A lightweight state-machine for validating use case descriptions

John Kanyaru and Keith Phalp
School of Design, Engineering and Computing, Bournemouth University, UK
(jkanyaru, kphalp)@bournemouth.ac.uk

Abstract

This paper presents a tool to provide an enaction capability for use case descriptions. Use cases have wide industry acceptance and are well suited for constructing initial approximations of the intended behaviour. However, use case descriptions are still relatively immature with respect to precise syntax and semantics. Hence, despite promising work on providing writing guidelines, rigorous validation of use case descriptions requires further support.

One approach to supporting validation is to use enaction. Indeed, enactable models have been used extensively within process modelling to clarify understanding of descriptions.

Given the importance of requirements validation, such automated support promises significant benefits. However, the need to produce formal descriptions, to drive enaction, is often seen as a barrier to the take-up of such technologies. That is, developers have traditionally been reluctant to increase the proportion of effort devoted to requirements activities. Our approach involves the development of a lightweight state-machine, which obviates any need to create intermediate formal descriptions, thereby maintaining the simple nature of the use case description.

Hence, this 'lightweight' approach, which provides an enaction capability 'for minimal effort', increases the likelihood of industrial take-up.

1. Introduction

The software engineering community has long understood the importance of stakeholder involvement in validation of requirements and specifications [1, 2, 3]. Tool support may help to bridge the communication gap between engineers and customers, by providing appropriate models to enhance shared understanding. This paper focuses on providing tool support to enhance the validation of use case specifications.

Use cases have gained widespread adoption mainly due to their presentation (with natural language) of system behaviour from the viewpoint of its users. In particular, the use case description details the interaction of users (actors) with the proposed system. This viewpoint is crucial, especially when validating the adequacy of the specification. However, UML use cases have several shortcomings that curtail their expressiveness in specifying behaviour. Whereas the use of natural language makes use cases easy to construct and understand, it is also a weakness, since natural language specifications can be ambiguous. The UML specification of the use case does not offer any guidelines for writing use case descriptions [4, 5]. Whereas authoring guidelines are a crucial issue for use cases, our focus is the inability of use cases to describe state-dependent requirements. There are no provisions in the UML specification for describing interdependencies amongst use case events. Indeed, the UML specification (see [6]) states that every use case should express a sequence of interactions that are independent of any other use case. That is, use cases specifying the same system must not communicate or have associations with one another. UML however, describes three types of relationships between use cases: Generalization, <<include>> and <<extend>>. Generalization relates general use cases to special-case ones. 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. <<extend>> on the other hand means that the extending use case is inserted, at a designated extension point, if a particular condition is true. It is clear that the property of independence of use cases cannot hold where decomposition of a system is crucial to its understanding. UML does not model interactions between actors, that is, communication and any associations between actors in a use case or across use cases are not allowed. Intra-use case

dependencies such as "event E requires that event Q has been previously executed" cannot be expressed in UML. Moreover, inter-use case dependencies such as "use case A requires that use case X has been previously executed" cannot be formally expressed in UML. In reality, however, use cases and use case elements do interact. Indeed, other authors have noted that the independence rule is often flouted in industrial practice [2].

This paper presents an approach for creating behavioural descriptions of a system with state-based use cases. An extended structure of the use case description is proposed. This extension allows inclusion of both intra-use case and inter-use case dependencies, whilst also incorporating actor interactions. Simply put, the contributions of the paper are threefold. First, we describe a structure suited to authoring state-based use case descriptions that exhibit inter-relationships amongst constituent elements. Second, we provide a sound animation mechanism, which supports the authoring and prototyping of descriptions written in the proposed structure. Third, we provide support for grammar-check based on (the CP) rules to enhance writing of intuitive and comprehensible descriptions. Thus, we provide the basis for a common approach to use case authoring and animation so desirable during the early stages of requirements and specification.

2. Extended structure and adopted approach

The UML semantics of the use case suggest that the dynamic requirements of a system as a whole can be wholly expressed with use cases alone [6]; that each use case specifies services rendered to its users and the service is a complete sequence. This implies that after its performance, the use case in general will be in a state in which the sequence can be initiated again. However, the UML-inspired Rational Unified Process (RUP) models requirements as use cases without allowing for any intra-use case or inter-use case state variables. This is clearly contradictory and insufficient for systems where states play an important role in controlling crucial interactions. Thus, in order to express use case interaction issues, one must be able to define states that can be accessed and modified by use case elements. The UML use case allows for only two states global to the whole use case, that is, the condition that is true of the system before the use case starts (precondition) and the condition that is true of the system after the use case finishes execution (post-condition) [7, 8]. This means that states for each event are ignored, by presuming that the event dependencies are linear. Furthermore, the two global states do not show contextual states of actors as they interact to execute the various use case events.

To address the above shortcomings we re-define the use case structure to allow for inclusion of state-based information pertaining to each constituent event. We explain our new structure as follows: a global precondition for a use case is the state of the system (or that of one of the actors) before the use case starts to execute. A global post-condition is the state of the system (or that of one of the actors) after the use case executes. Each use case event has a pre-condition and a post-condition. An event pre-condition is a condition that must be true of that event or the triggering actor before the event is triggered, and an event's post-condition is a condition that is true of the triggering actor or the event after the event is undertaken. Thus, when an actor triggers an event, the actor moves from a certain pre-state to a certain post-state. If the event affects the state of another actor, then the affected actor also changes state in its own specific way. The passive actor is termed secondary actor while the triggering actor is termed primary actor. By allowing for state changes of both primary and secondary actors, we support the description of interactions between system users where such interactions exist. Thus, our approach is premised on including state information in use case descriptions, while keeping the use case notation simple. By adopting this approach, the work borrows heavily from the process modelling community where business processes are modelled using state-based descriptions of work processes (e.g. [9]). In short, our resultant use case description incorporates state-based information to enable rigorous validation of stakeholder expectations. A benefit of our approach is that there is no need to use any intermediate formal grammar for describing the states and interactions.

3. Describing software behaviour with stateless use cases

A use case is a partial story describing a circumstance of system usage and how the system behaves while serving its external users. Stakeholders can write their own partial stories thereby contributing their own view of the desired behaviour to those of other stakeholders.

Determining an accurate behaviour of the system must involve scrutinizing the validity of each use case

event in relation to others. This leads to the question: How should a set of use case events be related? What is the underlying semantics of the relating element? In other words, regardless of whether a use case is a generalization of another use case, is extended by or inserted into another use case, relating events local to a use case is crucial to determining the behaviour that would lead to the execution of successive events in the use case and any other related use cases. These issues are not considered at all in the use case diagram or stateless textual use case specifications.

For instance, consider a situation where an academic registrar interacts with lecturers who volunteer for courses to teach. Suppose that the registrar is also involved with students who choose the courses they wish to study. Moreover, the registrar has to prepare the list of courses that students can choose from. Thus, there are three actors involved (registrar, lecturer, and student) each undertaking a task suited to their needs or role.

Below is a use case diagram depicting the above situation:

Figure 1: Course registration use case diagram

Figure 1 above is visually appealing to both the users and engineers as it depicts high-level real world needs of the three external users. The problem with the use case diagram generally is that it does not detail how the various elements relate with one another [10]. For example, in Figure 1 above, we cannot tell how the actions of the lecturer affect those of the student or registrar or vice versa. It is important for example for the registrar to know when they can prepare the course list, or indeed for students to know when it is right for them to choose the courses they wish to study. In other words, actors in use case must be aware of each other's context of actions to be able to proceed with events that interest them. Behavioural descriptions of systems normally involve determining how the system's constituent elements interact. In use case modelling terms, these elements are actors (system users) and the use cases. However, this is not possible in the UML use case model as it is not possible for the system (or part of it) to access the internal state of an actor. When modelling the behaviour expected of a system, it is important that a rich description is made, including interaction between actors [11]. For instance, engineers, customers and users must be able to determine and model possible behaviours of the lecturer depending on the actions of the student or registrar. However, UML cannot model such rich interactions because it forbids associations between actors. Indeed, [5] argues that the use case diagram is not expressive enough and should not be used on its own to describe software behaviour. This is a correct assessment; however, we observe that the nature of the textual use case suggested in [5] does not give any detail regarding the determination of interaction issues. For example, a textual specification depicting Figure 1 above is:

1. Lecturer volunteers for course
2. Registrar prepares course list
3. Student chooses course to study

If the domain is familiar and the problem is simple, then it might be easy to outline the sequence of events in a scenario like the one above. However, most software projects are complex and involve many interacting participants whose interaction patterns might vary. Stakeholders often find it hard to articulate their views clearly and sometimes need different combinations of tests to validate their understanding. It, therefore, becomes unclear whether the third event is dependent on the first, the second, neither or both. That is, can a student choose a course before any lecturer volunteers to teach it? Additionally, is it might be illogical for lecturers to volunteer for courses after students have made their choices? It is in such circumstances that knowing merely the actors and the events might not help solve the problem. Developers need further information to enable the teasing out of problem domain issues that will help clarify the interdependencies amongst use case elements.

The following section outlines our approach. Examples are used to demonstrate how the approach works.

4. Relating use case elements with states

4.1 Semantics of states

Our model assumes that states are crucial properties of actors, which determine whether the actor may

invoke (or participate in) an event. This means that for an actor to be able to invoke an event, it must be in a state matching that event's precondition, and after the event is successful, the actor changes state to the post-condition of that event. In other words, the textual use case specification is comprised of one central theme, the event, which in turn is accessible to the triggering actor (primary actor) and secondary (passive) actor. [Note that for simplicity we describe interaction between two actors, but the principle holds for any number]. A state based use case description need not be written in time-order as it mimics a state-machine whose order of event execution depends on the states of invoked and available events. Thus, default ordering of events is not presumed, as the order of execution is based solely on states. We have developed an application, called Educator, to help in authoring use case descriptions in this fashion. (An early version of the tool is described in [12]). Educator has functionality for including states and their amendment, to allow for testing of different combinations of possible behaviours. Stakeholders with differing views on the desirable behaviour can brainstorm on what they think is acceptable before any attempt on a working model is made.

Consider the course registration description in section 3 above. A state-based description of that scenario can be explained as follows: before the lecturer volunteers for any course, the lecturer is at an **initial** state, and after the lecturer has volunteered for the course they wish to teach, the lecturer is in the state **courses agreed.** This means that the lecturer has agreed with the registrar on some courses that the lecturer will teach. This implies that the registrar is involved in the lecturer's volunteering to teach some courses. We might suppose that the registrar is also at some **initial** state before reaching the **courses agreed** state with the lecturer. After this, the registrar can now prepare the list of courses available, so that students can choose the courses to study based on what is available. Thus, the registrar moves from **courses agreed** state to **list done** state. The student would have been at some **initial** state and will have to know that the course list is done so they also arrive at the **list done** state. In the end, the student can choose the courses they wish to study, thus, the student moves from **list done** state to **courses chosen** state.

The following shows the above description edited in our application:

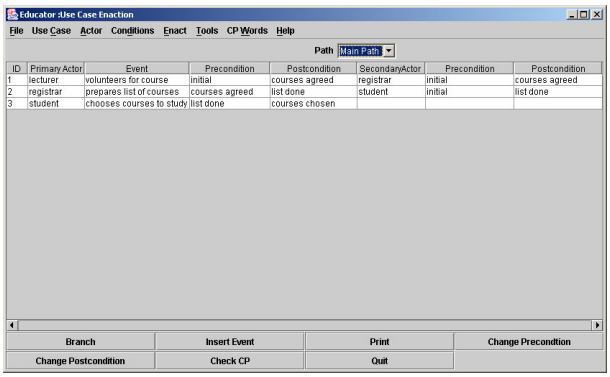

Figure 2: Course registration state-based description

Users can now animate the above description to view the various state changes of the actors as they perform their respective events. For instance the first animation window is:

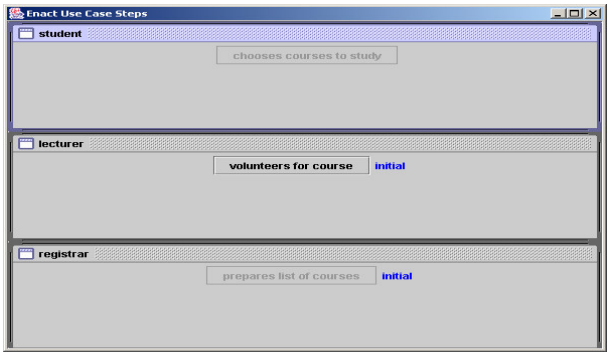

Figure 3: First animation window (registrar is secondary actor)

Figure 3 shows the lecturer in an initial state; ready to volunteer for courses to teach, and the registrar in the initial state. If this is not the desired behaviour, that is, if say it is the student who interacts with the lecturer when the lecturer is about to volunteer for course, then the description should be edited so that the student is the secondary actor for the first available event.

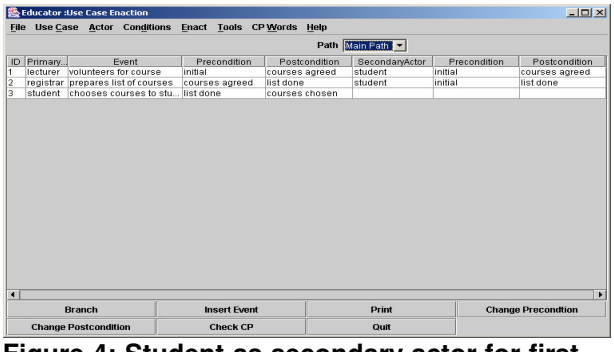

Figure 4: Student as secondary actor for first event

The first animation window shows that the student (not the registrar) is at initial state this time.

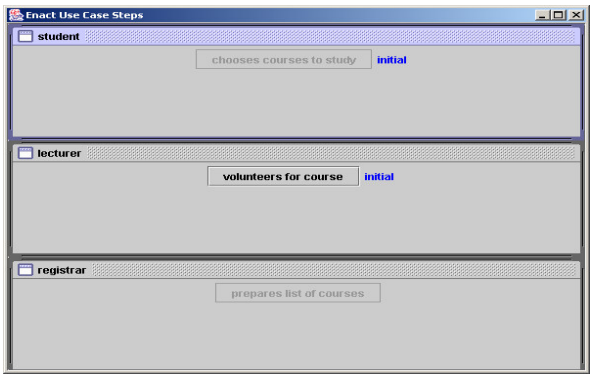

Figure 5: Student is now at initial (not the registrar)

It could be that after completing the animation, the users decide that the next available event is that the student will choose courses to study. To do this, the user simply changes the states of that event accordingly, that is, the precondition of the student's event will be matched with the post-condition of the lecturer's event. This is the essence of the tool usage. Author descriptions, animate them to clarify stakeholder expectations and revise the description (by changing states, rewriting events or adding new actors or states) to match the expectations of system stakeholders. The states represent conditions that are true of the respective actors for their respective contextual events. Ultimately, states are problem-domain specific rather than impositions of any programming language.

4.2 Adopted syntax

[13] observe that since use cases are written in natural language, their quality depends on disciplined use of natural language. Some researchers (e.g. [4, 5, 14]) have suggested grammatical structures to be followed for disciplined writing of use case descriptions.

[5] argues that a use case event should be simple and suggests the format:
Subject… verb… direct object … prepositional phrase.
For example, the first event in the use case constructed earlier would be "the lecturer volunteers for course". This format is similar to that suggested in [4] and we adopt it in our approach because it is simple and intuitive.

A recent study by [4] resulted in seven use case authoring rules, termed CP rules. These rules are an improvement of those of [14]. The main problem with most of the guidelines is their lack of automated support. We have incorporated some of the CP rules in our application to enforce authoring of comprehensible descriptions. For example, the first CP rule requires each use case sentence to appear on its own numbered line and we have supported this in our application (see Figure 2 for example). The second CP rule demands that the author should avoid the use of pronouns (e.g. he, she, and it). We support this by allowing the user to construct a working dictionary that contains disallowed words:

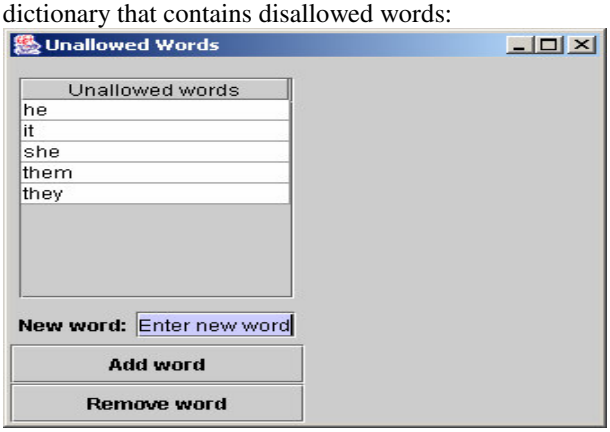

Figure 6: An example list of disallowed words

If the user writes a sentence that has any of the un-allowed words, then the application notifies the user of that and provides an option to re-write the sentence.

CP rule 6 requires that all verbs be in present tense format. We have provided functionality to check that users do not use words in the past tense. For instance, if the user writes the first event as "lecturer volunteered for course", the application reports the possible tense usage.

Figure 7: Feedback on CP rule 6

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, we have provided the functionality of constructing a dictionary of allowed words to ensure such words do not appear to flout CP rule 6 when used in the description:

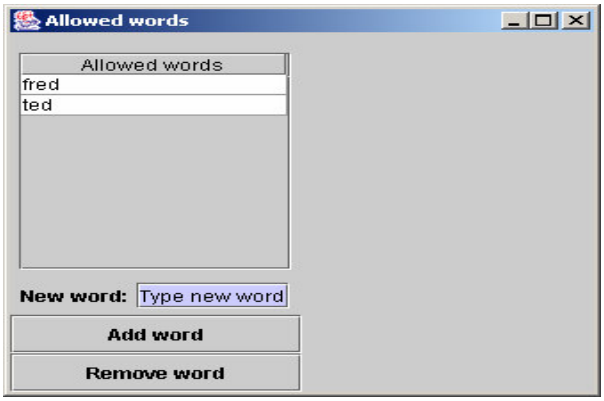

Figure 8: An example list of allowed words

The section below outlines our experiences on testing our approach and the animation environment with a group of MSc (Software Engineering) students.

5. Experiences so far

As indicated earlier, the aim of our animation tool is to aid the participation of both the customer and the requirements engineer in the validation process. An additional objective is to ensure the language used in making descriptions is as easy for customers as it is for the engineers. We avoid the use of formal grammars such as process-algebraic instantiations. When customers and general users participate in requirements validation tasks, the requirements engineer can easily show them the implications of the behaviour contained in the authored use case descriptions. Ultimately, participation of the customer is paramount and the increased understanding on the part of both parties has been the major strength of enactable process models [9, 15].

The animation tool was given to Masters students for use in software engineering projects. The students had experience on use cases via previous lectures and had been exposed to state-based use case descriptions and process models. We also discussed and demonstrated to the students how the tool works before leaving them to use it on their assignments. The eventual feedback was gathered with the help of structured questions. The general thrust of the questions was twofold:

1) To determine whether the tool was easy to use.

2) To determine whether the tool helped users with the clarification of the requirements for their software engineering projects.

During the gathering of the feedback, the authors were able to see demonstrations of the students use case descriptions edited and animated with the tool. Half of the students found the tool easy to use whereas the other half did not find it easy enough. However, all the students agreed that it was better using the tool for authoring and animation rather than use a notepad or typical word processor. Half of the students strongly agreed that using the tool for animating use case descriptions caused them to think a lot more about the appropriate behaviour of the final software for their projects and that they revised their descriptions over and over again to correct and test different implied behaviours. The other half of the students agreed the animation support helped them test their requirements before embarking on successive development efforts.

5.1 Further work

Thus far, the research has succeeded in defining an alternative approach to behavioural modelling with use cases. The supporting application works efficiently for use cases and their constituent elements. Alternative paths for the use cases are also taken care of where users might need to define alternative paths for accomplishing their tasks with the help of the system. We have resolved the issue of intra-use case relationships and tests undertaken indicate positive results. An industrial case study is also planned. Currently, we are working on supporting inter-use case relationships to ensure distinct use case descriptions can be associated where execution of one use case might impact or require the execution of another use case by the same or different actor.

Additionally, we intend to provide initial approximations of resultant classes based on edited use cases. We argue that once engineers have validated descriptions with other stakeholders, it is possible that the engineers can take a first-cut analysis of the description to derive classes for subsequent design. Our initial attempt on this is based on the UML concept that many actors form classes, this way, we provide functionality to name classes based on actors, and assign class properties based on actors' states and performed events.

5.2. Related work

Requirements Engineering (RE) research focused on use cases has been growing tremendously since the OMG standardised UML in 1997. This is not surprising given that the specification of appropriate system behaviour is the focal point of RE and use cases are the part of the UML that are solely geared to behavioural specification.

Interests in use case-oriented research have followed two distinct routes. Researchers have investigated use case structural issues pertaining to their authoring (e.g. [4], [5], and [14]); and have focussed on providing automated support for behavioural modelling (e.g. [16], [17], and [18]).

The central argument in the first group of researchers is that writing comprehensible use case descriptions is crucial to understanding the expectations of the stakeholders ([19]and [20]). The problem is that the UML does not give any guidance on how to write effective use cases [5] and it is left to the author to write descriptions in their own desired way. It is apparent that many engineers and customers may write ambiguous or incomprehensible descriptions if no writing guidelines are followed. The work of [14], and [4] which suggest various use case construction guidelines could be beneficial if adopted by industry. We argue that most of the guidelines suggested should be supported by an automated authoring application to enforce them (we are supporting some of the CP rules guidelines suggested by [4]); otherwise it will be difficult to advance them for use in practice.

The concept of augmenting textual descriptions with formality, execution and animation has gripped research and industry for well over a decade. [18] outlines an approach termed play-in/play-out where executable scenarios of system usage are played-in by users by help of a graphical user interface of the intended system. The scenarios are captured as message sequence charts (MSCs). The play-out process constructs a working model based on the scenarios played-in by users. In other words, the developer constructs a dummy interface of the application, lets users play with it (while MSC construction proceeds behind the scenes). Consequently, users get a view of their actions from a generated working model via the play-out process. Simply put, the engineer makes users execute their own use cases as if it was indeed a working system.

[17] describes executable use cases for a pervasive health care system based on three tiers. The first tier consists of informal descriptions of the elicited requirements and the relevant parts of the problem domain. This is done with the UML-style use cases. The second tier is a formal model providing execution capability of the descriptions made in the first tier. Various modelling languages (e.g. UML statecharts, activity diagrams, or even a programming language) can be used in this tier. The central theme for any language chosen for this tier is that it should be able to model states and the actions that can be performed in each state. This being largely a technical tier, it is a preserve for developers. The third tier is a graphical representation of the second tier to enable users to animate the formal model to clarify whether the model meets their expectations.

[16] describes an approach for verifying the behaviour of concurrent systems by using scenario-based state-machines that produce a combination of all possible behaviours of the designed system components. This can be viewed as more of a verification effort rather than validation as the focus is on testing whether inter-component communications match the derivative specification. The scenarios showing the communications (or apparent behaviour) are constructed using a dialect of process algebra called Finite Sequential process (FSP) and the written scenarios can be animated in a tailor made application called Labelled Transition System Analyser (LTSA).

[9] describes RolEnact, a tool for creating and enacting state-based business process descriptions. The descriptions are created using a formal language, Enact which is expressive enough to model roles, their states and the processes they take part in. [15] describes a graphical approach based on pi-calculus to make graphical models of software systems. The aim is to reduce the effort on the part of modellers when creating and reasoning about the behaviour exhibited by the models.

The above approaches are similar to ours in one important aspect, that is, the quest to involve customers in the validation of software requirements. However, a fundamental difference exists between our approach and these others. The LTSA three-tier model and the play-in/play-out approaches take a formal approach to modelling behaviour from natural language use cases with formal languages. The states of the resulting description are based upon the implementation bias of the deployed formal languages. Hence end users have relatively little involvement in creation or identification of flaws in such descriptions. On the other hand, our lightweight state machine is simple enough to allow a greater variety of stakeholders to be involved in production and validation of the description. Thus, users themselves are capable of identifying appropriate

states because of their apparent knowledge of the problem area. In doing this, we do not detach the description making process.

6. Conclusion

The specification of software behaviour is a complex task prone to subtle errors that can have serious ramifications. Behaviour modelling has proved to be successful in helping tease out and correct flaws in design artefacts; however, it has not had similar success in requirements specification. The two main reasons for this are as follows: firstly, constructing models for behavioural analysis remains a difficult undertaking requiring considerable expertise. Secondly, the validation benefits appear at the end of the (often lengthy) construction effort, and users often have little involvement in the construction of models.

The approach described in this paper, together with the supporting tool, demonstrates that it is feasible to produce enactable models of use case descriptions without delving into any formal specification techniques. That is, we produce behavioural models of use cases that are amenable to automated analysis for clarifying stakeholder expectations. The essence of obviating any need to create intermediate formal descriptions is to maintain the simple nature of the use case description. The supporting application enables the prototyping of state-based descriptions thus providing an early 'feel' of what stakeholders would get from the resulting software.

In our view, seamless development starting from the early stages of requirements elaboration should involve general users and customers. Hence, the tool allows for rigorous validation of use case descriptions, whilst still maintaining crucial user involvement.

7. References

1. 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.
2. Pfleeger, S., *Software engineering: theory and practice*. 2nd edition ed. 2001: Prentice Hall.
3. Leonhardt U, et al. *Decentralised process enactment in a multi-perspective development environment*. in *Proceedings of the 17th international conference on Software engineering*. 1995. Seattle, Washington, United States: ACM Press.
4. 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.
5. Cockburn, A., *Writing effective Use cases*. 2001: Addison-Wesley.
6. OMG, *Unified Modelling Language Specification version 1.5*. 2002, OMG; http://www.omg.org.
7. Stevens, P. and R. Pooley, *Using UML:Software Engineering with objects and components*. 2000: Addison Wesley.
8. Scheneider, G. and J.P. Winters, *APPLYING USE CASES: A Practical Guide*. 1998, Reading: Addison-Wesley.
9. Phalp, K., et al., *RolEnact: role-based enactable models of business processes.* Information and software Technology journal, 1998.
10. Chonoles J. M and J.A. Schardt, *UML 2 For Dummies*. 2003: Wiley Publishing, Inc.
11. Jackson, M., *Problem Frames: Analyzing and structuring software development problems*. 2001: Addison-Wesley.
12. Phalp, K. and K. Cox. *Using Enactable Models to Enhance Use Case Descriptions*, in *International Workshop on Software Process Simulation Modelling (in conjunction with ICSE 2003)*. 2003. Portland, USA.
13. Kulak, D. and E. Guiney, *Use Cases:Requirements in Context*. 2000: Addison-Wesley.
14. Rolland C and B. Achour, *Guiding The Construction of Textual Use Case Specifications*. 1998, CREWS Report Series.
15. 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.
16. Kramer J., J. Magee, and S. Uchitel, *Synthesis of Behavioural Models from Scenarios.* IEEE Transactions on Software Engineering, 2003. **29**(2).
17. Jorgensen B. J and C.Bossen, *Executable Use Cases: Requirements for a Pervasive Health Care System.* IEEE Software, 2004.
18. Harel D., H. Kugler, and R. Marelly, *The Play-in/Play-out Approach and Tool: Specifying and Executing Behavioral Requirements.* The Israeli workshop on programming languages and Development Environments; Haifa, Israeli., 2002.
19. Cox, K., *Heuristics for Use Case Descriptions, PhD Thesis*, in *School of Design, Engineering & Computing*. 2002, Bournemouth University, UK.
20. Glinz, M. *Improving the Quality of Requirements with Scenarios*. in *Second World Congress on Software Quality*. 2000. Yokohama.