business process modelling - see [46]), allow these types of process associations. As an example, consider the widely cited (e.g., by [38], and [37]) ATM system scenario. A cash withdrawal use case might need the behaviour (functionality) of a check balance use case, or a print statement use case depending on what a customer withdrawing cash might want to do during the withdrawing process. Hence, the restriction on the number of associations that can be made among use cases seems a drawback on the expressiveness of the use case notation, since real world processes interact in complex, and often, in ways that are dependent on the state of the actors (or roles) in the process. Another constraint on associating use cases in UML is that use cases cannot have associations to use cases specifying the same system or subsystem. In other words, UML forbids

any associations between use cases describing partial behaviour of the same system (or subsystem).

Additionally, UML requires that a use case cannot include use cases that directly or indirectly include it. In other words, if a withdraw cash use case includes a check balance use case, then check balance cannot include withdraw cash use case. This does not make logical sense as a check balance use case can inform the concerned ATM user on how much they have in their bank account so that the ATM user makes a decision on how much to withdraw. In other words, such prohibition is not practical for many real world applications.

3.4 Application of process modelling

A benefit of some process modelling techniques such as those reported in [49] and [47] is that business processes are built around the concept of interactions among roles in the business context. The process descriptions are state-based and interactions among roles during performance of the processes can be animated within an automated environment. Such approaches have been shown to enhance shared understanding between modellers and business stakeholders (see [51], [111], [47]).

3.4.1 Role-based process models

Within business process modelling, a widely used notation for business process representation is the Role Activity Diagram (RAD) [46]. RolEnact [51] provides an environment for constructing business process descriptions that are equivalent to a given RAD model. RolEnact has its own formal specification language for writing process descriptions. The essence of the RolEnact environment is to provide functionality for writing business process descriptions within an environment that affords the capability of enacting the descriptions. Enaction constitutes stepping through the process descriptions to reason about the logic of the process. As an example, consider the interactions between two users of a pen, a *Lecturer* and a *Student*. This description was discussed briefly in section 3.2. Again, suppose that, to start with, the *Lecturer* has the pen. The *Lecturer* performs the *gives pen* event and moves from the *hasPen* state to the *noPen* state. The student moves from the *noPen* state to the *hasPen* state after participating in the event. The Student can in turn initiate the *gives pen* event; hence the *Student* moves from the *hasPen* state to *noPen* state, while the *Lecturer* moves from the *noPen* state to *hasPen* state. A RAD representation of this scenario is shown in Figure 3.5:

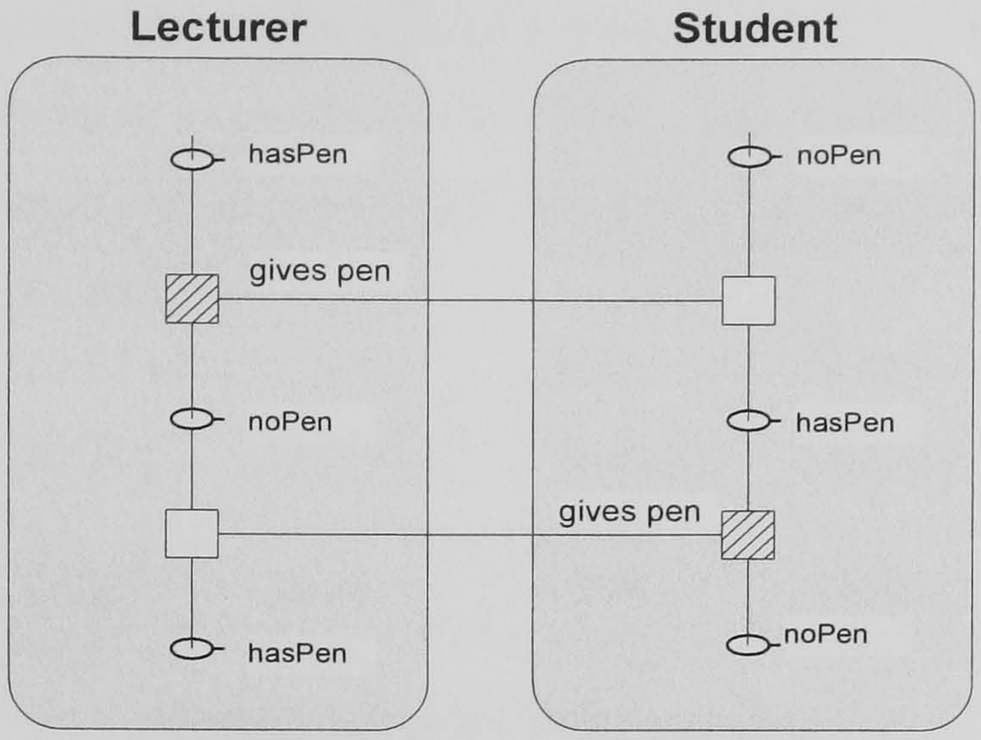


Figure 3.5: RAD for Pen exchange process

The RolEnact description equivalent to the RAD in Figure 3.1 is shown below:

```
Interaction Lecturer.gives pen
        Me(hasPen→noPen)
        Student(noPen→hasPen)
End
Interaction Student.gives pen
        Me(hasPen→noPen)
        Lecturer(noPen→hasPen)
End
```

A problem with the RolEnact description is that the process description has to be written with the RolEnact process specification language and compiled within the RolEnact environment. This thesis argues that use case specifications that are amenable to automated analysis should be constructed without resorting to an intermediate formal translation of use cases with another language.

3.4.2 State-based use case descriptions

A use case equivalent of the RAD shown in Figure 3.5 is constructed such that pre and post states are considered for each primary actor of the event. Moreover, consideration is made of the secondary actors (and their states) that may participate in an interaction with the primary actor. By considering named pre and post states associated with constituent events (the states actually belonging to the actors), the Educator approach adopts an accessible means for constructing state-based models of use cases that will enhance rigour in validation. This augmentation appears to the user as a simple addition to each use case step (see Table 3.1). However, the additions are actually controlling the event (though for our approach steps include only single events, and are therefore equivalent in practice). In addition, the states are of the actor, even though it is their

association to each event (and hence each use case step), that will be the focus of scrutiny. This state-based model is regarded as accessible to modellers since it does not require adoption of a formal specification language. As an illustration of these ideas the complete Pen exchange use case description is shown in Table 3.1:

Primary Actor	**Event**	**Pre state**	**Post state**	**Secondary Actor**	**Pre state**	**Post state**
Lecturer	gives pen	hasPen	noPen	Student	noPen	hasPen
Student	gives pen	hasPen	noPen	Lecturer	noPen	hasPen

Table 3.1: Pen exchange use case

The concept of state-based use case descriptions is not common within the UML community. Where consideration has been made regarding expression of dependencies in use case descriptions (e.g., Glinz [112]), the inclusion of states in each use case step (or event) is viewed by some [112] as a drawback to the clarity of the description. However, Glinz's [112] view is based on the assumption that state-based information is written using a formal specification language (e.g., a process algebra). In contrast, the results of the study reported in chapter six indicate that it is not necessary to use formal specification to add state information to use cases, and this obviation of formality reduces the additional effort required.

The author recognises that thinking about states may be harder than working with default use case descriptions, but the benefits of increased rigour (with use of states) may outweigh the extra specification work. Moreover, many proponents of UML use cases (e.g., [33] and [27]) recognise the importance of expressing interactions between primary actors and secondary actors, but do not offer any support for clarifying such interactions. The approach proposed in this thesis considers the need for elaborating dependency issues, and interactions among actors as a means to validation.

3.4.3 Intra-use case and inter-use case dependencies

In the use case description shown in Table 3.1, the invocation of each use case event is dependent on the state of the invoking actor. Intra-use case dependencies arise in a use case due to consideration of interactions among the involved actors within the use case, and for the events within that use case. However, often, a system may have several use cases, which may be inter-dependent. Dependencies across multiple related use cases are termed inter-use case dependencies. In order to understand further why such dependencies are important, or might

occur, consider how one might view interactions among roles within processes in a Role Activity Diagram (RAD).

RADs (including the RolEnact environment) group activities into roles (similar to actors). Hence the process is decomposed into roles, rather than decomposed by functionality, with actions spread across those roles. In contrast use cases decompose by (high level) function, and thus actors can be associated across, and associated to different, use cases. That is, the same actor, may exist in different use cases. Hence, it is often the case, that, in contrast to the advice of the UML, actions (for us events) of an actor in one use case impact the events of that same actor (or indeed other actors) in other use cases.

Hence, there is no duplication of roles within RAD, but for use cases an actor can exist in any number of related use cases. These issues will now be illustrated by consideration of the same example, grouped:

a) By roles, as for a normal RAD

b) By use case (ignoring roles or actors)

c) By separating each actor that is involved in multiple use cases into separate unique roles, where each role represents that actor for a particular use case, and is named accordingly.

That is, for the final example, c, the actor name is changed to include the use case to which it belongs. This both clarifies the link to use case, and provides a way of identifying this role uniquely.

Example

Consider the Pen exchange use case shown in Figure 3.5, and the description shown in Table 3.1. Also, suppose that there is another use case describing a Course registration scenario. The Course registration description is shown in Table 3.2.

Primary Actor	**Event**	**Pre state**	**Post state**	**Secondary Actor**	**Pre state**	**Post state**
Lecturer	volunteers for courses to teach	initial	coursesAgreed	Registrar	waiting	coursesAgreed
Registrar	prepares course list	coursesAgreed	listDone	Student	waiting	listDone
Student	chooses course to study	listDone	coursesChosen			

Table 3.2: Course registration

The *Student chooses course to study* event in the Course registration use case can occur only under the following circumstances:

- The *Registrar prepares course list* event in the Course registration use case has occurred and hence the *Student* is in *listDone* state.
- The *Lecturer gives pen* of Pen exchange use case has occurred and the *Student* is in *hasPen* state.

In other words, if the *Registrar* has prepared the course list, but the *Student* has no pen, the *Student* cannot choose courses. The assumption is that the *Student* needs the pen to be able to choose courses, perhaps since they will have to write their name on a list.

To show this distinction, a RAD view for the use cases outlined in Tables 3.1 and 3.2 shown in Figure 3.6:

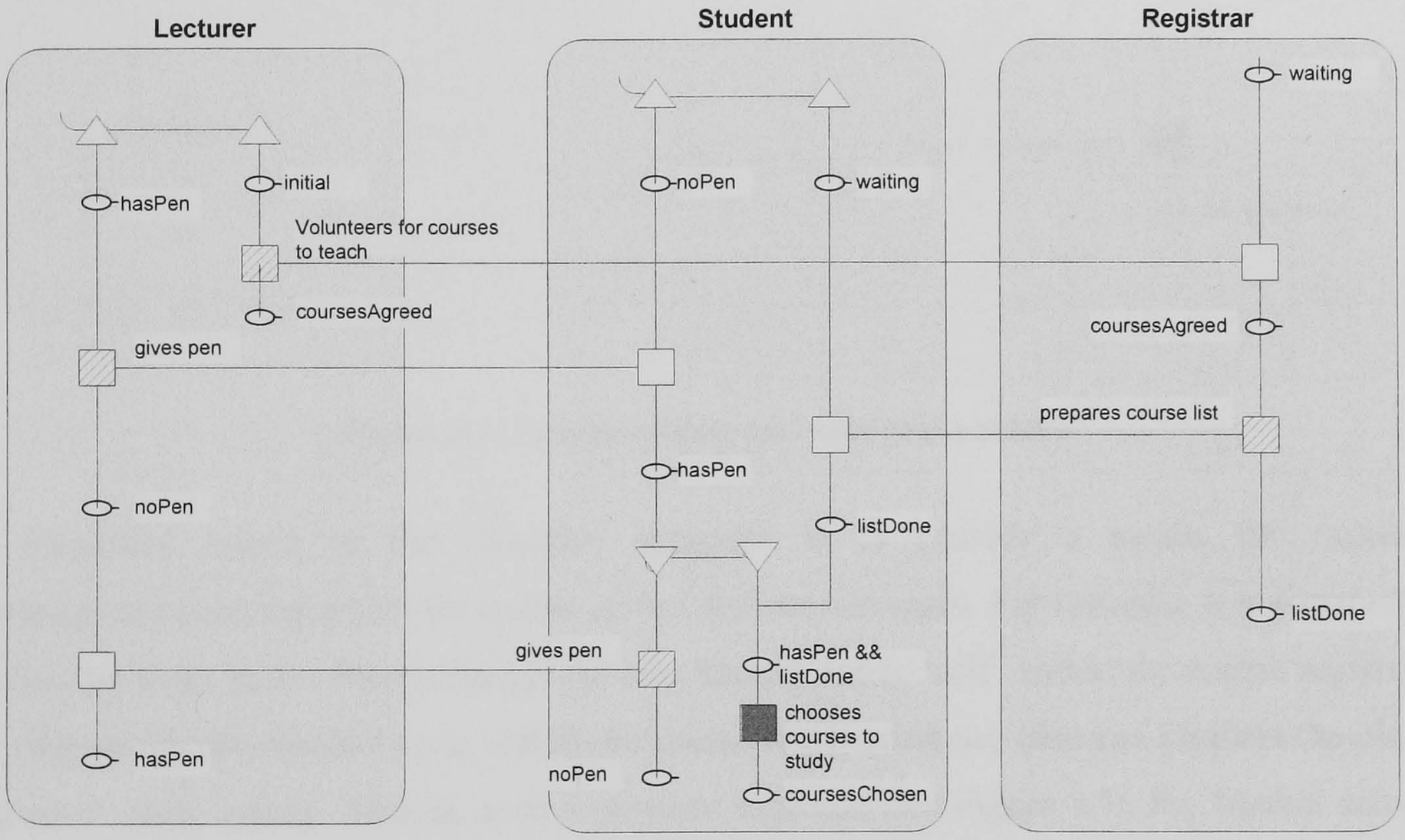


Figure 3.6: RAD for the Exchange pen and course registration use cases

In Figure 3.6, the *Lecturer* role has two threads, one thread in the Pen exchange use case and the other in the Course registration use case. The *Student* role has two threads, each in both descriptions. The *Registrar* role has only one thread within the course registration use case description.

In constructing a RAD representation of these two use case descriptions it is important to take into account the fact that the *Lecturer* and the *Student* actors appear in both use cases. Figure 3.7

is a RAD depicting the use cases of Tables 3.1 and 3.2 where the exact use case view (rather than RAD view) is maintained:

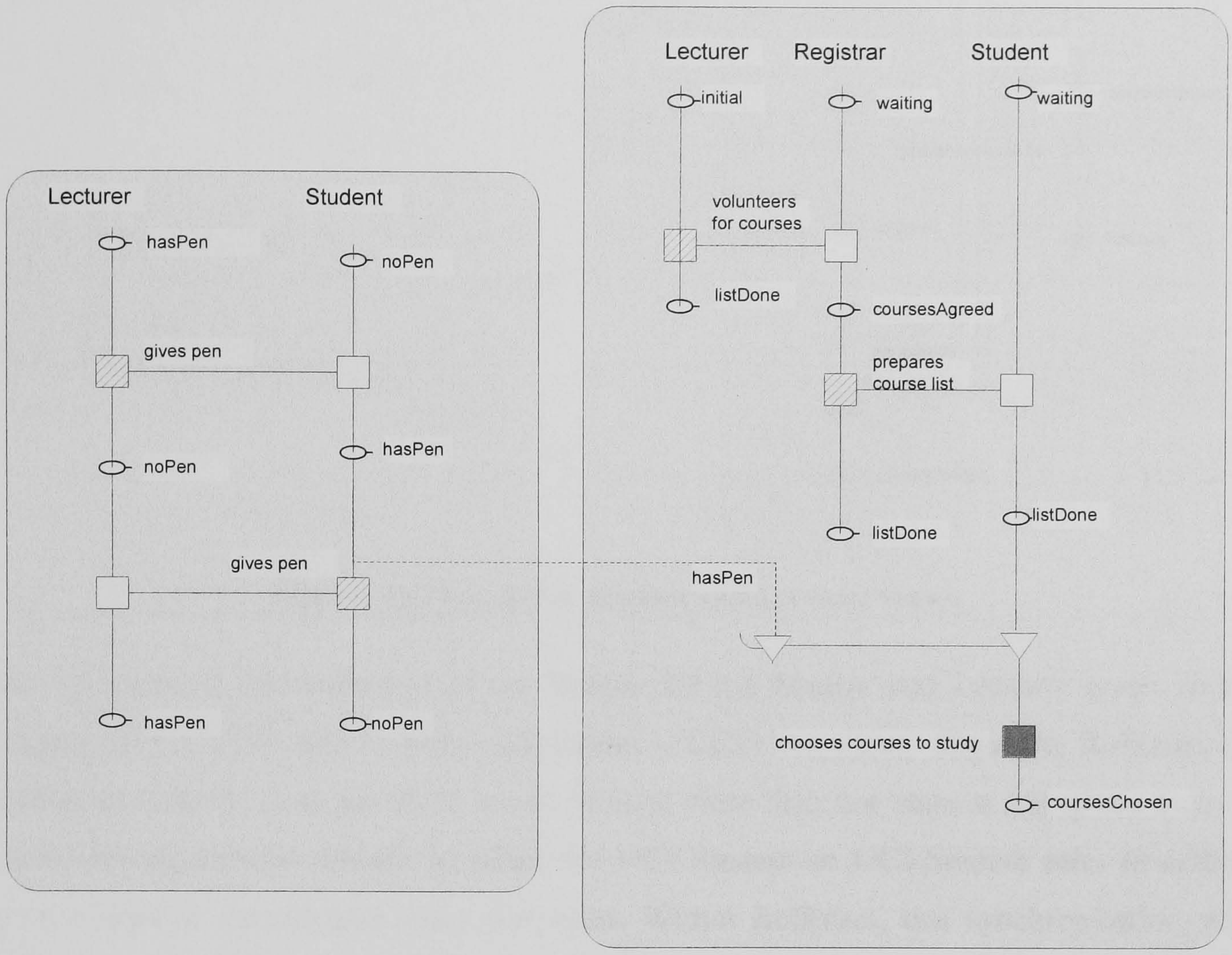


Figure 3.7: Synchronising use cases at event level

An important aspect of the Educator approach is to provide a means for expressing communication among actors for events across distinct use cases. For instance, it is apparent that the *Student* actor in the Pen exchange use case should notify 'self' within the course registration use case so that the *Student* actor within the course registration use case can perform the *chooses courses to study* action. That is, as in a previous example (see Figure 3.4), the *Student* actor has to have a means of 'informing self' that they now have a pen, so that once the list is done (by the *Registrar*), then the *Student* can choose courses to study. In the Course registration use case (UC2), the *Student* (UC2.Student) is a secondary actor in the Registrar's event for preparing course list. Hence, the UC2.Student moves from the waiting state to the *listDone* state after the *prepares course list* event has occurred. In this example, the *Student* has to be in an AND state that is a compound state of *hasPen* and the *listDone* which is arrived at by constructing a pseudo-interaction between UC1.Student and UC2.Student (the *inform has pen interaction*).
These distinct threads are depicted in the RAD in figure 3.8.

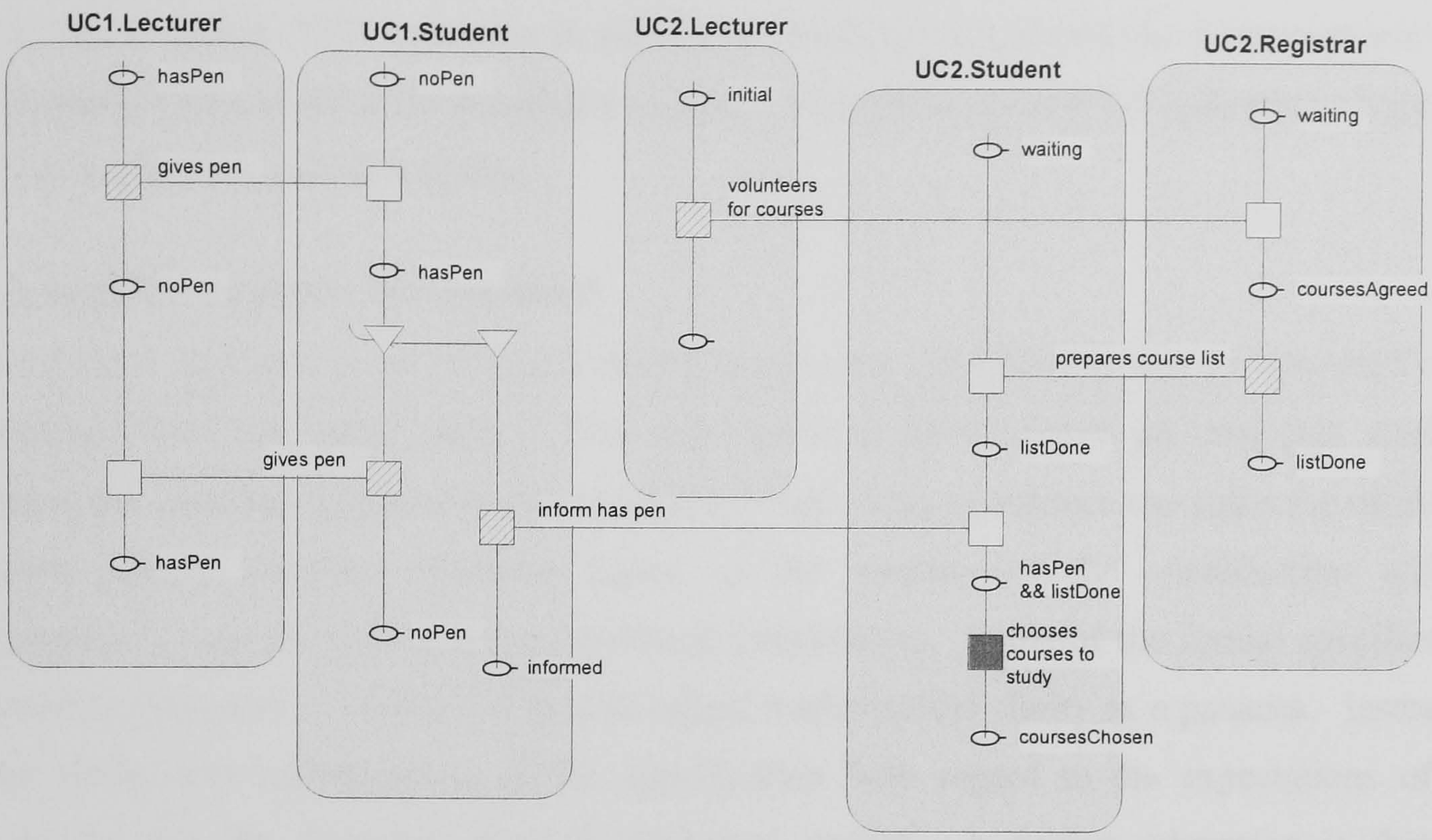


Figure 3.8: Threads for Student and Lecturer actors

Figure 3.8 shows a representation of the threads for the *Student* and *Lecturer* actors in both (Exchange pen – UC1) and (Course registration – UC2) use cases. Since the RolEnact (and Educator) state model does not allow a role to have more than one state at any given time, this precludes having parallel threads in either the UC1.Student or UC2.Student role. In addition, there is a need to synchronise these two roles. Within RolEnact, this synchronisation would normally be achieved by constructing an additional (pseudo) interaction between the respective roles, so that one of the roles informs the other of its state.

For example, an interaction *inform has pen*, would allow UC2.Student to be aware of the state of UC1.Student. However, for the interaction to take place the pre-states for both UC1.Student and UC2.Student are prescribed (being respectively *hasPen* and *listDone*). Hence, the interaction can only take place when both of these states are correct. On completion of this interaction UC2.Student would be in a state *hasPen and listDone* and would then able to take part in the *chooses courses to study* action,

While this approach works for the diagrammatic (and RolEnact) representations it is a clumsy mechanism to choose to communicate states for the implementation of enaction. Hence, within Educator, the synchronisation of both actors is achieved simply by making the UC1.Student state accessible to the relevant actors in UC2. The Educator approach simplifies this mechanism further by allowing association of distinct use case descriptions at event level. This mechanism

means that a logical AND operation is performed on the events across the concerned use case descriptions to reason about the possibility of a particular use case event being dependent upon an event in another use case description.

3.6 A need for support environment

The Educator approach is an enhanced means to use case specification where dependency and interaction issues are made explicit. The view taken in using named pre and post states to augment use case steps (events) is that an approach that seeks to enhance use cases for validation purposes should consider affording rigour in the scrutiny of the specification without compromising ease of reading by non-technical stakeholders. Some of the formal specification approaches discussed in section 2.4 tend to regard mathematical clarity as a panacea. Instead, it is the clarity and understanding of the specification with regard to the expectations of the stakeholder that the Educator approach considers crucial. A further advantage is that the approach offers the capability for the enaction of descriptions by the provision of simple tool support. Hence, the following outlines the motivations for the support environment.

3.6.1 Motivation

In software engineering, construction of models that depict interactions of design artefacts (e.g., components) is a common practice. Attempts have been made to provide precise semantics for scenarios to make them amenable for model construction and behaviour description. For example, [113] describes a process algebra (Finite State Process (FSP)) for specification of scenarios and an automated environment in which FSP scenarios are parsed to provide an animation of the described component's behaviour. The novelty of such an approach is the construction of behavioural models of components and the animation of model descriptions. Moreover, where the emphasis is primarily on model construction during design, the use of formal specification techniques (e.g., FSP) may not be a drawback as the audience could be engineers or academics with experience in such techniques. However, during the requirements process, participation by stakeholders with little or no experience in formal specification techniques is common. Hence, an alternative light-weight approach should be adopted to construct behavioural models and the animation of those models to 'tease out' the implied behaviour. There are two main factors that inhibit the application of formal techniques in the requirements process:

- Constructing models for behavioural analysis remains a difficult undertaking requiring considerable expertise ([73], [39]).

- Given the difficulty stated above, the validation benefits appear at the end of the often lengthy construction process, and users often have little involvement in the construction of models, and hence do not 'own' them.

Whereas adoption of a wholly formal approach is an impediment to non-technical stakeholders, an entirely informal approach would also produce ambiguous model descriptions that are difficult to validate. This thesis argues for the adoption of a semi-formal approach where controlled natural language is used, augmented with states in a natural and accessible manner.

Dependency information is an important part of the specification process, and tools to help identify and clarify dependencies are needed. Whilst considering dependencies is essentially a human activity, a range of options need to be explored whereby the human participants can use a visual model to share information regarding dependencies. That is, constructing specifications from initial participant's comments and demonstrating to the participants the implications of their statements is crucial to involving the relevant stakeholders. This thesis argues that animation of specification models provides a coherent mechanism by which all stakeholders can visualise implied behaviour and revise the specification as necessary.

Additionally, given adequate background knowledge about the domain, it should also be possible to provide a degree of automated critiquing, to supplement the manual critiquing process. This might involve the enforcement of syntactic guidelines that underpin the way in which specifications should be written. It is noted in [73] that automating the clerical work of detailed checking of specifications is an ideal way to supplement the human activity.

There are four key points that make it necessary to consider the development of tool support for the Educator approach:

1) To provide a vehicle that illustrates the use of state-based information as a means for intra-use case and inter-use case dependency analysis.

2) To offer automated support for authoring descriptions in the Educator approach.

3) To provide enactable functionality as a means to validating descriptions via visual prototypes.

4) To support authoring rules such as those suggested in the CP guidelines.

3.6.2 Overview of features

In addition to a coherent specification model, consideration needs to be made regarding what kind of automated support is necessary for requirements and specification tasks. Automated tools should form an environment in which the knowledge collected can be organised, manipulated and interrogated.

This thesis envisages and presents (see chapters 4, 5 and 6) an environment that comprises functionality for recording aspects of the domain that might not necessarily feature in the software specification. The tool has functionality to store notes regarding recorded specification steps. The information repository and the existing specification state is continually added to, and/or revised; hence any reasoning is non-monotonic in that new knowledge may invalidate previous conclusions. The incremental refinement of descriptions inevitably involves adding details such as exceptional case behaviour to fix problems that are discovered when descriptions are validated.

3.6.3 Enacting Educator descriptions

Educator descriptions are state-based, and Enaction is the process of stepping through a description to further scrutinise implied behaviour. This stepping through is possible given that pre and post states are used to control interactions among actors as they perform constituent steps (event) of the use case. This is analogous to the way in which states are used to control analysis of state-based business process descriptions within the RolEnact environment. This section discusses Enaction as the Educator approach's mechanism for fostering rigour in the scrutiny of dependency issues within use case specifications.

An example

Consider a safety-critical example- a Cardiac Pacemaker. A detailed description of a cardiac pacemaker is available in [114] and a partial specification using state charts can be viewed in [99]. Brief documentation of a pacemaker is given below:

A cardiac pacemaker is an implanted device that assists cardiac function when underlying pathologies make the intrinsic heart rate too low or absent. In a nutshell, a pacemaker refers to a pacing system which comprises of two subsystems:

1) The pacemaker subsystem which encases a pulse generator with a battery, a timer and a sensing device.

2) Pacing lead which carries tiny electrical pulses from a pacemaker to the heart (pacing) and also relays information about the heart's electrical activity back to the pacemaker (sensing).

Depending on medical diagnostics, the doctor can programme the pace maker to pace the Atrium (mainly right atrium), the ventricle (mainly right ventricle) or both (dual pacing) right ventricle and right atrium. To pace either an atrium or ventricle, the doctor must set the right pace maker component to monitor the heart's electrical activity from that chamber. Depending on the sensed heart activity, the pace maker can either be inhibited (so that no pacing is done) or triggered (so that pacing takes place).

From the above narrative of the general working of a pacemaker, a requirements engineer would want to construct a concise specification of the behaviour detailing how the supporting software should behave to assist the working of an artificial pacemaker. In use case terms, this involves identifying the actors that are involved in the pacing process, and their associated events. A 'first-cut' use case description of a pace making system is as follows:

1. Physician programmes pacemaker's impulse timer
2. Pacemaker interrogates the heart's electrical activity
3. Physician sets pacemaker's mode
4. Pacemaker sends pacing impulse to the heart
5. Heart conducts pacing impulse
6. Pacemaker ignores cardiac events for some time
7. Pacemaker resumes monitoring cardiac events

Figure 3.9: Pace maker use case description

Given the standard use case description of Figure 3.9, the basic question that a requirements engineer would want to address is: under what circumstance does an event (e.g., event 2 or event 3 occur)? That is, is event 3 dependent on say, event 2, 1, 4, all of them, or none? Unless the problem being specified is well known, the time-order dependency assumed when writing use case descriptions can be wholly misleading, and often dangerous.

The essence of the Educator approach is to inter-relate events as means of facilitating a controlled interaction model for scrutinising event occurrences amongst primary and secondary actors. Table 3.3 is a state-based version of Figure 3.9:

Primary Actor	**Event**	**Pre state**	**Post state**	**Secondary Actor**	**Pre state**	**Post state**
Physician	Programmes impulse timer	initial	timerSet	Pacemaker	initial	timerSet
Pacemaker	Interrogates heart activity	modeSet	awaitingHeartRate	Heart	awaitingPace Request	pacingRequestS ent
Physician	Sets pacemaker's mode	timerSet	modeSet	Pacemaker	timerSet	modeSet
Pacemaker	Sends pacing request to the heart	awaitingHeartRate	heartPaced	Heart	pacingReques tReceived	heartPaced
Heart	conducts pacing impulse	heartPaced	normalRateAchie ved	Pacemaker	heartPaced	normalRateAchi eved
Pacemaker	Ignores cardiac events for a set period	normalRateAchie ved	atRefractory State			
Pacemaker	Resumes monitoring cardiac events	atRefractoryState	awaitingHeart Rate			

Table 3.3: Pacemaking use case (Educator version)

In Table 3.3, event 3 has the pre-state *timerSet* and the post-state *modeSet.* This means that prior to the *Physician* setting the pacing mode, the *Pacemaker*'s timer must be set first (*timerSet),* then the pacing mode can be set (*modeSet*). The *Pacemaker* is a secondary actor since it is the device whose mode and timing is being set. The pre-state associated with event 2 (*timerSet*) is the post-state associated with event 1, which was the initially available event. Thus, the inclusion of states enables the mimicking of a state machine model where the order of execution of events is determined by correspondence of states.

A RAD equivalent of the state-based pacemaker description is shown in Figure 3.10:

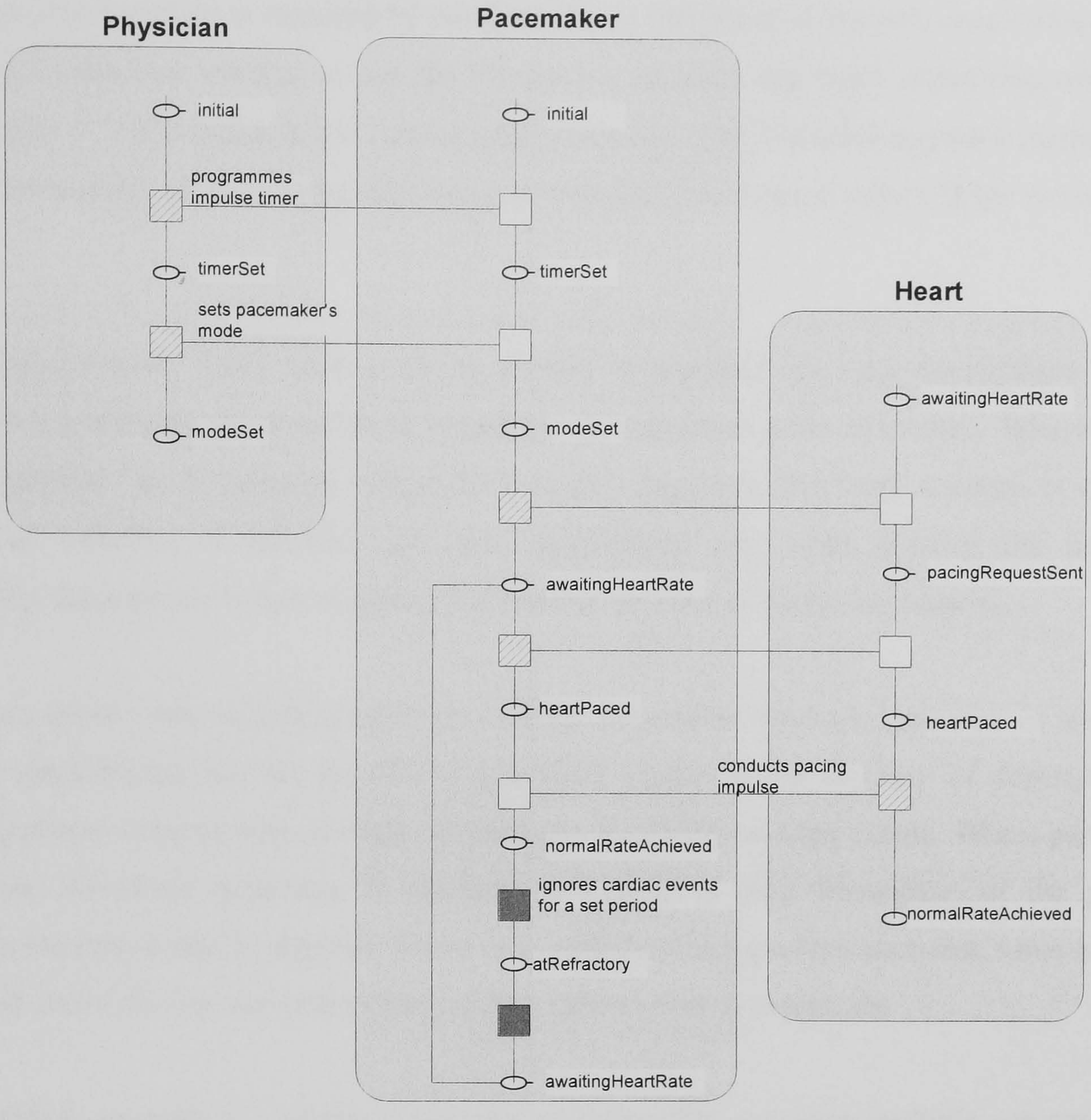


Figure 3.10: RAD for the pacemaker use case description

The RAD above (Figure 3.10) and the description in Table 3.3 indicate that the second possible event is the *Physician sets pacemaker's mode* and not the *Pacemaker interrogates the heart's electrical activity* as indicated in Figure 3.9. The main benefit for considering states as suggested in the Educator approach is to foster rigour in reasoning about consequences for each event to occur in relation to other events.

The above RAD depicts the actors in Table 3.3 as roles, and their state changes are shown both prior to and after participating in the respective events. Figure 3.10 and Table 3.3 indicate that upon the Pacemaker reaching the *awaitingHeartRate* state, the Pacemaker performs the event *interrogates heart activity* again. Hence the *Pacemaker* interacts with the Heart again. Notably, it is not possible to highlight this type of behaviour in the standard use case. In this example, a problem of missing out such a specification issue is that it would impact on the *Pacemaker* such that the pacemaking process would not be specified such that it would perpetually monitor and

provide pacing impulses as required by the heart. From the standard use case description, it is not possible to determine whether or not the *Pacemaker* resumes any heart monitoring and pacing activity after it has completed the current pacing activity. The Educator approach facilitates the explicit delineation of such behaviour using state-based information which helps to inter-relate events.

It is noted, however, that enaction can be applied to standard use case descriptions whereby modellers step through the description following the sequential order of events. Whereas this is not the intended use of enaction within the Educator approach, the work reported in chapter 6 shows that enaction of standard use case descriptions may often provide the benefit of discovering dependency issues as opposed to manual scrutiny of those descriptions.

This thesis argues that, as indicated in the Cardiac pacemaker example, enaction of state-based use case descriptions has the benefit of providing rigour in the scrutiny of descriptions by reasoning about consequences of event occurrences in relation to other events. Where participants realise that visualised behaviour is incorrect according to their knowledge of the problem, corrective measures may be applied. These may include altering actors such that some actors are associated with different use case events, adding extra events or actors, etc.

Combinatorial enaction is a means to use case enaction that considers multiple use cases being enacted in some desired combination. For example, some use cases may be required to occur concurrently, or one after the other (sequential), or that a modeller may choose which one to perform given a set of plausible use cases. The consideration of combinatorial enaction arose from the work reported in chapter 6, and an example is provided in Section 6.6.

3.6.4 Educator process

The typical process of constructing Educator-based specifications is outlined as follows.

An initial understanding of a problem may be gained by the modeller and a first-cut standard use case description constructed. The constituent events of standard use case description are then augmented with pre and post states, while also considering possible secondary actors (and their states) for each event. This augmentation process may result in adding further events or actors, or even further use cases. Hence, additional use cases may be authored as associates of the use case already under consideration.

The modeller then applies enaction to visualise the state changes of actors as they participate on performing the respective events. Enaction may result in revision of use cases as modellers gain further insight into the behaviour described by the use cases.

3.7 Chapter summary

This chapter has outlined the motivation for the Educator approach to use case specification and validation. In particular, the chapter has demonstrated the extent to which the Educator approach is informed by the RAD and RolEnact approaches, and the key differences between Educator and these approaches. The chapter has also provided a discussion of various terms used within the chapter (and the thesis) that have different connotations in other areas.

Within this chapter, an argument has been made that formally based techniques, whilst providing rigour in specification construction, sacrifice readability of the specification by non-technical stakeholders, an issue of importance during validation. Hence, the Educator approach seeks to provide rigour in scrutiny of specifications while obviating any need for formal specification techniques.

The chapter has provided an outline of the nature of Educator descriptions, including the syntax for a use case step. Aside from retaining the simple nature of the use case description, an additional benefit of the Educator approach is the production of use case specifications that are amenable to visual prototyping. Production of visual models at specification and validation stages offers the extra benefit of enhancing shared understanding among participants.

4. Development of tool support

This chapter outlines the rationale for developing the EducatorTool. In particular, the chapter discusses examples of EducatorTool's usage as a demonstration of how the tool provides support for the Educator use case approach.

Additionally, a background to the controlled natural language adopted for writing Educator-based use case descriptions is presented and the support for some of the CP ([114] and [37]) rules is outlined.

4.1 Rationale for a bespoke tool

A use case is a partial story describing a circumstance of system usage and how the system behaves while serving its external users. Two advantages afforded by UML use cases are that:

- the use case notation offers an intuitive way of describing software behaviour [117];
- use cases are easy to construct [30].

Industrial strength UML tools (e.g., Rational Rose [118] and TogetherSoft [119]) do not focus on the use case description primarily, nor do they offer any mechanism for considering dependencies. Whereas these tools lack such focussed functionality, one would think that the more bespoke, research-oriented tools such as those reported in [31] and [84] might consider specialised support for use case descriptions. The main shortcoming with most of those attempts is that they do not provide a mechanism to support their core findings. For example, despite [84] arguing for the need to adopt Cockburn's (see [33]) concept of primary and secondary actors, there is no way of supporting that scheme using the tool proposed in [84].

The general aim of CASE tools in software engineering is to provide automated support for various phases of software development (e.g., in unit-testing [120], or user interface engineering [121], [122]). Much of the focus on automated software engineering research has been on the later stages of the software process, namely verification of components' behaviour (see [123]) and model checking ([124], [125]). Software model checkers (see [126]) typically work by extracting a state machine (e.g., in the form of a transition system) from code. The (often incorrect) assumption of such efforts is that the specification phase will have been completed successfully [127].

The Rational Unified Process seems to recognise the role of states in specifications and recommends that a use case description should include a precondition, flow of events (basic and alternatives), and a post condition [26] at the use case level. The problem with this simplistic format is that dependencies amongst use case events are assumed to be linear (or not thought of at all).

Another area that has seen successful application of state-based information is business process modelling [49]. Whereas many process modelling approaches have a formal-methods orientation, the same approach cannot be used with use cases given that the uptake of use cases is due to their ease of use and understanding [128]. This thesis, therefore, posits that use cases can benefit from state machine representations of the software behaviour that they describe in order to rigorously validate stakeholder expectations. Furthermore, just as in much of model checking and business process modelling, appropriate (and simple to use) tool support is needed to enforce the incorporation of this type of information in use cases. Indeed, the role of tool support within software engineering, especially, at the requirements phase, is to enhance communication between stakeholders and development engineers [111].

The EducatorTool offers a mechanism by which modellers and stakeholders can enhance their shared understanding by stepping through visual models of the use case descriptions.

4.2 The EducatorTool

It has been stated elsewhere (section 4.2) that a use case description is a partial story describing a circumstance of system usage and how the system behaves while serving its external users (actors). A bespoke support tool, must meet at least, the function of writing these partial stories. The basic elements of a use case description are actors and events ([33], [37]). Additional elements suggested in the Educator approach are pre and post states for events, secondary actors, and pre and post states for those secondary actors. Hence, a most basic requirement of the EducatorTool is to support the construction of Educator-based descriptions as specified on the Educator model (chapter 3, section 3.4.3).

An additional requirement for the tool is the need to support behaviour prototyping based on the edited descriptions. That is, the EducatorTool provides functionality to enact (animate) use case descriptions. Enaction is a way of providing visual models whereby description authors can view the implications of their statements as actors interact in performing the events. Stakeholders can

therefore develop visual models of their understanding of the system to other stakeholders during validation of the specification.

Since use cases might have alternative courses, the tool provides functionality for including alternative paths. Alternative paths are not independent descriptions, as their execution depends on whether the execution of the base use case has followed the alternative path. Additionally, the tool provides ways to inter-relate distinct use case descriptions. The purpose of this is to synchronise distinct behaviour descriptions and enact them to visualise pre-defined interactions amongst different system parts.

4.3 Supporting CP rules

4.3.1 Overview of CP rules

The CP rules are covered in [37] and their essence reported in ([129], [114]) and were largely informed by those of [113]. The CP rules are a set of syntactic rules to be followed when authoring use case descriptions. The aim of the rules is to enable the production of commonly comprehensible use case descriptions given that the UML does not provide any guidelines regarding how use cases may be written. There are thirteen CP rules (see [37]) briefly described as follows:

1. Each sentence (or event) of the use case should be in a new line – this rule is aimed at fostering a consistent writing structure where the detail of the use case is clearly stated.
2. Alternative paths should be in a separate section (from the basic path) and the sentence (or event) number should agree – this rule aims to ensure alternative path details are not mixed up with those of the basic path, nor are they lost such that one cannot tell which alternative path is associated which event.
3. That present tense should be used in naming actions of actors.
4. Pronouns are to be avoided.
5. Adverbs are to be avoided.
6. Adjectives are to be avoided.
7. Negatives are only to be used in exceptional flows (error handling).
8. Explanation to be provided where it is needed to provide further detail.
9. Use case descriptions should be coherent.
10. A modeller should always employ the adjacency pair (Question -> Reply to Question).
11. Within a description, a modeller should show related use cases by underlining their names.
12. Only atomic events should be allowed: *subject verb object*

13. Also, atomic events of this form may be allowed: *subject verb object prepositional phrase*.

This thesis is not about use case authoring rules or guidelines. It is about dependency analysis in use case specifications. However, as articulated in [37] comprehensible descriptions are vital to validation. This thesis adheres to some of the rules (e.g., 1, and 12). Moreover, a few of the CP rules have been selected to further demonstrate that automated support can be provided for such rules.

The support of some of the CP rules by the EducatorTool is an additional benefit of automated support, and not a central issue of this thesis. As noted already, Educator descriptions are written such that each event is on a new line, thereby adhering to the first CP rule. This requirement is supported. The CP rule that demands that use case description authors to avoid pronouns (e.g., he, she, and it) is one of the other CP rules supported by the EducatorTool. This rule is enforced by allowing the tool user to construct a working dictionary that contains the words that are not allowed for use in the description:

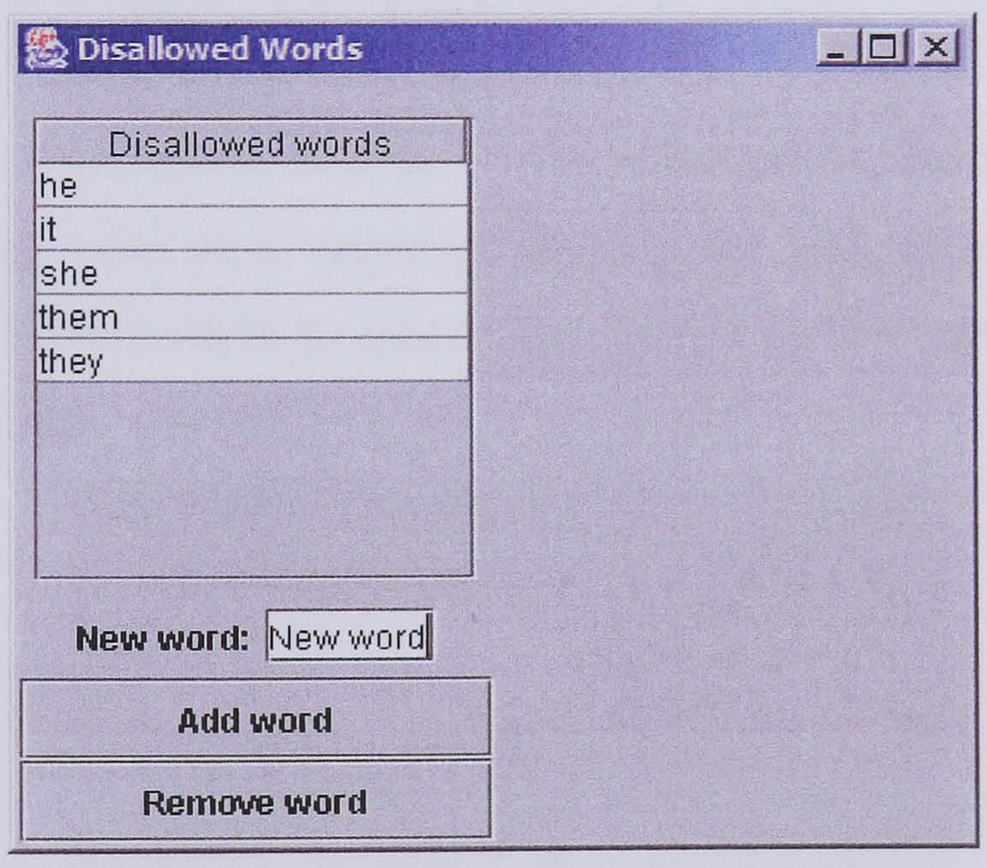


Figure 4.1: Disallowed words

If the user writes an event that has any of the disallowed words, then the application notifies the user of that word and provides an option to re-write the event using another word to be chosen by the user. For instance, if the user wanted to replace reference to "impulse timer" (in the description of Figure 4.15, pgs. 66-67) with the pronoun "it", the user will be prompted:

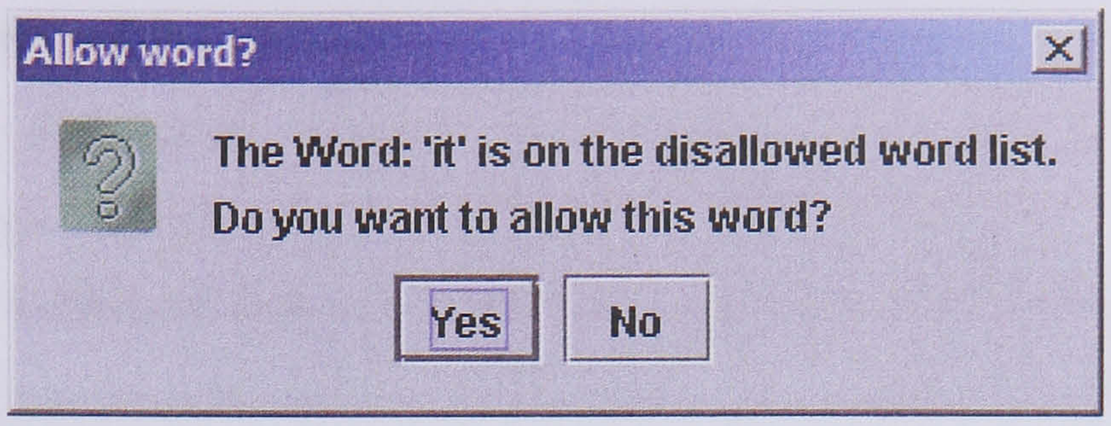


Figure 4.2: A word that may not be suitable

CP rule 6 requires authors to write their events in present tense. EducatorTool provides the functionality to check that users do not include words in the past tense. For instance, if the user writes the first event (of the Pacemaker-Table 3.3) as "Physician programmed the pacemaker's impulse timer", EducatorTool reports on the past tense usage:

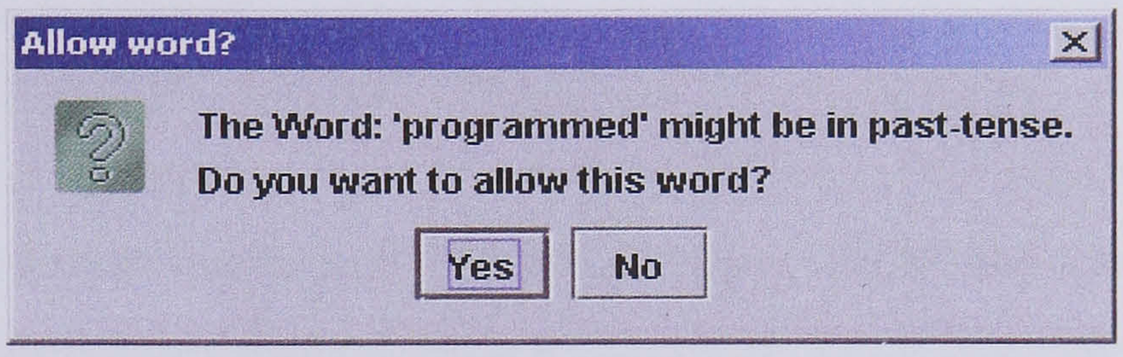


Figure 4.3: Checking use of past tense

This is enforced by use of an inbuilt checker of words that are in past tense. Since some words could appear to be in past tense (e.g., names of people), the tool allows the user to construct a working dictionary of allowed words to ensure such words do not appear to flout this CP rule when used in the description:

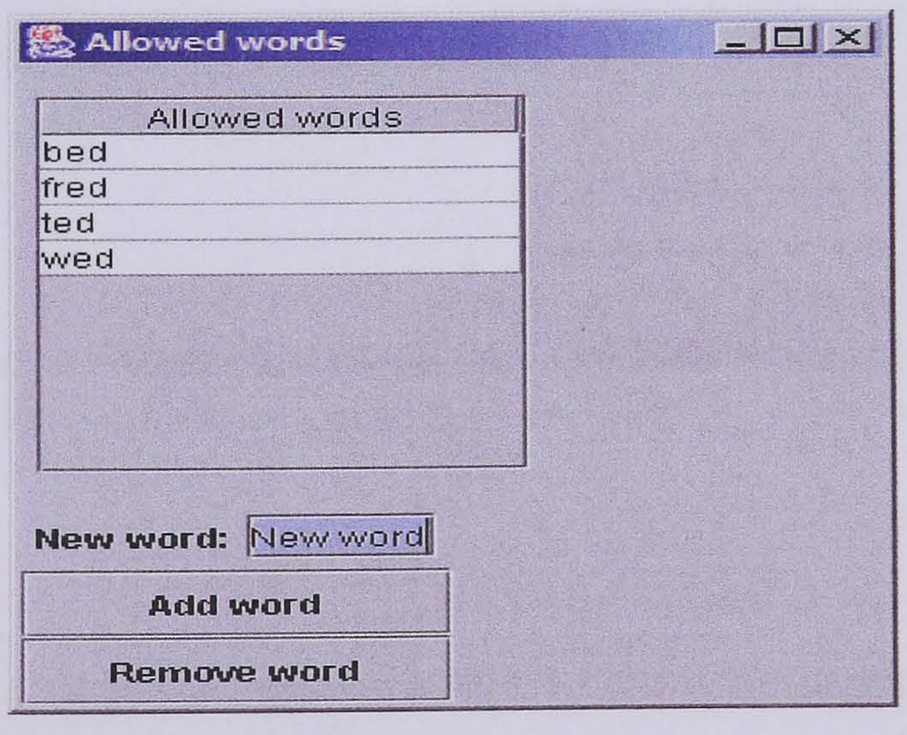


Figure 4.4: An example list of allowed words

The above CP rules are readily supported by the EducatorTool, and it is possible that the other CP rules can also be supported in similar specification environments like the EducatorTool.

4.4 Enactable functionality- examples

Enaction of Educator descriptions is one of the main functions of the EducatorTool. Enaction entails stepping through the logic of a description (or a set of descriptions).

Where the visualised behaviour does not match the expectations of the modeller, then modellers can revise the description (by adding extra events, actors or changing states).

Sections 4.7.1 through to 4.7.5 discuss the different enactable capabilities supported by the EducatorTool.

4.4.1 Default enaction

Default enaction is the simplest form of enaction offered by the tool. In this scheme, the description is enacted in a sequential order based on event positions, that is, from the first event, second event, etc. It is possible to invoke default enaction for an already state-based description. The assumption is that by stepping through a visual animation of the description, modellers may identify event orders that are erroneous. This thesis argues that the default enaction is simplistic and only useful as a first step towards a state-based model. As an example, consider the Cardiac pacemaker description of Figure 3.6.

Default enaction of use case descriptions entails stepping through the logic of the description without taking into account the pre and post conditions for constituent events. Hence, once the initial event has occurred, the next available event is the event in position 2:

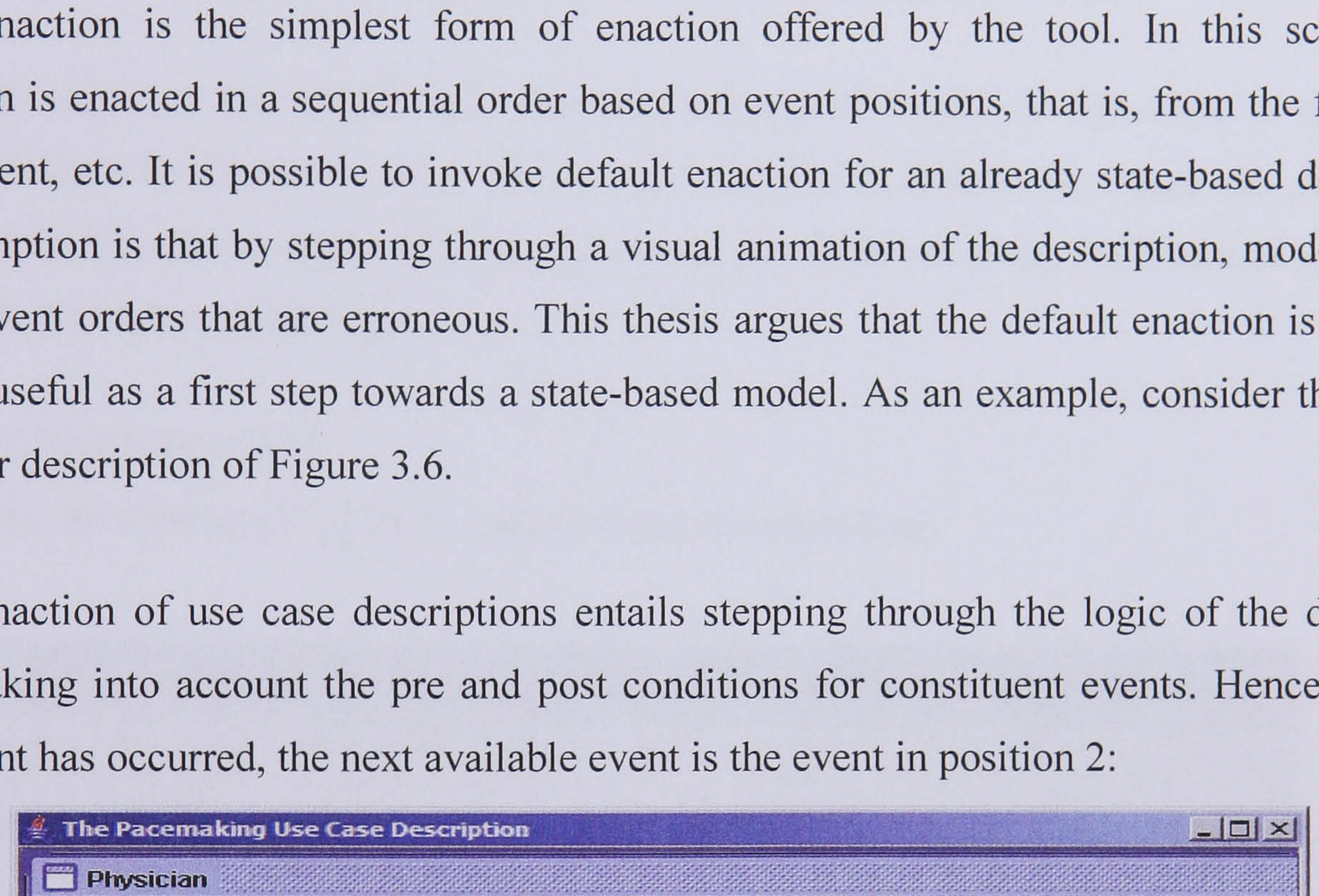


Figure 4.5: Default enaction (second event)

Overall, the Pacemaker process shown in Figure 3.6 would be visualised such that the constituent events are executed in sequence. The output of default enaction of the description is shown in Figure 4.6:

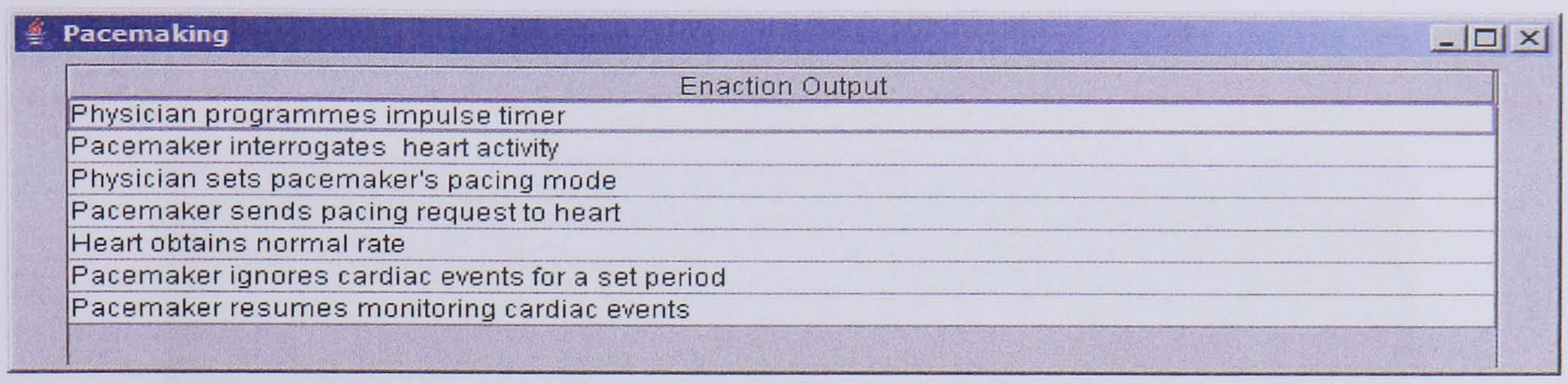
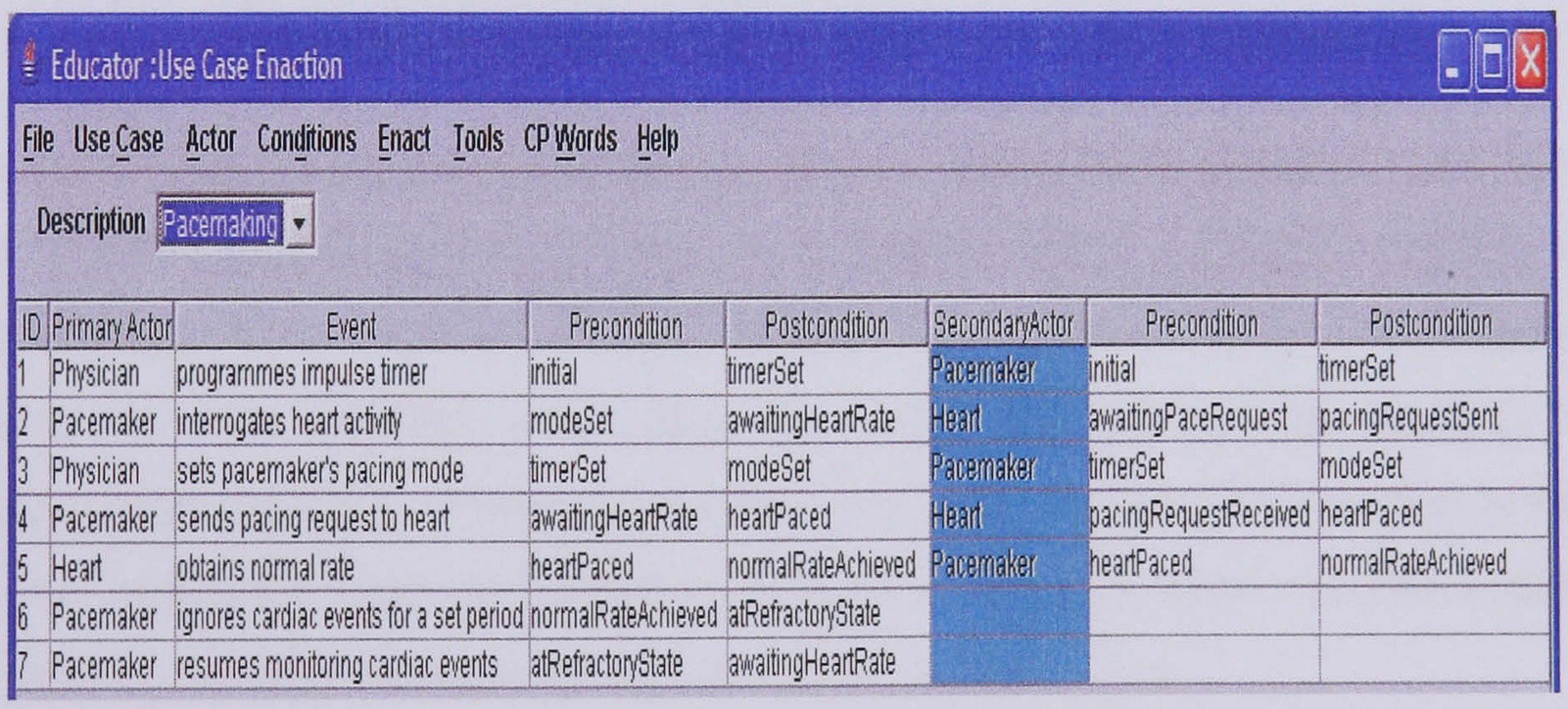


Figure 4.6: output of default enaction for the pacemaker

A problem with this assumption of linear dependency of events is that there is no rigorous scrutiny of the circumstances under which event occurs in relation to others (as discussed in section 3.4.3). This thesis argues that, for non-trivial specification tasks, the Educator approach be adopted for dependency analysis.

4.4.2 State based enaction

Consider the description of Table 3.3 edited in the EducatorTool:



ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	Physician	programmes impulse timer	initial	timerSet	Pacemaker	initial	timerSet
2	Pacemaker	interrogates heart activity	modeSet	awaitingHeartRate	Heart	awaitingPaceRequest	pacingRequestSent
3	Physician	sets pacemaker's pacing mode	timerSet	modeSet	Pacemaker	timerSet	modeSet
4	Pacemaker	sends pacing request to heart	awaitingHeartRate	heartPaced	Heart	pacingRequestReceived	heartPaced
5	Heart	obtains normal rate	heartPaced	normalRateAchieved	Pacemaker	heartPaced	normalRateAchieved
6	Pacemaker	ignores cardiac events for a set period	normalRateAchieved	atRefractoryState			
7	Pacemaker	resumes monitoring cardiac events	atRefractoryState	awaitingHeartRate			

Figure 4.7: State-based Pacemaker process in EducatorTool

The essence of enacting a state-based use case description is to provide a means for further scrutiny of dependency issues by visualising actors' state changes as they perform events. Whereas enaction or some form of prototyping is considered important (see [80], [130]) for enhancing shared understanding, most industrial strength CASE tools such as RationalRose do not provide specific functionality for authoring use case descriptions let alone their animation. Bespoke tools such as that reported in [31] do not consider enaction. A major theme of the work

presented in this thesis is to provide a prototyping mechanism where use case descriptions can be animated to visualise the implied behaviour. Hence, the perceived benefit of enaction is the visual rendering of the description thereby depicting what the stakeholders would obtain from the delivered product. For instance, the Physician and the Pacemaker interact during the performance of event 1 (see Figure 4.8). Prior to event 1 occurring, Figure 4.8 shows the states of both actors:

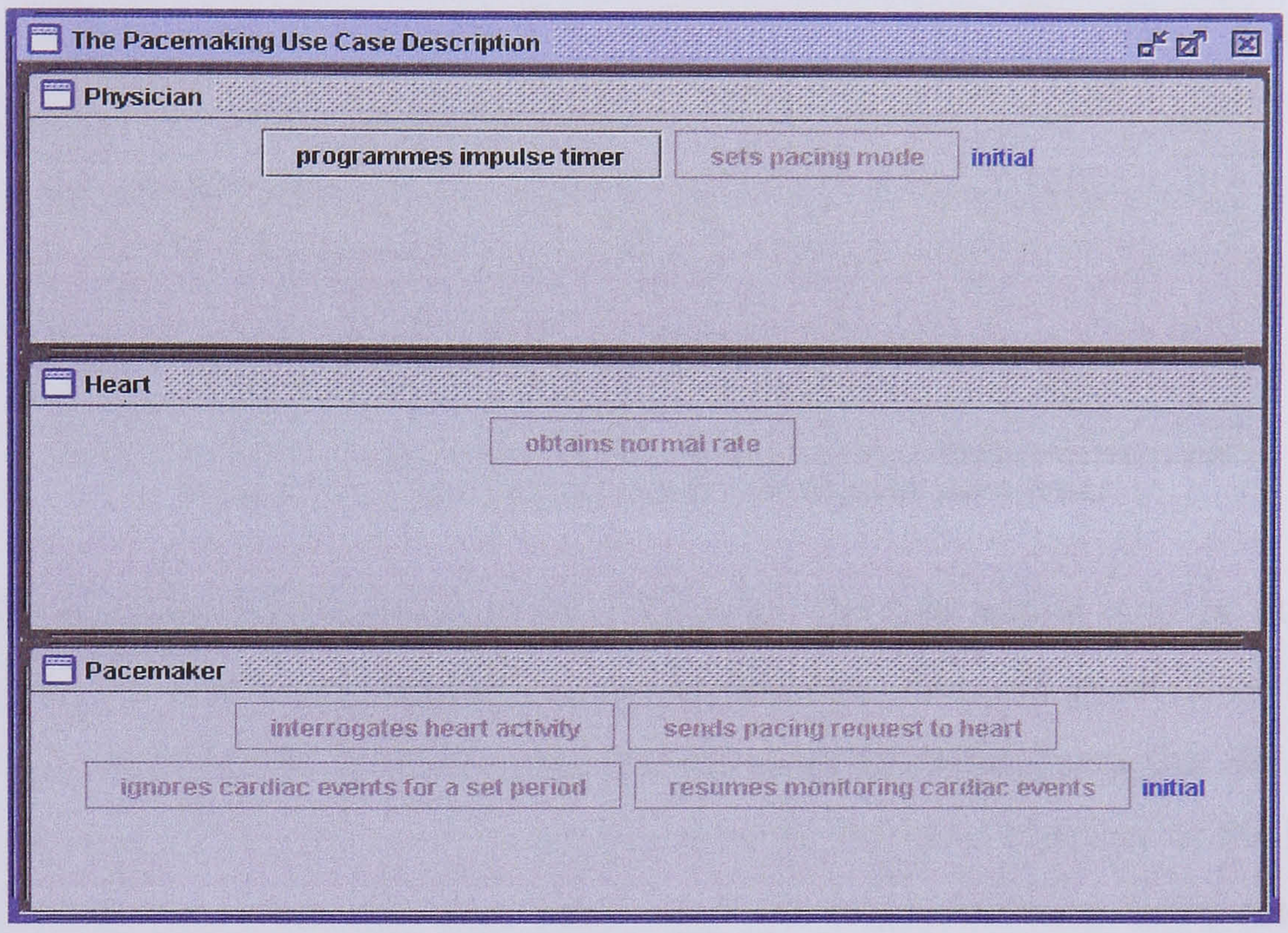


Figure 4.8: Physician and Pacemaker's states (prior to event 1)

By stepping through visual models of the description, stakeholders can revise any behaviour they deem undesirable by changing the primary or secondary actors, or even by adding or changing events and states. If the enaction of events followed the order of events in the stateless use case description, the next available event after the dialog above would be event 2, "Pacemaker interrogates heart activity". However, the inclusion of state based information reveals that the state of the Pacemaker after the execution of event 1 does not allow the Pacemaker to trigger event 2. Indeed, the next enaction dialog is as shown in Figure 4.9:

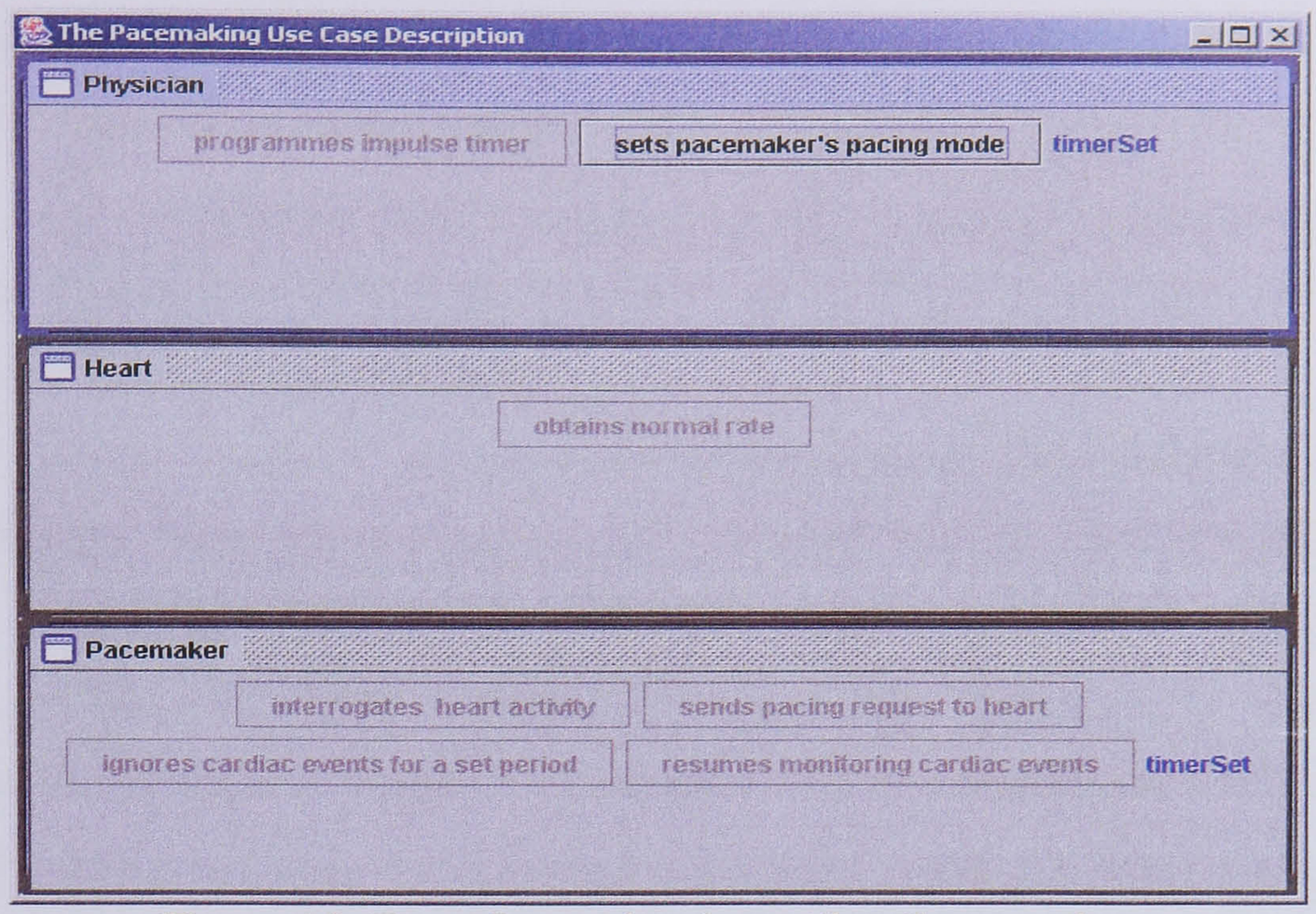


Figure 4.9: Event 2 must involve setting the pace timer

The literature on artificial Pacemakers is clear about the fact that pacing must be done based on the chamber of the heart that requires a pacing impulse. Thus, if event 2 of the stateless description is executed before the timer is set, and the mode determined (whether the left chamber or the right chamber of the heart is to be paced), then the incorrect chamber of the heart will be paced, hence inducing an exaggerated heart rate, the very condition being monitored for mitigation. Finally, the correct sequence of events after considering dependencies is shown below:

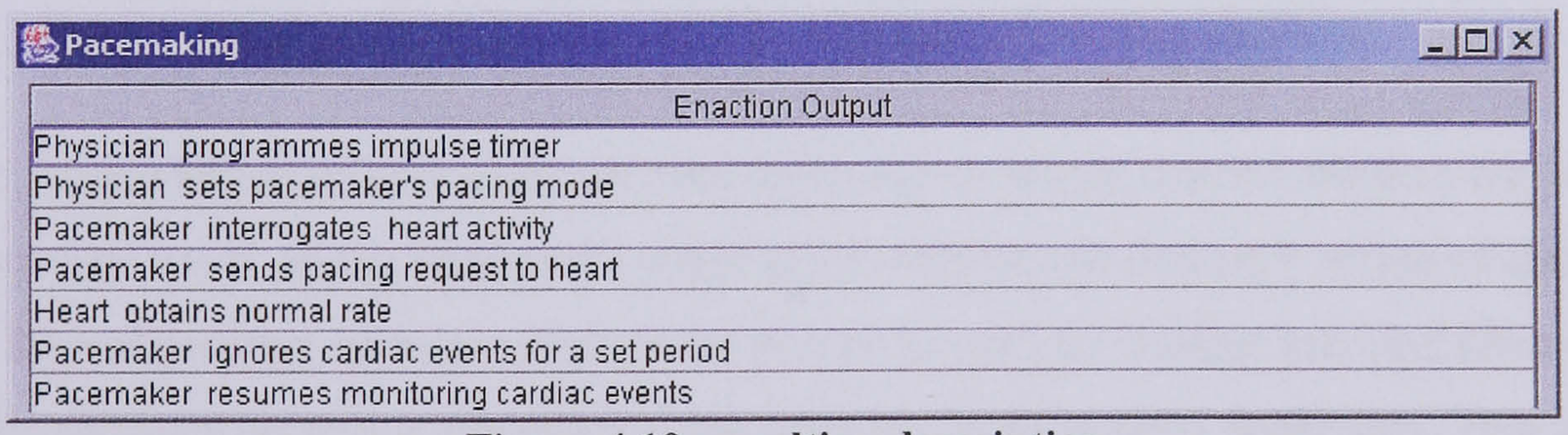


Figure 4.10: resulting description

Moreover, the description of the pacemaker will be able to mimic the continuous working of a pacemaker, whereby, after the final event, the pacemaker will resume monitoring again to ensure the heart is continually monitored and paced. The stateless description does not provide such a prototype of the exact behaviour expected of the working of the pacemaker, and it is difficult to determine, for example, the event at which pacing must resume.

4.4.3 Multiple use case enaction

The reason for considering enaction across multiple use case descriptions is to be able to determine inter-use case dependencies, and hence validate behaviour for sub-systems that interact in their course of execution. The UML specification of the use case does not consider such interaction issues across distinct use cases.

Consider the Exchange pen (UC1) and the course registration (UC2) use cases shown on pages 37 and 38 respectively. There are two ways in which distinct use cases can be associated with the EducatorTool. One way is to consider an event whose occurrence is dependent on local dependencies and those of events in other descriptions. For instance, in the discussion of the Educator approach (chapter 3, Figure 3.4) indicates that the Student (in UC2) cannot choose course to study based on events in UC2 alone. Choosing a course to study also depends on an event in UC1.

In a situation like this, the EducatorTool provides a modeller with the set of events from the Pen exchange use case. The modeller would then choose the specific event (Lecturer gives pen) of the Pen exchange use case that would be associated with the desired event (Student chooses courses to study) in the course registration use case:

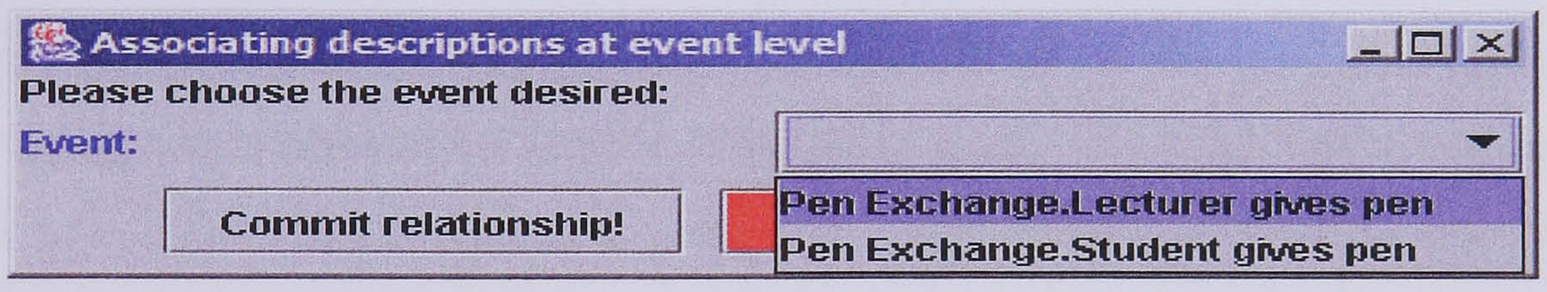


Figure 4.11: Event level synchronisation within EducatorTool

Again, an establishment of this inter-use case dependency means that the Student cannot choose courses to study based on the Registrar's event alone. Rather, the student's action of choosing the course is also dependent upon an event (Lecturer gives pen) in another use case (Pen exchange event). The meaning of such a dependency assertion has to be context dependent. That is, what is the essence of a student not being able to choose courses until the Lecturer has given pen? Perhaps, the student needs the pen during the choosing of the courses. Whereas this example is simple in nature and easy to reason about, real life applications and processes can be complex. Again, the synchronisation of processes has been a topic in computer science for many years. Issues such as resource sharing, multithreaded applications and the management of resource sharing amongst threads and processes can be seen as an area where this type of specification might be relevant. When the use cases are inter-related in the manner described here, there is choice between two events for the Student in the two descriptions. That is, the Student can

perform the event 3 in Course registration or event 2 ("gives pen") in the Pen Exchange use case. Figure 4.12 shows the interaction between the two descriptions within EducatorTool:

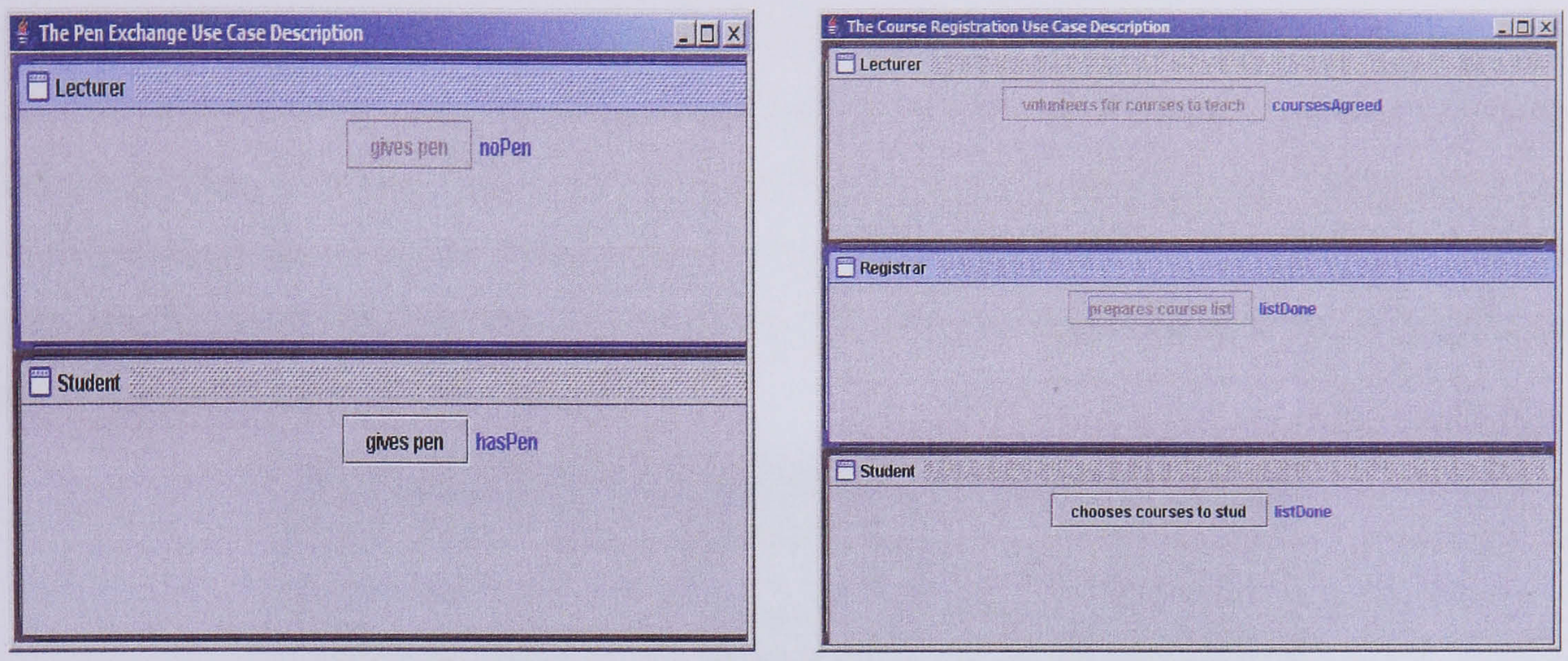


Figure 4.12: Interaction at event level between 2 use cases

The other form of multiple use case enaction is by considering a certain event in one use case where the behaviour of the other use case is to be invoked. That is, an interaction among actors in one use case leads to the invocation of a related use case. Consider the Exchange pen and Course registration use cases again. It may be that after Lecturer gives pen (in the Exchange pen use case), and the student is then in the *hasPen* state, the entire Course registration use case needs to be invoked before the Student can give pen (event 2 of Exchange pen use case) back to the lecturer.

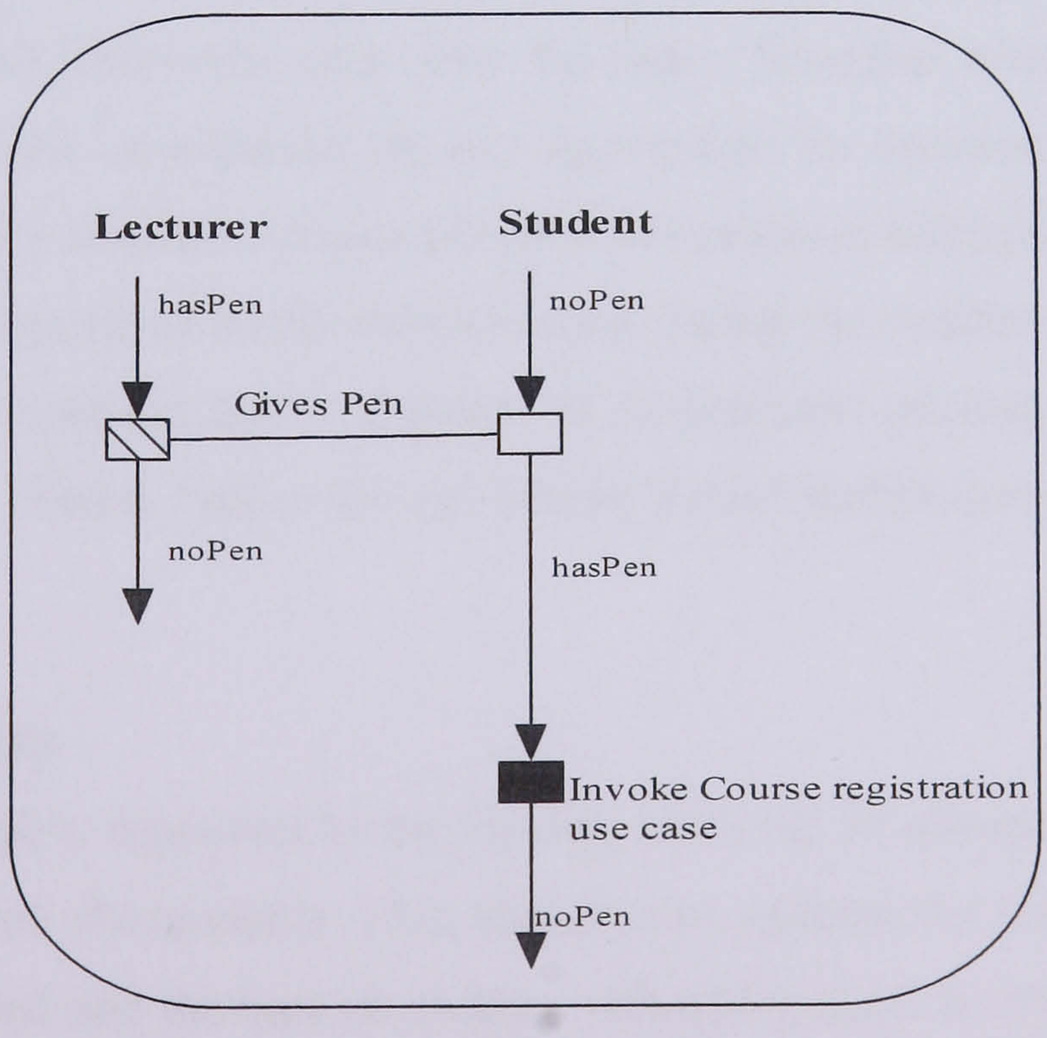


Figure 4.13: RAD showing invocation of a use case from another

Again, Figure 4.13, and the indication of a whole use case being invoked upon the execution of an event in another use case is an issue that can bear contextual interpretations depending on the processes described. Whereas the example used here is easy to outline and reason about, some real life processes require computed resources from other processes before they can execute their part of the system-wide goal. EducatorTool provides support for invoking use cases during the enaction of other use cases. Once a use case has been invoked, it overrides the invoking use case until it (invoked use case) has completed enacting. To invoke the Course registration use case within the Pen exchange use case, a user selects the event at which the invocation occurs, and then further selects the use case to call:

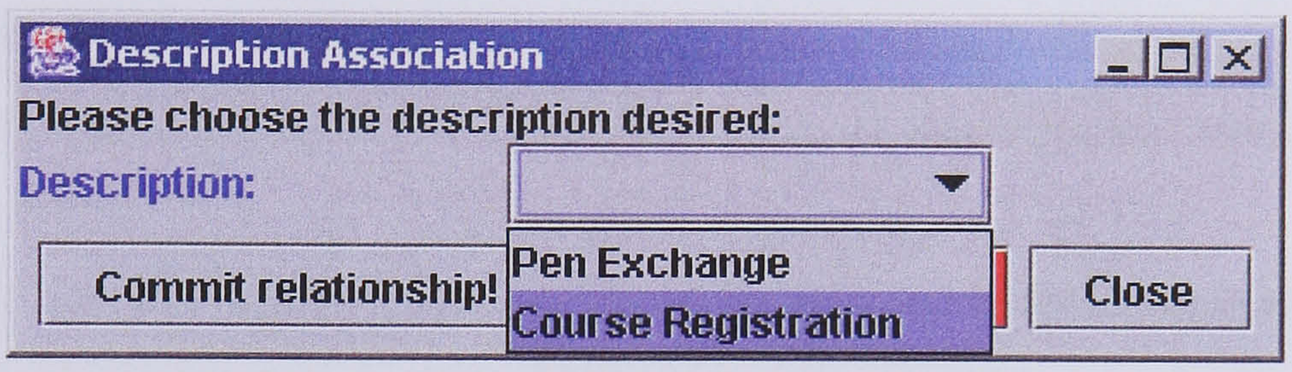


Figure 4.14: invoking Course registration at event 1 of Pen exchange

4.4.4 Modal (Combinatorial) enaction

Modal enaction is a scheme where different combinations of use case descriptions are enacted in either sequential, selection (choice), iterative, or concurrent order. That is two or more descriptions, regardless of their internal states can be enacted in sequential order, one description following the other. This forms the sequential mode of enaction. In iterative mode, two or more descriptions are enacted iteratively, each after the other. Selection involves making of choice between a set of available descriptions the one appropriate for enaction to realise the context-dependent behaviour of a situation. Concurrent enaction entails enacting two or more descriptions at the same time. The concept of modal enaction arose during the conduct of the industrial project. It became essential to consider modal enaction of descriptions because of the nature of some subsystems' functions. Hence, section 6.6 (of chapter 6) has further discussion (and illustration) of modal enaction.

4.6 Chapter Summary

The enactable functionality supported by the EducatorTool can be classed as either state-based or default (where states are disregarded). This chapter has outlined the various flavours of state-based enaction supported and the type of scrutiny offered by each. In stateless enaction the tool allows description authors to produce prototypes of a description following the default order of events. States play no role in this form of enaction. However, authors can change the ordering by

simply inserting events in different positions. The limitation of default ordering is that it disregards the notion of actors in a description "knowing" the context of actions of other actors. Rarely in any specification (whether of software or business process) are actions of one actor or role only important to that actor's or role's interests alone [14]. Thus, in order to enforce this form of interaction between actors this chapter argues for the inclusion of accessible states that determine which actor is responsible for initiating which action, and which actor is affected by the execution of the action. Thus, the state-based model makes states a crucial property of actors' events, which determine whether the actor may invoke (or participate in) an event.

There are two reasons for considering dependencies in use case specifications:

1) To determine a correct sequence of events given a set of requirements that a software system must meet.

2) To enhance the level of detail in use case descriptions since in their standard form, UML use cases offer little useful information to elaborate software behaviour.

The author does not impose a prescriptive adoption of the approach and tool presented in this thesis to systems development engineers. Rather, the thesis argues that where software engineers deploy use case descriptions in their specification tasks, then the approach and tool presented here are a useful means for validating requirements and specifications.

5. Evolution of EducatorTool

This chapter discusses feedback from various sources that informed the development of the Educator approach and tool. For instance, workshops with MSc Computing (Software Engineering) students provided an opportunity where the students used the tool for the specification of their software projects. The feedback obtained from these informal workshops is discussed in section 5.2. Further feedback was obtained from more formal workshops whereby presentations were made by this author to colleagues in the research group. Section 5.3 offers a discussion of the feedback arising from these presentations.

5.1 Rationale and activities undertaken

It has been pointed out in previous chapters that the Educator approach is motivated by the process modelling approach – RolEnact. Much of Educator's support environment was developed iteratively with use of example use cases from literature. The rationale for participating in workshops with students was to obtain feedback regarding the approach as presented within the tool. Preceding chapters have indicated that the essence of the Educator approach is to facilitate the consideration of dependency issues in use case specifications. The EducatorTool is a proof of concept tool that acts as vehicle for the approach. Hence, one of the things that the informal workshops sought to elicit from the MSc students was the extent to which they found the application of the Educator approach and the use of the support tool useful (or not useful) in "teasing out" dependency issues in the specifications for their software projects. The EducatorTool provides an enactable environment where Educator-based use cases can be visualised. Hence, the other issue that the author wanted to find out from the informal workshops was the extent to which the enactable functionality helped in validating the participant's specifications. A related matter is, of course, to obtain feedback regarding any enhancements that could be made to EducatorTool's functionality for authoring Educator-based use cases, their amendment, and enaction.

Additional workshops involving presentations to colleagues in the research group were conducted. These aimed at obtaining general critique of the approach and the tool, based on example demonstrations during these presentations.

5.2 Feedback from student workshops

5.2.1 Feedback gathering

The process of obtaining feedback from both workshops was informal in that no experimental design was involved to control participants and the treatments. Most of the feedback elicited was qualitative. For example, feedback on the use of the approach to validate specifications was obtained by discussing dependency issues discovered as result of applying the approach. An example from each workshop is provided in section 5.2.2.

Feedback gathering from the first workshop also involved the use of a questionnaire to ensure systematic responses by all participants at the early stage of the approach and tool development. Questionnaire design requires the consideration of the objectives for the study ([131], [132]) in order to be able to formulate the questions that will help address the issues being evaluated. Most questions were closed format with responses to choose from; the last question was open format where respondents could write comments regarding issues, that may not have been covered in the closed questions, that they wish to raise.

There were two main objectives for the workshops. First is to obtain feedback regarding the extent to which participants found enaction suited to dependency analysis. The reason for considering enaction is because it is the most explicit way in which the EducatorTool supports the Educator approach for dependency analysis. Second, the workshops had an objective of obtaining feedback regarding the use of the tool in constructing use case descriptions and revising them. Educator use cases can be constructed with the EducatorTool, and it is this construction of Educator-based use cases that the second objective was concerned with.

5.2.2 Educator approach

This section considers feedback, and examples from students and how such examples informed the Educator approach. Hence, some example use cases from students are presented. Their analysis is discussed with respect to how students found the Educator approach and tool useful (or not useful) in validating the use cases.

Example use cases & analysis

Example 1 is from the first workshop and example 2 is from the second workshop. Further example descriptions are provided in appendix F.3 and analysis of some of them is found in appendix K. The discussion of the examples is organised as follows. First, the augmented use

cases are presented, and any dependency issues found by students discussed. Second, presentation is made of the application of enaction (with EducatorTool), and any dependency issues found also discussed.

Example 1

One of the students was working on a project based on the UK's care home sector. The student had carried out background research regarding trends in the care sector. A disturbing finding was that whereas many care homes were vulnerable to closure (for various reasons), the number of people requiring care was increasing.

The student observed that since these care facilities are few, it is important that some of their activities are supported with a software system to increase the efficiency of the care staff. Some of the issues investigated and that needed addressing somehow were:

- Staff at many care homes often face difficulties obtaining leave (even sick leave) due to the many tasks that have to be attended to in person at the care home (e.g., administrative tasks like booking new clients).
- Managing staff issues (e.g., rota, wages, arranging and communicating departmental meetings, etc).

The student was specifying a care system where a family could make direct enquiries online without having to travel to the care home physically. The student started with an initial use case description depicting the sequence of events that a family would go through to make an enquiry (and hopefully, registration at a care home):

1. Family requests information on care home
2. Family sends information on elderly kin
3. Staff accesses relevant care information
4. Staff selects data to be included in report
5. Staff prints the report
6. Staff sends the report to the family

Figure 5.1: Care request use case description

The Educator approach proposes the augmentation of use case descriptions with state-based information. One of the reasons for developing the EducatorTool is to provide automated support for augmenting use case descriptions with the state-based information. Moreover, the Educator approach proposes the delineation of interaction issues by indicating any secondary actor for an

event and the pre and post state for the secondary actor involved in the event. The student considered the pre and post states for the description in Figure 5.1, and revised it to produce the state-based description shown in Table 5.1:

ID	Primary Actor	Event	Precondition	Postcondition	Secondary Actor	Precondition	Postcondition
1	Family	Requests information on care home	initial	careInfoRequested	***Staff***	initial	requestReceived
2	Staff	Requests family's kin details	requestReceived	kinInfoRequested	***Family***	careInfoRequested	kinRequestReceived
3	Family	Sends information on elderly kin	kinRequestReceived	kinRequestSent	***Staff***	kinInfoRequested	infoReceived
4	Staff	Accesses care information	infoReceived	careInfoAccessed			
5	Staff	Selects relevant care information	careInfoAccessed	careInfoSelected			
6	Staff	Prints the information	careInfoSelected	infoPrinted			
7	Staff	Sends the report to the family	infoPrinted	infoSent			

Table 5.1: Care request use case (revised in Educator approach)

Effects of augmentation

As a result of considering pre and post states for each event, the student brought to light some dependency issues. For instance, in Figure 5.1, it was not clear where the family requests the care home information (event 1) from. Hence, the inclusion of the staff as a secondary actor for that event clarifies the fact that the family interacts with a staff member from the care home. Furthermore, the state-based information enabled the student to realise that after event 1, the next available event could not be event 2 of Figure 5.1. The reason for this is that the care staff needed to know certain information regarding the person who needs care. This was considered useful for staff in determining whether they (staff) would be able to provide the care service and hence send the appropriate information to the family. An additional event was introduced:

Staff Requests family's kin details.

One of the issues that was associated with missing out the above additional event (and hence not carefully considering the family's kin circumstances), was that staff could take on people who need more specialised care (e.g., due to any existing medical condition or advanced age) than may be offered at the care home. Hence, this seemingly slight omission could mean admitting people for whom there were no adequate care facilities at the care home.

The set of events in the initial use case description (Figure 5.1) had the ordering changed by one position each starting at the second event. The importance of this has been discussed in the previous section. To reiterate, the introduced event was necessary to indicate what is the next the appropriate action for a member of staff who receives a request from a family regarding care service and facilities at the care home. The staff should know specific information about the family's kin to be able to determine the adequacy of the care services they offer with regard to the enquiring family.

Example 2

This example is from a workshop conducted with a different group of MSc students, mainly to gain further feedback on the extent to which the approach is supported by the EducatorTool.

A process considered by one of the students during this workshop is that of the monitoring of Lorries entering and leaving a depot. The student was attempting to describe the behaviour of the part of a system that could aid the depot operator in assessing and recording Lorries entering the depot to deliver goods, and those leaving the depot after delivery (or picking of) goods. Consider the use case description for recording Lorries exiting the depot:

1. Lorry arrives at depot exit
2. Operator drives to the depot exit office
3. Operator sets movement type (exit)
4. Operator checks entry load value for the Lorry
5. Operator requests lorry registration
6. Lorry driver provides lorry registration information
7. Operator weighs the lorry
8. Lorry exits the depot

Figure 5.2: Depot exit use case description

The state-based version of the use case in Figure 5.12 is shown in the following table:

ID	Primary Actor	Event	Precondition	Postcondition	Secondary Actor	Precondition	Postcondition
1	Lorry	arrives at depot exit	inSite	atExit	Operator	initial	driving
2	Operator	drives to depot exit office	driving	atExitOffice			
3	Operator	logs into the vehicle monitoring system	regReceived	opLogon			
4	Operator	sets	opLogon	movementSet			

		movement type (exit)					
5	Operator	checks entry load value for the lorry	movementSet	loadChecked			
6	Operator	requests lorry registration information	atExitOffice	regRequested	Lorry Driver	initial	regRequested
7	Lorry Driver	provides registration information	regRequested	regProvided	Operator	regRequested	regReceived
8	Operator	weighs the lorry	loadChecked	weighed			
9	Lorry	exits depot	infoSaved	exit	Operator	infoSaved	logsOff
10	Operator	saves lorry exit information	weighed	infoSaved			
11	Operator	shuts the vehicle monitoring system	logsOff	systemShut			

Table 5.2: state based version of Figure 5.12

Revisions to the initial description:

- Three added event:

  Operator logs into the vehicle monitoring system.

  Operator saves lorry exit information

  Operator shuts the vehicle monitoring system

  There were no added actors.

The additional events were considered important. For instance, the "Operator logs into the vehicle monitoring system" was a crucial event for ensuring entry information is checked on the existing records. The "Operator saves lorry exit information" is important as the determined exit load has to be recorded and saved too. The "Operator shuts the system" is important to ensure no active logons are left as that would expose the system to misuse whereby drivers can record falsified information.

5.2.3 EducatorTool

Considering Example 1

This section considers the application of the EducatorTool on the student examples. The main feature of the EducatorTool that is central to the Educator approach is the enaction of descriptions. Hence, this section provides a discussion of enaction and a discussion of whether or not students found enaction useful (or not useful) in further validation of specifications.

The EducatorTool was used to produce enactable models of the state-based use case of Table 5.1.

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Care Information Request

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	Family	Requests information on care home	initial	careInfoRequested	Staff	initial	requestReceived
2	Staff	Requests family's kin details	requestReceived	kinInfoRequested	Family	careInfoRequested	kinRequestReceived
3	Family	Sends information on elderly kin	kinRequestReceived	kinRequestSent	Staff	kinInfoRequested	infoReceived
4	Staff	Accesses care information	infoReceived	careInfoAccessed			
5	Staff	Selects relevant care information	careInfoAccessed	careInfoSelected			
6	Staff	Prints the information	careInfoSelected	infoPrinted			
7	Staff	Sends the report to the family	infoPrinted	infoSent			

Figure 5.3: use case of Table 5.1 within EducatorTool

The enaction dialogue prior to the execution of event 1 is shown in Figure 5.4:

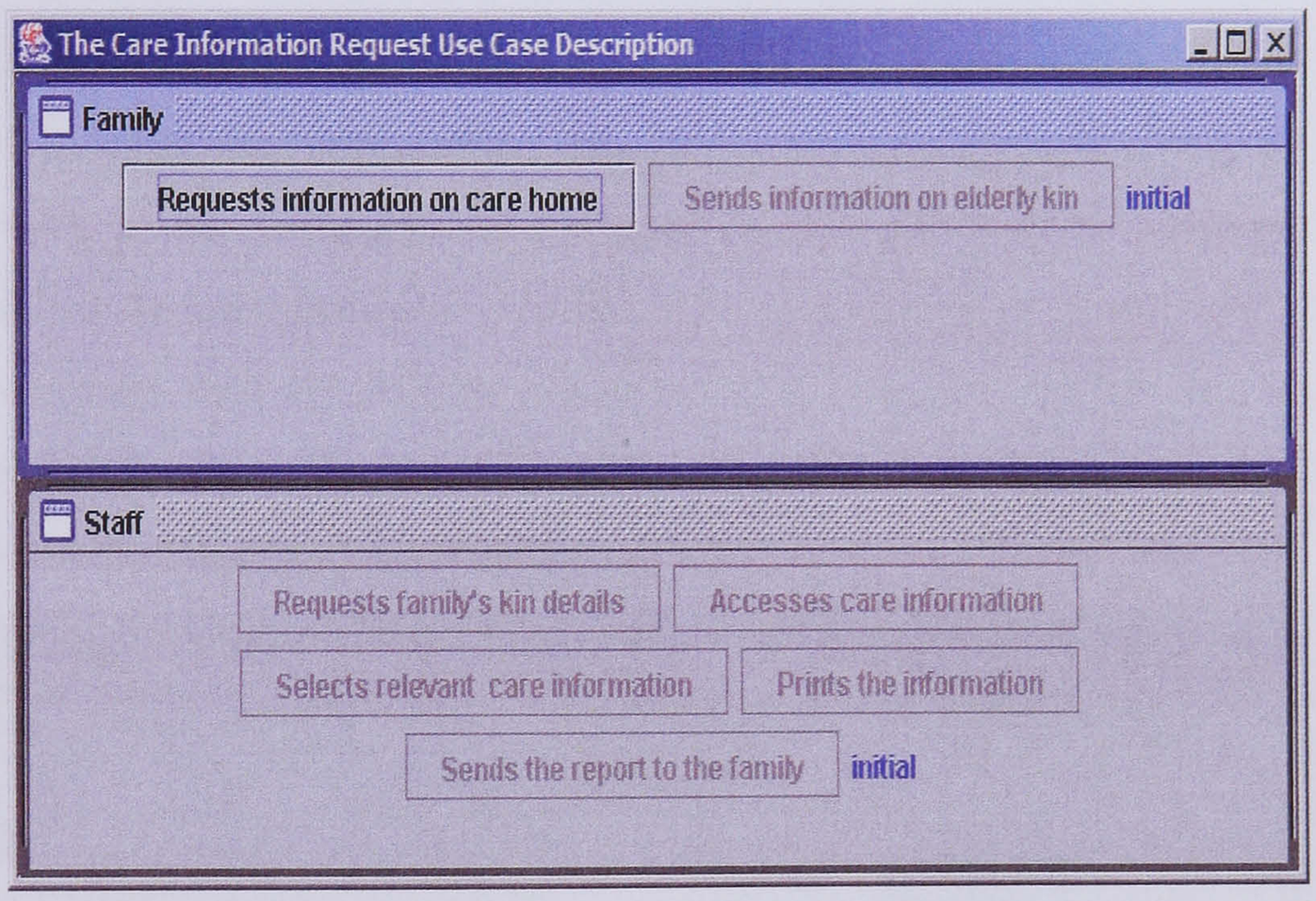


Figure 5.4: First enaction window

When the family requests information on the care home, the following dialogue is reached:

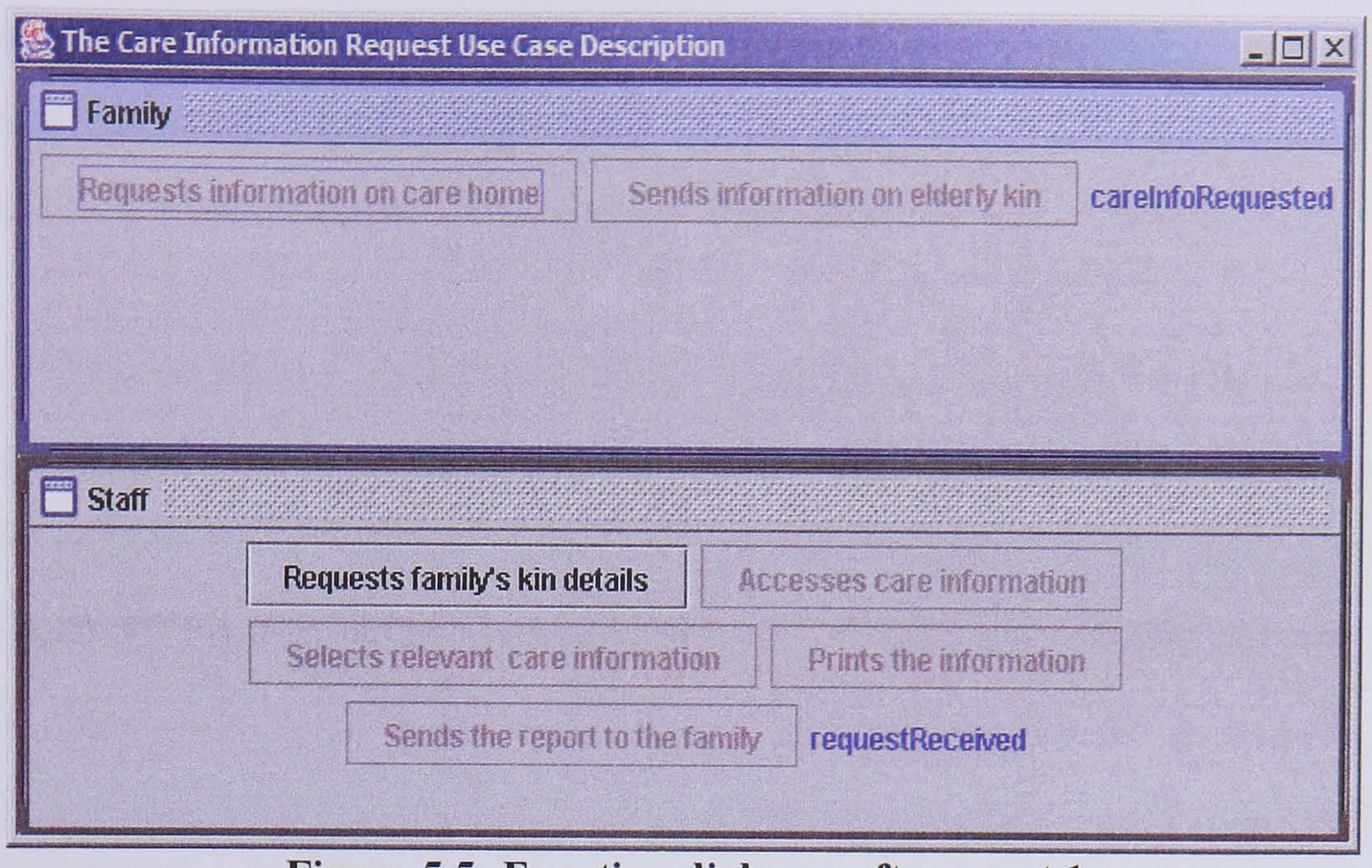


Figure 5.5: Enaction dialogue after event 1

Upon the enaction of event 5, the student argued that a manager of the care home had to be involved when relevant care information has been identified for a family's kin. The selected information (event 5) has to be sent to the manager who then opens a temporary file for the family's kin before the information is sent to the family. This is to ensure that when the family contacts the care home with any further queries (or with an indication that they want the service), the manager is able to obtain the information for easy reference and decision making. Hence the following is the revised use case description:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Care Information Request

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	Family	Requests information on care home	initial	careInfoRequested	Staff	initial	requestReceived
2	Staff	Requests family's kin details	requestReceived	kinInfoRequested	Family	careInfoRequested	kinRequestReceived
3	Family	Sends information on elderly kin	kinRequestReceived	kinRequestSent	Staff	kinInfoRequested	infoReceived
4	Staff	Accesses care information	infoReceived	careInfoAccessed			
5	Staff	Selects relevant care information	careInfoAccessed	careInfoSelected			
6	Staff	Gives manager the information	careInfoSelected	infoGiventoManager	Manager	waiting	selectedInfoReady
7	Manager	Opens temporary file for family kin	selectedInfoReady	familyFileOpened	Staff	infoGiventoManager	familyFileOpened
8	Staff	Prints the information	familyFileOpened	infoPrinted			
9	Staff	Sends the report to the family	infoPrinted	infoSent			

Figure 5.6: revised version of Figure 5.3

Revisions to the description of Figure 5.3:

- Two added events:

  Staff gives manager the information.

  Manager opens temporary file for the family's kin.
- One added actor:

  Manager

- Moved (affected) events

From Figure 5.3, event 3 moves to position 8 (Figure 5.6).

From Figure 5.3, event 7 moves to position 9 (Figure 5.6).

The necessity of the additional actor and events was seen as a means to ensuring integrity check by the manager of the care home to ensure any outgoing information is validated, and records kept for future reference. Hence, the additional revisions of the initial description due to the application of enaction were deemed important to the specification of the appropriate behaviour of the system.

The output of the enaction produced the following description which was considered an accurate representation of the intended behaviour:

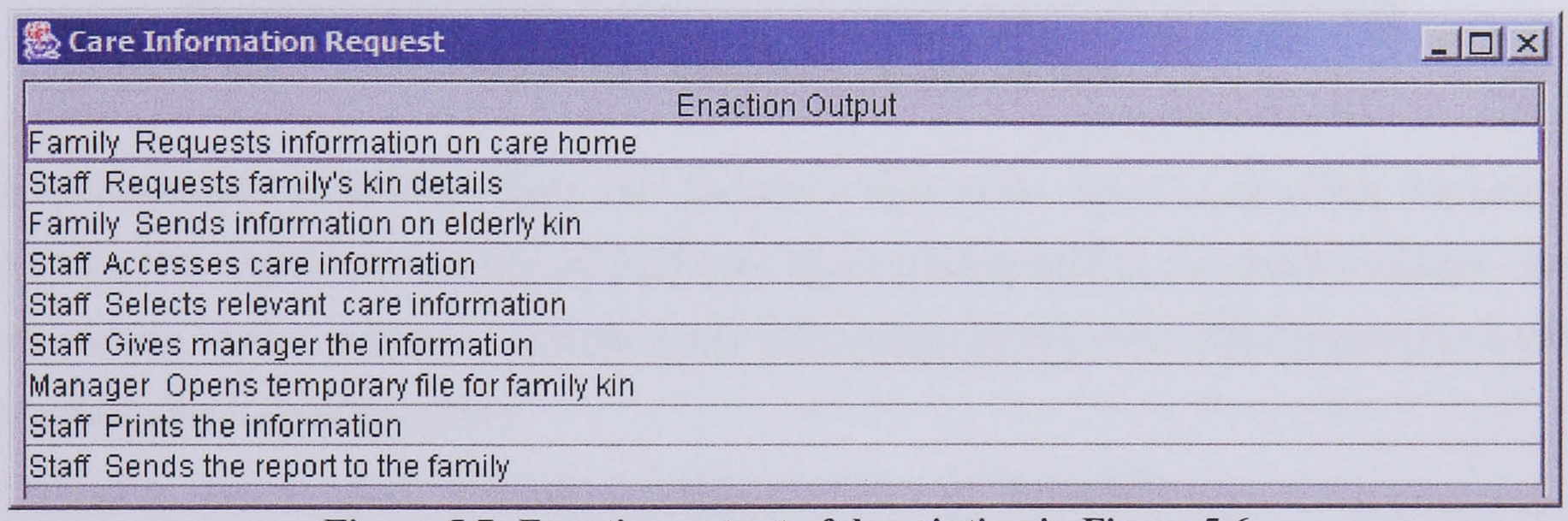


Figure 5.7: Enaction output of description in Figure 5.6

The use of the approach and the application of enaction by other students during this workshop raised a number of issues. These issues pertained to the functionality provided by the EducatorTool for both enaction and editing of descriptions. The issues are briefly discussed in section 5.2.4.1.

Considering Example 2

The enaction of the state based description (Table 5.2) with the EducatorTool produced a different order of events from the order of events depicted in the initial description (Figure 5.13). For instance, whereas the third event in Table 5.2 is "Operator sets movement type", enaction shows that the third event is:

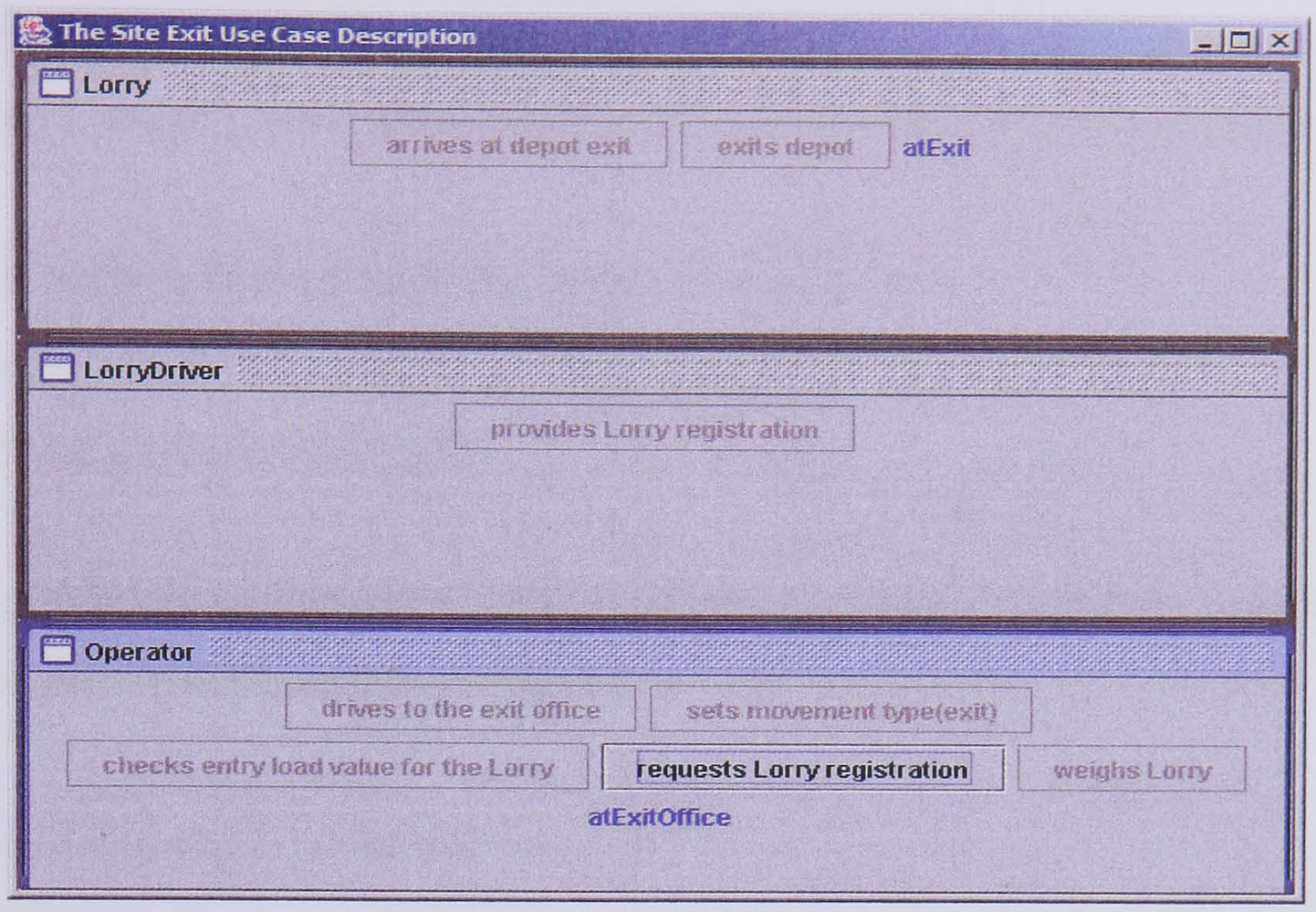


Figure 5.8: third event during enaction

This was explained as follows. Once the Operator arrives at the depot's exit office, the first thing is to check the registration details of the Lorry against those held in the depot's system. Setting the movement type is signalling that the Lorry registration details exist in the system (as a vehicle that entered depot legally before).

The output of enaction is the use case description shown in Figure 5.9:

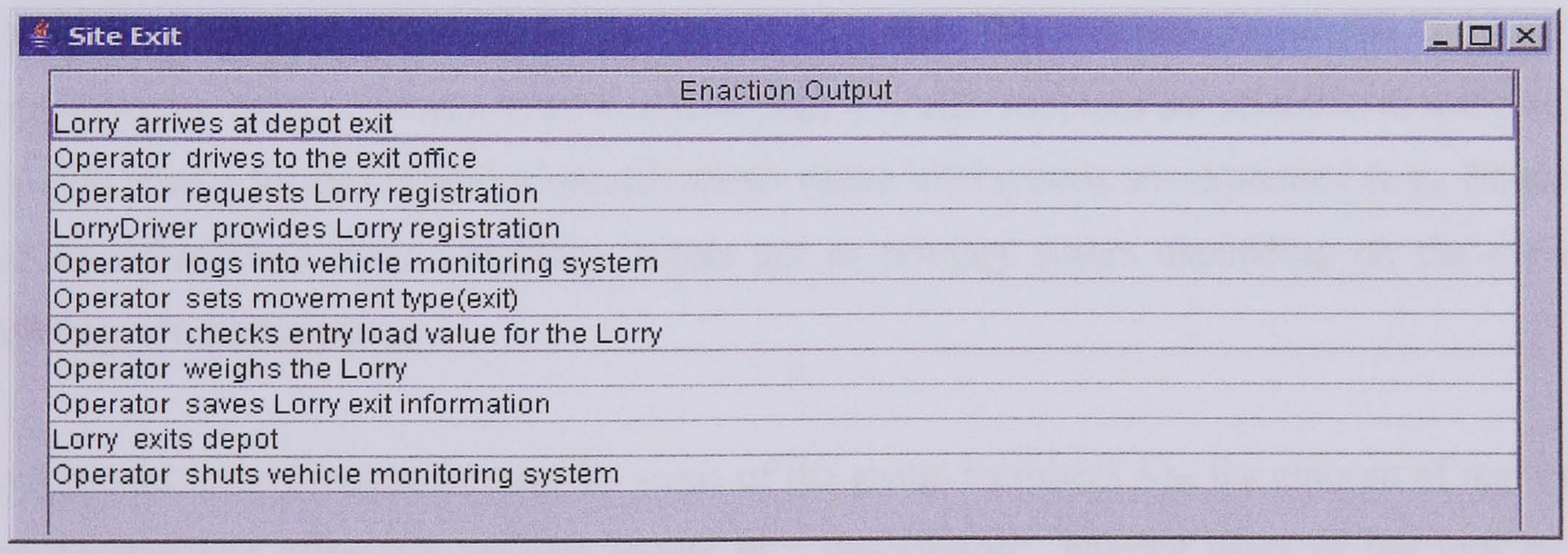


Figure 5.9: enaction output for depot exit use case

5.2.4 Discussion of issues

5.2.4.1 Tool issues

An issue reported was that there was no clear distinction between events for a main use case path and those of its alternative paths. The reason for this lack of distinction is that all events had the same font and print size. This was a problem, especially during enaction where a user needed to know which events belong to which use case path. This issue was resolved by changing the font

type of the alternative path events to italic with a font size of 12. Also, alternative path events have been given a colour yellow. Main path events have the times new roman font with size 12.

NB: alternative paths are named using the dot notation as shown:

<BasicPathName.AlternativePathName>

For example, if there is an alternative path called *ArrangeMeeting* in the use case of Figure 5.6, this alternative path could be named as follows:

Care Information Request.ArrangeMeeting.

Some students correctly pointed out that preconditions should be replicated and made available for selection as post conditions, and vice versa. That is, once preconditions are added, these preconditions should be made available to description authors for choosing as postconditions. The reason for this is that in a state-based specification, the precondition of one event might as well be the postcondition of another event. This functionality has since been implemented. The main benefit of this is that modellers are afforded greater efficiency since there is little scope for mistakes (e.g., spelling mistakes) as typing is minimised. This functionality works the same for actors, whereby once a primary actor is added, then it is also supplied for selection as a secondary actor. The reason for this is that in specifications where interactions are expressed (e.g., Educator-based specifications) some secondary actors act as primary actors depending on the event in which they are participating.

An additional issue brought forward by some of the group members was the amount of real estate that EducatorTool offers for writing events in a description. Several users of the tool in this group felt that EducatorTool was restricting their expressiveness for writing descriptions. The brevity with which EducatorTool enforces on event authoring was found to be restricting how much users could write. The Educator approach recommends a specification approach where modellers are able to clearly define the actors for their events; for this reason, the EducatorTool does not allow events to be added before a primary actor for the event is chosen. The event itself is written such that it is clear what the **verb** and the **object** in the event is. This model is again a recommendation within the CP guidelines. Functionality is provided within EducatorTool to write notes for each event where such additional notes may be needed. Whereas reasoning about

software behaviour in terms of atomic events with their participating actors might be more difficult compared to standard use case specifications, this approach is geared to enhancing clarity in considering dependencies amongst the events. Furthermore, articulating software behaviour during specification construction is often not an easy activity. Hence, the main reason for advocating brevity in writing use case events is to enhance clarity on the part of the description author (as suggested in the work of Cox [37] and Rolland [32]). Educator descriptions follow the CP writing guidelines (see section 4.4 of chapter 4). Indeed, the CP rules emphasise the issue of comprehensibility at great length. Moreover, where event text is too long to be viewed on the main display, a tree structure is provided where the events can be displayed alongside their actors and states.

5.2.4.2 Discussing the questionnaire

The questionnaire can be found in Appendix F. The questionnaire had suggestions stated in a way that the participants could respond in varying strengths of agreement or disagreement. In [132], it is argued that clarity is important in formulating questions and responses. The recommendations outlined in [132] were followed in constructing the questions. Furthermore, there are brief explanatory notes to each question. The responses were obtained in terms of whether the participants agreed, strongly agreed, disagreed, strongly disagreed, were undecided, or had no opinion. Short notes were written under each suggestion to explain each of the suggestions further. The confidentiality of the respondents was assured.

The first 4 suggestions required the respondents to tick an appropriate box indicating their answer. The fifth part required participants to write brief notes regarding their recommendations for improvements, and any constraints that they thought the approach and the tool placed on specification construction. Most of the comments written in part 5, and additional ones written in parts 1 through 4 have been discussed in the preceding "Issues and discussion" sections.
Figure F.1 (in Appendix F) shows that one of the respondents was undecided as to whether the tool was easy to use or not. This is because the respondent did not use the tool as much as the other respondents. Indeed, the respondent did not return any descriptions written using the tool despite using it during the initial trial session. Others found it easy to use as can be seen in the graph. Moreover, all respondents agreed (some even strongly agreed) that the scheme adopted for writing descriptions in a state-based fashion is well supported by the tool.

Enaction was found to provide feedback to the participants enabling further scrutiny of descriptions compared to mere reading of static textual use cases. Hence, the responses to suggestion 3 were indicative of the participants' agreement with the suggestion.

One participant disagreed with suggestion 4. The participant observed that the functionality to amend actors or states once enaction reveals that such changes are necessary was not straightforward. Participants recommended that all pre-states are also allowed to be selected as post-states, and vice versa. This issue has been discussed in the preceding "Issues and discussion" sections.

During the conduct of this workshop, the tool could only construct use cases and their alternative paths. The association of distinct use case descriptions had not been implemented. Some respondents pointed out to the necessity of this inclusion in the part 5 of the questionnaire where it was left open for the participants to describe any extra functionality they wished the tool to provide. Handling of multiple use case descriptions has now been implemented and is discussed in chapter 6.

Some participants wanted a further constrained way of writing descriptions. For instance, some argued that an intuitive starting point would be writing the actors, then the events for the actors and then the states. The tool has been built such that there must be a triggering actor for each event, and states can be added to events after the events have been written or during the writing of the events. Logically, the construction of an Educator description is such that a state cannot exist without an associated event, nor can an event exist without a triggering actor. Events can exist without states (as in default use cases) or without secondary actors.

5.3 Research group seminars

5.3.1 EducatorTool seminar 1

This seminar was conducted at the end of the first six months of the research reported in this thesis. The audience consisted of members of the Empirical Software Engineering Research Group (now Software Systems Modelling Research group). The research group comprised of 1 senior lecturer, 1 lecturer, 3 research fellows, 2 PhD students and 1 professor of software engineering. The general aim of the presentation was to report the progress made on the definition of the Educator approach and the support tool. The emphasis of the presentation is summarised as follows:

a) Clarifying to the group the nature of the state-based use cases as described within the Educator approach.

b) Demonstrating to the group how Educator descriptions are written within the EducatorTool.

c) Gaining feedback on any limitations of the approach and functions of the tool.

5.3.1.1 Discussion of issues

One of the issues that elicited discussion was the entry and exit points into and from a description during enaction. In other words, given that a description has more than one event, at what event does a modeller enter the enaction and at what point does a modeller exit enaction. This issue was explained as follows.

Educator descriptions can be written in two schemes: the standard (or default) scheme where states are not included in events and the state-based scheme where each event has a pre and post state. In the standard scheme, enaction starts at the event on position 1 (the first written event), proceeds to the second event, third event, and so on, until the event in the last position is enacted. In the state-based scheme, enaction of events is based on the event's state. Like in any other state-machine model, there is an initially available event, where enaction is entered from. The subsequent event is dependent on whether its pre-state matches the post state of the previous event; enaction is terminated (and exit point reached) in this scheme if there is no other event with a pre-state matching the post state of the currently available event.

Another issue was the extent to which information represented in a RAD model is transferable into an Educator description. This question arose from the fact that some of the research group members have a background in process modelling research. In particular, the Educator approach has been largely informed by the business process modelling approach whereby interactions between roles is depicted using state-based process models.

The author explained this issue as follows (also, others in the audience had similar views). RAD models of business processes provide a rich depiction of the business process being modelled. RAD models show roles, the responsibilities for those roles and their interactions as they execute various business tasks. On the other hand, there is no approach to date (as far as the author knows) that models interactions between actors within the UML use case in as a rich manner as business processes are modelled using RADs and associated tools. The Educator approach is a step toward providing this modelling capability. However, it should be noted that Educator use cases, like standard UML use cases can only model processes that require software support. Hence, much of a RAD model may not be described in an Educator use case simply because not

all roles (in a RAD) have a direct interest in a specific software system. A discussion of these issues can be found in [133].

5.3.2 EducatorTool seminar 2

This seminar was a presentation to the research group during a time when the group was hosting collaborators from a German university. The participants from the German university were one professor of software engineering and a prospective PhD student.

Discussion of issues

One of the issues raised during the presentation was whether use cases are well suited to constructing behaviour models depicting state changes between events. That is, is the deficiency of use cases typically a problem with the UML or that use cases are not meant for constructing interaction based models. The issue was raised with UML state charts in mind. First, construction of state chart models presumes a concise use case to have been produced in the first place. Secondly, the use of state-charts to enhance use case specifications introduces a layer of complexity to the specification process.

An additional issue that was raised was whether EducatorTool would provide further scrutiny of descriptions by constructing a trace of state transitions in diagrammatic form. Whereas this capability can be provided in latter versions of EducatorTool, the author set out to produce a proof of concept tool that enables enaction of state-based descriptions to support scrutiny of dependency issues. The general functionality for the enaction capability has been provided, and it would be worthwhile investigating these other issues (e.g., diagramming of state transitions) in further work on the approach and tool.

5.4 Chapter summary

This chapter has discussed several sessions that were conducted by the author to gain feedback on the approach and the tool proposed in this thesis. In particular, the chapter has described the changes that were made to the EducatorTool resulting from the feedback gathered.

The workshops and seminars provided invaluable feedback that became useful in evaluating the efficacy of the approach and the proof of concept tool. Given the need for automated support for many software engineering activities, it was important to obtain direct feedback regarding the EducatorTool as it plays the role of demonstrating the efficacy of the Educator approach.

This chapter does not by any means imply that the current tool is a final or complete version of a typical state-based use case authoring and enaction tool. There is scope for extension of the tool, especially to move towards the construction of a 'first cut' design model (e.g., a class diagram). Moreover, it is feasible to provide functionality that allows the production of some of the UML diagrams used for dynamic modelling (e.g., a sequence diagram or statechart) as a means of augmenting the class model of the system design. A further discussion on the extensions deemed necessary is provided in the final sections of chapter 6.

6. Industrial evaluation

6.1 Introduction

This chapter describes an industrial project undertaken as proof of concept for the Educator approach and the associated tool. The Educator approach is outlined in chapter 3. Chapters 4 and 5 discuss some aspects of the approach that pertain to initial feedback gathering and design. The goal of this chapter is to bring these ideas together in the form of a single industrial study. State-based use case specifications and the subsequent validation via early prototyping are demonstrated by constructing specifications of parts of a real-time monitoring system.

Use cases and state machine approaches address a wide range of software engineering concerns (e.g. behaviour specification, validation, automated analysis). It is, therefore, difficult for a single project to address all aspects of the use case notation or to demonstrate all of the claimed benefits of state-based approaches. The work reported in this chapter aims to demonstrate the rigour and scrutiny afforded by the proposed approach and tool in an industrial application. In particular, the analysed subsystems demonstrate the possible benefits of constructing enactable models of use cases for dependency analysis, and the ramifications of not considering such dependencies.

6.2 Aims and objectives

6.2.1 Aims

The workshops reported in chapter 5 highlight interesting features of the Educator approach and tool. In particular, the workshops and the seminars provided an opportunity to gain feedback, and, hence, improve upon the way in which the Educator approach is supported by the associated tool. However, the development of tools of this nature is often driven by industrial demand or a need to extend the novelty of existing notations; hence, an industrial project is a necessary test of fit. The basic aim for the work described in this chapter is to provide conduct an evaluation of the Educator approach and tool based on an industrial software project.

6.2.2 Objectives

There are two main objectives for undertaking an industrial specification project as part of the research described in this thesis. First, is to conduct an evaluation of the Educator approach to determine whether applying the approach within industrial projects is helpful in dependency analysis. It is to be noted that the application of the approach entails both augmentation of use cases with state-based information and the enaction of such use cases. The second objective is to evaluate the EducatorTool to determine its suitability for editing Educator-based use cases,

including the support of some of the guidelines outlined in chapter 4. A breakdown of these objectives is provided below:

1. To evaluate the extent to which the application of the Educator approach (including the mechanism of enaction) helps clarify dependency issues that were not found without application of the approach.
2. To evaluate the extent to which the EducatorTool provides an environment suited to authoring Educator-based descriptions, and the automated checking of the supported CP rules.

6.2.3 Evaluation activities

The evaluation activities for objective one will entail the augmentation of standard use case descriptions with pre and post states, and the consideration of secondary actors for events. An additional activity will be the enaction of such augmented use cases. The discovery of any (additional) of the following elements would be an indication that the application of the approach has helped highlight dependency issues:

1. Any additional events discovered.
2. Any additional actors discovered.
3. Any events whose order of occurrence is altered.

For any of the above items, a qualitative discussion is made to describe the ramifications for missing the additional specification element, or the significance of reordering the occurrence of events.

The evaluation activities for objective two will comprise the automated syntax checking of descriptions to be in conformance with the supported CP rules. Additionally, this objective will consist of evaluating the extent to which the EducatorTool supports the editing of descriptions. This editing involves adding new actors, or events, deleting events, changing actors for events, rewriting events, altering states for events, and editing associations among descriptions.

6.3 Evaluation strategy

The model evaluation framework presented in [134] provides a basis for evaluating software bidding models. Various model qualities (e.g., syntactic quality, semantic quality, model value and test quality) are outlined for consideration in the evaluation process. The application of the framework for specific evaluation circumstances would require that the framework be tailored to the specific model being evaluated. Table 6.1 lists the quality aspects outlined in initial model

evaluation framework [134], the initial definition of those qualities, and the way in which those qualities are tailored for evaluating the Educator approach and tool.

Quality aspect	Generic application (by Kitchenam)	Application for evaluating Educator approach/tool
Syntactic quality	That the language used (e.g. predicate logic or algebraic equations) in producing model constructs is syntactically correct.	That the controlled natural language adopted for constructing Educator descriptions is correct as per the adopted CP guidelines. These can be checked by using the automated CP checker within the tool.
Semantic quality	That model statements are correct and relevant to the problem domain. That such statements are complete in that no more statements about the problem are necessary for the model.	That Educator descriptions are correct and relevant to the problem being specified since such descriptions are validated by domain experts. An aid to validation is augmenting such descriptions with states, and enaction.
Pragmatic quality	A model quality concerned with whether roles (or stakeholders) interested in the model can relate their understanding of the process/task being modelled (e.g., bidding process in Kitchenam's case) to the model itself (bidding model)	For the Educator approach and tool, this will be concerned with whether domain experts do offer qualitative explanation of their reason regarding changes they recommend during specification construction and validation. That is, do the participants identify specification issues that are consistent with domain concerns?
Test quality	The use of simulation to assess whether the model's outcomes are consistent with specific (pre-defined) situations.	There is no sense in which this is necessary for the industrial project undertaken to evaluate the Educator approach and tool.
Value	The extent to which the use of the model improves an organisations process (e.g., bidding process)	The extent to which the use of the approach and tool is seen to improve the specification and validation process. That is, what is the significance of revisions made to specifications as a result of applying the Educator approach and using the tool?

Table 6.1: Quality aspects for evaluating Educator (based on Kitchenam's framework)

For each quality aspect, there is an associated goal (or goals) to be achieved by the evaluation. Moreover, further adaptation of the initial framework (see [135]) introduced the need to consider means for achieving the goal (by model developer) and means to assess the achievement of the goal (by the model evaluator). For the Educator approach and tool, the model development (specification of Educator approach) and evaluation were mainly conducted by this author, with participation and input (in terms of use case descriptions) being provided by staff from the company where the project was undertaken. Table 6.2 outlines the enhanced framework tailored to provide goal definitions for the evaluation of the Educator approach and tool. The table also provides brief descriptions of the means to achieve the stated goals and the means to assess their achievement. In [135], a further enhancement to the initial framework of [134] was the

delineation of the object of evaluation. The object of evaluation is either a conceptual model (e.g., the Educator approach) or a software tool supporting the conceptual model (e.g., EducatorTool).

Quality aspect	Object	Goal	Definition	Model properties	Means to achieve goal	Means to assess achievement of goal
Syntactic quality	Model and tool	Syntactic correctness of Educator descriptions	All the descriptions adhering to the Educator approach and authored with the support tool	Defined syntax – includes structure of Educator events, and supported authoring guidelines	Support for authoring Educator descriptions, and checking of adherence to guidelines	Review of each deficiency and each suggested correction with an option to obey rules or override them
Semantic quality	Model	Feasible completeness; feasible validity	The model is feasibly complete and feasibly valid	Traceability to domain	Descriptive notes of model elements (e.g. secondary actors) linking the model to the domain; checking of models' consistency based on pre/post states	Interviews and discussions with company staff to elicit their views on descriptions
Pragmatic quality	Model	Feasible comprehension	As far as possible, Educator approach is understood by its target audience	Description elements (e.g., actors, events, states)	Presentation of Educator approach using demonstrations and examples	Eliciting of understanding achieved by staff via interviews
	Model	Feasible understandability	As far as possible, Educator approach is presented in a format that is understandable to its target audience	Structuredness of Educator descriptions	Not applicable	Not assessed
Test quality	Tool	Feasible test coverage	Adequate testing of the tool in terms of feasible test coverage to assure support for Educator approach	Executability and enaction i.e. ability to execute the tool in order to do enaction	Example usage of the tool (test cases) demonstrating tool requirements and showing how the tool supports each type of user for authoring simple or enhanced descriptions	Evaluation of implied behaviour of the use case descriptions with staff
Value	Model	Practical utility	The extent that the approach facilitates rigour of scrutiny of when authoring specifications	Not applicable	Assessment of additional description elements produced as a result of using the approach	Interviews with staff to determine their views on the use of the approach
	Tool	Practical utility	The extent that the tool improves the efficiency and effectiveness of the user organisation	Not applicable	Appropriate user interface design for authoring descriptions with EducatorTool, online user manuals.	Not assessed as tool was mainly used by this author during the study

Table 6.2: Educator approach and tool evaluation framework

For the industrial study reported in this chapter, the quality aspects assessed for the Educator approach and tool will be highlighted in various sections of the chapter where a specification activity constitutes the evaluation of that quality aspect.

6.4 The project

6.4.1 The company

The company concerned (based in the UK) is a software house specialising in the development and maintenance of real-time monitoring and control software for its clients (other companies distributed across the UK). During the time of the study, the company had a staff base with the following roles:

1 Software developers: - staff who are involved with the development and maintenance of the monitoring and control systems.

2 Software Projects manager: - a member of staff who works closely with software developers on prioritising and agreeing on the projects and project deadlines.

3 Operations manager: - responsible for recruitment of new customers for the company; this role also determines and liaises with the customers regarding the various fixtures that would be supported.

4 Applications manager: - charged with the deployment of systems at customer sites, and the management of software development milestones for new releases or upgrades.

5 The managing director: - the main decision maker and has the final say on the customer to be supported, the fixtures to be allowed for support and also follows project completion and success on deployed sites.

6 Incident operators: - these are operators at the control centre office who provide support to remote customers. Their work involves notifying customers (through telephone calls or email) of any incidents that need to be attended to, or indeed, receiving requests from customers and acting upon those requests. There are at least 4 fulltime incident operators and an unknown (to the author) number that work on shift basis.

The description of the various systems and their organisation, including a discussion on efforts to re-engineer the systems for centralisation to the company site is provided in Appendices A.1, A.2, A.3, and A.4. Appendix B provides a description the type of details the company typically stores with regard to their client organisations. Further use cases can be found in Appendix C.

6.4.2 Purpose of the project

There were two main reasons for conducting the study:

1) The company concerned realised that they needed to construct a comprehensive specification for their existing software systems for maintenance purposes.
2) The author took the opportunity to conduct an empirical evaluation of the Educator approach and associated tool.

Regarding the first purpose stated above, the company had no existing documentation or specification of their systems. Given the workload on development tasks for the existing development team, the company decided to involve the author in the specification of their systems. One key purpose for the specification was stated to be the continued maintenance of the systems. Another essence of the work was to produce specifications for systems depicting new initiatives for centralising most systems' functions at the control centre. That is, the company wanted to reduce the amount of functions distributed to remote customer sites on various control system components. The operations manager advised that centralisation of systems would reduce travel overheads as the company often had to send out engineers to customer sites to resolve apparently simple problems.

Hence, the author undertook the opportunity to participate in the project in order to conduct an evaluation of the Educator approach using industrial applications. A difficulty in conducting the study was the conflict between the author's need for the study and that of the company. The company wanted all the systems specified and documented as quickly as possible. The author wanted to determine the extent to which the Educator approach and the EducatorTool could help clarify dependency and interactions issues within industrial setting. This conflict of interest meant that it was not possible for this author to organise participants in a way that would suit some aspects of the study design, rather, the company had their own project agenda which had to be adhered to.

6.4.3 How the project was carried out

The study involved a number of visits (by the author) to the company offices where interviews were conducted. The staff that participated in the interviews included two software developers, one software projects manager, one applications manager, one operations manager, and the managing director. The first interview was with the managing director, the applications and operations managers who provided an overview of the existing systems, associated databases, the nature of the customers supported, and the 'logical' preview of the differences among systems at customer sites and those at the control centre offices. Subsequent interviews involved the

developers, the software projects manager and the applications and operations managers. The general process of gathering data followed the following pattern. Given that the author was unfamiliar with the control engineering domain, the initial data collected entailed a brief description of each system. This was then followed by a detailed description of the systems in the form of use cases that depict the behaviour of the system. The use case descriptions were then revised by the author using the proposed state-based approach. The state-based information was elicited from the participants. Resulting state-based use cases were subsequently enacted (animated) to visualise the implied behaviour. Moreover, the author inspected the existing code for the subsystems studied to determine the extent to which the specifications matched the existing functionality. The functions of the inspected code were further discussed with the developers from the company to ensure the author's understanding of the code matched that of the developers. Additionally, the reverse engineering tool, CodeLogic, was used to produce class diagrams from code to gain further insight into what the developers had in place. Again, the class diagrams were verified with the developers. This demonstrates that the Educator approach and associated tool support can be used alongside other software design/development tools, in order to provide different perspectives of a system architecture to support maintenance efforts.

6.5 Discussion of study issues

6.5.1 Data gathering and procedure

The data collected for the study was use case descriptions detailing the behaviour of various subsystems. These descriptions were elicited via interview with participants from the company where the study was undertaken. The author had been provided with a document outlining the systems supported by the company. Hence, the interview involved the author mainly asking questions regarding the functions of these systems, and the participating staff (mainly developers) addressing the questions. The analysis involves augmentation of use cases with pre and post states, and the enaction of the use cases.

The analysis process took the following general procedure. An understanding and documentation of the system was obtained first. This understanding was gained through a discussion with participating company staff where an overview of each monitoring system was provided (by the staff to the author). The author wrote this overview using MS-Word on a laptop computer. Further elaboration of the functions of each system was sought by the author, and this is where use case descriptions were constructed (with the help of participating company staff) to describe the functions of each system. That is, an initial use case description (standard use case) was

written with help of the participating staff following an overview of the subsystem's function. Following that description, a state-based one would be constructed again with the help of the participating staff. Often, this author would work separately to produce state-based use cases which would later on be verified with the company staff. Additionally, enaction would be applied to try and clarify any specification issues that might not have been highlighted with augmentation alone.

The study is to be considered a success if augmentation of use cases, and enaction helps 'tease out' dependency issues that would otherwise not have been revealed without the application of the Educator approach or the use of enaction within the EducatorTool. The consideration of the extent to which the study succeeded (or did not succeed) is discussed in the "**Explanation**" sections following each analysed use case description.

To ensure the approach and tool were applied correctly, the author took part in every interview. As indicated above, the interview comprised of an explanation (by participating staff) of functions for certain systems (based on a question from the author). Hence, a brief overview of each system/subsystem was constructed in Microsoft Word first (the author had a Laptop computer where the descriptions were typed); the use case descriptions of each system/subsystem were then edited within EducatorTool. In effect, the author did all the editing while the participants provided the information pertaining to the systems; the participants also provided input on states for various events in the use case descriptions. The use cases provided within this chapter are those considered most important by the company staff; part G of the appendices has further analysed use cases.

6.5.2 Analysis issues

A typical way for the use of the approach and the tool is that standard use case descriptions are revised using the proposed approach and enacted within the EducatorTool. Each resulting description is then compared with the initial use case without the application of the Educator approach or EducatorTool. The selection of the systems to be studied and analysed was based on recommendations from participating staff from the company regarding the parts of the control systems they regarded most crucial to them and their customers. In general, the study evaluates the extent to which the application of the approach and tool support helps gain insight into dependency issues that would otherwise not have been considered without this type of analysis. Whereas the study is broken down into various subsystems, the problem domain is that of real

time monitoring systems. That is, the organisation concerned is a software house within the real-time monitoring systems domain. The company is typical of many other software houses in terms of use of software development technologies. For instance, like many other software development companies, the concerned company uses technologies such as C#, Java, C++ and SQL server. The company also constructs design models such E-R models, UML diagrams, etc. Hence, the study results are generalisable to domains similar to that of the studied company. As stated above, many software houses are typical (e.g., in construction of use case specifications during requirements analysis) of the one where this study was conducted, and the results of this study are equally applicable to those other companies (and system types) since similar issues can potentially be investigated using the approach and tool described in this thesis. That is, whereas the target clientele and domain is often different for different software development houses, the application of the use case notation for requirements and specifications is common across software houses. It is for the validation of such use case specifications that the results of this study are considered generalisable.

Considering learning effect

It is recognised ([136], [137]) that learning effect can affect the results of an empirical study (e.g., experiments and case studies). For the industrial study reported in this chapter, the author made consideration to address the learning effect issue. One way to rule out learning effect would be to conduct an experimental design where the analysis of use case descriptions is conducted as follows. Consider any two use case descriptions as follows:

Use case			Dependencies
1	Standard A	Augmented B	*n*
2	Augmented B	Standard A	*m*

Table 6.3: Considering learning effect

The table above shows use cases 1 and 2. For use case 1, the dependencies *n* is the number of dependencies revealed as a result of producing Augmented B following the construction of Standard A. For use case 2, the dependencies *m* is the number of dependencies revealed as a result of producing Augmented B first and then Standard A. Whereas this experimental design would assure the discovery of dependencies is not due to learning effect on the part of the modeller, the experiment is practically impossible for various reasons. First, the standard use case description is always a subset of the augmented one, hence, starting with augmentation and then producing a standard use case is not possible. The reason for this is that augmented use case descriptions have all the information that the standard ones have, and more (state-based

information, and notes). Second, the Educator approach and tool work by moving from an initial sequence of events (the standard use case) that is then augmented and enacted to determine any dependency issues. Hence, there is no sense in which a modeller would find it useful to start with an augmented use case as a first-cut analysis of the problem in hand.

6.6 Evaluating the Educator approach

This section considers the subsystems studied and the constructed use cases which helped provide an evaluation of the Educator approach. The Educator approach is proposed as a means for considering intra-use case and inter-use case dependencies. This section is organised to highlight and address these types of dependencies. Enaction is part of the Educator approach, and this section also provides an assessment of the extent to which enaction of Educator-based descriptions highlighted dependency issues. Additionally, this section highlights the quality aspects assessed as a result of the specification activities undertaken. References to the quality aspects outlined in Table 6.2 will be made where necessary.

6.6.1 Intra-use case dependencies

This section discusses examples that help to highlight the importance of considering intra-use case dependencies.

Example1: the alarm receiving process

This section provides an example to address objective 1. The section demonstrates the Educator approach by analysing the alarm handling process. The aim is to scrutinise and determine dependencies amongst the steps of the alarm handling process. One of the key functions of the company is the monitoring, receiving and recording of alarms. The process revolves around an alarm handling subsystem. Besides receiving and recording alarms, the subsystem also provides suggestions on appropriate alarm resolution actions for an operator at the customer's site. The following use case description was produced during interview with developers and applications manager at the company:

1. AlarmReceiver receives alarm data
2. AlarmReceiver records alarms
3. Operator accesses existing alarms
4. Operator views recommended alarm actions
5. Operator resolves alarm

Figure 6.1: Alarm Receiver use case

The essence of the Educator approach is to facilitate consideration of dependencies by producing a state machine model of the use case description. This state machine model requires that each event has a pre and post state; moreover the modellers should consider whether an event has a secondary actor participating in the execution of the event. The use case of Figure 6.1 revised to adhere to the Educator approach is shown below:

ID	Primary Actor	Event	Precondition	Postcondition	Secondary Actor	Precondition	Postcondition
1	AlarmReceiver	requests alarm data	initial	requested	***Scheduler***	waiting	requestReceived
2	Scheduler	sends alarms data	requestReceived	dataSent	***AlarmReceiver***	requested	waiting
3	AlarmReceiver	receives alarms data	waiting	received			
4	AlarmReceiver	records alarms data	received	recorded			
5	Operator	accesses existing alarms	initial	accessed			
6	Operator	views viable alarm actions	accessed	actionsViewed			
7	Operator	resolves alarm	actionsViewed	alarmResolved			

Table 6.4: Alarm Receiver use case (revised in Educator approach)

6.6.1.1 Effects of augmentation

There are a number of dependency issues that came to light as a result of considering pre and post states for each event (without application of enaction). For instance, in Table 6.4, event 1, it was not clear from where the alarm receiver receives its alarm. Consideration of states for an alarm receiver, before and after receiving an alarm, caused the introduction of the Scheduler actor which is responsible for relaying alarm data to the alarm receiver. However, before sending any alarm data to the AlarmReceiver, the Scheduler must have received a request regarding alarms from the alarm receiver. Hence, receiving of an alarm by the AlarmReceiver is dependent on two actions:

1) The AlarmReceiver must first request the alarm data

2) The Scheduler must send the alarm data to the AlarmReceiver

The participating staff indicated that the consequence of missing out the Scheduler in the alarm process is that active alarms could not be received. Hence leaving out the functions of Scheduler in the alarm process would mean that fire alarms are not received and recorded for the operator to act upon. Moreover, intruder alarms would not alert staff for action where intrusion had occurred. A summary of the revisions made to the initial description, due to the consideration of states for each event is given:

Revisions to the initial description:

Two added events:

AlarmReceiver *requests alarms data.*

Scheduler *sends alarms data.*

One added actor:

Scheduler.

Moved (affected) events:

After augmenting the use case of Figure 6.1 with states, the initial five events were each moved 2 steps down. The revisions and clarification of initial description in the way shown above was regarded as important by the staff participating in the project. It was mentioned that an omission of the Scheduler (actor) subsystem would technically curtail the working of the AlarmReceiver.

6.6.1.2 Explanation

In Table 6.4, the AlarmReceiver is at *initial* state prior to triggering event 1, and moves to *requested state* after requesting alarm data. Alarm data requests are made to a middle ware subsystem, Scheduler, which is a secondary actor during the first available event. Scheduler moves from a *waiting* state to a *requestReceived* state during its interaction with AlarmReceiver. Scheduler then performs the event whose pre-state is *requestReceived*; this event involves sending the data to the AlarmReceiver. Scheduler moves from *requestReceived* state to *dataSent* state, whereas AlarmReceiver moves from *requested* state to *waiting* state. The AlarmReceiver then receives the alarm data, and moves from *waiting* state to *received* state. Once received, the alarm is then recorded, and the AlarmReciever moves from *received* state to *recorded* state.
Site-based operators are able to access recorded alarms (event 5), view actions recommended for resolving alarms (event 6) and subsequently act to resolve the alarms (event 7).

6.6.1.3 Changes to event positions

The initial set of events (derived from first interview-Figure 6.1) had their ordering changed by two positions each. This was regarded as a significant change given that events such as "*AlarmReceiver records alarm*" cannot happen until the alarm has been requested (event 1-Table 6.4) and received (event 3 -Table 6.4). Hence, the use of augmentation helped highlight dependency issues regarding an essential actor and important events that were missed in Figure 6.1.

The augmentation of use case events with pre/post states has helped reveal dependency issues that were not explicit without the augmentation of the use case. The use of state-based information within use case descriptions forces the modeller to reason about the consequences of each event to occur, and hence follow the logic of the events. It is this rigorous reasoning within a state-machine model that the Educator approach endeavours to foster.

Quality aspects assessed

As a consequence of considering pre and post states for events in the Alarm Receiver use case, several addition description elements were revealed. These include additional actors and events, and the altering of the initial order of occurrence of events based on feedback from the participating staff. Hence, in validating the Alarm Receiver use case the Educator approach quality aspects assessed include both **pragmatic** and **semantic** quality. The semantic quality aspect of the approach is considered in the situation where additional actors (e.g., Scheduler) are deemed necessary for the description to be complete within the specific real time monitoring sub-system. The pragmatic quality aspect is deemed to have been assessed because the participating staff were able to provide input in constructing and amending the use case for the Alarm Receiver sub-system. Discussions with the participants indicated that they understood the Educator-based description of the sub-system, hence providing a positive assessment of pragmatic quality.

6.6.1.4 Enacting state based use cases-further dependencies analysis

This section considers enaction to further address objective 1. The part of objective 1 addressed is that of enaction as a constituent part of the Educator approach. Educator models are visual prototypes of behaviour where the occurrence of events and the ensuing state changes of involved actors are visualised. It is recognised by many researchers (e.g., [138], [139], [140], [130]) that some form of visual simulation (or prototyping) enhances clarity and shared understanding during requirements validation tasks. Enaction has to be performed using an automated environment, and the EducatorTool provides such support. The use case of Table 6.4 is shown within the EducatorTool in Figure 6.2:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Alarm Receiving

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	AlarmReceiver	requests alarms data	initial	requested	Scheduler	waiting	requestReceived
2	Scheduler	sends alarms data to AlarmReceiver	requestReceived	dataSent	AlarmReceiver	requested	waiting
3	AlarmReceiver	receives alarms data	waiting	received			
4	AlarmReceiver	records alarms data	received	recorded			
5	Operator	accesses existing alarms	accessingAlarms	accessed			
6	Operator	views recommended alarm actions	accessed	actionsViewed			
7	Operator	resolves alarms	actionsViewed	alarmResolved			

Figure 6.2: Use case of Table 6.4 edited in EducatorTool

An enaction dialogue prior to the execution of the first available event is shown in Figure 6.3:

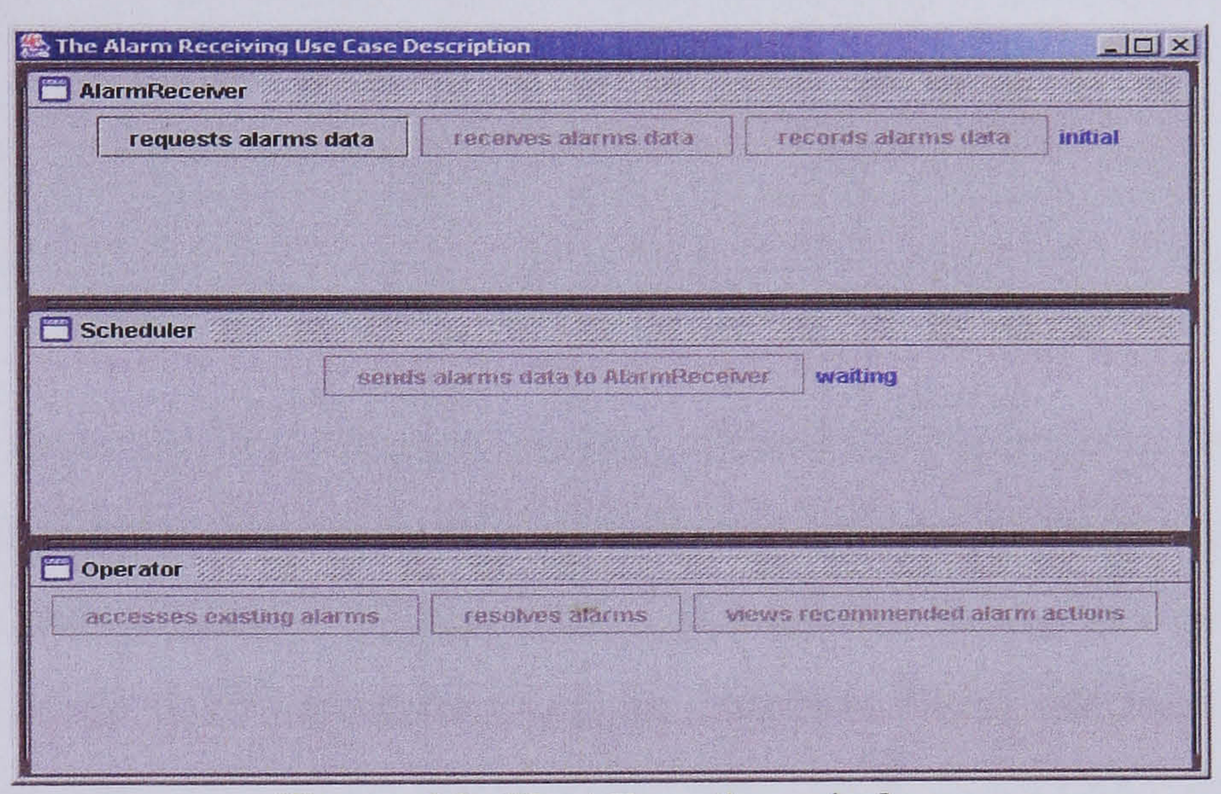


Figure 6.3: First Enaction window

When the alarm receiver receives alarm data then the data should be recorded, hence the dialogue in Figure 6.4:

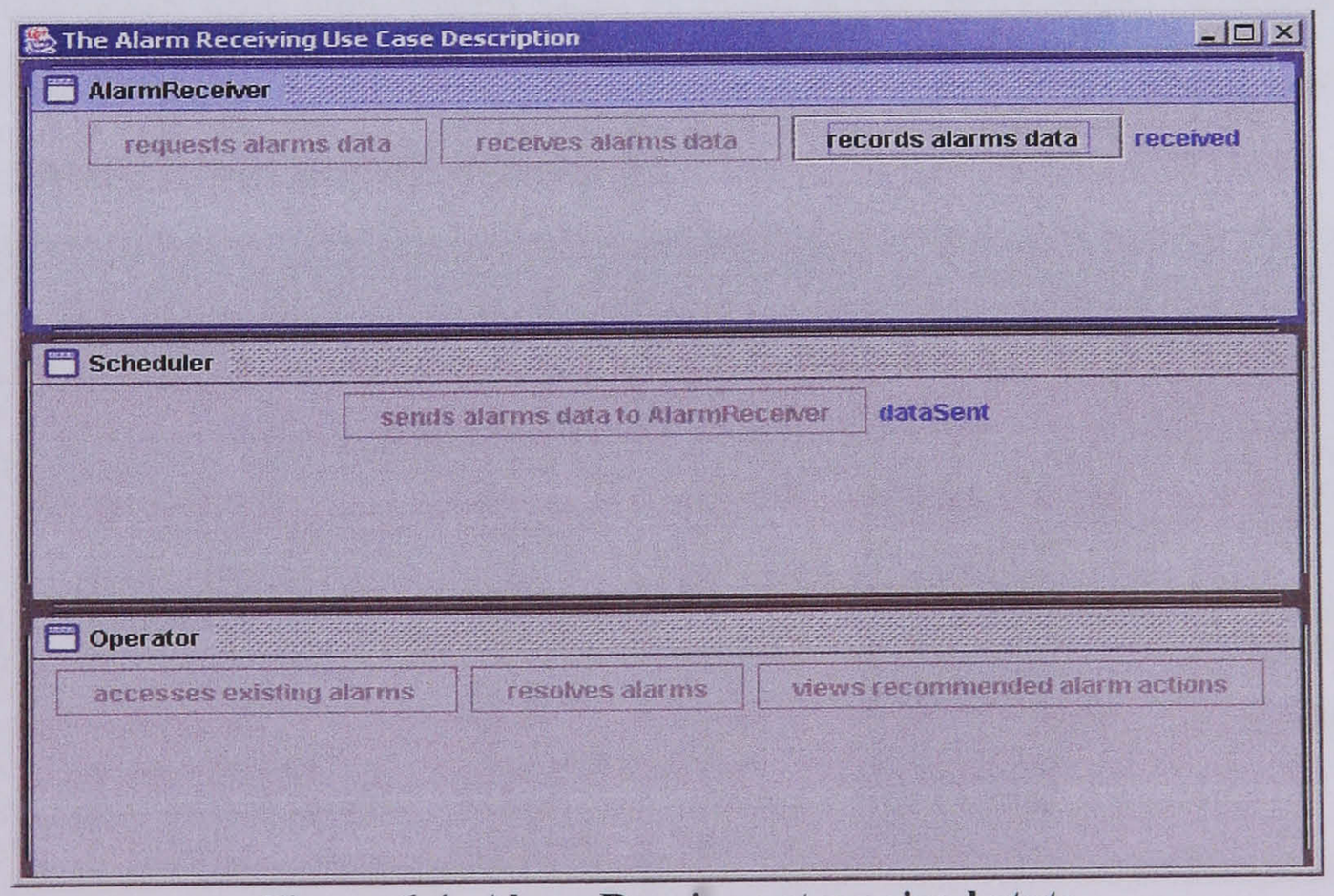


Figure 6.4: AlarmReceiver at received state

Following this state machine model, the occurrence of the event in Figure 6.4 above produces the dialogue in Figure 6.5 where there is no further enactable event:

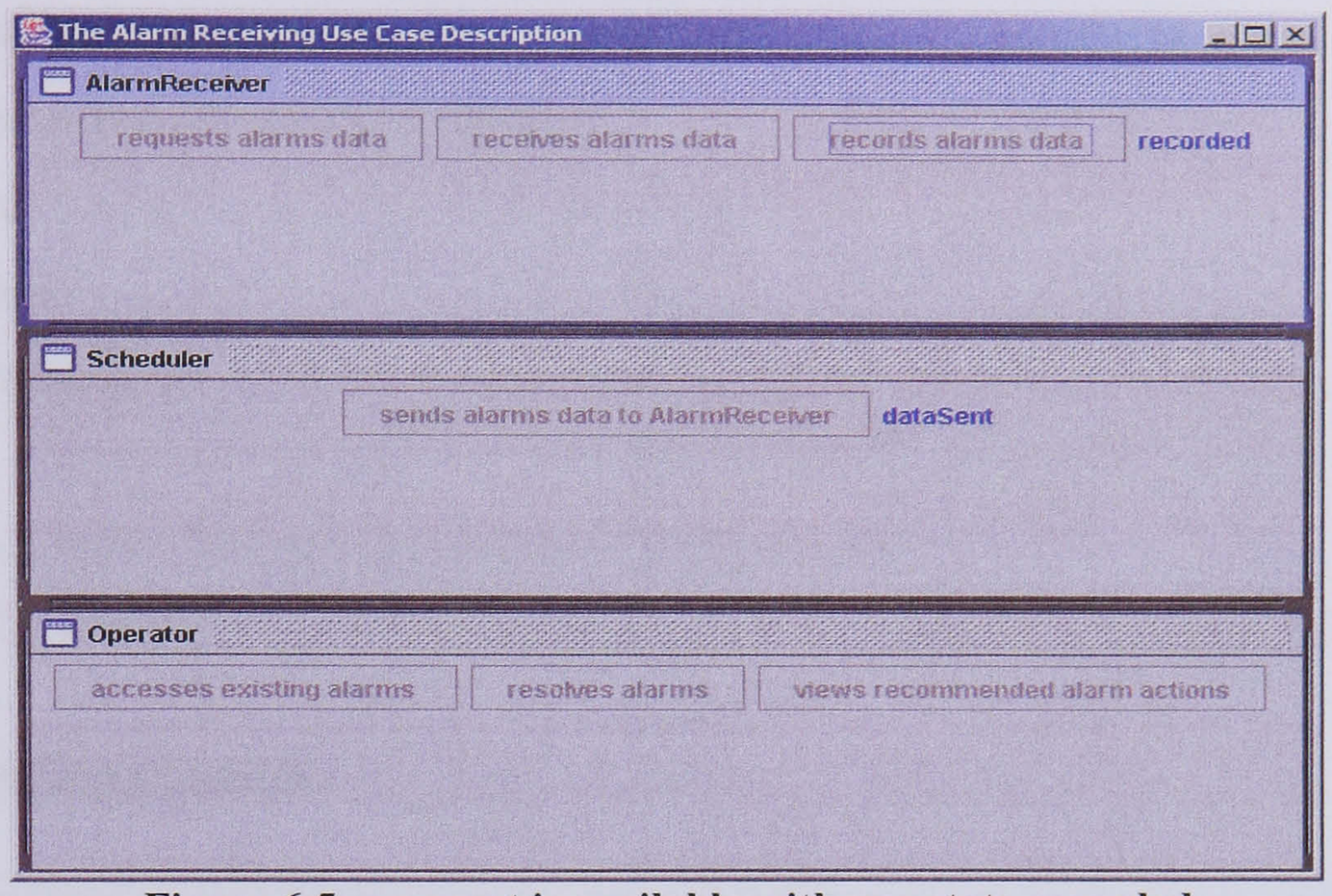
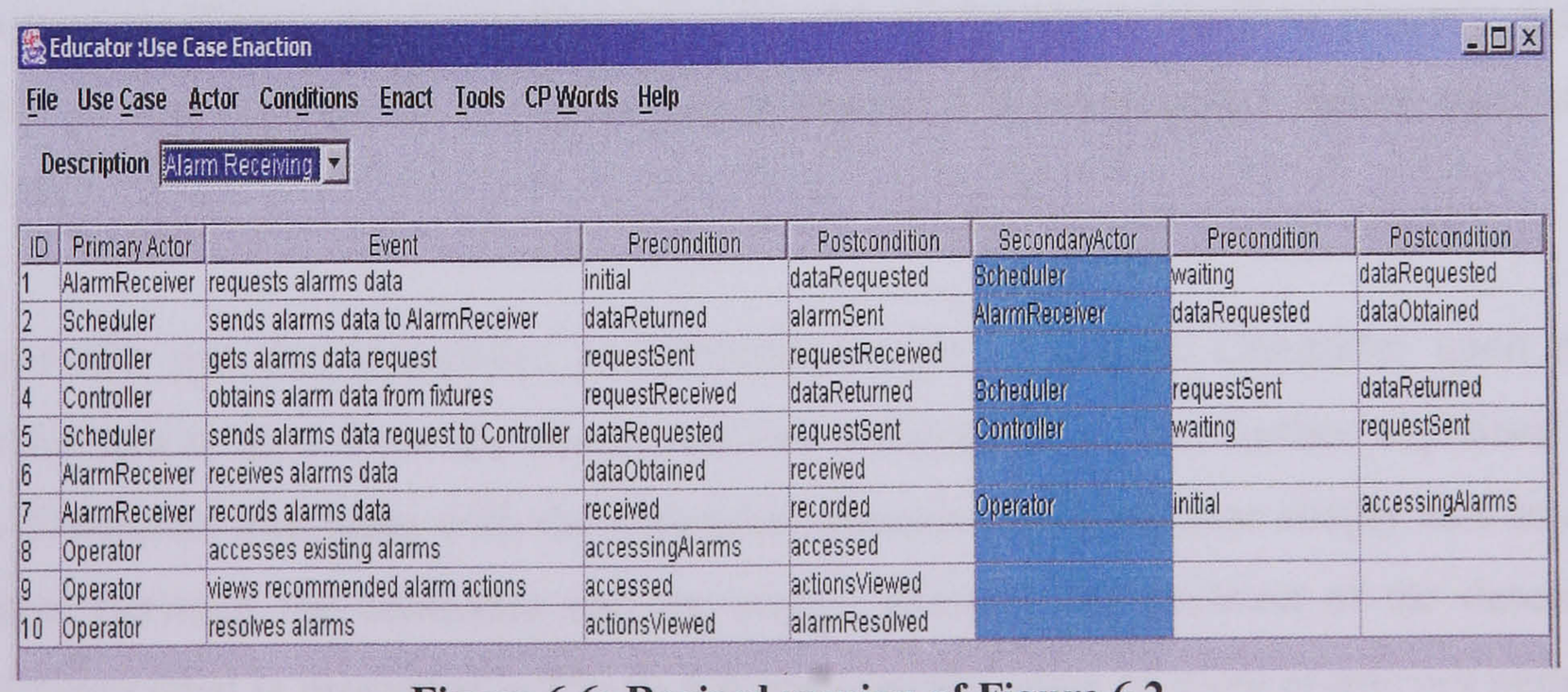


Figure 6.5: no event is available with pre-state recorded

There was no such event and this situation was only noticed during enaction. In other words, once the alarm is recorded by the AlarmReceiver, nothing else happens to the alarm. The consequence of a stalemate like the one shown in Figure 6.5 is that neither event 4 nor event 5 would occur. Hence, any alarms could not be attended to by the operator. Where the alarm receiving module does not raise alarms when they occur, the consequences can be catastrophic (e.g., in the case of a fire). Furthermore, it was pointed out that the specification of a correct sequence of events, and the knowledge about the states for each involved component was important for any new development staff; such new staff would need to learn about interactions between systems to be able to support the systems. A further scrutiny of the inter-relationships among events caused the revision of description to produce the following:



ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	AlarmReceiver	requests alarms data	initial	dataRequested	Scheduler	waiting	dataRequested
2	Scheduler	sends alarms data to AlarmReceiver	dataReturned	alarmSent	AlarmReceiver	dataRequested	dataObtained
3	Controller	gets alarms data request	requestSent	requestReceived			
4	Controller	obtains alarm data from fixtures	requestReceived	dataReturned	Scheduler	requestSent	dataReturned
5	Scheduler	sends alarms data request to Controller	dataRequested	requestSent	Controller	waiting	requestSent
6	AlarmReceiver	receives alarms data	dataObtained	received			
7	AlarmReceiver	records alarms data	received	recorded	Operator	initial	accessingAlarms
8	Operator	accesses existing alarms	accessingAlarms	accessed			
9	Operator	views recommended alarm actions	accessed	actionsViewed			
10	Operator	resolves alarms	actionsViewed	alarmResolved			

Figure 6.6: Revised version of Figure 6.2

Revisions to the description of Figure 6.2:

Three added events:

Scheduler *sends alarms data request to controller*

Controller *obtains alarms data from fixtures*

Controller *returns data to Scheduler*

One added actor:

Controller.

Moved (affected) events

From Figure 6.2, event 3 moved to position 6 (in Figure 6.6).

From Figure 6.2, events 5, 6, and 7 moved to positions 8, 9 and 10 respectively (in Figure 6.6).

6.6.1.5 Explanation of results

The following table summarises the results of augmentation alone and augmentation coupled with enaction for the AlarmReceiver use case.

Variable	Augmentation	Enaction	Augmentation + Enaction
No. of events discovered	2	3	5
No. of events moved	5	4	9
No. of actors discovered	1	1	2

Table 6.5: Summary of changes due to augmentation and enaction

The introduction of the network **Controller** actor and its events meant that there was clear description of how the Scheduler obtains alarms data from the fixtures in order to relay the data to the AlarmReceiver. These issues were deemed significant to correct execution of the alarm receiving process. Moreover, the re-arrangement of events 3, 5, 6, and 7 (of Figure 6.2) was made possible by considering the Controller actor as part of the alarm handling process. Again, this consideration was highlighted by visualising interactions between actors during enaction of the description.

It was pointed out that Scheduler communicates with a network Controller upon receiving requests for alarm data from the alarm receiver subsystem. It was further explained that the network Controller interfaces with the networked fixtures and Scheduler simply acts as a type of middleware between the controller and the alarm receiver. The revision of the description of Figure 6.2 into what is presented in Figure 6.6 resolved the issue of some events not being able to execute. For instance, the issue visualised in Figure 6.6 was resolved when it was pointed out that

a site-based operator must access recorded alarms and subsequently act to resolve them. Hence an **Operator** actor is involved in an interaction with the **AlarmReceiver** during the execution of the event in Figure 6.5. The **Operator** is a secondary actor in that event. Hence, the events 8, 9 and 10 are now able to execute.

Figure 6.7 shows output of the enaction after the description was corrected through enaction and augmentation:

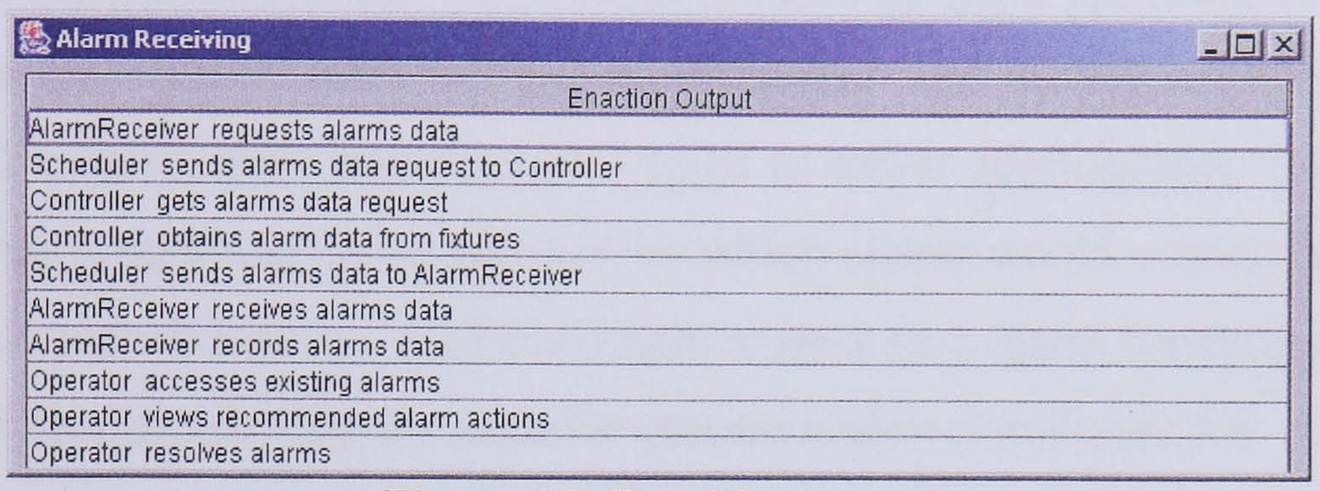


Figure 6.7: Resulting output

Upon consideration of dependencies (via augmentation and enaction) the final description has 4 more events and two more actors added to the description that had initially been constructed (in Figure 6.1) at the first interview. It is this kind of rigorous scrutiny of specifications that the Educator approach seeks to emphasise as means to clarifying issues concerning dependencies and interactions.

Quality aspects assessed

As a consequence of enacting the Educator descriptions, several addition description elements were revealed. These include an additional actor (Controller) and 3 additional events. Thus, in the further validation of the Alarm Receiver use case the quality aspects of the approach, including **pragmatic** and **semantic** quality were assessed. The semantic quality aspect of the approach is considered in the situation where additional actors (e.g., Controller) are deemed necessary for the description to be deemed complete and valid. Again, the pragmatic quality aspect was assessed because the participating staff were able to provide input in the revisions to the description following information arising from the enaction. It was apparent to the author that the participants understood and followed the enactable models, further providing a positive assessment of pragmatic quality. **Syntactic quality** was assessed in that descriptions in this section are edited with the EducatorTool, which has automated checking capability for various CP rules. Hence, it is reasonable to argue that syntactic correctness was assessed.

Example 2: client-server connection process

The example considered in this section helps evaluate objective 1.

The company wanted to move most of the monitoring systems' functions from remote sites to reduce the travel overheads incurred by engineers each time they need to solve issues at disparate customer locations. The envisioned change of the infrastructure was described by the operations and applications managers. The site-based systems (seen as client applications in a new client-server infrastructure) would hold reduced functionality and would have a communication mechanism with control-centre based systems (seen as server applications). There needed to be established a generic communication protocol by which clients would connect to servers to seek the enhanced functions that clients need to do routine tasks (e.g. alarm handling and reporting). A first-cut use case specification for client-server process is shown in Figure 6.8:

1. Client looks up domain naming service
2. Client creates connection socket
3. Client connects to server
4. Server requests clients login data
5. Client sends login data to Server
6. Server authenticates client login data

Figure 6.8: Client connection use case

The state-based version of the use case in Figure 6.8 is shown in the following table:

ID	Primary Actor	Event	Precondition	Postcondition	Secondary Actor	Precondition	Postcondition
1	Client	looks up Domain Naming Service	initial	addressResolved	***Server***	waiting	addressResolved
2	Client	creates connection socket	dataAuthenticated	socketCreated			
3	Client	connects to server	socketCreated	connected			
4	Server	requests clients login data	addressResolved	dataRequested	***Client***	dnsResolved	dataRequested
5	Client	sends login data to Server	dataRequested	dataSent	***Server***	dataRequested	dataReceived
6	Server	Authenticates client login data	dataReceived	dataAuthenticated	***Client***	dataSent	dataAuthenticated

Table 6.6: Revised client-server use case

Revisions to the initial description:

Added events:

None

Added actors:

None

Moved (affected) events:

None

6.6.1.6 Explanation

The author and the participants did not find any amendments necessary for the above description. It was not clear whether this was because the client-server process was a new process being initiated or because the description was an accurate representation of the intended behaviour. A general point to be inferred from this is that, often, modellers may construct standard use case descriptions that depict the behaviour of a system accurately even without further scrutiny with the Educator approach. Hence, for well known systems, a standard use case description might suffice to showing the desired behaviour of the system.

6.6.1.7 Enacting the client-server use case- further dependency analysis

This section provides an analysis of the client-server connection use case by way of enaction to provide further evaluation of objective 1.

EducatorTool was used to provide enaction of the description. Hence the description of Table 6.5 edited within EducatorTool is shown below:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Client_Server connection

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	Client	looks Domain Naming Service	initial	addressResolved	Server	waiting	addressResolved
2	Client	activates connection socket	dataAuthenticated	socketActivated			
3	Client	connects to server	socketActivated	connected			
4	Server	requests client's login data	addressResolved	dataRequested	Client	addressResolved	dataRequested
5	Client	sends login data to server	dataRequested	dataSent	Server	dataRequested	dataReceived
6	Server	authenticates client data	dataReceived	dataAuthenticated	Client	dataSent	dataAuthenticated

Figure 6.9: Client Server connection use case

An enaction window indicating interaction between the client application and the server application during the first event shown below:

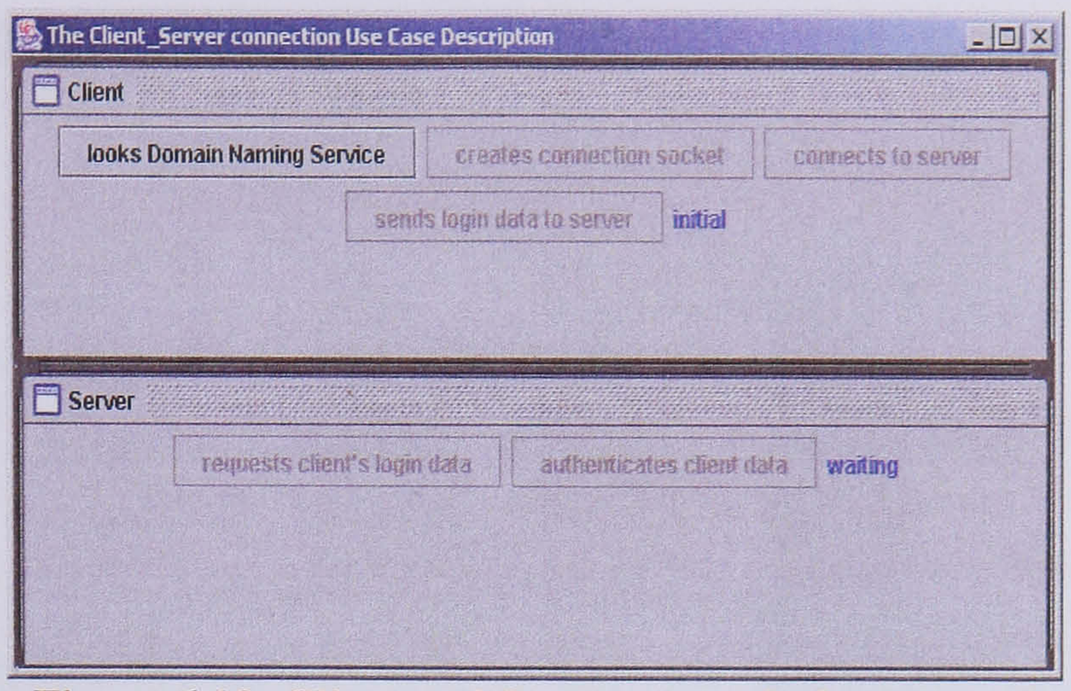


Figure 6.10: Client and Server states before event 1

The execution of the above event leaves the involved actors in the *addressResolved* state. Essentially, once the Domain Naming Service has been resolved, it implies that the Client application has identified the appropriate Server application to send or request data or other information from. Hence, the next available event is one whose pre-state is addressResolved (Server requests client's login data) and not the event in position 2. The next enaction dialogue is shown in Figure 6.11:

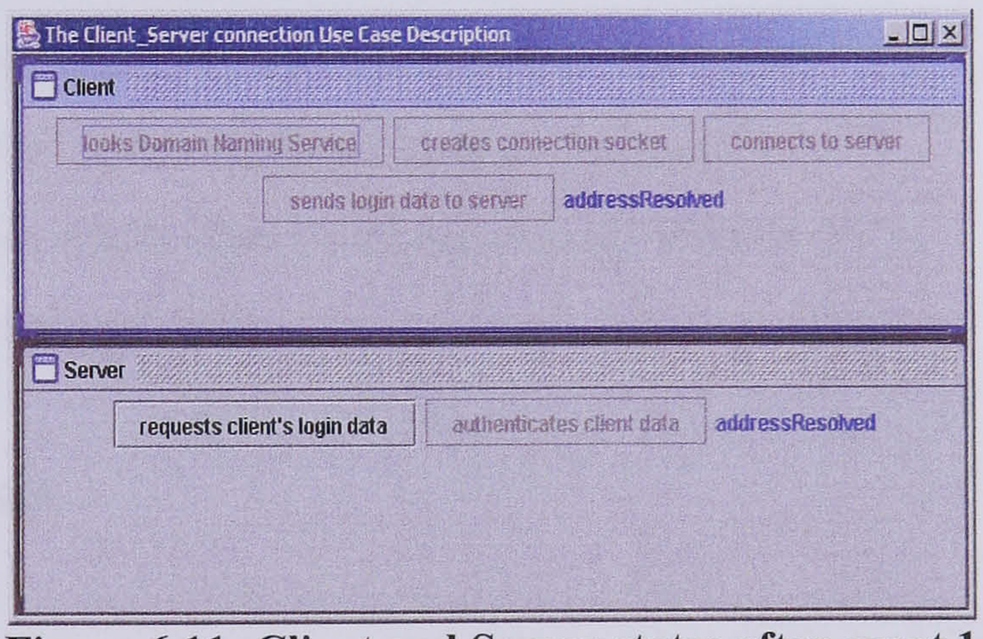


Figure 6.11: Client and Server states after event 1

During enaction, it was pointed out that in the current distributed infrastructure, Clients always have to look up domain naming service but don't always connect at the instant the address is resolved. In the proposed client-server connection infrastructure, however, clients must request connection to server applications explicitly. Moreover, the "middleware" for establishing inter-component communications (Scheduler) must initialise the client connection sockets to override any previous server data. Finally, the description was amended to the following:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description Client_Server connection

ID	Primary Actor	Event	Precondition	Postcondition	Secondary Act...	Precondition	Postcondition
1	Client	looks Domain Naming Service	initial	addressResolved	Server	waiting	addressResolved
2	Client	requests conenction	activated	connectionRequested	Scheduler	connectionRequested	connectionRequested
3	Scheduler	initializes the connection socket	connectionRequested	socketInitialized			
4	Client	activates connection socket	dataAuthenticated	activated	Scheduler	waiting	activated
5	Client	connects to server	atHandshake	connecting	Server	dataAuthenticated	connecting
6	Server	acknowledges connection	connecting	connected	Client	connecting	connected
7	Scheduler	registers network handler with Server	handlerCreated	handlerRegistered			
8	Scheduler	creates network handler for client	socketInitialized	handlerCreated			
9	Server	requests client's login data	addressResolved	dataRequested	Client	addressResolved	dataRequested
10	Scheduler	establishes client and server 'handshake'	handlerRegistered	atHandshake			
11	Client	sends login data to server	dataRequested	dataSent	Server	dataRequested	dataReceived
12	Server	authenticates client data	dataReceived	dataAuthenticated	Client	dataSent	dataAuthenticated
13	Client	executes tasks	connected	executingTasks			

Figure 6.12: revised client-server connection use case

Revisions to the description of Figure 6.9:

Seven added events:

Client requests connection

Scheduler initializes the connection socket

Client activates connection socket

Scheduler registers network handler with Server

Scheduler creates network handler for client

Scheduler establishes client and server 'handshake'

Client executes tasks

Added actors:

Scheduler.

Moved (affected) events:

From Figure 6.9, the following events moved from their current positions to new positions in Figure 6.12 as follows:

Figure 6.9: Figure 6.12:
Event 2 *moved to* ***position 4.***
Event 3 *moved to* ***position 5.***
Event 4 *moved to* ***position 9.***
Event 5 *moved to* ***position 11.***
Event 6 *moved to* ***position 12.***

Again, consideration of dependencies by augmenting use case events with pre and post states enabled the discovery of additionally actors and events. The inclusion of these extra elements in the initial use case description changes the way in which the execution of events occurs. Inevitably, such scrutiny was regarded significant as new information regarding important subsystems was revealed.

6.6.1.8 Explanation of results

In the Client-Server connection use case the role of Scheduler was not apparent when the description was augmented with pre and post states. For instance, the participants had not thought about the express necessity of the connection socket to be initialised each time a connection is requested by a client. Moreover, Scheduler was again highlighted as an important middleware for providing a "handshaking" between clients and servers. Hence, the visual enactable models produced with the EducatorTool provided cues within the participants regarding additional events (seven of them) and the additional actor (Scheduler). Whereas in the previous example, augmentation of the description alone provided insight into dependency issues, in this example augmentation alone did not offer any further clues into the nature of relationships between events. Rather, the enaction of the augmented description provided extensive and important insight into issues that were not determined in the default description, or the augmented description. The significance of combining augmentation with enaction within the Educator approach can be seen by observing the importance of some of the changes made to the descriptions. For instance, if a client does not perform event 2 (of Figure 6.12), the Scheduler will assume that a previous connection exists. If such a situation is not the case, then the client application will not find a server application. Where such a server application is an alarm receiver such as the one seen earlier, the implications are enormous. Another crucial change was the need for a Scheduler to create a connection handler for a client and register it for the specific server that the client wants to connect to. These events are crucial for offering a stable connection that does not experience deadlocks. For instance, a network handle for alarm handling should not be confused with that of data logging or vice versa. Finally, the correct sequence of events upon the consideration of dependencies is shown in Figure 6.13 below:

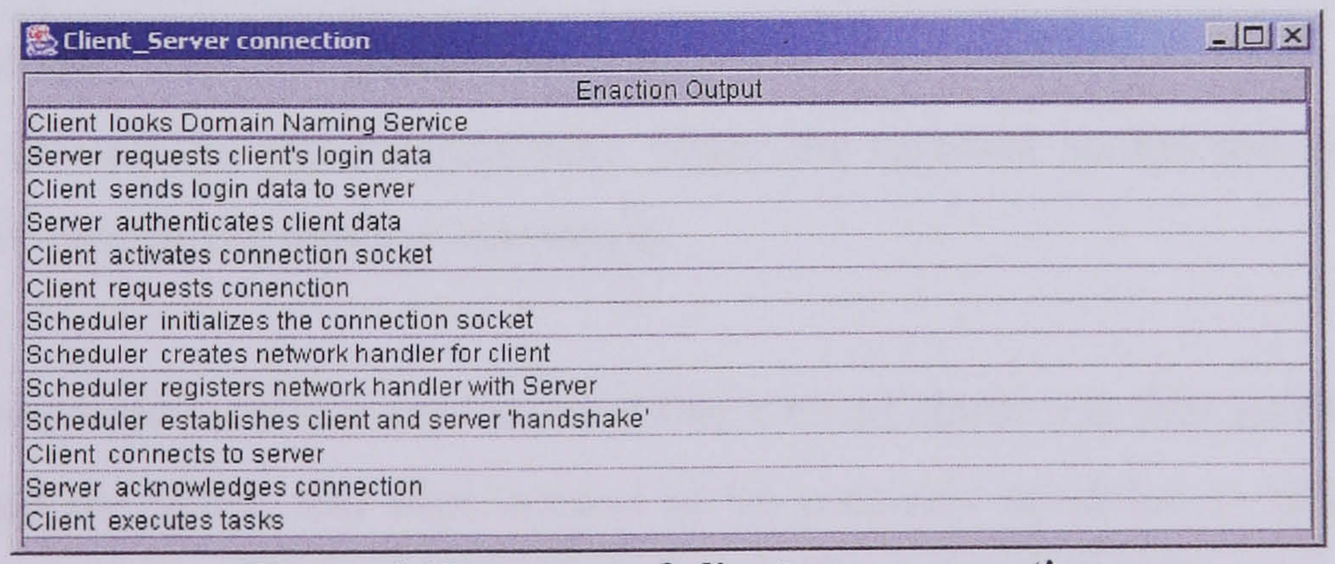


Figure 6.13: output of client-server enaction

Various quality aspects were assessed during the analysis of the client-server connection process. For instance, the use of the EducatorTool in editing descriptions is an indication that the supported authoring guidelines were automatically checked. Again, this aligns to the **syntactic**

quality aspect of the evaluation framework. Moreover, the information provided by participants, and associated discussions during the validation of descriptions provided an assessment of **semantic quality** since participants were able to provide insight about specifications based on their problem domain knowledge. Discussions arising from the revisions to the descriptions indicated that the participants felt the application of enaction provided a means to rigorously scrutinise specifications. This can be mapped to the **value quality** aspect in the evaluation framework.

6.6.2 Inter-use case dependencies

This section discusses examples that highlight the importance of considering inter-use case dependencies. Inter-use case dependencies clarify dependencies among distinct use cases of the same or different systems (or subsystems). One of the aspects of the industrial project was to describe the manner of interaction between two control-centre based systems. These systems are:

a) A control-centre alarm receiver.

b) A control centre collection service (CCCS).

The control centre alarm receiver has the same functions as the alarm receiver at customer sites (seen earlier in the analysis of intra-use case dependencies). An important difference between the two alarm receivers is that the one at the control centre creates a text file with the alarm data and saves the text file on a local computer disk. The use case description of the control-centre alarm receiver is shown in Figure 8 of Appendix C.

The CCCS on the other hand is a utility module for polling the text files containing alarm data, formatting the alarms into a standard view, and executing functions (typically stored procedures) to resolve the alarms.

NB: The stored procedures and the databases where the resolved alarms and alarm actions are stored is a subject outside the remit of this thesis.

The aspect of the control centre alarm receiver that created alarm text files and the aspect of the CCCS that polled the alarm text files needed to be precisely specified to delineate points of synchronisation between the two modules. Figure 9 of Appendix C shows the use case description for the CCCS module.

Schemes for inter-use case dependencies

There are two ways in which distinct use case descriptions can be associated to analyse dependencies amongst them. One of the ways of synchronising distinct use cases is to invoke one use case at a certain event of the related use case. Consider the CCCS and the Control Centre AlarmReceiver again. The CCCS's functionality can be invoked when the AlarmReceiving use case executes the "alarm text file creation" event according to one of the developers at the company. The RAD in Figure 6.18 shows the CCCS being invoked as described above:

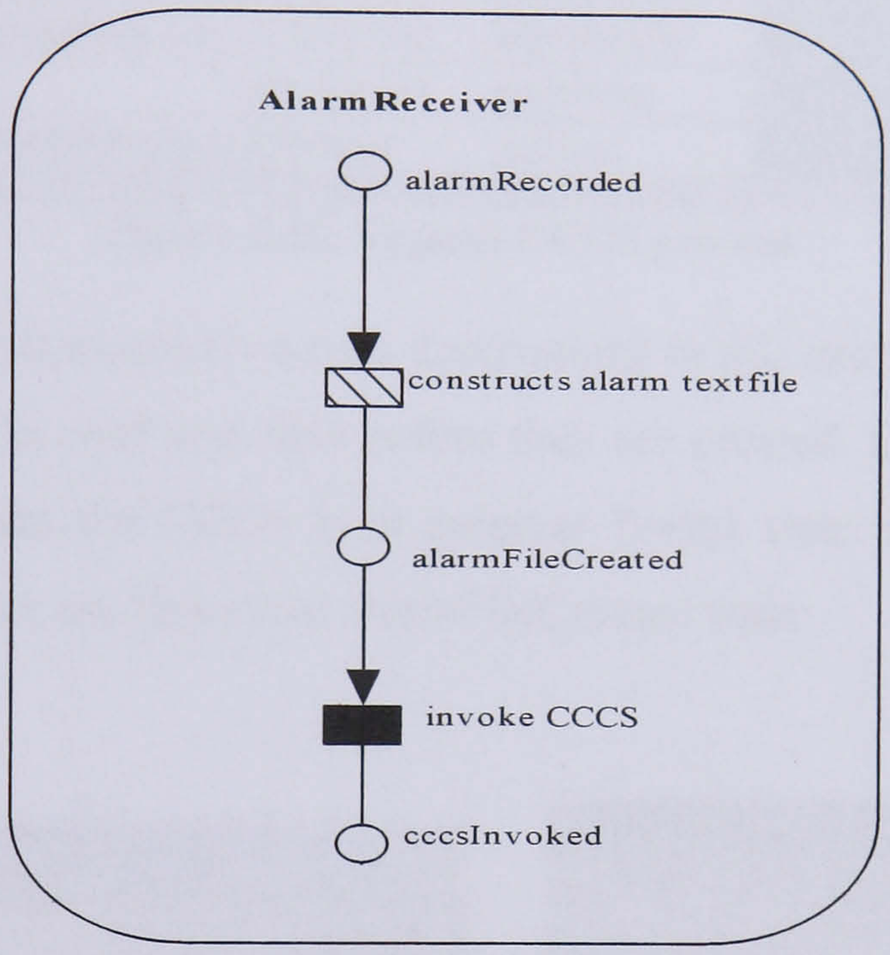


Figure 6.14: invoking CCCS within AlarmReceiving use case

In Figure 6.14, the invocation of the CCCS process cannot happen unless the alarm receiver has constructed the alarm text files. A second scheme of inter-use case dependency analysis is one where events in one active description are synchronised with those of another active description. In this scheme, an event in the CCCS use case description is able to execute if and only if another event in the AlarmReceiver has executed. In this case, the CCCS cannot "read text files" (event 2) before the AlarmReceiver has constructed the text files (even 8 of Control centre AlarmReceiver).

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Control Centre AlarmReceiver

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	Fixture	raises alarm at remote site	alarmOk	alarmActive	Controller	initial	alarmActive
2	Controller	obtains alarm data	alarmActive	dataObtained			
3	Controller	relays alarm data to Scheduler	dataObtained	dataSent	Scheduler	waiting	dataReceived
4	Scheduler	connects to AlarmReceiver	dataReceived	connected			
5	Scheduler	relays alarm data to AlarmReceiver	connected	alarmSent	AlarmReceiver	initial	alarmReceived
6	AlarmReceiver	returns alarm received ackowledgement	fileSaved	ackSent	Scheduler	alarmSent	ackReceived
7	AlarmReceiver	records alarm on database	alarmReceived	alarmRecorded			
8	AlarmReceiver	constructs alarm textfile	alarmRecorded	alarmFileCreated			
9	AlarmReceiver	saves textfile on local HDD	alarmFileCreated	fileSaved			
10	Scheduler	drops connection with AlarmReceiver	ackReceived	disconnected			

Figure 6.15: Control Centre AlarmReceiver use case

Consider the CCCS description:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Call Centre Collection Service

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	CCCS	polls alarm receivers	initial	receiverPolled			
2	CCCS	reads text files	receiverPolled	fileRead			
3	CCCS	calls standardisation procedure	fileRead	procedureCalled			
4	CCCS	produces recommendations to resolve alarm	procedureCalled	actionsRecomd	Operator	waiting	actionsRecomd
5	Operator	resolves alarm	actionsRecomd	almResolved			
6	CCCS	stores unresolved alarm in activeAlarms table	almResolved	almStored			

Figure 6.16: Typical CCCS process

An example of event-level associations across descriptions in this example is the requirement that CCCS should not attempt to read text files before they are created. During enaction, event 2 of CCCS can only occur when the CCCS is at receiver Polled state and the AlarmReceiver (in control centre alarm receiver use case) is at alarmFileCreated state:

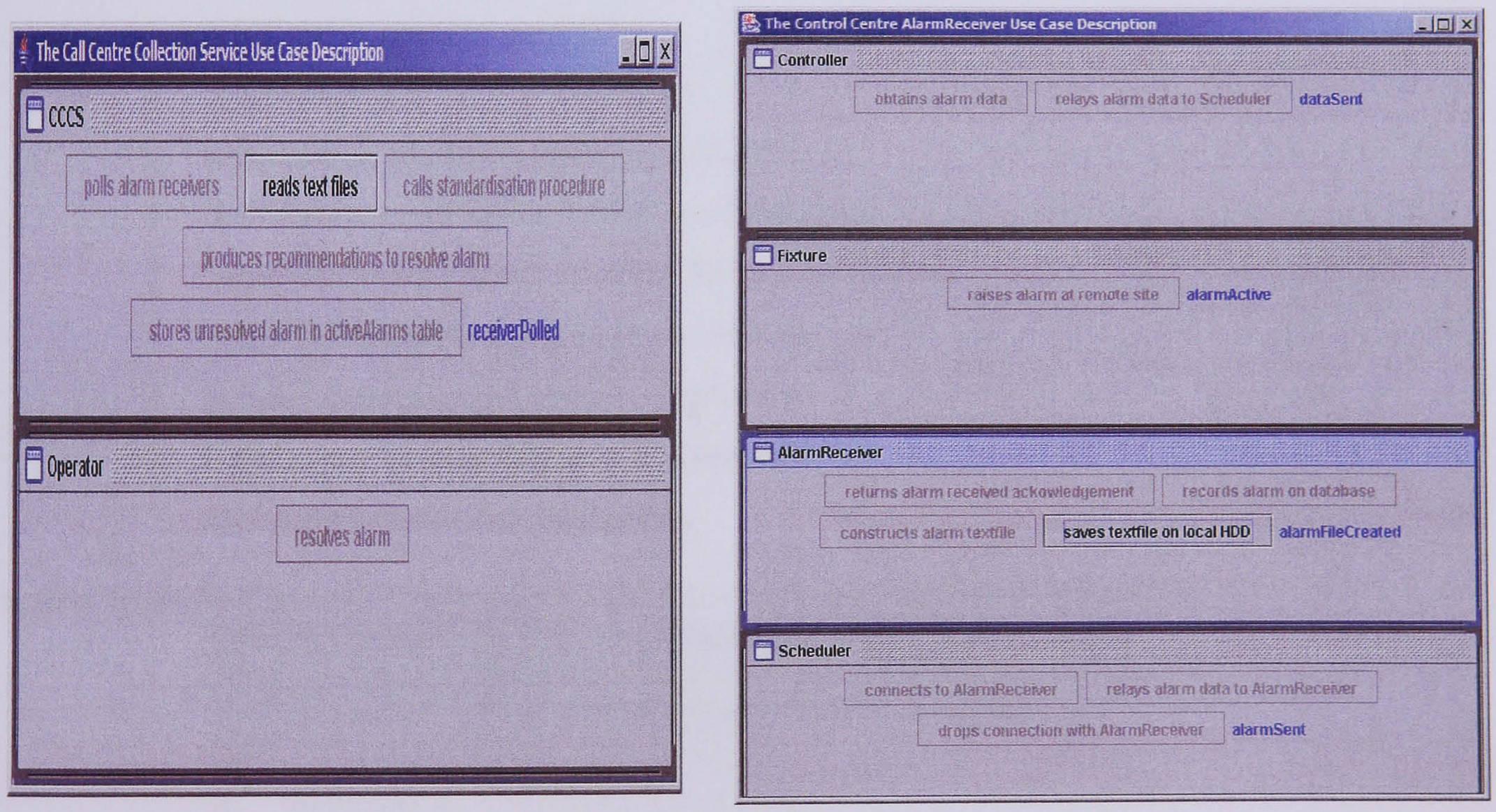


Figure 6.17: Synchronisation of inter-use case events

In the above descriptions (Figures 6.15 and 6.16), the enaction (shown in Figure 6.17) indicates that the second event (with precondition **receiverPolled**) in the CCCS description occurs only after the eighth event (with postcondition **alarmFileCreated**) in the Control Centre AlarmReceiver occurs. Hence, the

two processes run together, but only synchronise on those events since the processing of alarms at the control centre is guided by recommendations from the CCCS. However, the production of those recommendations is dependent on the alarm receiver constructing text files containing data regarding the alarms. Whether modellers enact their descriptions using any of the flavours offered within EducatorTool is purely dependent on the interactions envisaged between systems and also on the level of development of the specification.

6.7 Evaluating the EducatorTool

Editing descriptions

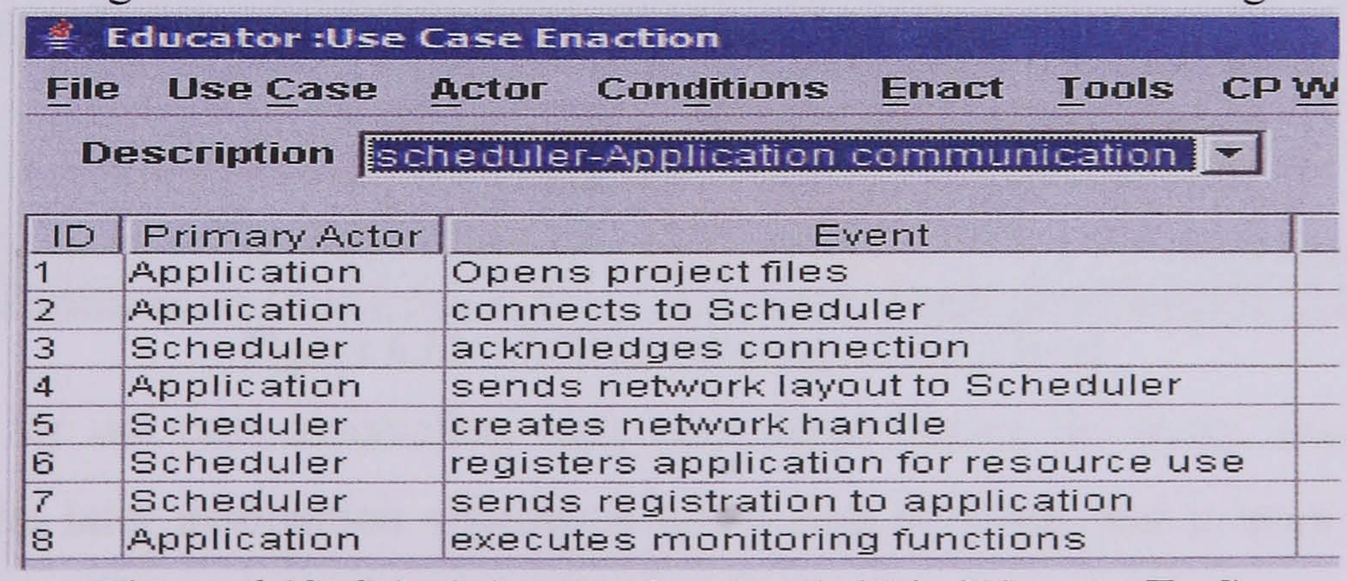
This section provides an example to address objective 2. The example outlined in this section was used as a means to demonstrate to the participating company staff how descriptions are edited with the tool.

The example comprises the Scheduler-Application process. Scheduler is a subsystem that was said to be central to the monitoring process in that it is a type of middleware that manages communication between subsystems, where these subsystems are generally termed "Application". The following standard use case description was constructed during interview:

1. Application opens project files
2. Application connects to Scheduler
3. Scheduler sends acknowledges connection
4. Application sends network layout to Scheduler
5. Scheduler creates network handle for the application
6. Scheduler registers application for resource use
7. Scheduler sends registration to application
8. Application executes monitoring functions

Figure 6.18: Scheduler-Application

The description of Figure 6.18 edited within the EducatorTool is shown in Figure 6.19.



ID	Primary Actor	Event
1	Application	Opens project files
2	Application	connects to Scheduler
3	Scheduler	acknoledges connection
4	Application	sends network layout to Scheduler
5	Scheduler	creates network handle
6	Scheduler	registers application for resource use
7	Scheduler	sends registration to application
8	Application	executes monitoring functions

Figure 6.19: Scheduler-Application (within EducatorTool)

Again, the aim of editing the description of Figure 6.18 in the tool (as shown in Figure 6.19) was to demonstrate the basic usage of the tool to author descriptions. In this editing, emphasis was laid on the <actor> <event> scheme discussed in chapter 3. At this initial stage, no further demonstrations were made regarding elements such as secondary actors, and states since these were going to be introduced during further analysis of the specifications.

Associating descriptions

In consideration of inter-use case dependencies, one way to associate distinct use cases is to consider an event in one use case where a related use case should be invoked. This is shown in Figure 6.18. The EducatorTool supports this functionality by providing a modeller with all the descriptions in to choose in constructing such associations. The Figure below shows a snapshot of EducatorTool usage in such an association:

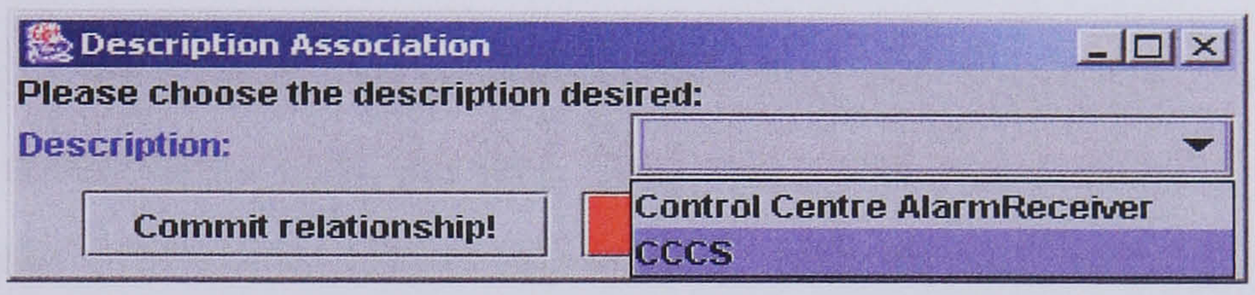


Figure 6.20: Relating CCCS to an event of AlarmReceiver

To do this, the tool offers the functionality to insert a description at an event in another description where its functionality is specifically needed (in the case of description in Figure 6.15, event 8). When the alarm receiving use case reaches event 8, then the CCCS (see Figure 6.16) is invoked automatically, so that it can access the alarm and recommend actions to resolve it.

Another way the EducatorTool supports associations among descriptions is event-level associations. For example, to perform the event level association mentioned in the paragraph below Figure 6.15, would highlight event 2 of CCCS and chose the AND (event 8) event from the Control Centre Alarm Receiver

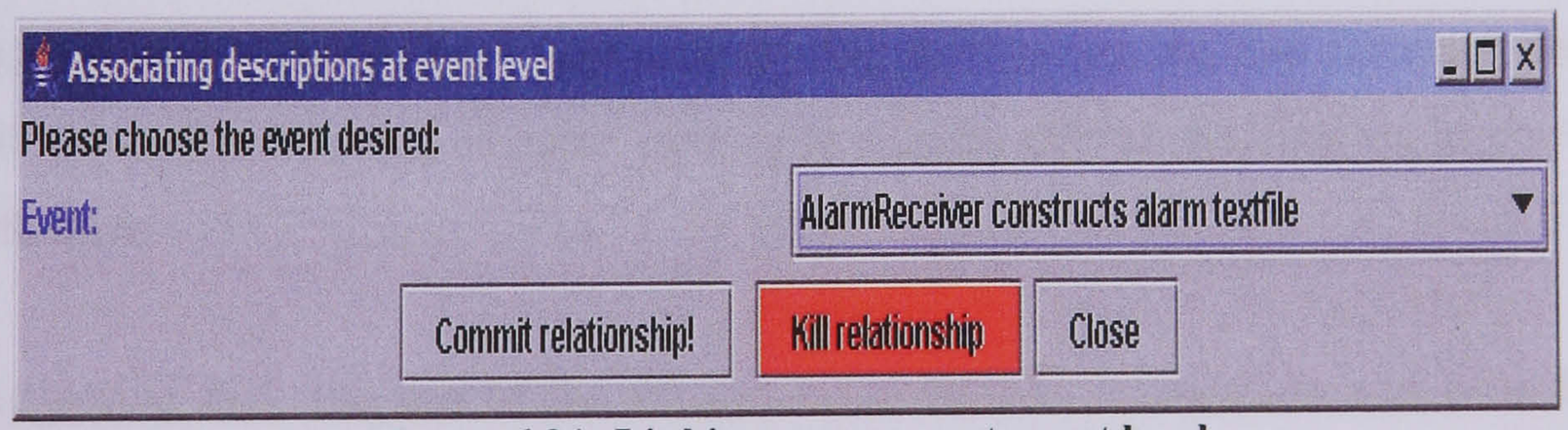


Figure 6.21: Linking use cases at event level

These schemes for use case association were considered during the study, and the developers indicated that the later association scheme is more elegant given that it does not require the invocation of an entire use case unnecessarily.

6.8 Review of objectives & evaluations

The application of the Educator approach in dependency analysis has been discussed throughout sections 6.6.1 (with examples based on 2 sub-systems) and 6.6.2. An assessment of the subsystems studied shows that the augmentation of use case events with state-based information provided insight about dependency issues that was not apparent within the standard use case specification for the subsystems. For instance, the analysis of the Alarm Receiver subsystem (see Figures 6.1 and Table 6.4) highlighted some dependency issues which are discussed in sections 6.6.1.1, 6.6.1.2 and 6.6.1.3. Enaction is considered in section 6.6.1.4 for this sub-system and again, several dependency issues are highlighted.

The various quality aspects of the Educator approach assessed during the study of this sub-system are as follows:

Syntactic quality – the syntactic correctness of the Educator descriptions was assessed at the point of editing descriptions with the tool. This was often not conducted explicitly, rather, checking of Educator descriptions against adherence to supported guidelines is inbuilt in the tool.

Semantic quality – discussions with participating staff indicated that analysis of descriptions via mere augmentation, and the application of enaction provided further clarity about the sub-systems to be able to consider such descriptions complete and valid specifications of the sub-systems.

Test quality – the tests regarded as having been "indirectly" conducted were those concerning the use of the tool to perform enaction and associations of descriptions. Such aspects of the tool were executed and discussions with participating staff indicated that such automated support was useful during validation and authoring of descriptions.

Value –the value associated with the model and the tool is the facilitation of rigorous scrutiny specifications. In most of the sub-systems studied, the application of the approach provided insight into aspects of the sub-systems that would possibly be missed without such rigour.

Pragmatic quality – discussions with participating staff suggested that Educator descriptions were well comprehended, and in most cases, participants altered descriptions to correct found issues with them.

For the objective two, the use of the EducatorTool to edit descriptions was considered well supported. For instance, the associations among use cases that are often made to consider inter-use case dependencies were deemed useful in validation of specifications.

6.9 Discussion of findings

Software engineers are often not experts in most application domains within which they are required to develop software. On the other hand, most domain experts are not experts in software engineering. Despite the communication gap that normally impedes informed elaboration of business needs, and the possibilities of their support with software, few approaches exist that lend themselves to bridging this gap without introducing costly learning overheads. Whereas the use case notation has been widely adopted due to its ease of use and intuitiveness, its lack of rigour and detail has been a stumbling block to wider application in non-trivial systems development projects. The study presented in this thesis is a reasonably complex test for the proposed approach where use cases are augmented with state-based information and supported with an automated environment for their authoring and animation. By enhancing the level of detail in use case descriptions and by providing a mechanism for stepping through the logic of the descriptions, the validation of use case specifications against stakeholder expectations can be done prior to any further development takes place.

An interesting outcome (and to some extent unexpected) of the study was that, some information that resulted from the enaction did not fit in either as an actor, event or state. For instance, information regarding the duration between a client's request of a server resource and the time taken by the server to respond cannot be represented with UML use cases. Developers however emphasized that such timing attributes must be described because a delay time of more than 10 seconds by the server is deemed a failed 'resource seek'. The author has since introduced a mechanism to write short notes to explain aspects that cannot be represented in an interaction step of a use case. Indeed, whereas timing requirements are critical to some domains (e.g., real time monitoring and control), use cases are not suited to describing such requirements. This thesis does not claim that state-based use cases would be suited to describing software requirements dealing with timing of executions for certain functions. The thesis demonstrates, however, that state-based use cases are better suited (compared to traditional use cases) to detailed description of software behaviour where dependency and interaction issues ought to be made explicit.

Additionally, developers were keen to ensure some subsystems were described so that they executed their tasks concurrently. An example given was that a new customer can be registered while alarm receiver and alarm display clients for existing customers continue their functions. Some subsystems also execute in a given sequential order regardless of their internal states. This type of sequential execution was mainly necessary to ensure an alarm receiver client only requested alarm resolution stored procedures after a reliable client-server connection had been

established. Some subsystems needed to execute every so often. Thus, it was important to include functionality for iterative enaction. The other mode of enaction is one where choice must be made between subsystems to execute. For example, customers can resolve alarms or in some circumstances, control engineers can resolve the alarms for the customer. There needs to be functionality to support the making of choice between specific processes. This combinatorial enaction scheme has since been incorporated in the support tool. Figure 6.21 shows a typical usage of the tool where different combinations of partial specifications can be made depending on functional demands of the users:

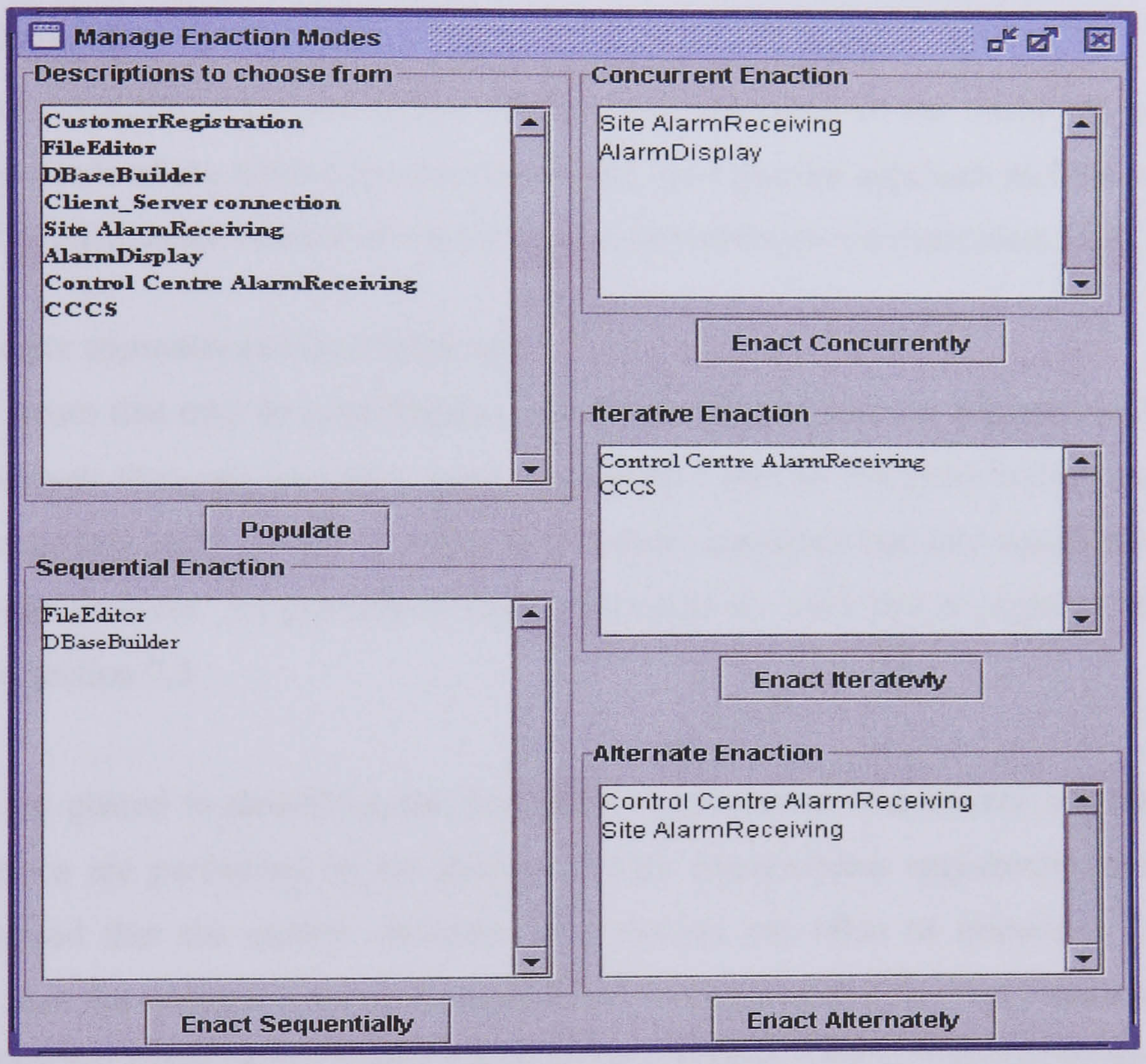


Figure 6.22: Modal (combinatorial) enaction

It is argued in [109] and [90] that application of state-based approaches in use cases (and scenarios) can tedious for large systems because there is bound to be states 'explosion'. Whereas this may indeed be an issue, this thesis observes that progressing software development on the basis of a flawed specification does not justify avoidance of a rigorous means for specification validation. Moreover, where larger systems are encountered, it may require creativity on the part of the modeller to break down the specifications into smaller parts that are specified and analysed individually. These distinct specifications can then be integrated using one of the means described within the Educator approach.

Most existing tools that are oriented toward behaviour modelling tend to describe the processes being modelled using formal grammars (e.g., process algebras for the LTSA [38] tool, and the PlayIn/PlayOut Engine [34]). Other tools such as L'Critoire [31] do not consider enaction at all. This study shows that by maintaining the simple nature of the use case notation, and allowing much of the appeal regarding process and tasks to be captured within the use cases, toolsets of this nature can gain industrial uptake.

6.10 Limitations

This section considers some limitations of the work presented in the thesis. In particular, a discussion is made of the limitations associated with the Educator approach and the support tool. Additionally, a discussion is made of the limitations pertaining to the evaluation study.

6.10.1 Educator approach and tool limitations

One of the issues that may be considered as a limitation of the Educator approach is the inability to associate more than one secondary actor to an event. That is, if a specification requires more than one secondary actor per given event, the Educator approach and tool would not be able to represent such an event. An example of such a circumstance (and how it might be addressed) is discussed in section 7.5.

Use cases are geared to describing the functional requirements of a system and not how well those functions are performed by the system. Within requirements engineering, however, it is well recognised that the quality attributes of a system are often as important (if not more important) than the systems functional capabilities ([141],[140], [123]). The Educator approach, however, does not focus on the quality attributes of a system. That is, integrating non-functional requirements with the functional requirements is not within the initial premise of the Educator approach.

Another limitation of the Educator approach is the lack of emphasis on non-functional requirements. It is to be noted however, that, whereas non-functional requirements may be quite critical in some domains, use cases are not geared to describing such requirements. Hence, there are no explicit Educator approach elements to describe such requirements. However, it is possible to write down notes regarding some aspects of a use case specification with the EducatorTool to ensure some of the quality-based information is not lost.

6.10.2 Research methodology

The essence of the industrial project described in previous parts of this chapter is to provide an evaluation of the Educator approach and tool. The analysis of the use cases provided indicates that the approach and the tool helped (in this particular project) highlight dependency issues. One of the limitations of the study is that it is not possible to determine which dependency issues could have been highlighted during later stages of development (or with further scrutiny) without application of Educator approach and tool. The author recognises that perhaps, some of the dependency issues might have come to light during design or implementation of the specifications. However, there is also a risk of such issues being missed (as they were missed in these cases), and it is such rigour in scrutiny that Educator approach is aimed at fostering so as to avoid the risk of missed information.

An interesting way to conduct the evaluation of the approach and tool would be to adopt a case study design where the author remained mainly as an observer. Multiple case studies may provide an opportunity to evaluate the approach and tool in multiple organisations, across multiple projects, thereby offering greater confidence in results. That is, such an approach might be useful in affording greater opportunity for multiple participants (from different companies) to apply the approach and tool in the validation of their specifications. It is hoped that multiple studies would provide wider insight into the way the Educator approach and tool helps in dependency analysis. Moreover, consideration of multiple studies with multiple participants might provide an opportunity to assess issues such as whether more significant effort is required to construct Educator-based use cases as opposed to standard use cases.

6.11 Chapter Summary

This chapter has presented a software specification project demonstrating the application of the Educator approach and associated tool within industry. The study highlights the rigour afforded by treating use case descriptions as light-weight state machines and the power of enaction in providing cognition cues during dependency analysis. Whereas many proponents of UML only recognise the need for pre/post states global to a use case, the study has shown that this simplistic state space is insufficient for any non-trivial problem.

This thesis does not impose conversion of large monolithic use cases into state machine models; however, the Educator approach is useful in situations that require precise specification of behaviour and its validation for system parts that are not well understood or must be precisely

specified. Hence, the Educator approach and tool can be used in conjunction with (or to supplement) other approaches and tools.

Much of UML use case literature seldom considers dependencies beyond the typical <<*extend*>> and <<*include*>> relationships [10, 55, 106]. The (often dangerous) presumption normally is that events follow a time-order dependency. This was the assumption by the developers and managers during the outline of all the descriptions presented in previous sections of this chapter (and in the appendices). This thesis asserts that for simple problems and uncomplicated domains, it is possible to outline the correct sequence of events in such use cases with confidence. Most control systems are however, complex, and the synchronisation of events in a process, or indeed, the synchronisation of processes is usually far from obvious [107].

The study shows that consideration of states for constituent events of a use case and the states relating distinct use cases offers a mechanism for the explicit consideration of dependency issues. Dependency issues on the other hand enable rigour in the analysis of the behaviour implied in use case descriptions. Moreover, enaction of the augmented use case descriptions provides visual models that enhance shared understanding amongst description authors. In particular, consideration of dependencies causes modellers to evaluate further the ramifications of proceeding with development with the current specification. For instance, the consequences of flawed alarm receiving process can cost lives, and extensive damage to property and such considerations must be made prior to subsequent design and implementation.

The validation of the specifications for the various subsystems with the Educator approach and tool indicates that additional specification elements were discovered. It is possible, however, that some of those elements (e.g., additional actors, events, and their reorganisation) might have been discovered in later stages of development. For instance in Figure 12, the additional events, and the added actor (Scheduler) might have been noticed to be missing during design or implementation of the subsystem. However, it is also likely that such information regarding a crucial communication module might be missed (as it was missed during the specification stage). The risk of such an anomaly is the lack of communication between applications that are involved in the monitoring of fixtures. Rather than take such a risk, the Educator approach and tool provide a means to rigorously scrutinise implied behaviour at a stage when the cost implications are not so high.

7. Conclusions

This thesis set out to address the inadequacy of the UML use case as a specification and validation notation; in particular, the inability of the use case to provide a mechanism by which dependency issues can be delineated. In addressing this problem, the thesis has provided an enhanced use case structure (Educator approach) whereby state-based information is used to augment the specification as a means to 'teasing out' dependency issues. An additional component of the work done to address the above inadequacy of use cases is the provision of an automated environment in which the Educator-based use cases are authored, and enacted as a means to early prototyping of the implied behaviour.

This thesis has presented various examples from literature and an industrial software specification project that serve to demonstrate the outlined problem with use cases. The analysis of the examples and the industrial study also serve to demonstrate the application and benefits of the proposed solution. In short, the work described in this thesis has addressed two related issues. The first issue is the inadequacy of the use case notation as a vehicle for software specification and validation. To address this issue, the thesis has proposed the Educator approach which enhances use case descriptions with state-based information as a means to considering dependency issues. The thesis argues that by supporting the consideration of dependency issues, modellers tend to reason more rigorously about the behaviour implied in the use case description. The other issue addressed in this thesis is the commonly held view (e.g., by [97], [142], [140]) that to provide automated support for animation of specifications a formally-based language is needed for constructing such specifications. This thesis argues that such an approach is not necessary during the early stages of requirements and specifications, especially because many participants are often unfamiliar with such formal languages. That is, for specification and validation purposes, this thesis seeks to obviate the need for such techniques as a means of ensuring non-technical participants are able to comprehend and hence scrutinise the specification.

Empirical methods of research have long been adopted within social sciences, and have also gained wide acceptance within the software engineering community (e.g., [143], [144], [145], [146]). The strength of empirical research is the evaluation of research results using direct (or, often indirect) observation or experience. This thesis has endeavoured to evaluate empirically the proposed approach and to demonstrate the efficacy of the proof of concept tool. Initial evaluation

was conducted in the form of seminar presentations to colleagues and student workshops. An industrial proof concept study was conducted to provide evaluation of the research within an industrial setting.

Whereas the study reported in chapter six provide interesting evaluation of the Educator approach and tool, there are a few weaknesses of the evaluation study. For instance, it is not possible to argue that the only way that dependency issues would be revealed in the studied systems is by applying the Educator approach. Perhaps, participants would discover such issues during later stages of development. Whether this is true or not, it is not possible to argue either way based on the study. What is clear though, is that, at the time of considering the subsystems, many important issues regarding the subsystems were missed out. The important thing was that further analysis with the use of the Educator approach helped discover some missing information.

7.1 Review of objectives

The first objective (see section 1.5, pg. 16) of the work reported in this thesis was to provide an approach for constructing use case specifications whereby dependency issues could be made explicit. This constitutes the Educator approach. In essence, the state-based information, by which constituent use case events are augmented, acts as control for event executions. The second objective (see section 1.5, pg.16) of this thesis was to provide automated support for the approach of objective one. That is, a proof of concept tool was deemed necessary to illustrate the novelty and benefits for the application of the proposed approach. This automated support is developed within the EducatorTool.

An additional objective (again, see section 1.5, pg.16) of this thesis was to consider an empirical means for evaluating the benefits of applying the enhanced use case approach in specification and validation, and the extent to which the supporting tool elaborates the essence of the approach. The evaluation was seen as a means of determining whether Educator-based use case descriptions are amenable to dependency analysis, and hence, validation of the specification, and whether (easy to use) tool support can provide enactable functionality to help tease out such dependency issues.

Chapters 4, 5 and 6 provide discussions on the approach including the evaluation of the supporting tool.

7.2 Summary of findings

The chapter outlining the industrial evaluation (chapter 6) discusses various real-time monitoring system functions that were investigated during the study. For instance, the default enaction of the

Scheduler-Application (section 6.7) communication subsystem provided insight about subtle and tacit issues that had not been highlighted by the manual inspection of the specification. The application of the Educator approach on the 'Alarm receiving' process (section 6.6.1, example 1) provided further information regarding events and actors that were crucial to the subsystem but had not been thought of in the initial specification. Enaction of the 'Alarm receiving' process using the EducatorTool provided further insight into dependency issues by highlighting the need for inclusion of further events and actors to the augmented use case specification. In short, the analysis of subsystems from the industrial project indicated that augmentation of use case events with state-based information provided insight about dependency issues that was not found in the standard use cases. Moreover, the application of enaction on the augmented use case specifications provided further scrutiny about dependency issues than merely inspecting the state-based use cases. There was an exception in one of the subsystems (Client-Sever connection process – section 6.6.1, example 2) studied regarding the augmentation of the use case specification with states. For this subsystem, augmentation alone did not reveal any additional dependency issues that required addition of events or actors. However, the enaction of the augmented use case specification revealed some dependency issues that were not revealed by augmentation alone. Indeed, this thesis proposes that Educator descriptions be further scrutinised by application of enaction using automated tools such as the EducatorTool.

This thesis has shown that use case specifications must not necessarily be written in a formal language, especially when the focus is the early phase of requirements and specifications. The reason for this is that developers (and many non-technical stakeholders) are often reluctant to devote the proportion of effort needed to learn such languages in order to use the underlying approaches and tools ([73], [140]). Moreover, this thesis shows that it is possible to provide automated support for specification purposes (e.g., Educator approach) without delving into any formal specification languages.

It has been mentioned elsewhere in this thesis that the core theme of the use case specification is to describe software behaviour in terms user-system interactions. It is surprising to find, that despite this novel underpinning theme, there is little focus on the rigour and scrutiny of such interactions in terms of the consequences for performing each constituent step in relation to the others. These issues have long been investigated within process modelling (e.g., [47], [49]) and also within process algebras (e.g., [45], [147]). These approaches ([47], [49]), and [45], [147]) have not been applied primarily to requirements and specifications to address some of the

weaknesses mentioned already within the UML use cases, and this is indeed a surprise. This thesis has applied concepts from business process modelling to use case specifications in order to address dependency and interaction issues within use case specifications. Hence, the key finding from this work is that application of state-based approach (borrowed from business process modelling) and the automated production of visual prototypes helps development participants in clarifying dependency issues during specification and validation. Additionally, whereas the initial focus of the work was to investigate means for clarifying intra-use case and inter-use case dependencies, the work during the industrial study provided an opportunity to consider other ways in which systems interact. For instance, it was found out that consideration and provision needed to be provided (see section 6.6) to clarify sequential, concurrent, iterative, and choice-based interactions amongst systems. An additional finding is that production of such state-based specifications does not need to adopt a formally based technique like is common with much of process modelling approaches.

7.3 Thesis contributions

The contributions of this thesis are outlined as follows. The conduct literature survey (see chapter 2) has highlighted some of the inadequacies of use cases as a specification and validation notation. In particular, validation of use case specifications, and especially, clarification of dependencies is hardly supported by the UML specification of the use case or the approaches described in section 2.4. Consideration of process modelling approaches and the use of process algebras have provided insight about the way in which use cases can be enhanced to facilitate elaboration of dependency issues. The identification of these issues with use cases, and the way in which they can be addressed, is considered (by the author) as one of the contributions of this thesis.

To address dependency and interaction issues, the notion of pre and post states for use case events is introduced. The state-based information is used to augment constituent events of a use case to facilitate the clarification of both intra use case and inter use case dependencies. One of the key factors to dependency analysis is the consideration of interactions among actors when performing constituent steps of the use case. This thesis addresses interaction issues by allowing explicit addition of a secondary actor (with its pre and post states) for an event. Hence, the main contribution of this thesis is the provision of an enhanced structure for authoring state-based use case descriptions whereby dependency and interaction issues can be analysed. This state-based approach is the Educator approach (discussed in chapter 3). The Educator approach is supported by a proof of concept tool. The tool provides enactable functionality for producing visual

prototypes of the specification. The enaction of the Educator-based specifications is considered an important contribution of the thesis since visual models have long been accepted as a means to enhancing shared understanding among development participants.

The concept behind the Educator approach is informed by business process modelling approaches where some authors (e.g., [47], [51]) constructed business process models using a formal specification language. The resulting process descriptions are then enacted to produce visual models that business partners can scrutinise for validation purposes. A key focus of most of these business process modelling approaches is the production of visual (also see [48] and [49]), enactable models that participants can use as prototypes for clarifying implied behaviour. Unlike most UML notations (e.g., use cases), these process modelling approaches have not gained wide adoption within industry, and whereas other reasons might contribute to this, their inclination toward the use of formal techniques is again a drawback to their uptake.

To obviate the need for such learning effort, the Educator approach allows for the state-based modelling of use case descriptions in which the constituent steps of the use case are augmented with named pre and post states. Again, the essence of this approach is to force the rigorous scrutiny of intra-use case and inter-use case dependencies. Further to the state-based approach, the investigation of use case specifications and the outlined validation efforts has led to several considerations. For instance, this thesis has gone beyond the standard <<include>> and <<extend>> relationships to offer a mechanism for considering various forms of associations among use case descriptions. In this regard, the Educator approach and tool provide a means for synchronising descriptions at use case level and also at event level. Moreover, there is provision of a mechanism to perform combinatorial validation of descriptions based on the different ways in which subsystems in the real world might execute in relation to each other. The sets of combinations include selection of a description from several existing ones, iterative enaction, concurrent enaction, etc. Again, chapters 4, 5 and 6 provide examples where intra and inter-use case dependency analysis is performed based on event-level relationships within the same use case, event-level associations across different use cases, or indeed associations amongst use cases based on their need to execute in some sequential order, concurrent order, or even through iterations.

In summary, the major innovations are described as follows. The idea of constructing light-weight state models is pivotal to enhancing the structure of the use case description to be

amenable to dependency analysis. This constitutes the Educator approach. Moreover, the use of enaction is vital in providing visual models for scrutinising use case descriptions during specification and validation. It is considered (by the author) that the conduct of an industrial project as a proof of concept for the approach and tool is a significant part of the contribution of this thesis. Moreover, the EducatorTool provides support for some of the CP rules (see section 4.4) recommended (see [37] and [114]) for authoring of use case descriptions.

7.4 Conclusion

Whereas behaviour modelling has proved to be successful in helping 'teasing' out and correct flaws in business processes (e.g., [51] and [49]), it has not had similar success in software specification and validation. The two main reasons for this are as follows. First, constructing models for behavioural analysis remains a difficult undertaking requiring considerable expertise. Second, the validation benefits appear at the end of the (often lengthy) construction effort, and users do not devote time to learning the specification languages involved. The Educator approach is a compromise attempt that proposes the use of state-based information for rigorous specification, and enhances communication during enaction of the specification. Given the importance of behaviour validation, and the potential benefits of enaction, the adoption of such an approach and tool support promises many benefits. Software developers have traditionally been reluctant to devote proportionate effort to requirements activities, and this thesis contends that semi-formal approaches of this nature increase the likelihood for industrial uptake.

In its entirety, this thesis has demonstrated the essence of considering dependency issues during software specification and validation. The thesis has shown the extent to which the use of state-based information, and the enaction of the resulting specification can help in "teasing out" dependency and interaction issues in requirements and specifications. This was done by considering applying the Educator approach and enaction to use case specifications. In many cases, augmentation of use case specifications alone provided insight about dependency issues that had not been within the standard use cases. The application of enaction in the subsystems studied provided further insight into dependencies. Analysis in itself comprised the consideration of the additional events and actors (including the reordered events) that resulted from the application of the Educator approach and tool. Hence, the Educator approach has been proposed as one way of facilitating the consideration of dependency issues, and the EducatorTool has been discussed as a mechanism by which Educator-based specifications can be authored and enacted by participants during the scrutiny of the implied behaviour. Finally, the thesis has shown that it

is (technically) unnecessary to use intermediate formal grammars for specification authoring as is the case with many state-machine oriented approaches.

7.5 Further work

7.5.1 Educator approach

The Educator approach is currently based on a dual actor interaction per use case event. Cockburn [33] and Douglas [67] recognise the importance of considering both primary and secondary actors in use cases. Many other researchers (e.g., [56], [109], and [110]) however, do recognise the need to indicate the secondary actors in use case events.

The Educator approach adopts an event occurrence style where each event is triggered by a primary actor, and the secondary actor is an optional user who can participate in the event of interest. Whereas reasonably sized and complex systems have been investigated using the Educator approach, it is possible that an event can require more than two interacting actors. In such a case, Educator would be limited, and an extension of the approach may be necessary. As an example of where multiple secondary actors may be needed to describe an interaction is an automated response system for a library. Consider a situation where the system is used to send reminders to borrowers when items are due. Hence, the system would send a reminder to the borrower, who would respond by returning the due item to the library, where a member of staff would collect the item from the borrower. The use case event triggered by the library response system can be written as:

1. **System** sends reminder to borrower *initial reminderSent*
   Secondary actor 1: **Borrower** *initial reminderReceived*
   Secondary actor2: **Staff** *atWork itemRecieived*

In such a case, the automated system for sending reminders is the primary actor. Where such specifications are necessary, participants ought to decide how they organise the multiple secondary actors (perhaps as written above). However, this author observes that in many cases where multiple secondary actors per event are encountered, it is likely that a further event(s) is missing where one of the secondary actors is the primary actor. Consider the example described above. It is possible to re-write it as follows:

1. **System** sends reminder to borrower *initial reminderSent*
   Secondary actor: **Borrower** *initial reminderReceived*
2. Borrower returns item to the library *reminderReceived itemReturned*

Secondary actor: **Staff** *atWork itemReceived*

Decomposing the description further may enhance clarity regarding the process being described. But again, where it is crucial that multiple secondary actors are needed, the approach would require artistic deployment to incorporate further secondary actors. This is an aspect of the Educator approach that needs further work.

7.5.2 EducatorTool

One of the ideas that need to be considered for support with the EducatorTool is the issue supporting multiple secondary actors per event in a use case description (see section 7.5.1). Such an extension of the tool is regarded trivial, as extending the existing application classes is indeed likely to a straightforward implementation issue. It would be worthwhile of course considering matters pertaining to real-estate; that is, how use case events with multiple secondary actors are to be written to fit in a graphical interface without hiding important information.

An additional issue that arose during early demonstrations of the EducatorTool was the construction of state diagram equivalents of the state-based textual specification. That is, providing a means to depict graph-like state-transitions during enaction. Much more can be done to provide different ways of displaying actors and their respective events, and state changes. For instance, it is possible that use of graphical icons, or some form of multimedia facility during enaction can provide better means of viewing behavioural models of the use cases. However, this thesis set out to address dependency and interaction issues, and to show that a simple enactable tool can provide the benefit of visual scrutiny and clarification of implied behaviour. Hence, there is scope for extending the tool, and these possible enhancements will be considered for further work.

A further issue that may arise in an environment with multiple users is the need to maintain and track, and merge versions of descriptions. That is, given that the Educator approach allows for description association (in various ways) for dependency analysis, users may alter such associations, or states, and actors thereby changing the initial associations applied by other users of the descriptions. Hence, further work is needed to determine means for version control of Educator-based specifications in a multi-user environment.

References

1. Potts, C. *ScenIC: A Strategy for Inquiry-Driven Requirements Determination.* in *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*. 1999. Los Alamitos, CA: IEEE Computer Society press.
2. Ryser, J. and M. Glinz. *A Practical Approach to Validating and Testing Software Systems Using Scenarios*. in *Quality Week Europe*. 1999. Brussels: Software Research Institute.
3. Laney, R., et al. *Composing Requirements Using Problem Frames*. in *2th IEEE International Requirements Engineering Conference*. 2004. Kyoto, Japan.
4. Rapanotti, L., et al. *Architecture Driven Problem Decomposition*. in *12th IEEE International Requirements Engineering Conference*. 2004. Kyoto, Japan.
5. Loucopoulos, P. *Systems Co-development through Requirements Specification*. in *13th International Conference on Information Systems Development: Advances in Theory, Practice and Education*. 2004. Vilnius, Lithuania.
6. Loucopoulos, P. *Systems Co-development through Requirements Specification*. in *13th International Conference on Information Systems Development:Advances in Theory, Practice and Education*. 2004. September.
7. Bray, I., *An Introduction to Requirements Engineering*. 2002, Harlow, UK: Pearson Education Limited.
8. Maiden, N. and A. Sutcliffe. *Domain Abstractions in Requirements Engineering: an exemplar approach?* in *Proceedings 7th Knowledge-Based Software Engineering Conference*. 1992: IEEE Computer Society Press.
9. Jackson, M., *Software Requirements & Specifications:a lexicon of practice, principles and prejudices*, ed. ACM. 1995: Addison-Wesley.
10. Jackson, M., *The meaning of requirements.* Annals of Software Engineering, 1997. 3.
11. Jackson, M., *Problem Frames: Analyzing and structuring software development problems*. 2001: Addison-Wesley.
12. Maciaszek, L.A., *Requirements Analysis and System Design*. 2001: Addison-Wesley.
13. Sutcliffe, A., *Scenario-Based Requirements Analysis.* Requirements Engineering Journal, 1998. 3: p. 48-65.
14. Deraitus, M., *Customer Requirements and User Requirements: Why the Discrepancies?* IEEE Software, 2002.
15. Li Xiaoshan, et al., *Generating a Prototype From UML Model of System Requirements*. 2004, United Nations University.
16. Loucopoulos, P., et al. *Project Failures: Continuing Challenges for Sustainable Information Systems*. in *proceedings of 6th International Conference on Enterprise Information Systems*. 2004. Porto, Portugal.
17. Finkelstein, A. and J. Dowell, *A Comedy of Errors: the London Ambulance Service case study.* 8th International Workshop on Software Specification & Design IWSSD-8: IEEE CS Press, 1996: p. 2 -- 4.
18. Hunter, A. and B. Nuseibeh, *Managing Inconsistent Specifications:Reasoning, Analysis, and Action.* ACM Transactions on Software Engineering and Methodology, 1998. 7(4).
19. ITCortex, *Failure Examples of IT projects (http://www.it-cortex.com/Examples_f.htm)*. 2005.
20. Oates, J., *NHS IT: Over budget, overdue and unpopular (http://www.theregister.co.uk/2006/05/30/nhs_it_hated/)*, T. Register, Editor. 2006.
21. Nuseibeh, B., A. Finkelstein, and J. Kramer. *ViewPoints: meaningful relationships are difficult*. in *International Conference on Software Engineering*. 2003. Portland, Oregon.
22. Jacobson, I., ed. *The Use Case construct in object-oriented software engineering*. Scenario-based design: envisioning work and technology in system development, ed. J.M. Carroll. 1995, Wiley: New York.
23. Coleman, D., et al., *Object Oriented Development: The Fusion Method.* 1993, Engelwood Cliffs, NJ, USA.: Prentice-Hall.
24. Rumbaugh, J., I.Jacobson, and G. Booch, *The Unified Modelling Language Reference manual.* 1998: Addison-Wesley.
25. OMG, *Unified Modelling Language Specification version 2.0*. 2002, OMG-UML; http://www.uml.org/.

26. RationalCorporation, *The Use Case model (http://www.rational.com)*. 2003.
27. Jacobson, I., et al., *Object Oriented Software Engineering: A Use Case Driven Approach*. 1992: Addison-Wesley.
28. Scheneider, G. and J.P. Winters, *Applying Use Cases: A Practical Guide*. 1998, Reading: Addison-Wesley.
29. Jacobson, I., J. Rumbaugh, and G. Booch, *The Unified Software Development Process*. 1999, Harlow: Addison Wesley.
30. Fowler, M. and K. Scott, *UML Distilled*. 2nd edition ed. 2000: Addison-Wesley.
31. Rolland, C., *L'E-Lyee: Coupling L'Ecritoire and LyeeALL*. Information and software Technology journal, 2002.
32. Rolland, C. and B. Achour, *Guiding The Construction of Textual Use Case Specifications*. 1998, CREWS Report Series.
33. Cockburn, A., *Writing effective Use cases*. 2001: Addison-Wesley.
34. Harel, D. and R. Marelly, *Capturing and Executing Behavioral Requirements: The Play-In/Play-Out Approach*. 2001, The Weizmann Institute of Science: Rehovot, Israel.
35. Kulak, D. and E. Guiney, *Use Cases: Requirements in Context*. 2000: Addison-Wesley.
36. Grieskamp, W. and M. Lepper. *Using Use Cases in Executable Z*. in *3rd IEEE International Conference on Formal Engineering Methods*. 2000. York, England.
37. Cox, K., *Heuristics for Use Case Descriptions*, in *Design, Engineering & Computing*. 2002, Bournemouth University, UK.
38. Uchitel, S., J.Kramer, and J. Magee, *Synthesis of Behavioural Models from Scenarios*. IEEE Transactions on Software Engineering, 2003. 29(2).
39. Holloway, M. and R.W. Butler, *Impediments to Industrial Use of Formal Methods*. IEEE Computer, 1996. 29(4).
40. Parnas, D.L., *Mathematical Methods: What We Need and Don't Need*. IEEE Computer, 1996. 29(4).
41. Leveson, N.G., et al., *Requirements Specification for Process-Control Systems*. IEEE Transactions on Software Engineering, 1994. 20(9).
42. Kavakli, E. and P. Loucopoulos. *Goal Driven Requirements Engineering: Evaluation of Current Methods*. in *Eighth International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*. 2003. Klagenfurt/ Velden, Austria.
43. Loucopoulos, P. and N. Prekas. *A Framework for Requirements Engineering Using System Dynamics*. in *21st International Conference of the System Dynamics Society*. 2003. New York City.
44. Hnatkowska, B. and Z. Huzar, *Transformation of dynamic aspects of UML models into LOTOS behaviour expressions*. International Journal of Applied Mathematics and Computer Science, 2001. 11(2): p. 537-556.
45. Hoare, C.A.R., *Communicating Sequential Processes*. 2004: Prentice Hall.
46. Ould, M., *Business Processes: Modelling and Analysis for Reengineering and Improvement*. 1995, Chichester: Wiley.
47. Abeysinghe, G., et al., *RolEnact: Enactable Models of Business Processes*. Information and software Technology journal, 1998. 40(3).
48. Walters, R., *Checking of models built using a graphically based formal modelling language*. Journal of Systems and Software, 2005. 76(1).
49. Walters, R.J. *A Graphically Based Language for Constructing, Executing and Analysing Models of Software Systems*. in *26th Annual International Computer Software and Applications Conference*. 2002. Oxford, England.
50. Heimdahl, M.P. and N. Leveson, *Completeness and Consistency Analysis of State-Based Requirements*. IEEE Transactions on Software Engineering, 1996. 22(6): p. 363–377.
51. Phalp, K., et al., *RolEnact: role-based enactable models of business processes*. Information and software Technology journal, 1998.
52. Zimmerman, M.K., K. Lundqvist, and N.G. Leveson. *Investigating the readability of state-based formal requirements specification languages*. in *International conference in software engineering*. 2002. Florida, USA.
53. Homrighausen, A., H. Six, and M. Winter, *Round-Trip Prototyping Based on Integrated Functional and User Interface Requirements Specifications*. Requirements Engineering, 2002. 7.

54. Mannio, M. and U. Nikula. *Requirements Elicitation Using a Combination of Prototypes and Scenarios*. in *The 4th Workshop on Requirements Engineering*. 2001. Buenos Aires, Argentina.
55. Finkelstein, A. and A. Savigni. *A Framework for Requirements Engineering for Context-Aware Services*. in *First International Workshop From Software Requirements to Architectures; 23rd International Conference on Software Engineering*. 2001.
56. Sommerville, I., *Requirements Engineering, A Good Practice Guide*. 1997: Addison Wesley.
57. Finkelstein, A. and W. Emmerich, *The Future of Requirements Management Tools*, in *Information Systems in Public Administration and Law*, G. Quirchmayr, R. Wagner, and M. Wimmer, Editors. 2000, Oesterreichische Computer Gesellschaft.
58. Sutcliffe, A. and N. Maiden. *Use of Domain Knowledge for Requirements Validation*. in *Proceedings of IFIP WG 8.1 Conference on Information System Development Process*. 1993: Elsevier Science Publishers.
59. Spanoudakis, G., A. Finkelstein, and D. Till, *Overlaps in Requirements Engineering.* Automated Software Engineering, 1999.
60. Hunter, A., *Reasoning with conflicting information using quasi-classical logic.* Journal of Logic and Computation, 2000. 10(5): p. 677-703.
61. Kanyaru, J. and K. Phalp. *Supporting the Consideration of Dependencies in Use Case Specifications*. in *Requirements Engineering: Foundations for Software Quality (REFSQ'05)*. 2005. Porto, Portugal.
62. Zave, P., *Classification of Research Efforts in Requirements Engineering.* ACM Computing Surveys, 1997. Vol. 29(4): p. 315-321.
63. Nuseibeh, B. and S. Easterbrook. *Requirements Engineering: A Roadmap*. in *Proceedings of International Conference on Software Engineering (ICSE-2000)*. 2000. Limerick, Ireland: ACM Press.
64. Sutcliffe, A., J.Galliers, and S.Monica. *Human Errors and System Requirements*. in *Proceedings of 4th IEEE International Symposium on Requirements Engineering*. 1999. Los Alamitos, CA: IEEE Computer Society Press.
65. Sutcliffe, A. and S. Minocha. *Analysing Socio-Technical System Requirements*. in *4th International Symposium on Requirements Engineering*. 1999. University of Limerick, Ireland.
66. Hunter, A. and S. Parsons, *Introduction to uncertainty formalisms*, in *Applications of Uncertainty Formalisms: Lecture Notes in Computer Science*. 1998, Springer.
67. Hunter, A. and B. Nuseibeh. *Analysing inconsistent specifications*. in *Third IEEE International Symposium on Requirements Engineering*. 1997: IEEE Computer Society Press.
68. Sommerville, I., *Software engineering*. 6th edition ed. 2001: Addison-Wesley.
69. Wieringa, R.J., *Requirements engineering: frameworks for understanding*. 1995, Chichester: Wiley.
70. Tveito, A. and P. Hasvold, *Requirements in the Medical Domain: Experiences and Prescriptions.* IEEE Software, 2002. 19.
71. Graham, I., *Requirements engineering and rapid development: an object-oriented approach*. 1998, Harlow: Addison-Wesley.
72. Liu, S. and R.Adams. *Limitations of formal methods and an approach to improvement*. in *Proceedings of 1995 Asia Pacific Software Engineering Conference*. 1995. Brisbane, Australia.
73. Henderson, P. and R.J. Walters. *System Design Validation Using Formal Models*. in *Tenth IEEE International Workshop on Rapid System Prototyping (RSP99)*. 1999. Florida.
74. Letelier , P., P. Sánchez, and I. Ramos, *Prototyping a requirements specification through an automatically generated concurrent logic program*, in *Practical Aspects of Declarative Languages, Lecture Notes in Computer Science*, G. Gupta, Editor. 1998, Springer-Verlag.
75. Sánchez, P., P. Letelier, and I. Ramos. *Validation of Conceptual Models by Animation in an Scenario-based Approach*. in *ACM Conference on Object-Oriented Programming, Systems,Languages, and Applications: Workshop on scenario-based round-trip engineering*. 2000. Minneapolis, Minnesota, USA.
76. Urban, J.E., *Software Prototyping and Requirements Engineering*. 1992, Data & Analysis Center for Software: Rome.
77. Pfleeger, S., *Software engineering: theory and practice*. 2nd ed. 2005: Prentice Hall.
78. Pressman, R., *Software engineering: a practitioner's approach*. 5th edition ed. 2000: McGraw-Hill.

79. Andriole, S.J., *Fast, cheap requirements prototype, or else!* IEE Software, 1994. 11(2).
80. Homrighausen, A., H.Six, and M. Winter. *Round-Trip Prototyping for the Validation of Requirements Specifications*. in *Requirements Engineering: Foundations for Software Quality*. 2001. Interlaken, Switzerland.
81. Kanyaru, J. and K. Phalp. *Requirements validation with enactable models of state-based use cases*. in *Empirical Assessment in Software Engineering conference (EASE'05)*. 2005. Keele University, UK.
82. Bruegge, B. and A.H. Dutoit, *Object-Oriented Software Engineering*. 2000: Prentice Hall.
83. Regnell, B., *Requirements Engineering with Use Cases- A Basis for Software Development*, in *Department of Commnunication Systems, Lund Institute of Technology*. 1999, Lund University, Sweden.
84. Some, S., *Supporting use case based requirements engineering.* Information and software Technology journal, 2006. 48(1).
85. Sutcliffe, A., et al., *Supporting Scenario-based Requirements Engineering.* IEEE Transactions on Software Engineering, 1998. 24(12): p. 1072-1088.
86. Dennis, A., *Systems analysis and design: an object-oriented approach with UML*. 2002, Chichester: Wiley.
87. Glinz, M. *Improving the Quality of Requirements with Scenarios*. in *Second World Congress on Software Quality*. 2000. Yokohama.
88. Jarke, M., T. Bui, and J.C. J, *Scenario Management; An Interdisciplinary Approach.* Requirements Engineering Journal, 1998. 3(4): p. 155-173.
89. Jørgensen, J.B. and K.B. Lasen. *Aligning Work Processes and the Adviser Portal Bank System*. in *1st International Workshop on Requirements Engineering for Business Need and IT Alignment, in conjunction with RE'05*. 2005. Paris, France.
90. Glinz, M., *An Integrated Formal Model of Scenarios Based on Statecharts*, in *Software Engineering – ESEC'95*, W. Schafer and P. Botella, Editors. 1995, Springer: Berlin. p. 254-271.
91. Harel, D. and A. Naamad, *The STATEMATE Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology, 1996. 5(4).
92. Carroll, J.M., ed. *Scenario-Based Design: Envisioning Work and Technology in System Development*. 1995, Wiley: New York.
93. Amyot, D. and A. Eberlein, *An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development.* International Telecommunications Systems Journal,, 2003. 24(1): p. 61-94.
94. Maiden, N. and D. Corrall, *Scenario-driven systems engineering*. 2000, IEE Seminar: London.
95. Uchitel, S., T. Systa, and A. Zundorf. *Scenarios and state machines: models, algorithms, and tools*. in *Workshop on Scenarios and State Machines: models, algorithms, and tools: Proceedings of the 24th International Conference on Software Engineering*. 2002. Orland, Frolida.
96. ITU, *Message Sequence Charts.* International Telecommunication Union, Telecommunication Standardization Sector, 1996.
97. Whittle, J. and J. Schumann. *Generating Statechart Designs from Scenarios*. in *22nd International Conference on Software Engineering*. 2000. Limerick, Ireland.
98. Kruger, I., et al., *From MSCs to Statecharts*, in *Distributed and Parallel Embedded Systems*, F.J. Rammig, Editor. 1999, Kluwer Academic Publishers.
99. Douglass, B.P., *Real-time UML: Developing Efficient Objects For Embedded Systems*. 2000: Addison-Wesley.
100. Uchitel, S. and J. Kramer. *A Workbench for Synthesising Behaviour Models from Scenarios*. in *proceeding of the 23rd IEEE International Conference on Software Engineering*. 2001. Toronto, Canada.
101. Damm, W. and D. Harel, *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design, 2001. 19(1).
102. Czerny, B.J. and M.P.E. Heimdahl. *Integrative Analysis of State-Based Requirements*. in *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*. 1998. Honolulu, Hawaii.
103. Thomas, D. and A. Hunt, *State Machines.* IEEE Software, 2002. 19(6).
104. Maiden, N., *Discovering Requirements with Scenarios: The ART-SCENE Solution.* Special Theme: Automated Software Enginering, 2004.

105. Subramaniam, K. and B. Far. *Automating Transition from Stakeholder requests to Use cases*. in *IEEE Canadian Conference on Electrical and Computer Engineering*. 2004. Niagara Falls.
106. Jørgensen, J.B. and C. Bossen, *Executable Use Cases: Requirements for a Pervasive Health Care System*. IEEE Software, 2004.
107. Yu, E. *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering*. in *3rd IEEE Int. Symp. on Requirements Engineering*. 1997. Washington D.C., USA.
108. Macaulay, L., *Requirements Engineering*. 1996: Springer.
109. Glinz, M. *Statecharts for Requirements Specification-As Simple as Possible, as Rich as Needed*. in *Proceedings of ICSE; workshop on scenarios and state machines; models, algorithms and tools*. 2002.
110. Polya, G., *How to Solve It*. 1957: Princeton University Press.
111. Phalp, K. and K. Cox. *Using Enactable Models to Enhance Use Case Descriptions*. in *Proc. of ProSim'03, Int. Workshop on Software Process Simulation Modelling (in conjunction with ICSE 2003)*. 2003. Portland, USA.
112. Glinz, M. *Problems and Deficiencies of the UML as a Requirements Specification Language*. in *Proceedings of the 10th International Workshop on Software Specification and Design*. 2000. San Diego.
113. Rolland C and B. Achour, *Guiding The Construction of Textual Use Case Specifications*. 1998, CREWS Report Series.
114. Phalp, K. and K. Cox. *Supporting Communicability with Use Case Guidelines: An Empirical Study*. in *6th International Conference on Empirical Assessment and Evaluation in Software Engineering*. 2002. Keele University, Staffordshire, UK.
115. Uchitel S., Kramer J., and J. Magee, *Synthesis of Behavioural Models from Scenarios*. IEEE Transactions on Software Engineering, 2003. 29(2).
116. BetterHealthChannel, *Artificial Cardiac Pacemakers*. 2001.
117. Stevens, P. and R. Pooley, *Using UML: Software Engineering with objects and components*. 2000: Addison Wesley.
118. Quatrani, T., *Visual Modelling With Rational Rose 2000 and UML*. The Addison-Wesley Object Technology Series, ed. J.R.a.I.J. Grady Booch. 2000, London: Addison-Wesley.
119. IBM, *Role-Based Visual Modeling Platform for Software Teams (http://www.borland.com/us/products/together/)*. 2005.
120. Xie, T. and D. Notkin, *Tool-assisted unit-test generation and selection based on operational abstractions*. Automated Software Engineering, 2006. 13(3).
121. Savidis, A. and C.Stephanidis, *Automated user interface engineering with a pattern reflecting programming language*. Automated Software Engineering, 2006. 13(2).
122. Elkoutbi, M., I. Khriss, and K. Keller, *Automated Prototyping of User Interfaces Based on UML Scenarios*. Automated Software Engineering, 2005. 13(1).
123. Padmanabhan, P. and R. Lutz, *Tool-Supported Verification of Product Line Requirements*. Automated Software Engineering, 2005. 12(4).
124. Heimdahl, M., Y. Choi, and M. Whalen, *Deviation Analysis: A New Use of Model Checking*. Automated Software Engineering, 2005. 12(3).
125. Li, H., S. Krishnamurthi, and K.Fisler, *Modular Verification of Open Features Using Three-Valued Model Checking*. 12, 2005. 3.
126. Taghdiri, M. and D. Jackson. *Inferring Specifications to Detect Errors in Code*. in *the 19th IEEE Conference on Automated Software Engineering*. 2004. Linz, Austria: IEEE CS press.
127. Babin, G. and F. Lustman, *Application of formal methods to scenario-based requirements engineering*. International Journal of Computers and Applications, 2001. 23(3).
128. Sutcliffe A., et al., *Supporting Scenario-based Requirements Engineering*. IEEE Transactions on Software Engineering, 1998. 24(12): p. 1072-1088.
129. Phalp, K. and K. Cox. *Guiding use case driven requirements elicitation and analysis*. in *7th International Conference on Object-Oriented Information Systems*. 2001. Calgary: Springer-Verlag.
130. J. Magee, et al. *Graphical Animation of Behavior Models*. in *Proceedings of the 22nd International Conference on Software Engineering*. 2000.
131. Bulmer, M., ed. *Questionnaires*. SAGE benchmarks in social research methods. Vol. IV. 2004, SAGE: London.

132. Frazer, L. and M.Lawley, eds. *Questionnaire design & administration: a practical guide*. 2000, Wiley.
133. Kanyaru, J. and K. Phalp. *Aligning business process models with specifications using enactable use case tools*. in *Requirements for business need workshop (at RE'05)*. 2005. Paris, France.
134. Kitchenam, B., et al., *A framework for evaluating a software bidding model.* Information and software Technology journal, 2005. 47.
135. Kitchenam, B., Stephen Linkman, and S. Linkman, *Experiences of using an evaluation framework.* Information and software Technology journal, 2005. 47(11).
136. Flyvbjerg, B., *Five Misunderstandings about Case Study Research.* Qualitative Inquiry, 2006. 12(2).
137. Easterbrook, S. and J. Aranda. *Case Studies for Software Engineers*. in *28th International Conference on Software Engineering*. 2006. Shanghai, China.
138. Seybold, C., S. Meier, and M. Glinz. *Evolution of Requirements Models by Simulation*. in *International Workshop on Principles of Software Evolution*. 2004. Kyoto, Japan.
139. Eshuis, R. and R. Wieringa, *Tool Support for Verifying UML Activity Diagrams.* IEEE Transactions on Software Engineering, 2004. 30(7).
140. Heymans, P. and E. Dubois. *Some thoughts about the animation of formal specifications written in the Albert II language*. in *3rd IEEE International Symposium on Requirements Engineering*. 1997.
141. Amyot, D. and A. Eberlein. *An Evaluation of Scenario Notations for Telecommunication Systems Development*. in *The 9th International Conference on Telecommunication Systems*. 2001. Dallas, TX, USA.
142. Tuok, R. and L. Logrippo, *Formal Specification and Use Case Generation for a Mobile Telephony System.* Computer Networks, 1998. 30(11).
143. Basili, V. and R. Selby, *Paradigms for Experimentation and Empirical Studies in Software Engineering.* Reliability Engineering and System Safety, 1991. 32(1-2).
144. Basili, V. *The Role of Experimentation in Software Engineering: Past, Current, and Future*. in *Eighteenth International Conference on Software Engineering* . 1996. Berlin, Germany.
145. Selby, R., V. Basili, and T. Baker, *Cleanroom Software Development: An Empirical Evaluation.* IEEE Transactions on Software Engineering, 1987. 13(9).
146. Shull, F., et al. *Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem*. in *First International Symposium on Empirical Software Engineering*. 2002. Nara, Japan.
147. Milner, R., *Calculus of Communicating Systems*. 1980, Springer-Verlag.

Appendices

A. Additional data

A.1 The systems

The organisation installs the main monitoring application at each customer's site. The monitoring application has the following sub-systems: alarm receiver, alarm display, network controller drivers, scheduler (a middleware for inter-component communication), a web-client for data display, database builder, and a file editor. The databases (created automatically by database builder sub-system) are: Logs database, Alarms database, and User Tracking database. The following is a brief description of the main functions of each subsystem:

- Alarm receiver obtains active alarms from alarmed fixtures and records the alarms into Alarms database.
- Alarm display pops active alarms onto a customer's computer monitor.
- Database builder automatically creates the above mentioned databases upon the installation of the entire system.
- File editor subsystem provides an interface for an operator at customer site to construct project files containing attributes of monitored fixtures and network layout.
- Scheduler supports communication between the subsystems (including the network controllers) for data exchange.
- Web client for displaying alarm handling statistics (number of alarms resolved, pending alarms, who resolved them, etc.)
- The Alarms database stores data regarding alarms.
- The User tracking database stores information regarding site-based operators and their usage of the various sub-systems.
- The Logs Database stores records of monitored data (e.g., temperature, refrigerant liquid levels, etc.)

At the time of the study, the control systems were dividend into two broad categories, namely, the systems deployed at customer sites (hence termed site-based systems) and the systems deployed at the Company's office (also called the Control Centre Systems). For each site, there is at least one individual (operator) termed a key holder. The key holder is a person associated with a site who can be contacted by the operators at the control centre office. Thus, each site must have at least one key holder (or customer name for contact) associated with the site. The information

necessary to fully describe a site is outlined in part B of these appendices.

The following sections present the descriptions of various site-based and control centre-based systems (and subsystems).

A.2 Site-based Systems

The site-based systems are for data logging and reporting on activities within the monitored fixtures. All the data logging and monitoring functions are encapsulated within designated subsystems that interact, exchange data and messages during the execution of their monitoring tasks. The subsystems are outlined below (respective use cases can be found in part B of the appendices).

a) FileEditor: When a developer (or a support engineer) from the Control Systems Company installs the site-based system on the customer's host machine, an operator at the site creates a text file comprising information regarding the monitored fixtures, the network profile of any other communication software, and the profile of users at the site. A crucial communication software whose virtual network profile must be stored in the text file is Scheduler (see part e). This subsystem is a type of middle-ware for inter-component communication, especially, sitting at the interface between network controllers and other subsystems.
Thus, the main function of the FileEditor subsystem is to enable the creation of essential properties of monitored fixtures and save them in a text file for access during the monitoring process. In a nutshell, FileEditor allows a user (customer) to produce project files (as text files) that contain the following information:

- Virtual network layout
- Scheduler's IP address
- Database connection information
- List of data points to be logged and where they should be logged
- Information regarding users of applications and their login details and privileges.

The project files are stored in a directory on a host machine at the customer's site.

b) DBaseBuilder: This subsystem automatically creates databases based on information contained in the text files created by the FileEditor subsystem. The information relevant for creation of the

databases include fixture details (e.g., code, name, location etc). For example, the alarms database stores information regarding any arising alarms, or any alarms pending actioning, or indeed previous alarms that have been resolved (including who resolved them and the action taken to resolve them). Another database created by this module is the Logs Database which records the data variables being monitored (e.g., temperature, refrigerant liquid levels etc.).

c) AlarmReceiver: This subsystem obtains active information regarding active alarms (e.g., AlarmID, AlarmType, AlarmTime, FixtureCode, FixtureName) and records it on the Alarms Database. This subsystem provides important functions to the customer and should be available all the time.

d) AlarmDisplay: Once the alarms have been recorded on the Alarms database, the AlarmDisplay subsystem pops the active alarms on the customer's PC monitor. This enables the customer to know which fixture has raised the alarm, what time the alarm occurred, and the recommended action for resolving the alarm. Depending on the type of alarm (e.g., open door, or smoke), the customer can resolve the alarm straightaway, or a control engineer can resolve it for the customer (e.g., if it is an alarm due to a fault in the control systems).

e) Scheduler: this subsystem offers a communication interface between applications that need environmental data and the serial ports. It (Scheduler) connects the AlarmReceiver and the data logging subsystems) with the external networked artefacts (e.g., fixtures) that are being monitored. Thus, the stable communication between Scheduler and monitoring applications (the applications needing network access) is critical to the monitoring process.

f) DataLog: This is a data logging application. DataLog simply records points of data onto the Logs database. This data concerns the variables that the control systems monitor on the fixtures. Simply put, DataLog accesses project files to gain the structural layout of the information it needs. Subsequently, DataLog requests the information from Scheduler which in turn (Scheduler) contacts a network controller to gather the requested information. The controller returns requested values to Scheduler which in turn (Scheduler) returns information to DataLog. The information returned to DataLog is then recorded into the Logs database.

g) ActivityTracker: This subsystem records information regarding activities being undertaken at particular sites by customers using various site-based subsystems. For example, once a customer

resolves an alarm, the action undertaken to resolve the alarm must be recorded, including the details of the customer who acted on the alarm. All such information is stored in the User-Tracking database.

A.3 Control Centre Based Systems

Many control-centre based subsystems are copies of those deployed at the customer sites. For instance, at the Control Centre, the engineers run Alarm Receiver subsystems for each customer site.

An additional component of the Alarm Receiver subsystem for the control centre operators is the Call Centre Collection Service (CCCS). The communication between the Alarm Receiver and the CCCS is crucial. This is explained as follows: The Alarm Receiver constructs an alarm text file containing the information (alarm identifier, alarm type, alarm time, fixture identifier, site code, site name) about the alarm arising at the customer site. Every fifteen seconds, the CCCS accesses the folder in the hard disk of the host machine to fetch alarm text files in which the alarm information is encoded. CCCS then executes a SQL standardization script, which decodes the alarm information from the file. The alarm information that the standardization procedure is concerned with is AlarmTime, AlarmID, SiteCode, FixtureType, FixtureCode and AlarmType. The standardization procedure also executes a script that is used to determine the appropriate alarm resolution action. The alarm record constructed by the CCCS is stored in the Standardised Alarms table of the database. The table stores these alarms in a queue awaiting an operator to resolve them. That is, CCCS first decodes alarm information from alarm text files (created by the alarm receiver) and stores the alarms in a standard alarms table. These are the alarms that are in the process of being dealt with. All the alarms here should send warnings to the supervisor/senior operator if they are overdue with being dealt with.

Thus, unlike site-based alarm handling where an alarm receiver records an alarm, then the alarms display subsystem pops the alarm for the customer to view and resolve (manually), the Control Centre has further automation where the CCCS intervenes to 'suggest' alarm resolution strategy.

An additional component of the control-centre based system is a web client which provides real-time data regarding alarm activity in all the supported sites. This information is accessed by control centre staff who then prioritise the sites and alarm types that need immediate attention if the Alarm Receiver-CCCS process did not culminate in a successful resolution of the alarm.

Figures 8 and 9 of part C (of the appendices) shows the textual use cases for the control-centre alarm receiver and CCCS processes.

A.4 Further initiatives

The company proposed the reengineering of its control systems. This proposal was communicated to the author by the operations and applications manager. The essence of the reengineering work was stated as follows. Maintenance overheads were increasing due to the need for engineers (the developers and occasionally, operators) to travel to customer sites most of the time there was a task to be resolved. The company therefore realised that they needed to investigate and specify the aspects of the existing systems that needed reengineering for adaptation into a centralised infrastructure. The applications and operations managers indicated that the centralisation effort was going to result in applications at customer sites becoming clients to server applications administered centrally by the Control systems company. The databases were to be centralised too, so that all the client applications did was to request data/services from the respective remote server. This way, the engineers only needed to ensure continued stability of server applications as any problems with clients could be resolved via telephone call or the customer being given access to a remote copy of the necessary subsystem. Besides centralisation, an additional functionality was to be bundled into server applications to enable the addition of new networks for new customers, registration of associated network controllers, and the creation of project roots for these networks and for new customers. This aspect of the new infrastructure needed to be specified.

The following is a description of the capabilities that needed to be centralised in different functional modules to act as servers for light-weight client applications based at customer sites:
ConnectionBuilder: Given the necessary communication between remote clients and servers, the ConnectionBuilder's first task is to establish a stable network connection between the client application (e.g. a site-based alarm receiver) and the server (a Control Centre-based Alarms database with stored procedures for resolving alarms). This module therefore has the further demand to support construction of such objects as standard alarms and database connections. That is, remote client applications make requests to a module charged with the function of standardising alarms, and storing the alarms in a database table based on the site whose client application sent the alarm data. Thus, a server application needs to be clearly set out with functionality to enable the receiving of alarm data from the remote client, establishing a database connection, standardising the alarm, determining the appropriate response and storing the alarm

on the customer database quota. This ensures that all requests are handled at the control centre (even if it appears to a remote customer as if every response is local to their site).

ProjectBuilder: A project builder is central to the activities of adding new customers to the Company's database of the customers being supported. Once the usual business negotiation has been done between the managing director (of the Control Engineering company) and the business director of the new customer's company, the next task is for a project task to be initiated where various functions are executed to enable the support of the new customer. For example, depending on the fixtures being supported its network type is determined, network controllers are configured, the controllers are registered for the new network. Briefly, the incorporation of a new customer must be completed by executing a number of functions which are described in part D of the appendices.

This module also has the functionality to implement on the fly a controller interface that allows a controller to register an event with the interface requesting a connection to another network type, requesting data from that network, and un-registering the network. This is to ensure that any client application that requires data from multiple network types can dynamically connect to these network types via capabilities being proposed in the new infrastructure.

This module also has the functional responsibility of creating a description of each client for all customers. This description includes client name, client socket type, client protocol type, and address family. These attributes are necessary when a client needs to establish communication session with remote server, network controller or other clients.

The first issue of concern for the proposed client-server architecture became the establishment of a generic ubiquitous client-server connection process whereby client applications would request data/services from server applications. For the actual communication between clients and servers, we needed the input of the developers who had a greater depth of lower level issues than the managers. Further scrutiny resulted in a systematic client-server connection use case description shown in Figure 10, part D of the appendices.

As mentioned previously, an additional demand was to enhance the process of registering a new customer for support with the control company's real-time monitoring systems. It was imperative, given the proposed centralised infrastructure that the registration of a new customer is made as easy as possible with appropriate software support. In this process (registration of a new customer) the three members of staff involved in the initial decision making are the managing director, the operations manager and the applications manager. Further discussions yielded a generic customer registration use case shown in Figure 11, part D of the appendices.

B. Site information

- Site Code- a unique alphanumeric identifier for a site (e.g. SW100XT)
- Site Name- the name of the site (the same name that would be used in the mailing address)
- Site Address- the mailing address for the site
- Site Phone Number- the telephone number for the site
- Site Fax Number- the fax number for the site
- Site E-mail Address- an email address for the site
- Site Opening Times- the opening (and closing) times for each day of the week
- Customer Name- the name of the contact customer for the site
- Fixtures-the fixtures being monitored in the site
- Fixture Sub Systems-subsystems monitoring a fixture(depends on the functions of the fixture)
- Fixture valuation- the cost associated with the fixture for stock taking
- Controllers- network controllers for the site
- Controller Name- the name assigned to the controller
- Controller Address- the network address of the controller
- Controller Type- the type of the controller depending on the network it interfaces.
- Alarm Methods- the specified actions for solving different alarms for the site
- Key Holders - individuals who carry the site's keys
  - Name
  - Phone Number
  - Fax Number
  - E-mail Address
  - Position Held
- Site Check Frequency- the frequency that the site needs to be checked by control centre staff.

C. Site-based systems use cases

The use case description detailing the desired behaviour of the project files creation by the FileEditor subsystem is outlined in Figure C.1:

1. User creates text file (project file) with FileEditor
2. User adds virtual network layout to project files
3. User adds database connection information to project files
4. User adds list of point types to be logged to project files
5. User sets up login information for all applications
6. User adds Scheduler IP address to project files
7. User stores project files on host machine

Figure C.1: Project Files creation

Figure C.2 is a use case description for the database creation process:

1. User creates project files with FileEditor
2. User saves project files on host machine
3. DBaseBuilder reads project files
4. DBaseBuilder obtains database connection information from project files
5. DBaseBuilder obtains point types to be logged from project files
6. DBaseBuilder obtains Scheduler IP address from project files
7. DBaseBuilder executes SQL commands to create Logs DB
8. DBaseBuilder executes SQL commands to create Alarms DB
9. DBaseBuilder executes SQL commands to create Tracking DB

Figure C.2: Database creation

Figure C.3 is a use case description of the site-based alarm receiving process (final use case):

1. Alarm Receiver requests alarm data (FixtureCode, AlarmID, AlarmType, AlarmTime, SensorValue, SiteCode, SiteName, etc.) from Scheduler
2. Scheduler sends alarm data request to the Controller
3. Controller gets alarm data request
4. Controller obtains alarm data from fixtures
5. Scheduler sends alarm data to Alarm Receiver
7. Alarm Receiver records alarms data (in alarms database)
8. Operator accesses existing alarms
9. Operator views recommended alarm actions
9. Operator resolves alarm

Figure C.3: Site-based Alarm Receiving

Figure C.4 is a use case description of the AlarmDisplay subsystem:

1. Fixture triggers alarm

2. Controller sends alarm data to Schedule
3. Schedule relays alarm to alarm receiver
4. Alarm Receiver records alarm in Alarms DB
5. Alarms Display reads project files
6. Alarms display reads user settings from project files
7. Alarms display accesses alarms table in Alarms DB to read active alarms
8. Alarms Display pops alarm on user's PC monitor [alarm details include FixtureCode, AlarmTime, AlarmType, SiteCode, and AlarmID]
9. ActivityTracker records alarm resolution actions on User Tracking Database

Figure C.4: Alarms Display use case

The general process for connection between applications at the customer site and Scheduler is shown in Figure C.5:

1. Application opens project files
2. Application connects to Scheduler
3. Scheduler acknowledges connection
4. Application sends network layout to Scheduler
5. Scheduler creates network handle for the application
6. Scheduler registers application for resource use
7. Scheduler sends registration to application
8. Application requests information from resource
9. Resource sends information to application
10. Application executes monitoring functions

Figure C.5: Scheduler-application communication

Figure C.6 is a use case description of the data logging process:

1. DataLog obtains Scheduler IP from project files
2. DataLog requests connection to Scheduler
3. Scheduler acknowledges connection to DataLog
4. DataLog relays virtual network layout to Scheduler
5. Scheduler checks network layout against existing network handler
6. Scheduler registers DataLog with network handler
7. Scheduler sends acknowledgment to DataLog
8. DataLog sends points data request to Scheduler [e.g. temperature, refrigerant level, etc]
9. Scheduler forwards data to DataLog

10. DataLog reads database connection information from project files
11. DataLog records data into the points table in Logs database.

Figure C.6: Data logging

Figure C.7 is a textual use case description for activity tracking subsystem:

1. Customer executes a site-based task
2. ActivityTracker reads user information from project files (user name, role, profile of systems allowed to access etc)
3. ActivityTracker records user details into User Tracking DB
4. ActivityTracker reads recorded alarms from Alarms DB
5. ActivityTracker reads user actions taken to resolve alarms
6. ActivityTracker records user's actions against the user in Tracking DB

Figure C.7: Activity Recording use case

D. Control centre-based systems use cases

Figure D.1 shows a use case description of alarm receiving at the control centre

1. Fixture raises alarm at customer site
2. Controller obtains alarm data [site code, site name, date/time of alarm, system/fixture name, sensor name, current sensor value, alarm set point]
3. Controller relays alarm data to Scheduler
4. Scheduler connects to AlarmReceiver
5. Scheduler relays alarm data to Alarm receiver
6. Alarm receiver returns alarm received acknowledgement
7. Alarm receiver records alarm on database
8. Alarm Receiver constructs alarm textfile
9. Alarm receiver saves the alarm textfile on local HDD
10. Scheduler drops connection with AlarmReceiver

Figure D.1: Control Centre Alarm Receiver use case

Figure D.2 shows the use case description for the CCCS process:

1. CCCS polls alarm receivers (every 15 seconds)
2. CCCS reads alarms text files (on local HDD)
3. CCCS calls the standardisation procedure

[Standardisation procedure writes alarm information into a standardised alarms table.
This is to make a copy of original alarms, store them else where for actions thereby leaving the original raw alarms data unchanged]

4. CCCS produces recommendations for resolving the alarm

5. Operator resolves alarms in Active table manually
6. CCCS stores unresolved alarms in active alarms table.

Figure D.2: Bureau Alarm Collection Service use case

NB: Some alarms are resolved automatically by appropriate scripts executed by CCCS.
Some other alarms require intervention by an operator at Next Control.
Some alarms however are resolved by customers at their sites.

E. Further use cases

Functions of the ProjectBuilder:

1. AddNewController- a new controller is added to ensure client applications are able to connect to the specific network types.
2. AddNewNetwork- a network for a given category of fixtures is set up by the engineers and the necessary server applications configured for it.
3. AddNewProjectsRoot- a network profile, including folders for storing different access privileges and data.
4. AddNewSession- a method for ensuring connection session attributes are always accessed and verified to establish a stable communication session.
5. AddNewDatabase- this function is executed by specific embedded SQL commands for the construction of various databases for a new customer (e.g. AlarmsDB, LogsDB, UserTrackingDB).
6. CreateDatabase- this is an overridden version of the AddNewDatabase function. It ensures that during the writing of any data to a database element, such a database is checked to ensure it exists; otherwise, a new database is created.
7. ConfigureDatabase- this function ensures the correct settings, including the class path (directory structure), database elements (tables, views and stored procedures), are done depending on the database being created.

Figure E.1 is a client-server connection use case:

1. Client looks-up the Domain Naming Service
2. Client creates a connection Socket
3. Client connects to server
4. Server requests login data
5. Client sends login data
6. Server authenticates login data

Figure E.1: Client-Server connection use case

Figure E.2 is a centralised customer registration use case:

1. Operations manager prepares fixtures list.
2. Managing director ratifies fixtures.
3. Operations manager creates customer's profile.
4. Applications Manager sets up client applications network information.
5. Applications manager stores client applications in customer's profile.
6. Applications manager links client applications to server applications.
7. Applications manager configures Server applications.
8. Applications manager initialises customer's client applications.

Figure E.2: Registering a new customer

F: Questionnaire used to obtain student feedback

F.1 The Questionnaire

1. The EducatorTool application is easy to learn

(This refers to how easily you were able to learn to write your actors, events and states; it also includes saving, printing, adding extra events and extra states etc)

0. No Opinion	1. Strongly disagree	2. Disagree	3. Undecided	4. Agree	5. Strongly Agree

Comments

2. The EducatorTool application is efficient

(This refers to how well suited to the task the application is. That is, compared to say, writing the description with pen and paper; and whether you think it took you too long doing some tasks such as choosing actors and/ or states)

0. No Opinion	1. Strongly disagree	2. Disagree	3. Undecided	4. Agree	5. Strongly Agree

Comments

3. Enaction enabled me to scrutinise the description further

(This refers to state-based enaction. Did the state changes during enaction cause you to think a lot more about your description, did you find that you needed to revise the description because the enaction produced state changes that were not appropriate according to your knowledge of the problem represented by your description)

0. No Opinion	1. Strongly disagree	2. Disagree	3. Undecided	4. Agree	5. Strongly Agree

Comments

4. Revising the description using the tool is easy

(This follows directly from question 3 above. After deciding to revise the description, e.g. by adding more states, or appending secondary actors to events, or even changing primary actors for some events, was it easy to effect those changes)

0. No Opinion	1. Strongly disagree	2. Disagree	3. Undecided	4. Agree	5. Strongly Agree

Comments

5. Please describe any changes to functionality or added functionality that you would like to see:

The graph and table shown below presents the participants responses to each suggestion. A graphical representation of these responses is also given.

F.2 Questionnaire analysis

The data collected from the participants was laid on a table showing the number of respondents that chose a particular option for each question. The table then was translated into a bar graph. The two figures are shown overleaf:

	No Opinion	Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
Suggestion 1	0	0	1	0	7	0
Suggestion 2	0	0	0	0	3	5
Suggestion 3	0	0	0	0	2	6
Suggestion 4	0	0	1	0	4	3

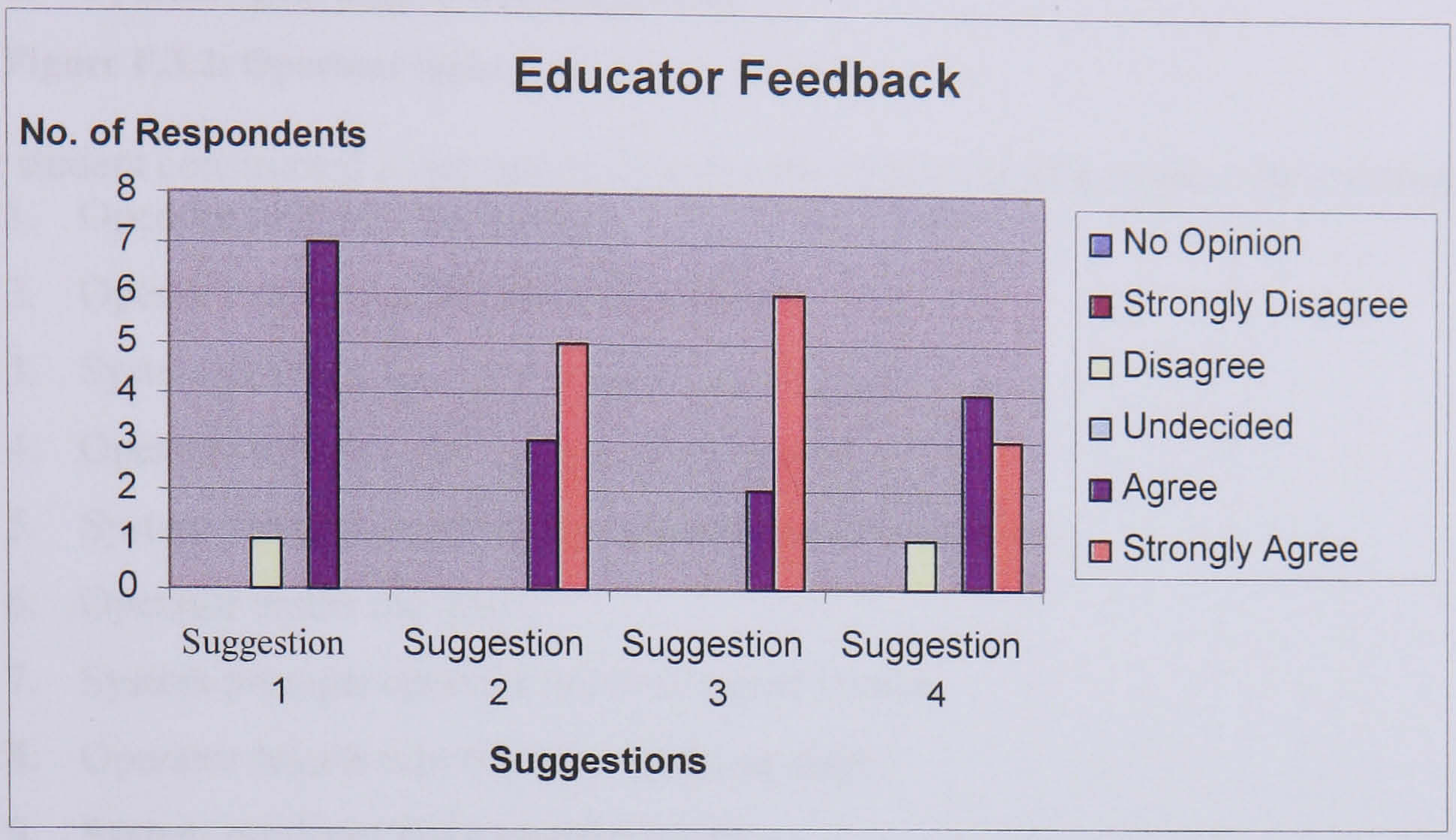


Figure F.2.1: Graph of respondents against suggestions

F.3 Further use cases from student workshops

The use case in Figure F.3.1 shows a use case for Operator validation. It is a use case to be performed when an operator of a depot entry system wishes to login to undertake one or more functions with the system. The use case was constructed by one of the students during the second workshop.

1. Operator selects system login function.
2. System prompts operator to enter login details.
3. Operator enters their login details.
4. System searches for operator details on database.
5. System validates operator details.
6. System confirms operator details ok.
7. Operator logs into the system.

Figure F.3.1: Validate operator details

Another student constructed a use case to describe the login behaviour as follows:

1. System prompts operator to enter their username.
2. Operator enters their username.
3. System prompts operator to enter their password.
4. Operator enters their password.
5. System verifies the operator's username.
6. System verifies the operator's password.
7. System accepts operator login details.

8. Operator gets logged into the system.

Figure F.3.2: Operator login

Another student constructed a use case to describe the behaviour of a module for creating reports:

1. Operator logs into the system.
2. Operator selects create report function.
3. System prompts operator to select report type.
4. Operator selects report type (e.g. movements).
5. System prompts operator for report dates (from and to).
6. Operator enters the dates.
7. System prompts operator to select report format.
8. Operator selects report format (print or view).
9. System produces the requested report.

Figure F.3.3: Create transaction reports

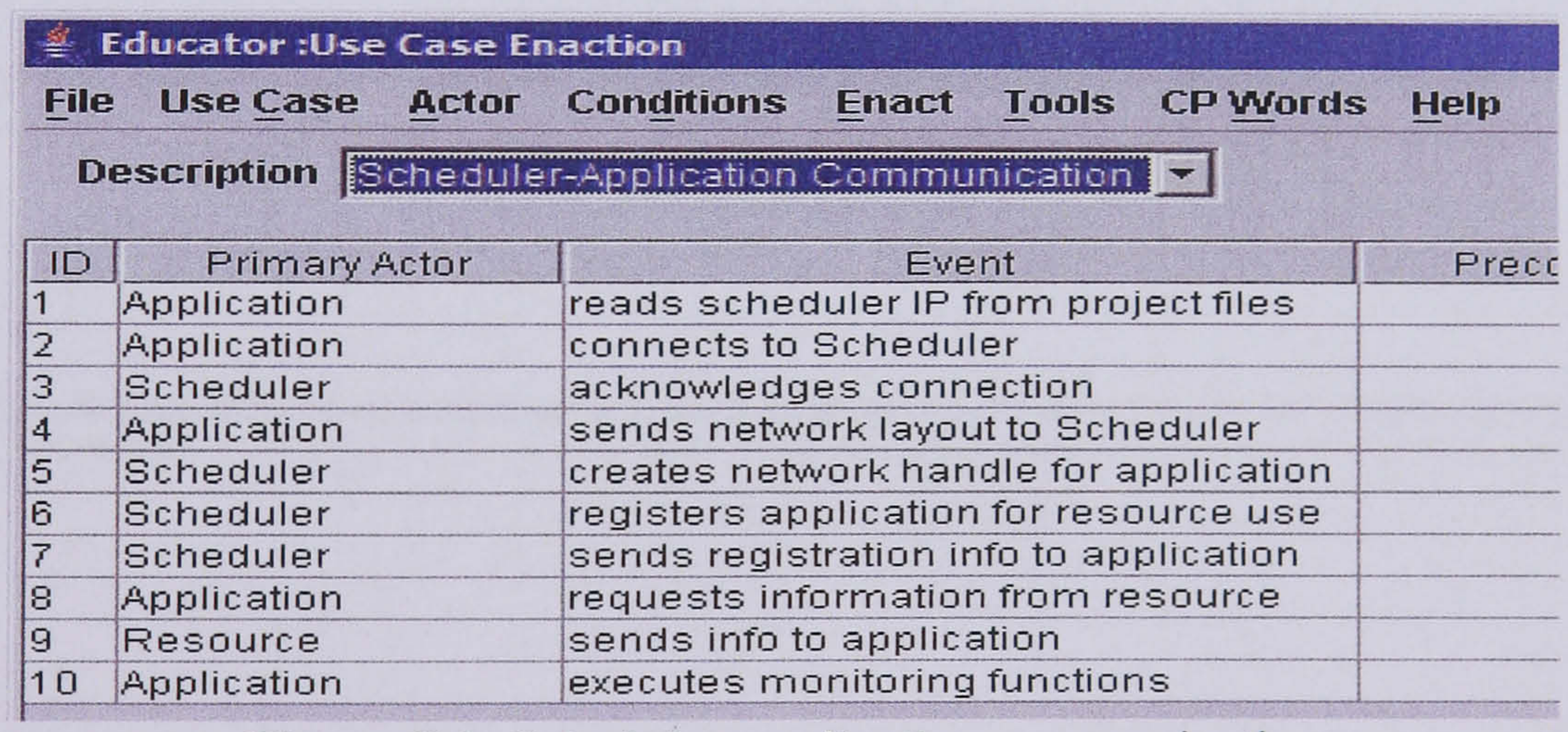
Examples showing the analysis on the above use cases can be found in appending K.

G: Additional (analysis) examples from the industrial study

<u>Example 1</u>

Consider the Scheduler-application communication use case description of Figure C.5.

The default use case edited within EducatorTool is shown in Figure G.1:



ID	Primary Actor	Event	Precc
1	Application	reads scheduler IP from project files	
2	Application	connects to Scheduler	
3	Scheduler	acknowledges connection	
4	Application	sends network layout to Scheduler	
5	Scheduler	creates network handle for application	
6	Scheduler	registers application for resource use	
7	Scheduler	sends registration info to application	
8	Application	requests information from resource	
9	Resource	sends info to application	
10	Application	executes monitoring functions	

Figure G.1: Scheduler-application communication

The essence of the description in Figure G.1 (and C.5) is to show the steps that constitute a communication process between a site-based application and the Scheduler subsystem. Enaction of the above default use case description was done to further scrutinise any dependency issues without application of state-based information (further addressing objective 2.1).

During enaction of the description in Figure G.1, the initial enaction dialog is shown in Figure G.2:

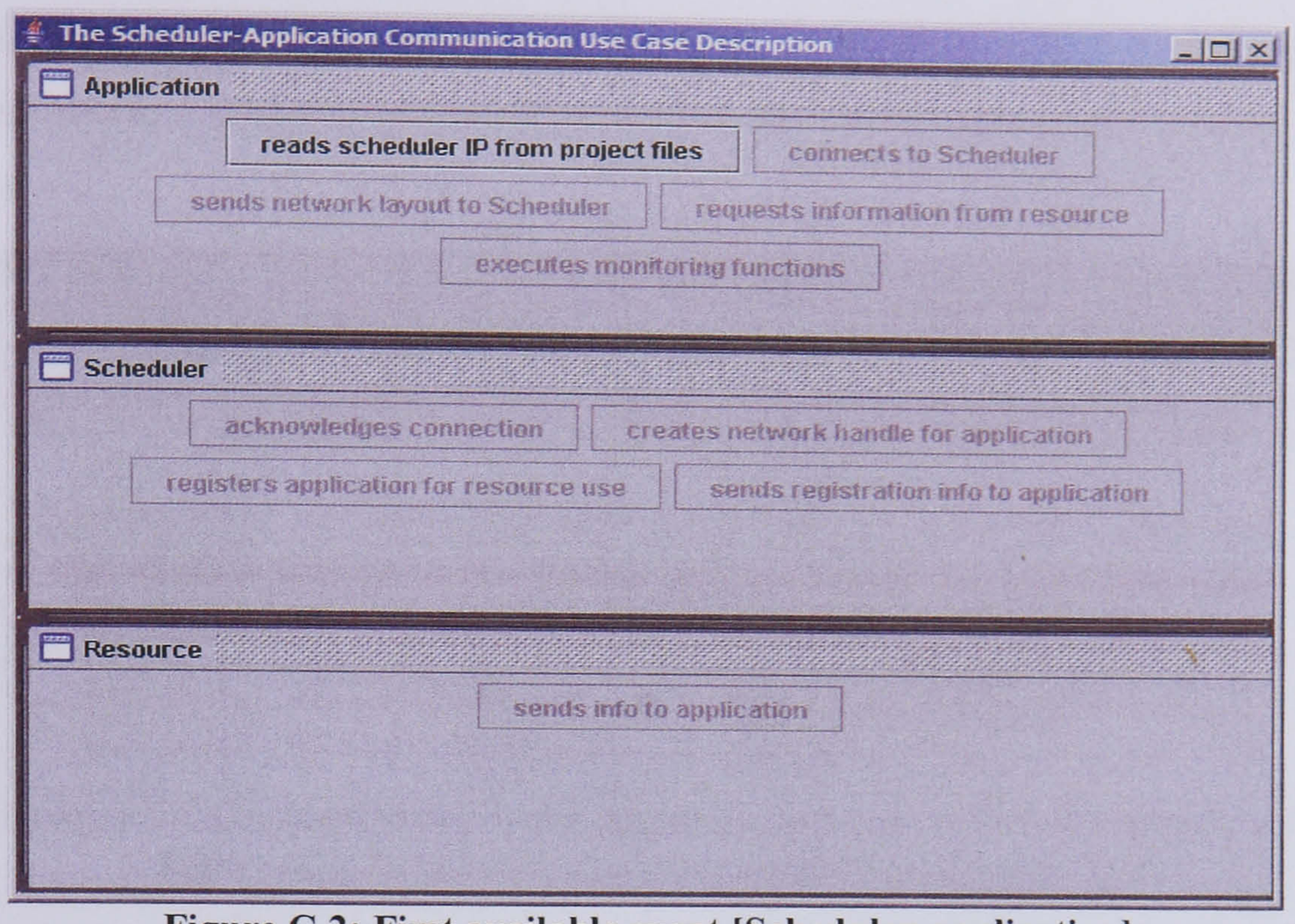


Figure G.2: First available event [Scheduler-application]

Given that default enaction entails stepping through the events in the order they are written, the next available event would be "Application connects to Scheduler":

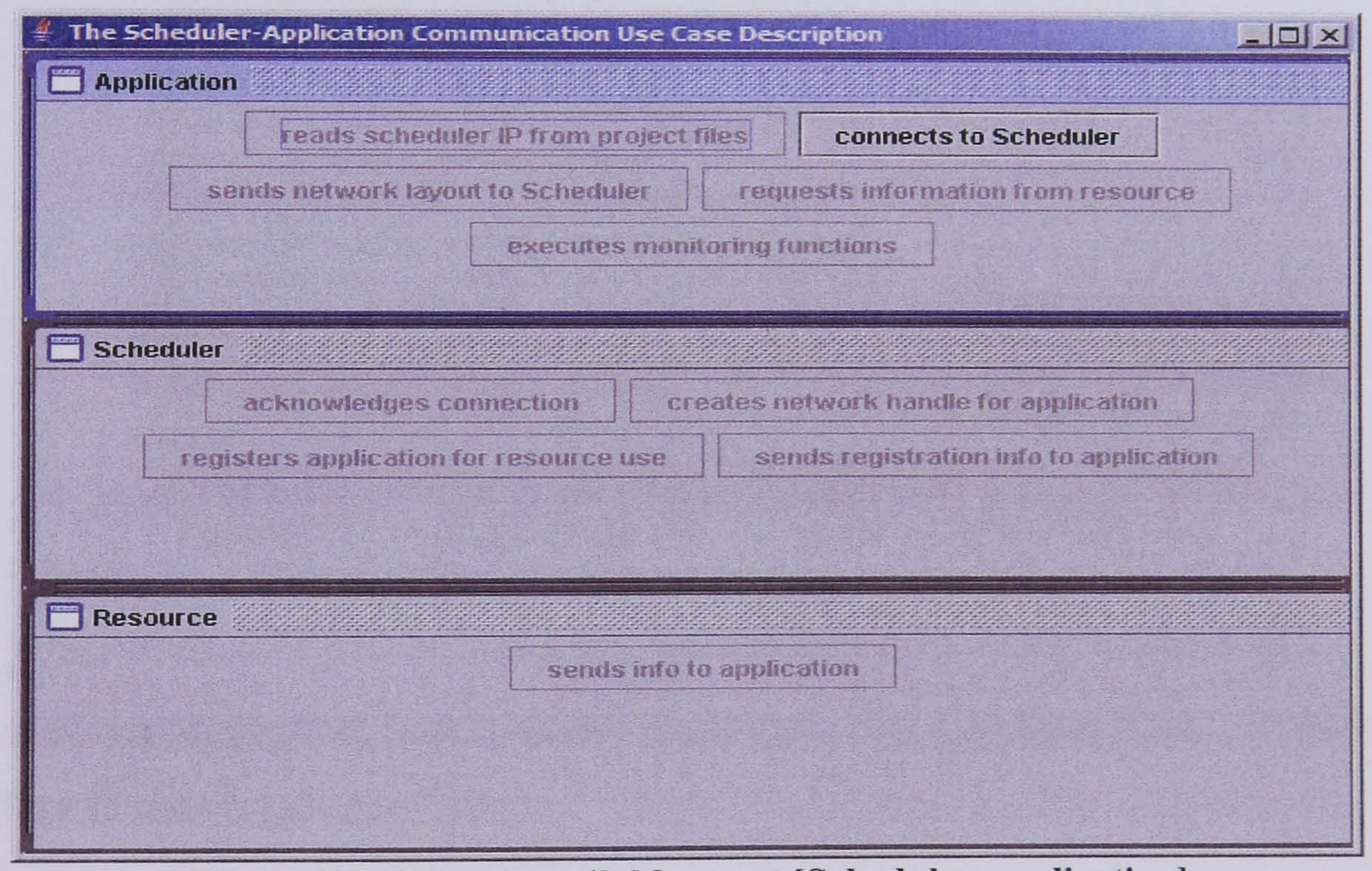


Figure G.3: Second available event [Scheduler-application]

One of the developers observed that prior to the event in Figure G.3 happening, that is, before connection is done and acknowledged, the application must first request the connection to be established. The reason for this is that Scheduler could have been "held" by other applications, and a straightforward connection would disrupt previous, perhaps more urgent communication. Moreover, it was argued that prior to Application sending the network layout to Scheduler (event 4, Figure G.3), Scheduler must request the applications network layout first. The reason for this is

that, Scheduler has an inbuilt network layout for some of the most crucial applications, and does not need them resent. Hence a revised Scheduler-application communication use case is shown in Figure G.4:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Scheduler-Application Communication

ID	Primary Actor	Event	Prec
1	Application	reads scheduler IP from project files	
2	Application	requests connection	
3	Application	connects to Scheduler	
4	Scheduler	acknowledges connection	
5	Scheduler	requests network layout	
6	Application	sends network layout to Scheduler	
7	Scheduler	creates network handle for application	
8	Scheduler	registers application for resource use	
9	Scheduler	sends registration info to application	
10	Application	requests information from resource	
11	Resource	sends info to application	
12	Application	executes monitoring functions	

Figure G.4: Revised scheduler-application communication

The output of enaction for the description in Figure G.4 is shown below:

Scheduler-Application Communication

Enaction Output
Application reads scheduler IP from project files
Application requests connection
Application connects to Scheduler
Scheduler acknowledges connection
Scheduler requests network layout
Application sends network layout to Scheduler
Scheduler creates network handle for application
Scheduler registers application for resource use
Scheduler sends registration info to application
Application requests information from resource
Resource sends info to application
Application executes monitoring functions

Figure G.5: Enaction output for revised scheduler-application

Example 2

Consider the Data logging use case description of Figure C.6. The default use case edited within EducatorTool is shown in Figure G.6:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description DataLogging

ID	Primary Actor	Event
1	DataLog	obtains Scheduler IP from project files
2	DataLog	requests connection to Scheduler
3	Scheduler	acknowledges connection to DataLog
4	DataLog	relays virtual network layout to Scheduler
5	Scheduler	checks network layout against existing network handler
6	Scheduler	registers DataLog with network handler
7	Scheduler	sends acknowledgment to DataLog
8	DataLog	sends points data request to Scheduler
9	Scheduler	forwards data to DataLog
10	DataLog	reads database connection information from project files
11	DataLog	records data into the points table in Logs database.

Figure G.6: Data logging use case description

The data logging use case (Figures G.6 and C.6) who the data logging process as executed by the DataLog subsystem. Like many other monitoring subsystems, DataLog needs the services of the Scheduler application. Again, default enaction (or enaction of standard use cases) entails stepping through the events in the order they were written. One of the issues raised during the enaction of the use case in Figure G.6 regarded event 9 (Scheduler forwards data to DataLog):

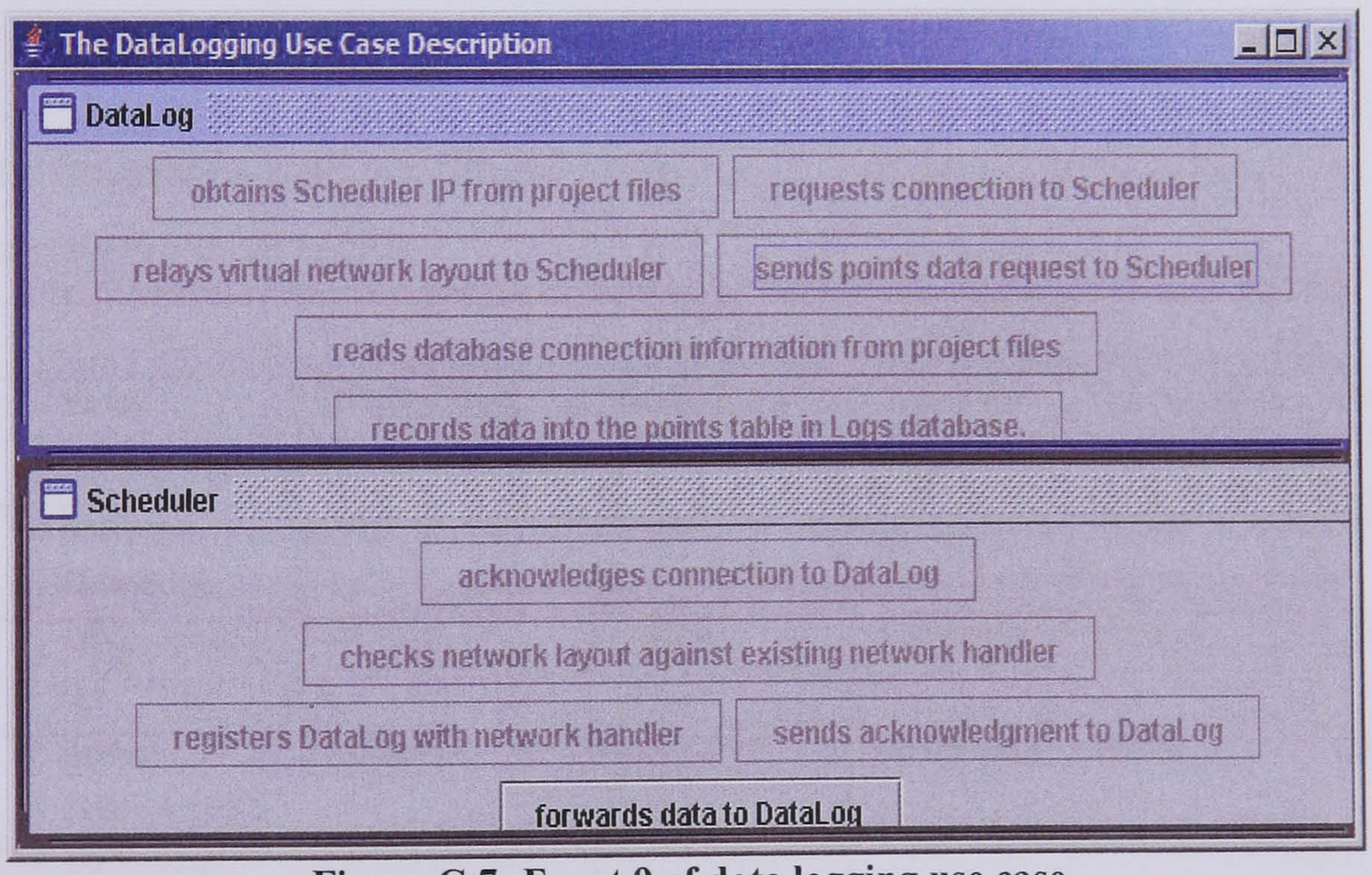


Figure G.7: Event 9 of data logging use case

It was observed by the developers that the above event (Figure G.7) cannot happen without Scheduler requesting such data from a network controller. In other words, Scheduler communicates with the controllers on behalf of the DataLog. The Controller must then return the requested data to the Scheduler, which in turn relays the data to DataLog.

Hence, the revised use case description Data logging is shown in figure G.8:

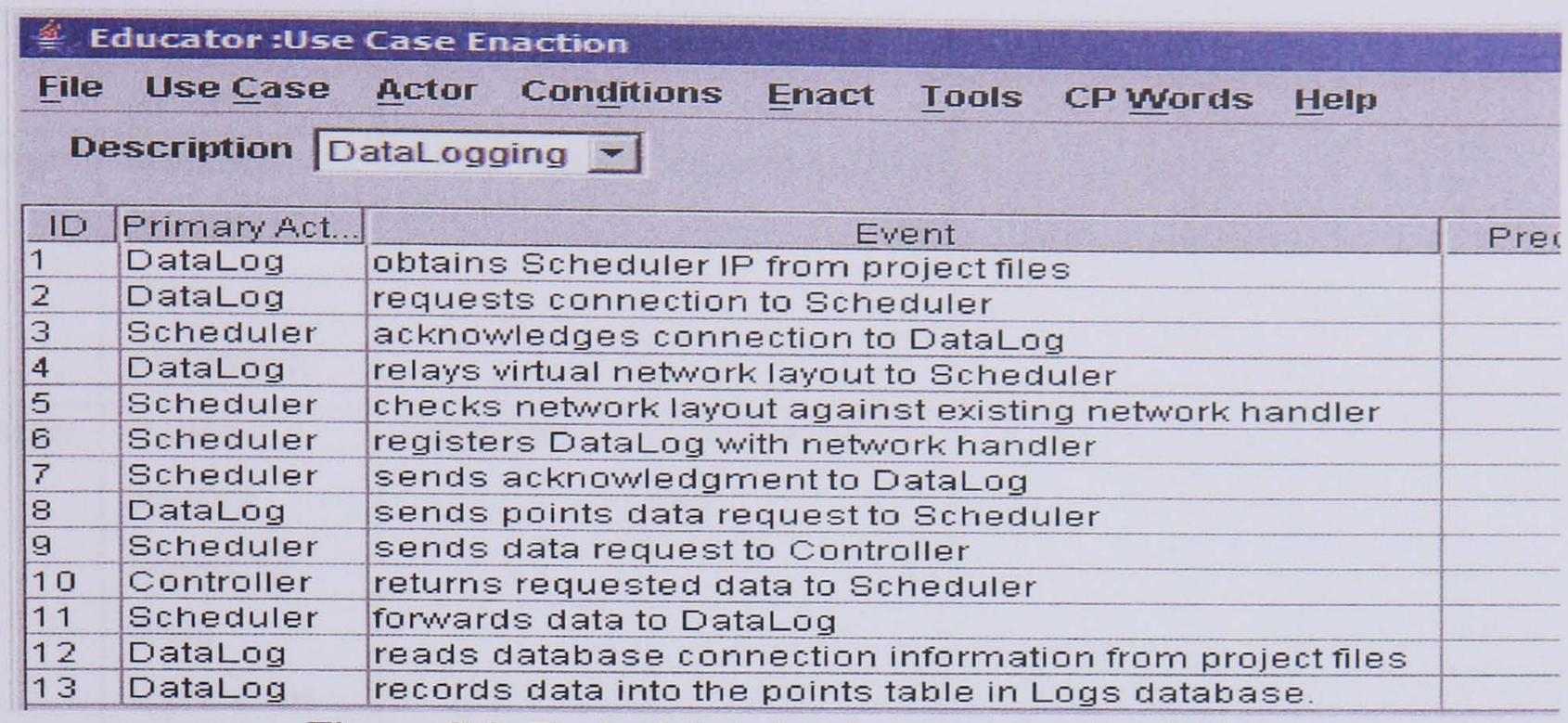


Figure G.8: Revised data logging use case description

Two additional events were added as a result of enaction:

- Scheduler sends data request to Controller
- Controller returns requested data to Scheduler

An actor that was not within the initial description was added, the Controller.

Example 3

Consider the Customer registration use case of Figure E.2. A state-based version of the description is shown in table G.1:

Primary Actor	**Event**	**Precondition**	**Postcondition**	**Secondary Actor**	**Precondition**	**Postcondition**
OpsManager	prepares fixtures list	initial	listReady	MD	waiting	listReady
MD	ratifies fixtures list	listReady	listRatified	OpsManager	listReady	listRatified
OpsManager	creates customer profile	listRatified	profileCreated	AppsManager	waiting	profileCreated
AppsManager	sets network info for client applications	clientsInitialised	networkInfoSet			
AppsManager	stores client applications in customer profile	listenerRegistered	clientsStored			
AppsManager	links clients applications with server applications	serverAppsConfigured	linkingDone			
AppsManager	configures	networkInfoSet	serverAppsConfig			

	server applications		ured			
AppsManager	initialises customers client applications	profileCreated	clientsInitialised			
AppsManager	registers network listener for client applications	linkingDone	listenerRegistered			

Table G.1: state-based version of customer registration use case

Discussion

Table G.1 shows the state-based version of the use case in Figure E.2 (Customer registration).

The description in Table G.2 has one extra event:

- AppsManager registers network listener for client applications.

The additional event was considered necessary due to the fact that it is possible for customer's client applications to be set up in most ways, and leave them "inactive" in terms of accessing resources across the network. This is possible mostly when the registration process has not been completed, especially when the control engineering company is still awaiting for some relevant information for the new customer.

Another notable change in the state-based description (Table G.1) is that events do not necessarily execute in the order in which they were written. Consideration of the pre and post conditions for each event, including the secondary actors, resulted in a different execution sequence as compared to the one that would have occurred if default order was assumed:.

1. OpsManager prepares fixtures list
2. MD ratifies fixtures list
3. OpsManager creates customer profile
4. AppsManager initialises customer's client applications.
5. AppsManager sets up client applications network information
6. AppsManager configures server applications
7. AppsManager links clients applications with server applications
8. AppsManager registers network listener for client applications
9. AppsManager stores client applications in customer profile

Figure G.9: Revised customer registration use case

Some observations were made as result of augmenting the initial use case (Figure E.2) with states. These are explained as follows:

Client applications must be initialised (event 8, Figure E.2) prior to network information for the clients being set up. Initialisation means determining whether the application has an assigned user (customer) and whether the details for an operator from the customer's company are available for any subsequent contact. In short, the states enabled the rigorous reasoning about the implications of each step happening in relation to other steps, and the actual monitoring processes.

Hence, the use of state-based information alone resulted in the addition of one event and the re-ordering of events' sequence (see Figure G.9).

Example 4

Consider the state-based description in Figure G.1 edited in EducatorTool:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Customer Registration

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	OpsManager	prepares fixtures list	initial	listReady	MD	waiting	listReady
2	MD	ratifies fixtures list	listReady	listRatified	OpsManager	listReady	listRatified
3	OpsManager	creates customer profile	listRatified	profileCreated	AppsManager	waiting	profileCreated
4	AppsManager	sets network info for client applications	clientsInitialised	networkInfoSet			
5	AppsManager	stores client applications in customer profile	listenerRegistered	clientsStored			
6	AppsManager	links clients applications with server applications	serverAppsConfigured	linkingDone			
7	AppsManager	configures server applications	networkInfoSet	serverAppsConfigured			
8	AppsManager	initialises customers' client applications	profileCreated	clientsInitialised			
9	AppsManager	registers network listener for client applications	linkingDone	listenerRegistered			

Figure G.10: Customer registration use case (edited in EducatorTool)

The participating developers observed that this is an important part of the new initiatives. Further scrutiny if the description by way of applying enaction was conducted.

During the enaction, the occurrence of the event where client and server applications are linked together (or clients associated with respective servers), it was observed that the MD needed to act at that point to add the new customer into MD's list of the existing customers. The event concerned is shown in Figure G.11:

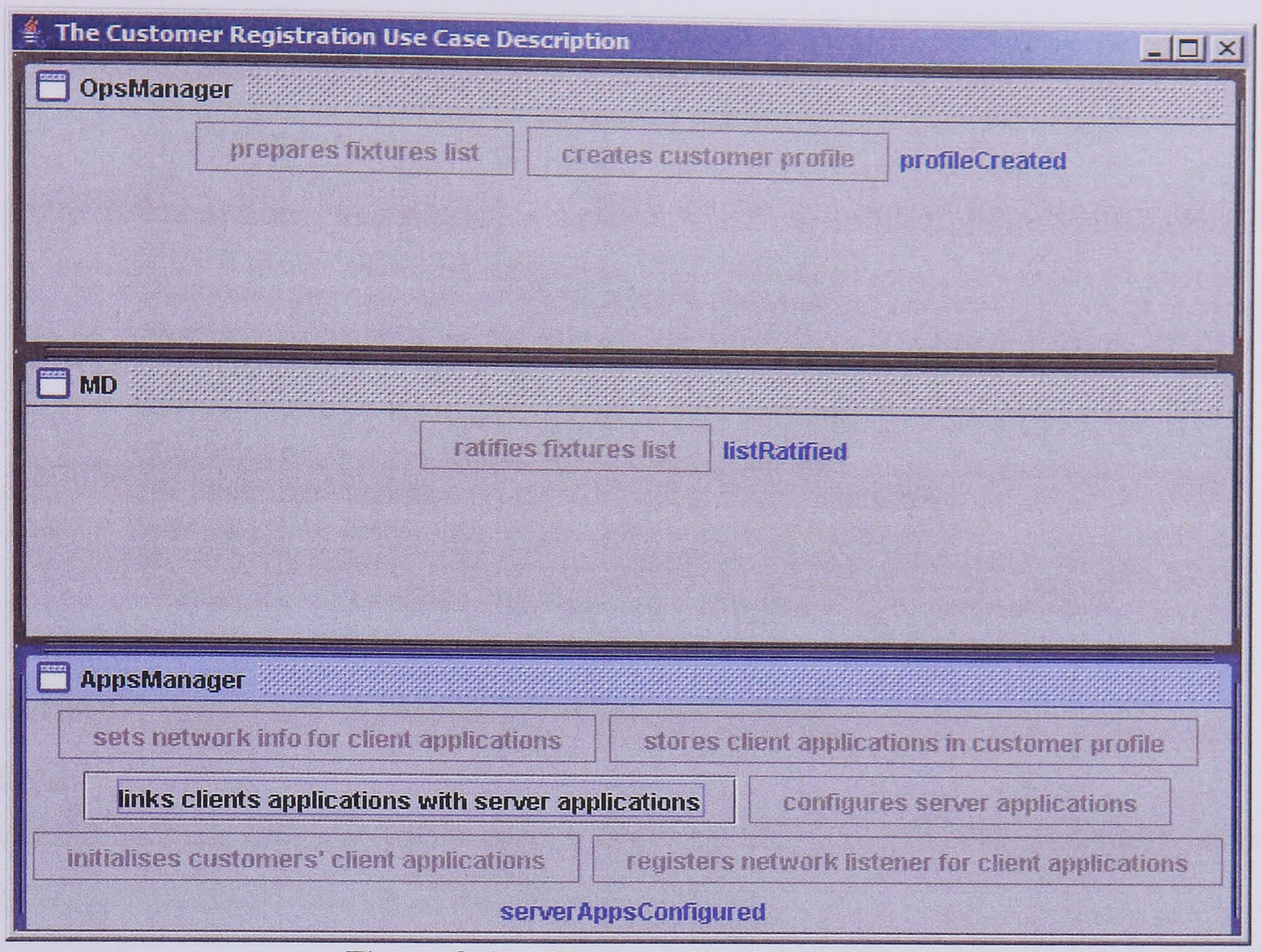


Figure G.11: clients-applications linkage

An additional issue brought forward after considering the additional event mentioned above, was the need to assign a control engineer as a custodian for the new customer. This assignment is done by the operations manager, who considers the best suited engineer based on the customer's business, and whether the engineer has been supporting similar customers before.

A revised use case description, based on these two arising issues is shown on Figure G.12:

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description: Customer Registration

ID	Primary Actor	Event	Precondition	Postcondition	SecondaryActor	Precondition	Postcondition
1	OpsManager	prepares fixtures list	initial	listReady	MD	waiting	listReady
2	MD	ratifies fixtures list	listReady	listRatified	OpsManager	listReady	listRatified
3	OpsManager	creates customer profile	listRatified	profileCreated	AppsManager	waiting	profileCreated
4	AppsManager	sets network info for client applications	clientsInitialised	networkInfoSet			
5	AppsManager	stores client applications in customer profile	listenerRegistered	clientsStored			
6	AppsManager	links clients applications with server applications	serverAppsConfigured	linkingDone			
7	MD	adds new customer to the list of existing customers	linkingDone	customerAdded	OpsManager	listRatified	customerAdded
8	OpsManager	assigns an engineer to the new customer	customerAdded	engnrAssigned	AppsManager	linkingDone	engnrAssigned
9	AppsManager	configures server applications	networkInfoSet	serverAppsConfigured			
10	AppsManager	initialises customers' client applications	profileCreated	clientsInitialised			
11	AppsManager	registers network listener for client applications	engnrAssigned	listenerRegistered			

Figure G.12: Customer registration - revised after enaction

The output of the enaction produced a revised description that participants agreed to (see Figure G.13).

Customer Registration

Enaction Output
OpsManager prepares fixtures list
MD ratifies fixtures list
OpsManager creates customer profile
AppsManager initialises customers' client applications
AppsManager sets network info for client applications
AppsManager configures server applications
AppsManager links clients applications with server applications
MD adds new customer to the list of existing customers
OpsManager assigns an engineer to the new customer
AppsManager registers network listener for client applications
AppsManager stores client applications in customer profile

Figure G.13: Enaction output for the customer registration use case

K: Additional (analysis) examples from the workshops

Example 1

Consider the Validate operator details use case of Figure F.3.1. The concerned student produced the following state-based version of the initial description:

Primary Actor	Event	Precondition	Postcondition	Secondary Actor	Precondition	Postcondition
Operator	selects system login function	initial	loginSelected	System	idle	loginSelected
System	prompts operator to enter login details	loginSelected	detailsRequested	Operator	loginSelected	waiting
Operator	enters their login details	waiting	detailsEntered	System	detailsRequested	detailsObtained
System	searches operator details on database	detailsObtained	searchingDB			
System	validates operator details	searchingDB	detailsValidated			
System	confirms operator details ok	detailsValidated	confirmed	Operator	detailsEntered	confirmed
Operator	logs into the system	confirmed	logged	System	confirmed	functionPrompt
System	requests operator to choose function to execute	functionPrompt	waiting	Operator	logged	functionPrompt
Operator	selects a	functionPrompt	functionSelected	System	functionPrompt	functionSelected

	function (e.g. create report)					
System	records the selected function into usage log	functionSelected	recorded			

Table K.1: state-based version of validate operator details use case

Discussion

Table K.1 shows the state-based version of the use case in Figure F.3.1 (Validate operator). The description in Table K.1 has three extra events:

- System requests operator to choose function to execute.
- Operator selects a function (e.g. create report).
- System records the selected function into usage log.

The student argued that the additional events were necessary to ensure that authentic usage of the depot system monitored. For instance, each operator needed to be able to indicate why they were logging into the system (by selecting a system function). The system function would then be automatically recorded for security purposes. It was not considered desirable to just login and not use the system in anyway as that would leave it vulnerable to unauthorised use.

Hence the revised description is shown below:

1. Operator selects system login function.
2. System prompts operator to enter login details.
3. Operator enters their login details.
4. System searches for operator details on database.
5. System validates operator details.
6. System confirms operator details ok.
7. Operator logs into the system.
8. System requests operator to choose function to execute.
9. Operator selects a function (e.g. create report).
10. System records the selected function into usage log.

Figure K.1: Revised validate operator details use case

Example 2

Consider the Create transaction reports use case shown in Figure F.3.3. The student performed an enaction of the default use case to try and find out whether there were any specification issues that could be further elaborated. Figure K.2 shows the description edited in EducatorTool

Educator :Use Case Enaction

File Use Case Actor Conditions Enact Tools CP Words Help

Description Create report

ID	Primary Actor	Event
1	Operator	longs into the system
2	Operator	selects create report function
3	System	prompts operator to select report type
4	Operator	selects a report type (e.g. movements)
5	System	prompts operator for report dates (from and to)
6	Operator	enters the dates
7	System	prompts operator to select report format
8	Operator	selects report format (print or view)
9	System	produces the requested report

Figure K.2: Create transaction report

To reiterate, the description in Figure F.3.3 (and K.2) describes and sequence of steps that the user of the depot system would follow to produce a report with the system. Enaction of the above default use case description was done by the student to further scrutinise any dependency issues without application of state-based information.

During enaction of the description in Figure K.2, the initial enaction dialog is shown in Figure K.3:

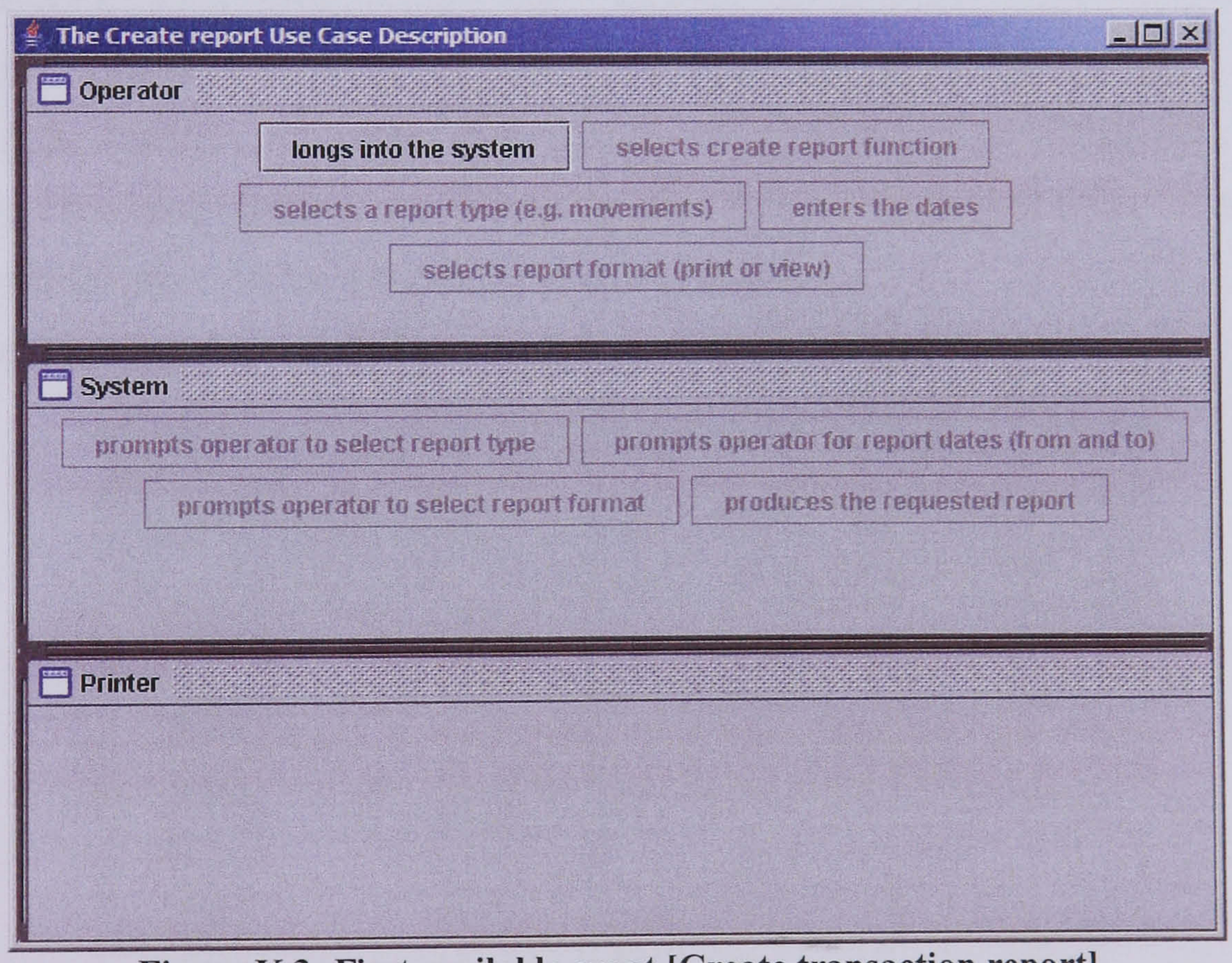


Figure K.3: First available event [Create transaction report]

Default enaction entails stepping through the events in the order they are written and the next available event is "Operator selects create report function":

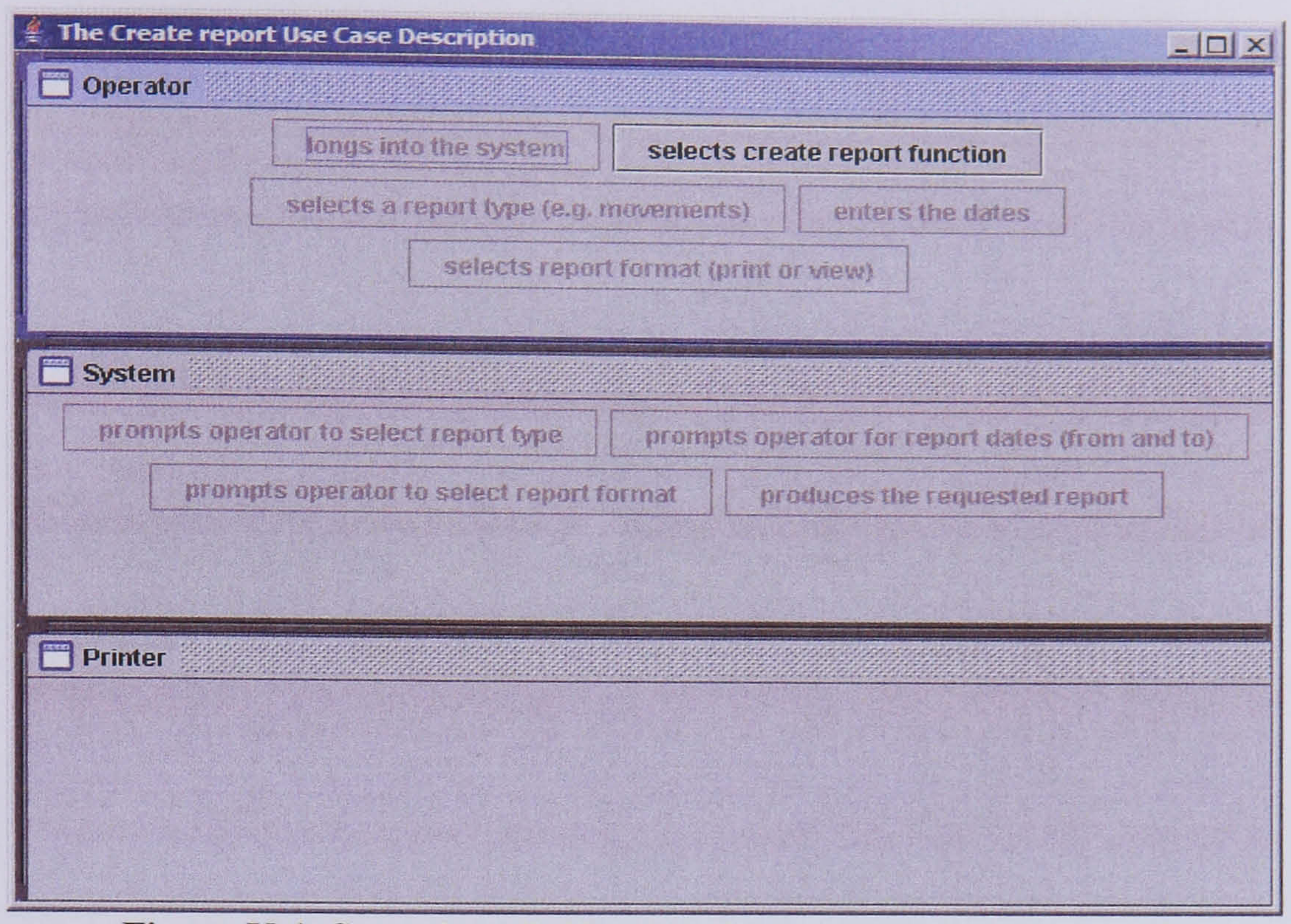
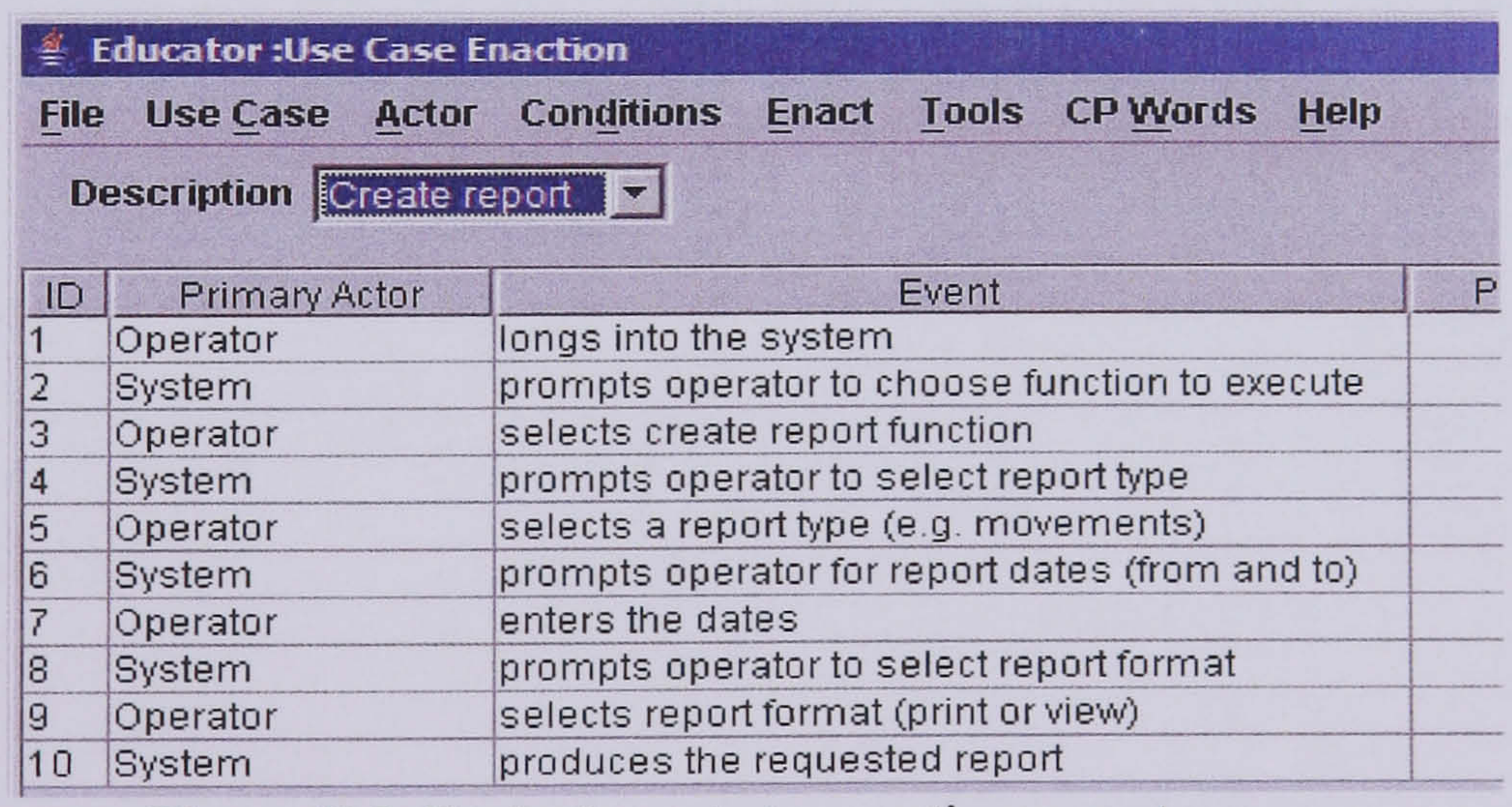


Figure K.4: Second available event [create transaction report]

The student observed that it is likely that after login (see Figure K.3), the system has a range of options (besides creating report) that the user (operator) can choose to execute. For instance, the student argued that prior to printing a report, the operator might want to perform some data processing (e.g., summarise all depot entries for the day) so that such processed data is reflected in the report. This meant that the student considered an addition event for the *System* actor:

- System prompts operator to select a function to execute.

Hence a revised Create transaction report use case is shown in Figure K.5:



ID	Primary Actor	Event	P
1	Operator	longs into the system	
2	System	prompts operator to choose function to execute	
3	Operator	selects create report function	
4	System	prompts operator to select report type	
5	Operator	selects a report type (e.g. movements)	
6	System	prompts operator for report dates (from and to)	
7	Operator	enters the dates	
8	System	prompts operator to select report format	
9	Operator	selects report format (print or view)	
10	System	produces the requested report	

Figure K.5: Revised create transaction report use case

The output of enaction for the description in Figure K.6 is shown below:

Create report

Enaction Output
Operator longs into the system
System prompts operator to choose function to execute
Operator selects create report function
System prompts operator to select report type
Operator selects a report type (e.g. movements)
System prompts operator for report dates (from and to)
Operator enters the dates
System prompts operator to select report format
Operator selects report format (print or view)
System produces the requested report

Figure K.6: Enaction output for revised create report use case