Using a formal method to model software design in XP projects

Christopher Thomson
University Of Sheffield
c.thomson@dcs.shef.ac.uk

Mike Holcombe
University Of Sheffield
m.holcombe@dcs.shef.ac.uk

***Abstract*— A software engineer depends on an established best practice toolbox in order to build quality products. Extreme programmings practices tear away many well known techniques to aid developer efficiency; however this can mean that a project is poorly documented. This is a particular problem in our taught courses where all the developers leave on a yearly basis! Extreme X-Machines are a formal model that we have successfully introduced as a documentation method. As a lightweight and change resistant method it is easy to use, but it is also formalised allowing for a succinct and accurate representation of a software system. We have found that the developers benefit from both the description that they make of the software and any that are supplied as documentation during maintenance projects.**

***Index Terms*— X-machines, metaphor, system documentation, lifecycle management, formal model, Extreme X-Machine, XXM, Extreme Programming.**

I. INTRODUCTION

AT the University of Sheffield for the past five years we have been studying the way that software is developed by student teams working on large projects with external clients. These real business problems provide a tough challenge which our students usually rise to. These projects have proved a fertile ground for research with a large number of papers published based on the data collected . Many of these have compared Extreme Programming (XP) and more traditional development methodologies.

We teach two courses that use XP: The Software Hut is a second level module lasting 12 weeks. The Genesys project is a fourth level module lasting 24 weeks. Both courses take projects from external industrial clients who pay for the development of a system that they require. Over the past three years we have collected data from just over 60 separate development projects undertaken by these students.

In teaching XP one of our major concerns has been to use as many of the practices as possible. As with many XP teams we have not been able to implement them all for practical purposes, so for example, as the students only work 15 hours a week on the projects, having an on site customer is unfeasible, instead regular meetings are encouraged. One practice that we thought should be feasible, though we have found hard to implement, is that of a system metaphor. We define the system metaphor in XP as an overview of the proposed system. In many cases this can take the form of a white board, or some shared common knowledge, however this does not provide a lasting or necessarily useful definition of the system.

A particular challenge for us in this respect is that on the Genesys project in which students change abruptly at the end of the academic year. This has in the past caused significant difficulties where systems are not adequately documented. The system metaphor is usually described as a way to have an overview and controlling vision for the whole system. This is a valuable commodity in a system that lacks a large amount of documentation; however this metaphor is hard to phrase if it is to be used to introduce an outsider into a project. Our primary goal during this research was to find a model that allowed the students to define the metaphor for a general class of software systems.

There was also a secondary goal for the model, to help with the general management of the system. Story cards in XP are used to define the system requirements and are implemented in small iterations; however they do not define (intrinsically at least) how they relate to each other. This leaves two problems; firstly how do we know which stories are interdependent if they change and secondly how do we test these relationships.

This then gives three areas that we consider in this paper relating to the metaphor. Firstly how can it be used as an effective knowledge transfer device? Secondly what is the relationship between stories and

ISSN 1109-9305

change? Lastly how can we test relationships following change?

II. Extreme X-Machines

The model that we have chosen to use is that of the Extreme X-Machine (XXM) [1]. This model is essentially similar to standard X-Machine (SXM) models which have been well published elsewhere [2]–[7] in particular Holcombe and Ipate have applied the model to XP test generation [8]. The XXM model has been designed to be both simple and flexible to use with some useful features. XXM models can be used in a similar way to use cases, although their structure is more closely related to other types of state diagram.

Perhaps the easiest way to visualise an XXM model is to consider the final software release of the project. Typically a XXM in this situation can be formed by running the application and noting the events that occur due to user actions. Typically the first screen to appear will contain some kind of menu giving the user options for the current session, the menu screen is represented by a state and each option by a transition (possibly to this state or another state) which is labelled by the name of the action and click to define the user event. This process can then be repeated for each screen that is presented to the user.

The model is expressed visually with circles representing screens and arrows between them representing transition functions. In the event that a single transition goes through to two different screens, then a square state is inserted into the diagram to represent an internal processing step. Whilst each element of the diagram can be annotated to describe the process taking place, a typical XP project will only name the elements.

A simple system that illustrates the main points of an XXM diagram is shown in Fig. 1. This diagram specifies a system that users can log into and view a secret account, new users can register if required. The diagram should be read from the start state on the left. The inevitable function display login executes and the login window is displayed. Here the user can login, when they click login the click(login) function is executed which initialises the internal state validate user. If the user exists the exists function is executed and the Display Account window is shown. When they logout click(logout) is executed and the XXM terminates, or returns to a calling XXM. Of course there are other paths through this XXM that can be described in a similar way.

By limiting ourselves to around seven states per page these diagrams can be kept simple and easy to maintain. If more detail is required, the screens, functions and states can be broken down in separate more detailed diagrams, to form a model hierarchy. The depth of the hierarchy is arbitrary and dependant on how much detail about the system that the author wishes to include.

We have found that there is typically a strong relationship between a story and a number of functions, where the screens are often the start and end points for the story, as well as the functions. The functions can then be implemented directly in the control code for the system. This allows a strong binding between the user documentation and the code, which can be lacking in some projects.

From the students perspective the XXM works well as it encourages the students to design their systems from a clients perspective, a technique normally only used by experts [9]. The visual approach is also useful with studies showing that diagrams are essential when the developers prior knowledge of the system is low [10].

If SXM had be used to represent these systems then in some respects they would have been more complicated, i.e. input in every day applications is very large and defining how it is processed could be onerous. And in others they would have been simpler, i.e. they do not represent functions where there is no input to determine which function to be executed. However a primary benefit of the XXM is that they fit better with the clients view of the system by focusing on screens and the processing required to move between them.

III. Students XXM

On our Genesys undergraduate course we exclusively use XP as the development methodology. This has advantages for the students as it allows them to start producing systems in much short timer spans as the length of the course is limited to 24 weeks. This means that fairly complex systems can be developed. However projects often run over year groups; and XP can lead students to poorly document their systems. We identified a major problem: the students found it hard to link the code to an overall understanding of the system this was caused by the small amount of high level documentation. We now use XXM to fulfil this role.

Our first successful use of this technique was with the Software Hut project in 2003/4. Many of the students on the course had problems visualising what a XXM might look like, they also felt that the models tended to be over simplistic. However some teams were more adventurous and presented far more complex diagrams. The diagram set collected and shown in Figs. 2-5 is such an example. Over time the diagram changes to reflect the system as it evolves. These changes are mainly related to the flow model of the user interface.

Students on the Genesys course in 2004/5 were encouraged to look at the previous years projects and

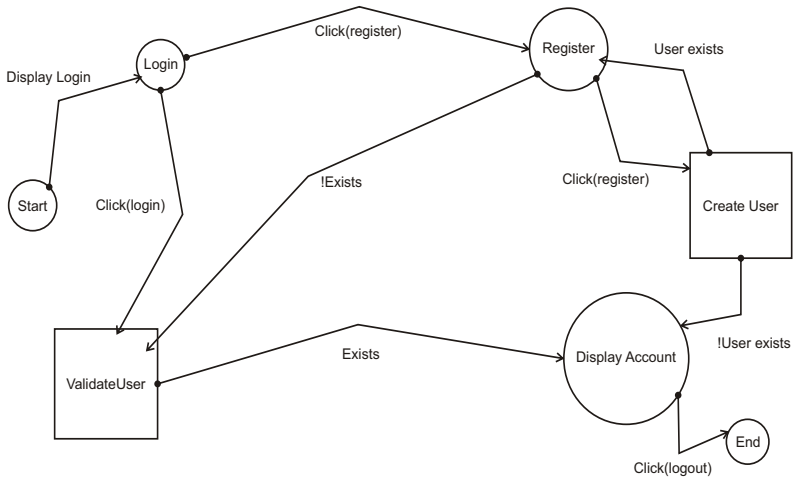

Fig. 1. An example of a XXM diagram. This represents a simple login system into a secure account area of a system.

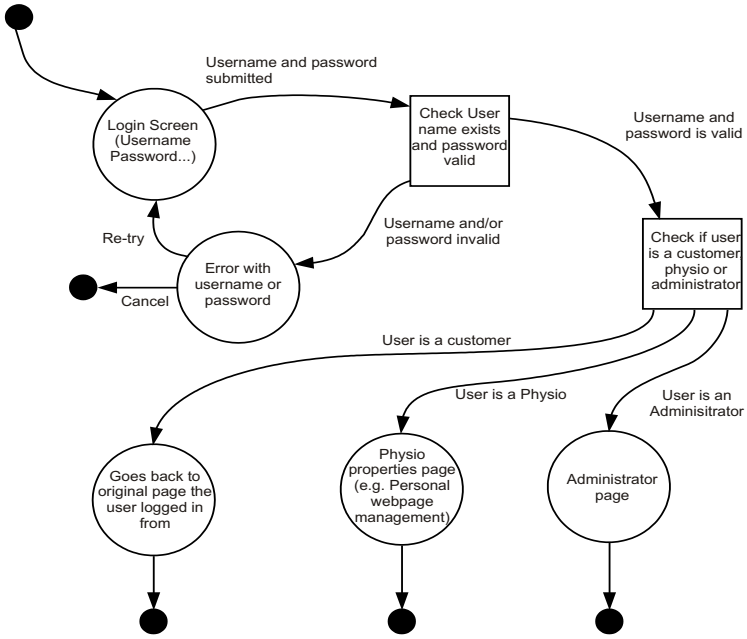

Fig. 2. XXM model for login, by Team 3 software Hut 2004, version 1.

rate them for understandability. This encouraged them to use XXM more proactively in this session, as they quickly found that they provided a useful clear outline to the project without trawling though code or story cards. Perhaps the most important factor here is that there is a clear relationship portrayed within the whole system. This has led the students to develop the XXM side of their story cards to make this relationship clear during the project as well.

IV. Managing and Describing Change

XP encourages us to embrace change and in our experience this seems to be a very wise move. Frequently we have observed projects where business models change or simply the clients interpretation of the business. If these changes are not implemented then the final solution will be less welcomed by the client. Change however is not without risk, and it is worth considering how change affects an XP process. There follows some case studies in which a XXM model may have helped

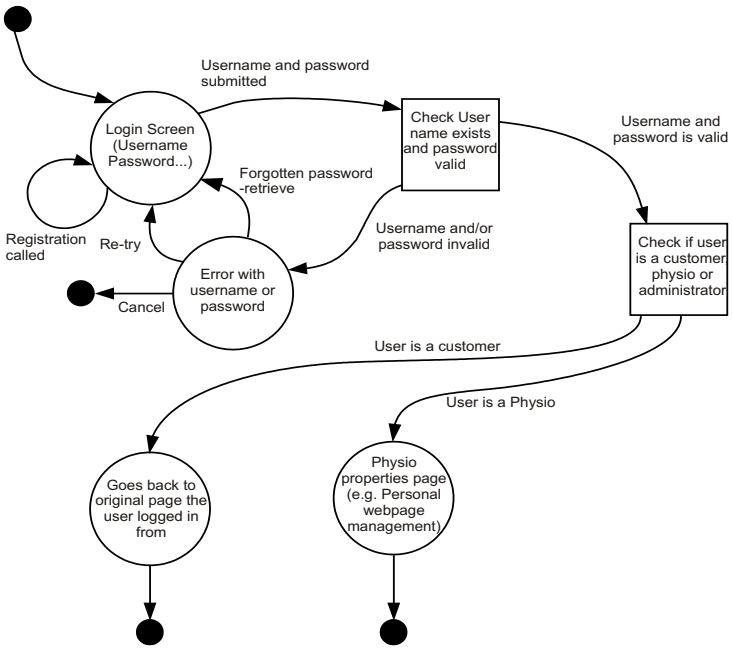

Fig. 3. XXM model for login, by Team 3 software Hut 2004, version 2.

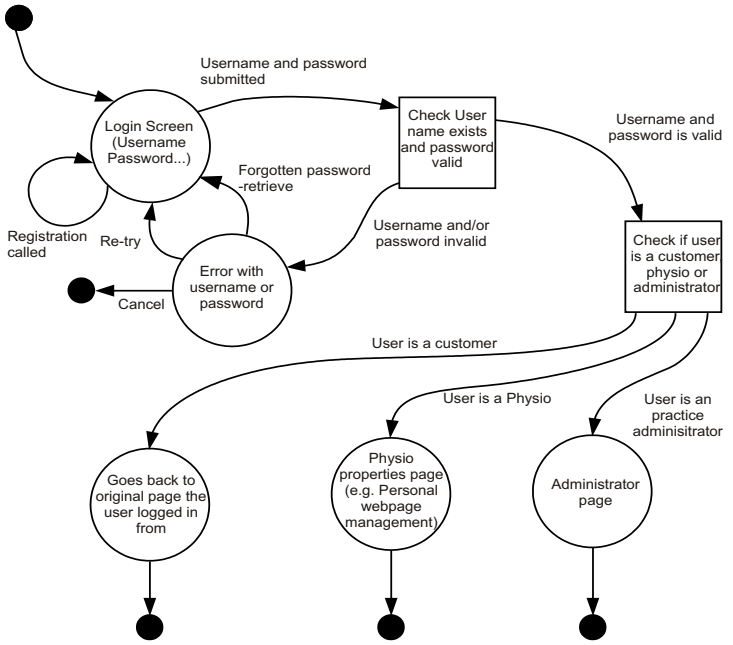

Fig. 4. XXM model for login, by Team 3 software Hut 2004, version 3.

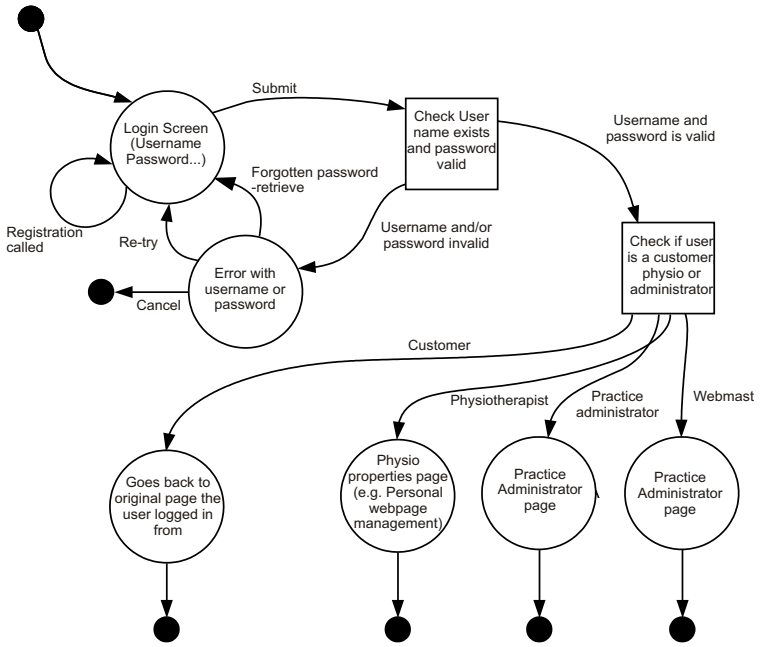

Fig. 5. XXM model for login, by Team 3 software Hut 2004, version 4.

to manage change related issues.

Let us first consider the Genesys group and the txt4offers 2003 project. The students were interviewed with seven questions; you can find their responses in the Table I. This project was a particularly hard one to complete because the client was unsure of his requirements and changed his mind frequently about key issues through most of the build. The students were encouraged to use a XP like development approach and found this useful; however this conflicted with the ideas of their client who generally wanted the project to be run his way.

In general on this project they found that the core of the system frequently changed. Interestingly this core was primarily concerned with interfacing external applications. The client had a general direction in mind, the actual specifics turned out to be far more challenging than he had imagined. This resulted in multiple changes to core parts of the program; this was in terms of both logical progression and external components used. This necessitated several major re-writes.

The main motivation for change was provided by the client. The changes experienced were focused on the creation of multiple prototype solutions so that the client could assess them particularly in regard to third party services. This generated extra work in two main areas; these were in coding and communication. The students found the communication most challenging as they learnt to continually talk to their client to confirm what direction he would like the system to go in. However this was identified also as the riskiest part of the project, it allowed the client to change his mind frequently.

The benefit of using an XXM in this type of situation is that it is fairly resistant to low level change. It models instead the visible functionality without worrying about how the code is internally designed.

The most obvious source of client driven change is the alteration of story cards. The story cards used by our Software Hut 2004 students capture more information than many to allow us to identify this change. An investigation against the electronic copies of the story cards reveals that the changes fall into two main categories functional changes and non-functional changes; these are shown in the Tables II and III. This is fairly logical because we are talking about the function of a program; however the detailed groupings are perhaps of more interest.

The changes related to functional change show a large bias towards the changes to do with testing. This is no surprise as the XP technique dictates that the tests define the system and give the programmer a specification to work towards, when the tests pass, the software is complete. These seem to be related to the changes in the description, which whilst not purely functional show a bias. The conditions category reveals interesting information; it defines how a story interacts with the other stories and is not normally captured by XPs lighter

TABLE I

GENESYS TXT4OFFERS INTERVIEW ABOUT CHANGE IN THE PROJECT (SUMMARY)

Question	**Response**
What percentage of your time has been spent handling changing requirements?	40% - SMS (3) PC (2) Parser (1) Web (1) DB (2) The motivators were: Prototyping (revise and writing again), Financial considerations, exploration of ideas (features and new ideas). It was hard to know what the client wanted at the start of the process he had cloudy vision. The students felt that clear vision would have been better, or if they had been allowed to do their own thing. They felt that the following indicators would spot projects that could suffer from high rates of change: Client cant come to ideas, does not accept advice, respect opinions or trust people/methods. This seems to be a psychological effect.
What activities have you done as a result of changing requirements?	Throw out old code (some classes as much as 3 times). Client liaison/communication. More team meetings. Stress release activities. Motivation and counselling. Third party communication. (Code 60%, meetings 20%, documentation 20%).
What techniques have you found to be useful when dealing with changing requirements?	Wait a week before writing documentation. Communications, regular phone calls, often every day. They speculate that the following may have also helped: Simulating the system. Thinking about the software engineering process mythology to use from the start.
How is the process that you adopted different from a traditional one?	Constant client feedback followed by changes. Rushed into coding without design (fundamental changes were added, maybe better design would have helped). Defining the vision. Doing many roles/motivation/system understanding.
What have you done that you have found most useful?	Client management
What changes have caused you the most problems?	Third party changes- they affect many parts of the system. Understand what parts of the system are fundamental.

TABLE II

FUNCTIONAL CHANGES COLLECTED FROM ELECTRONIC STORY CARDS FROM ALL TEAMS IN THE SOFTWARE HUT 2004.

Item changed	**Potential reasons**
Tests - Rewrite	Related to changes in the flow of execution in the program.
	Refinement of test cases to define extra functionality.
	Redefinition of tests.
	As a result of changes in interfaces.
Tests - Add	Additional cases not previously thought of.
	Quality attributes and functional tests added at the same time.
Tests - Exchange	Revised to reflect user experience and observable results.
Description - Redefine	When functional tests are added.
	Occasionally completely redefined.
Description - Remove	The reason for this was unclear.
Description - Expand	As a result of adding functional tests.
	For more information.
	Data to be collected had changed.
Conditions - Add	The reason for this was unclear.
Conditions - Redefine	It would appear that the location within the program of the modelled task changed.

TABLE III
NON-FUNCTIONAL CHANGES COLLECTED FROM ELECTRONIC STORY CARDS FROM ALL TEAMS IN THE SOFTWARE HUT 2004.

Item changed	Potential reasons
Description - Redefine	To reflect the users experience.
Description - Define	To make it more explicit.
Description - Elaborate	To make it more explicit.
Quality Attributes - Add	This was previously undefined.
Conditions - Replace	Changes in event/attributes and tests to suggest alterations for a more user centred approach.
Tests - Reword	To make it more explicit.
Entities - Rename	To reflect the users terminology.

story cards.

It is perhaps of interest that the non-functional requirement changes form only about a third of the total. This suggests that both the teams and the clients were a lot clearer of these requirements from the start. The majority of the changes here are to do with either renaming things or recentering previously expressed ideas in more user centred language.

Many of the functional changes are behind-the-scenes changes. This means that they will not be visible on any top level XXM. This is a useful feature as it means that the diagrams need only be updated infrequently. In the case that the flow of the system changes then this diagram will change. This is also useful as it allows developers to identify the knock-on affects of their changes.

Lastly Thomson spent some time attending the independent team meetings that were held by team 10 in the Software Hut 2004. The data collected from this process can be used to see how the team went about making decisions and dealing with change. Most of the early change that was experienced in this project will be familiar to any seasoned software developer. The changes are mainly to do with the team coming to an understanding of the system.

The project started with each client giving an overview of the problem that they wanted to solve, the teams created an initial informal design of the system based on this. After meeting the client individually for the first time and discussing the problem, a large number of conceptual changes were made to the system as they modified their expectation to the clients reality. After this point very few high level changes were made.

At the first meeting the client stated that he did not require a login to the system and all users should have equal access. This was against what the team thought. However later in the process the client reversed this decision. This suggests that the user over the period of the project gained knowledge about the process in his company, and as a result of this realised that a more advanced system was required. This type of discovery seems important in all the teams that we have considered.

The team were using XXM extensively as part of this project and found that by completing them at the same time as story cards they were able to visualise how the system fitted together. This allowed ideas about the structure of the system to be displayed and refactored quickly at the earliest stages of the project, as well as increasing the quality of the communication between team members.

V. IMPLEMENTING XXM IN A PRACTICAL COURSE

As we introduced XXM models to the student projects it became clear that it would be necessary to provide a CASE tool that gave at least minimal support in the creation of XXM like models and the generation of simple test sets. The department had recently been awarded two grants by IBM to develop XP related plug in components for the Eclipse framework as part of the Genesys project. Therefore we decided that it would be valuable to create the XXM case tool as an Eclipse plug in .

The Eclipse framework provides a basic integrated development environment, which is functionally enhanced for individual programming languages through the use of plug in components. It is currently hot property with a lot of industry involvement from various heavyweight companies such as IBM and Oracle. Plug in components are easy to install into the framework, with a fair amount of scope available to lock a plug in into the existing user interface. Whilst we found that developing plug in components was substantially harder than installing and using them we were pleased with the overall result.

The main goal of the plug in was to make the creation of and maintenance of XXM models easier as well as providing useful additional functionality. XXM can be easily expressed both graphically and textually so the plug in provides both views of the data through a taviewported editor. This allows competent users to quickly assemble a machine manually whilst

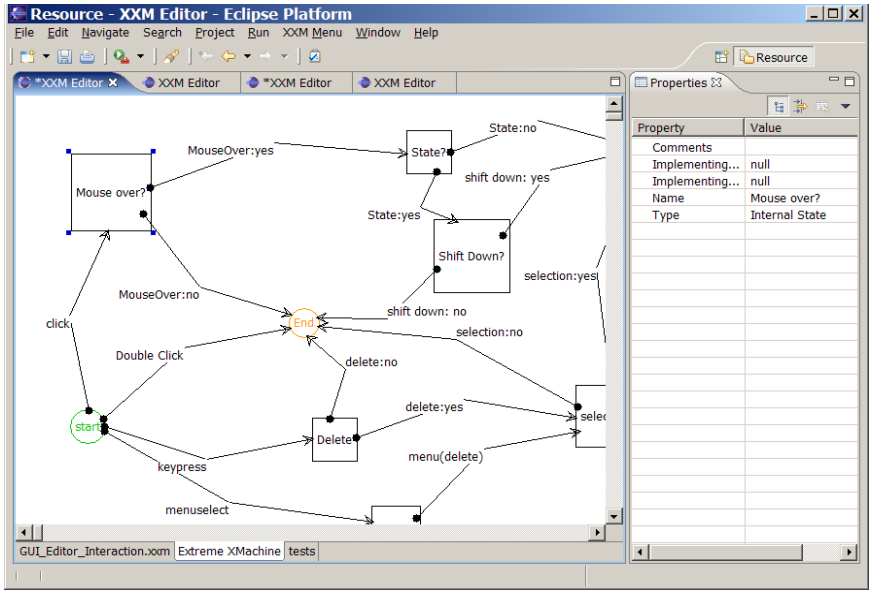

Fig. 6. A screen grab of the XXM plug in running in Eclipse.

less experienced users can use the graphical editor. Fig. 6 shows a screen shot of the editor in action.

The GUI Editor features a simple WYSIWYG interface that allows the diagrams to be drawn in a familiar point and click fashion. The editor ensures that almost impossible to draw an invalid diagram from the connectivity viewpoint as functions may only be paced between states. Further more users are warned about inaccessible and dead-end states by colouring them red. However no checks are made on the validity of any of the labels this allows the tool to be used flexibly when specifying different software.

The textual representation is a simple form of XML that specifies the various items in the diagram; one file is created for each diagram. The format captures the details of the visual representation (such as location) as well as the details of the XXM specification. Essentially each state and function has a tag in the XML file with the attributes of that tag defining the attributes of the state or function. As XML only allows the definition of tree like structures and XXM are represented by a graph structure the links between states and functions are defined by attributes of the function which define the ids of the start and end states. Whilst this is not ideal as ids must be maintained, it helps to avoid casual errors when naming two states with the same name. This allows for some basic error checking when the model is parsed, allowing the user to correct manually created XXM before editing in the GUI editor.

In order to make the tool more useful a test generation algorithm was also added. The algorithm transverses the XXM graph and documents a set of tests that will ensure function path coverage of the system; see the example in Figs. 7 and 8. In the future we hope to extend this following the method previously described in [8]. In the case that there are several XXM describing the whole system organised hierarchically the tests can be run against each part if the system independently. In most cases these tests have been used manually by the developers: the developer navigates through the program to the first screen in the test and ensures that data can be entered to reach each subsequent screen. However in some cases the developers were able to create JUnit tests [11] that automated this process. This can be used in a similar way to X-Machine testing [7] in order to make sure that an implementation has a similar internal XXM to our specification; this test set is less strong in order to make it easier to use. However we have yet to show the mathematical strength of this test set. The tests give several sequences of functions that together exercise the implementation when combined with appropriate input data. This is capable of showing if the implementation has an internal XXM that at least exhibits the functionality in the model; however it cannot show if there is any additional functionality.

Having used the tool with the students we found that the quality of their models increased as anticipated. We also found that the quality of their testing improved as they became aware of what they needed to test in the code. Whilst these tests are not as strong as other forms of X-Machine testing their increased usability suggests that they are more likely to be used.

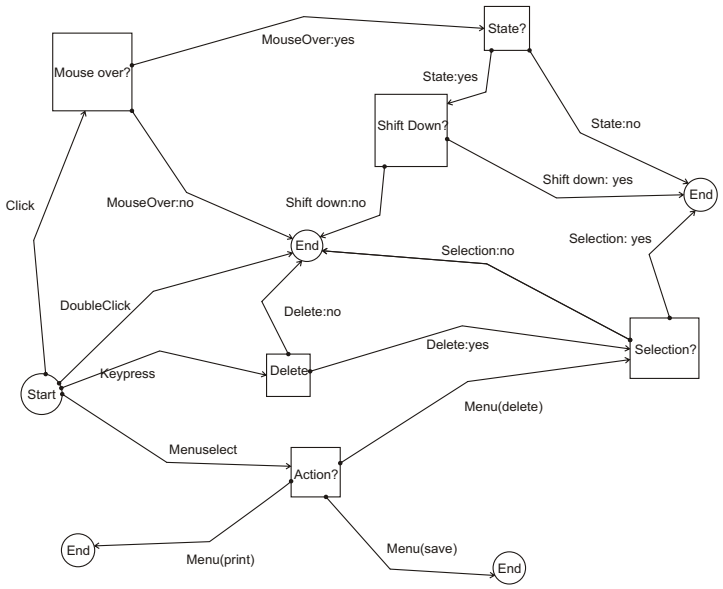

Fig. 7. An example XXM that shows some GUI interactions.

```
(start) > click > (Mouse over?) > MouseOver:no > (End)
(start) > click > (Mouse over?) > MouseOver:yes > (State?) > State:yes > (Shift Down?) > shift down: no > (End)
(start) > click > (Mouse over?) > MouseOver:yes > (State?) > State:yes > (Shift Down?) > shift down: yes > (End)
(start) > click > (Mouse over?) > MouseOver:yes > (State?) > State:no > (End)
(start) > menuselect > (action?) > menu(print) > (End)
(start) > menuselect > (action?) > menu(delete) > (selection?) > selection:yes > (End)
(start) > menuselect > (action?) > menu(delete) > (selection?) > selection:no > (End)
(start) > menuselect > (action?) > menu(save) > (End)
(start) > Double Click > (End)
(start) > keypress > (Delete) > delete:no > (End)
(start) > keypress > (Delete) > delete:yes > (selection?) > selection:yes > (End)
(start) > keypress > (Delete) > delete:yes > (selection?) > selection:no > (End)
```

Fig. 8. The test sequences generated by the eclipse plug in for the XXM in Fig. 7. Each line corresponds to a single test case and a functional path through the system. Items in brackets refer to the states, and between these are functions that need to be executed to show that the path exists.

> "We found the extreme machines very useful as they helped us to gain a fairly simple view of the whole system at a relatively early stage of the project. When a mistake/better design is found, we simply update the corresponding XXM. This process was quite efficient since the changes required the XXM were usually minimal, and as long as the XXMs were kept up to date, anyone in any doubt had a quick reference to what the system was supposed to be."
>
> Team 10, 2005
>
> "Extreme X-Machines were central to the development process, allowing fast turnaround of design documents in parallel with the equally rapid eXtreme implementation. The simple, structured nature of the X-Machines ensured our time was focused on producing relevant, useful plans in lieu of more complex design approaches."
>
> Team 6, 2005

Fig. 9. Students' views on the XXM diagrams were very positive.

VI. Conclusions

In the Introduction we defined three goals for our research which have been achieved. We identified the

XXM model as a way to effectively transfer knowledge about the system between teams. The model seems to be simple to maintain and easy to understand, allowing the story cards to be linked together conceptually. By creating an overview model of the system new team members can rapidly pick up the architecture and relate this to the other documentation and code. This is of most importance in projects like ours where a team will have to maintain a system that they themselves did not originally develop.

During this research XXM models have been used successfully to visualise small scale commercial systems. It is the authors’ opinion that as with other X-Machine models they could be applied to a variety of systems [2]–[7]. However there are no provisions in the current model to deal with either realtime constraints or concurrent systems, these omissions would necessarily restrict the systems that they could be applied to.

Whilst XP encourages practitioners to embrace change, the tools provided to manage change are limited. This study has highlighted that change occurs in projects is sometimes reflected in the story cards but not always. However the change is important as it can potentially cut across several areas of the system. This study has considered some projects were change has occurred, in some cases XXMs were used successfully and it appears that XXMs would be useful in the other cases as well. XXMs provide a model that documents this change and by generating test sets from that model we can easily test that change. This may be particular useful if we are using automated tests as it will help us to identify which of our tests need to be updated.

The students had very positive comments about the XXM diagrams when questioned at the end of the project. Two typical views are shown in Fig. 9. This success was due to the implementation of the Eclipse plug in which helped the students to draw valid diagrams. One of the greatest benefits identified by the students was the simplicity of the notation, and its resilience to change.

We plan to use XXM on further projects to validate its usefulness. We are doing that as part of our on going research; however the results would be more conclusive if others were to try it on their own projects.

ACKNOWLEDGMENT

The data for this work would not be possible with out the co-operation of the students on our courses, or the other teaching staff, including Simon Coakley and Dr. Marian Gheorghe. Sharifah Lailee Syed-Abdullah gave us advice on the cognitive process used with the XXM models. Christopher Thomson is supported by an EPSRC studentship. This project is part of the Sheffield Software Engineering Observatory.

REFERENCES

[1] Thomson, C, Holcombe, W., ”Applying XP Ideas Formally: The Story Card and Extreme X-Machines”, In Dranidis, D., Tigka, K. eds: Proceedings of 1st South-East European Workshop on Formal Methods, Thessaloniki, Greece, pp 57-71, South-East European Research Centre, 2003.

[2] Eilenberg, S., ”Automata, Languages and Machines”, Vol. A, Academic press, N.Y., 1974.

[3] Holcombe, M., ”X-Machines as basis for dynamic systems specification. Software Engineering Journal”, Vol. 3, No. 2, pp. 69-76, 1988.

[4] Holcombe, M., ”An integrated methodology for the formal specification, verification and testing of systems”, Proc. EuroSTAR 93, London, 1993.

[5] Holcombe, M., Ipate, F., ”Correct Systems: Building a Business Process Solution”, Springer Verlag Series on Applied Computing, 1998.

[6] Ipate, F., ”Theory of X-machines and Applications in Specification and Testing”, Ph.D. Thesis, University of Sheffield, .1995.

[7] Ipate, F., Holcombe, M., ”An integration Testing method which is proved to find all faults”, Intern. J. Computer Math. Vol. 63, pp. 159-178, 1997.

[8] Holcombe, M., Ipate, F., ”Complete Test Generation for Extreme Programming”, In Eckstein, J., Baumeister, H., eds.: The Extreme Programming and Agile Processes in Software Engineering: 5th International Conference, XP 2004, Volume 3092/2004 of Lecture Notes in Computer Science. Garmisch-Partenkirchen, Germany, June 6-10, pp. 274 - 277, Springer-Verlag Heidelberg 2004.

[9] Robilliard, P. N., ”The Role of Knowledge in Software Development”, Communications of the ACM, vol. 42, pp. 87-92, 1999.

[10] Schnotz, W, ”Commentary: towards an Integrated View of Learning from Text and Visual Display”, Educational Psychology Review, vol. 14, pp. 101-102, 2002.

[11] Object Mentor, Inc., ”JUnit, Testing Resources for Extreme Programming”, www.junit.org, 2006.