Applying XP Ideas Formally: The Story Card and Extreme X-Machines

Christopher Thomson* and Mike Holcombe

The Department of Computer Science, The University Of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield, S1 4DP. UNITED KINGDOM
{c.thomson, m.holcombe}@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/∼cthomson

Abstract. By gathering requirements on story cards extreme programming (XP) makes requirements collection easy. However it is less clear how the story cards are translated into a finished product. We propose that a formal specification method based on X-Machines can be used to direct this transition. Extreme X-Machines fit in to the XP method well, without large overheads in design and maintenance. We also investigate how such machines adapt to change in the story cards and propose how this could be further enhanced.

1 Introduction

Extreme programming (XP) is an agile development methodology that is in use by a number of software houses around the world. Kent Beck's [1] book "Extreme Programming Explained: Embrace Change" has done much to promote its use and describe the 12 tenets of XP known as practices. Four practices were of particular interest to us, firstly the "Planning Game" that describes an informal collection of requirements based around story cards which describe the users needs. Secondly "Metaphor" is meant to describe the proposed system in a simple way. Thirdly "Simple Design" is used to signify both that no unnecessary features should be added and that there is no formal design. And lastly in "Testing" unit and system tests are continually written and updated, normally before the code is modified for some feature.

In agile processes in general and in XP in particular formal design is normally ignored, so applying formal methods is a challenge. An agile formal method must be as adapt at coping which change as its host method. However a formal model would be useful, particularly if it were to help us model the system as a whole and help us identify suitable test sets. In XP story cards provide the overview of the system and provide the most suitable base for an applied formal method.

Story cards may also benefit from this as with a more formal structure they would be easier to mine for information, allowing part of a project to be constructed automatically. However this would make story cards much harder to understand, therefore we need to build formalism into a straightforward English vocabulary.

* Research supported by a EPSRC studentship.

Such changes however could upset members of the XP community, so we need to ensure that whilst we can generate formalisms from some information there remains the option for users to define their own way of doing things, preferably within some form of formal framework. Ultimately such provisions could lead to an natural language processing or understanding problem.

2 Informal Models

2.1 Story cards

The first problem to address is of what a story card represents and the granularity of a story card. It seems that there is no easy solution to this problem as some systems will fit better into high level stories, whilst others will use many low level stories. There is also the problem of deciding how many possible actions should be present in a story! This suggests that we might need a hierarchy of stories, in some cases these will take the form of physical story cards which decompose a more complex story into manageable chunks. In others this will take the form of virtual story cards, where several actions are described on a single story card, (in both cases these could be on paper or on-line).

A story card can then be said to represent: an entity, doing a job, which might have several stages. These then can be linked up to other entities doing related jobs, so in effect distributing the responsibility of some task, forming a complete story. This collection of stories which forms the system is called a story board as together they describe the functionality of the system.

2.2 Story Boards

Using story cards we are able to specify some of the functions of the system, but as yet we are unable to see how all these stories are composed together into a whole. In order to do this we can use a story board. Story boards are typically used when putting together the sequences of action in a film, in this context they are used to string together stories, and make interdependencies explicit.

The story board itself is coarse grained, so each story may have several attached stories, though it will not be possible to determine how these directly interface with this story. Therefore once stories have been linked the exact interactions need to be noted within the story. In many cases this should be easy to determine, and in cases where it is not it may show that the story is imprecise or errant.

This type of idea could be presented in two ways. Firstly on paper the stories could be physically pinned to a notice board so that strings could be attached between stories. Secondly they could be represented in a computer system by an overall diagram with the story cards hyper linked together.

3 Formal models

3.1 X-Machines

Having defined our system using story cards and the combined functionality using a story board it is now necessary to flesh in further details using a formal specification.

We promote a method based on an X-Machine (Eilenberg Machine), but enhance the technique so that it can be used to model real applications more effectively.

The X-Machine is based on the concept of a finite state machine however labels are replaced by functions, which in turn may be decomposed into further X-Machines. Originally was it defined by Samuel Eilenberg in 1974 [2] and it has been shown to have the computational power of a Turing machine. This was later used by Holcombe in 1988[3] to provide a hierarchical machine, and then expanded to include theorems for testing [4]. A subclass of the X-Machine, the Stream X-Machine (SXM) was later defined, in order to guarantee that sensible test cases might be produced, which led to a reliable theory of testing [5] [6] [7].

An X-Machine is composed of three key components: State, Memory and Functions. Functions act as the transitions between state and act on the memory. In an X-Machine state is only defined in terms of process and not in terms of memory. This allows loops of various types to be constructed easily.

3.2 A Practical Model

X-Machines can appear fairly abstract when we try to apply them to a real world software engineering task, so we motivate an approach based on story cards and boards which is more obvious. Each story describes a number of tasks, which will fall into three types, tasks which make a decision (and therefore a logical branch), tasks which do something and tasks which work on a data model. We will model these functions, with Extreme X-Machines (XXM), function flow diagrams (FFD) and data processing models (DPM) respectively. This will allow us to model the systems formally and as a benefit construct sensible test sets.

The translation of branching points in the story cards to XXM is simple, by using the predicate transition logic (an XXM specifies that from any state, if a predicate is used to guard a transition then the anti-predicate must reside on another transition). If it is a multiple branch then this can probably be written as a series of single branches in most cases. All other predicates and inputs which might act at the state are assumed to loop back into the state if not specified.

For the case of the action like tasks we use function flow diagrams. These define in detail the processing steps involved and any outputs of the system. Each step in the flow diagram can be considered a different function. This form of design allows us to produce automatically a decent form of modularised code, however this does not go as far as to proscribe where a function must reside.

Tasks which specify data storage operations (this may include searches), are represented by a data processing model. In the resultant system may take the form of something similar to a database. As databases are resistant to change in general (requiring a large amount of resource to change) we try to separate these access routines from the rest of the system [8].

How can we decide which parts of our stories translate to these different models? In the general case this is very hard to do automatically, however by hand this should be a fairly simple task. Should we want an automatic translation then we can establish a simple system which identifies keywords associated with different types of task.

To ensure that a tool understands the stories, we could limit the English used to a formalised subset of the language.

3.3 Extreme X-Machines

An Extreme X-Machine allows us to model the control structures of a story and ultimately the whole system. We might want to do this because it allows us to see in great detail where we expect the execution of commands in a finished system to lead us. Using this information we will be able to perform a test which will guarantee that a program implementation directly conforms to the XXM and therefore the story card model.

An XXM is an extended form of the X-Machine which allows each function to be guarded by a predicate which identifies a state of memory (this provides a more general definition), with the caveat that from the origin state there must also be a transition defined for the inverse predicate. The XXM has a complex memory structure, allowing us to specify variable names, arrays and arrays within arrays. In order to allow abstraction away from user interfaces which are difficult to test we extend the definition to allow XXM to reside at states as well as in functions. The XXM within states are limited such that they run when the state is entered and when they exit the state is exited (and not before). In effect then these XXM are run as functions just before the state is entered. Further more they only have readonly access to system memory and return values that are only current for the next function.

By using a predicate as a guard function on a state transition we gain an important abstraction from the standard SXM. The key factor here is that we can separate the memory that we are interested in from the rest of the memory, and in the case where this memory state defines a database it means that we do not have to consider many possible states of the database (which could be huge) and instead define what a valid state might look like. This has a potential impact on the testing function however because we may find it hard to define what we expect the current memory state to be following a complex function, and also because if the memory state space is large it is potentially very hard to test for extra functionality included with the implementation.

In the model described here the guarding predicate can be defined or partially defined by the validation set of the FFD which comprises this function (in the case that this function is represented by a XXM then deriving the validation set is a recursive problem) combined with an optional predicate if required.

Normally we use the XXM that are embedded in the states to represent elements of the user interface (screens) as they are shown to the user. This helps us to separate the user interface code from that of the system control code. This is useful as it is fairly easy to automatically test system control code, and comparatively hard to test user interface (UI) code because it contains many more non-functional requirements. We note that control code is normally only associated with three non-functional requirements: Reliability, Performance and Time, whereas the UI many have many more harder to quantify requirements such as 'ease of use'. In practice we can allow this to return to the origin state, though it is unclear as to the effect this may have on any function that may be executed in that state.

3.4 Function Flow Diagrams

Function flow diagrams are used to represent simple functions which do not contain control structures, or that contain reliable (that do not require testing) control structures (such as a library routine). In essence this means that a function must run from start to finish in a straightforward manner. However you may decompose the represented functions further into sub functions as long as they also comply to this rule. FFD's therefore do not allow more than one output path. If a FFD has no output then this signifies that the program halts, or does not terminate.

At the lowest level the functions on a FFD could correspond to those provided by the standard implementation of the target language. In practice many users may prefer to deal with this level of description by hacking code in the target language; tool support in this case may allow FFD to be produced automatically.

For each function on the FFD we need to specify its input criteria, i.e. the expected memory state and the inputs it receives, and the output criteria, i.e. the expected state of memory after the output and any outputs it might have. This will allow us to model these aspects directly in the XXM. In the case of an FFD the memory can be expressed directly as variables in the system or as predicates defining the variables. These criteria specify the contract that the particular function conforms to, so given an input value in the input range then an output value in the specified output range should be produced.

3.5 Data Process Models

Data process models model the structure of permanent data features in the application. In many cases these features of an application are expensive to change as they have knock on effects across a system. Therefore we separate the data design from the system early and stabilise the structure with public interfaces, this potentially also makes testing straight forward. Of course we could use XML files and Databases [9] as well so that if we do have to change the data model, then the migration cost is minimised. We ensure that the access routines and database are tightly bound, so that any change can be managed within this area of the software.

Story cards may not specify the complete data to be held in the system, however at some stage of the specification of the system this must be defined. One option is to allow the story cards to settle down so that the developers feel more confident about the data requirements and then explicitly add this information to the relevant cards.

DPM's capture two things the data access routines and the database, file or memory (data structure) design. We can capture this structure using the DPM, this is like a data entity diagram, but it also allows us to specify the format of the data, its relationships and its access routines.

To build a DPM the first thing we do is to capture the data access routines, that normally refer to an entity. In general terms these will be formed by a set of XXM which are in effect slotted into the larger model.

From the access routines we design the data model for that entity. This consists of an internal data model (similar to a class in object oriented languages) and a table in a database. Once we have done this for all the required entities we can normalise the database structure and define data integrity rules. We now have the single data unit used for the internal memory model and a multiple data unit that is most suited to database storage. For the purposes of testing the external memory model is just considered as another part of the internal memory of the XXM.

4 Unit and System Tests

An important aspect of agile methodologies is the automated test scripts that are used. The collection of models that are used here offer some useful methods to help us create them in a simple and often automated way.

4.1 Extreme X-Machines

The XXM will allow us to automate the system integration tests and help us to design test sets which cover enough of the system (guaranteed). Using the Stream X-Machine theory of testing we can generate a test set which passes though the whole system. This guarantees the exercising of all possible paths though the system and that there is no unexpected functionality (of the implemented system compared to the X-Machine model). Having run an SXM test set over the integrated system we can be sure that the system conforms to the XXM model and therefore the story cards.

In order to apply this test generation set we require that the implemented code satisfies some corresponding design for test conditions. Essentially we require completeness (an SXM is complete with respect to memory if it is always possible to apply an input which can trigger a function, i.e. all states are reachable) and output distinguishability (every function results in a output which is unique to the execution of that function). Both of these conditions are simple to implement and sensible conditions on a program. Both of these can be readily ensured by a tool, though determining if the first is possible may be an NP Complete problem. Because we define transitions with respect to the memory state, XXM can be harder to test, as there are potentially many more paths though the system. This maybe overcome however by defining how the inputs interact with the memory and using the original testing method.

Tool support should be able to identify the SXM testing paths though the XXM, and using the guard predicates it should be able to construct appropriate input data to ensure that a particular path is followed though a program execution. In order to specify the paths though the system we note how the input sets the memory in order to satisfy the transition predicates.

It is noted that the XXM specifies that XXM can be decomposed into the system states as well as the system functions. It is envisaged that these machines can be treated as independent from the rest of the system and can be tested in isolation without effect on the rest of the system in effect these would be integration tests on subsections of the system. The output of these systems is expected to consist of user interactions; valid versions of these can be constructed for the test sets. This has the benefit of separating system control and the code required to interact with the user, which is harder to test in an automated system.

4.2 Function Flow Diagrams

For each represented function we can define a validation set and a test set. The validation set is a description of all valid inputs into the function, this will normally be described by a predicate which defines a valid state of memory and the validity of any inputs. The test set gives values in the valid range along with the expected outputs, and changes to memory. In order to derive this test set it seems most valid to use the category partition method of testing, in some cases the inputs maybe determined automatically by tool support.

The test sets generated here can then be run against each function using the traditional method of unit testing, though tool support could make the construction of the test sets easier than some current systems. Of course we also have a model of how the functions interact with each other so we can extend this notion of unit testing to a series of functions which run together in an FFD.

4.3 Data Process Models

The information captured by the DPM allows us to create test cases for each of the access routines. We will test for the correct operations and for data integrity constraints as defined by the DPM. For each access routine we can ensure that the published functionality is intact and works as expected using the XXM model, given this we can be fairly sure that the system will integrate correctly.

When testing the rest of the system we do not use the routines that access the database, and instead use stub functions that return correct results for some query, this should allow us to perform tests on the remainder of the system with far more speed and certainty. In such a case we only test what happens to the data inside the processing part of the system. In some cases we may however want to use the real code (allowing us to test for performance and other related issues), and in this case test databases would be required which have known contents. We assume that the code provided by the database providers performs as expected when we use this style of testing. Furthermore we may wish to test that the data access routines work correctly; in this case we should be able to test these in a test harness.

5 Code Design

Once we have a detailed model and test set for the system as a whole we can start to build the code. In many cases we have already defined the structure of the code down to a very low level, this links the design stage directly to the code, such that changes we make at the code level directly effect the design. In order to keep maintenance costs down tool support will obviously be necessary.

It seems that the best way do deal with any code that we want to hack manually is to do this only at the function level, with some disregard to any module that it may reside in. We can achieve this by not holding module variables as such with the data for a module (from a FFD) separated into a second module (supplied by a DPM). In general a tool will decide where a function resides and construct the module automatically with appropriate references.

Ultimately the code generated is not as object oriented as one might wish for. The design method here sits somewhere between object orientated and imperative languages so a modular approach is more natural. In this case code resides in a code module that is concerned with the action of the method. This may be the owner of the story, a collection of utility routines, or a collection of routines which act on some data entity within the system.

6 A Brief Example of a Changing System

In order to grasp how change can effect a project we analysed the output from seven small industrial software development projects. The data varied in quality among the projects but allowed us to identify some causes of change, which are likely to be typical of such projects. The major causes of change are listed below.

1. Developing client understanding of their goals.
2. Developers misunderstand client requirements.
3. Error in recording of client requirements.

6.1 Requirements

The following case study shows how such changes effect a typical project, based on what we have learnt. We follow the case study through two iterations of development showing how change effects the proposed models. The example is based around the library problem presented by Kemmerer [10]. We have split part of his specification [p34] into two iterations of requirements change, this lacks many features of a more complete system as originally specified:

Iteration 1:
The example system considered in this paper is a university library database. The database transactions are as follows.

– *Check out a copy of the a book.*

The critical requirements that the database must satisfy at all times are as follows.

– *All copies in the library must be either checked out or available for checkout.*
– *A borrower may not have more than some predetermined number of books checked out at some time.*

Iteration 2:
The example system considered in this paper is a university library database. The database transactions are as follows.

– *Check out a copy of the a book.*
– *Get a list of titles of books in the library by a particular author.*

The critical requirements that the database must satisfy at all times are as follows.

– *All copies in the library must be either checked out or available for checkout.*
– *No copy of a book may be both checked out and available for checkout.*

6.2 Story Cards

There follow two iterations of story cards which represent the system. In the first the author does not capture the requirement for the search function. In the second this is added and the previous cards are refined. These changes reflect observations that have been made on student projects. It should be noted that the exact meaning of the stories is not necessarily precise, this will be refined in the XXM model.

Iteration 1:
Main Menu

- Allow the user to choose between the following stories:
- Check out
- On completion of a story the menu is shown again.

Check out

- The library user enters their borrower id, and the book id.
- If the borrower id is not valid fail.
- If the book id is not valid fail.
- If the number of books on loan is greater than max-books then fail.
- Otherwise
- increase borrower books on loan
- set book borrowed by borrower

Iteration 2:
Main Menu

- Allow the user to choose between the following stories:
- Check out
- List titles by author
- On completion of a story the menu is shown again.

Checkout

- The user enters their borrower id, and the book id.
- If the borrower id is not valid fail.
- If the book id is not valid fail.
- If the book is on loan then fail.
- Otherwise
- set book borrowed by borrower

Search for books by author

- Read in the authors name from the user.
- Search the book records in the description author field for the author.
- Print out the titles of books that match the authors name.

Each story is roughly divided into three sections. In the first the user is prompted for any required input. In the second the input is validated to ensure it is correct. And finally the requested action is performed, which often results in output, but not always to the user. As the specification changes (in terms of the stories) so additional items are added, deleted or changed in each of these sections.

6.3 Derived XXM

These story cards can then be drawn into an XXM. Here we illustrate the machines separately for each story and iteration as this allows them to be changed easier with respect to the story cards (see Figs. 1 to 5 in the appendix). The edges have been labelled such that the first item is a predicate that must be satisfied to perform this function, the second is the function name, and the third indicates the memory values changed.

The search by author XXM (see Fig. 5) is typical of most XXM. Control enters into it though the arrow at the top of the diagram. This is labeled with "/getuserInput('Author?')/search_author", this indicates that this edge represents an unconditional transition (nothing before the first slash) that executes the getUserInput function (between the slashes). This function modifies the memory value search_author (as indicated after the final slash). The state entered then has two possible transitions, both guarded. The lower transition "search_author!=null/..." allows the control flow of the program to continue, only when search_author has been set. The XXM goes onto iterate though all the instances of the requested author, writing the results to the screen. When there are no further instances the control flow reaches the SBA_End state and flow returns to the top level Main Menu XXM.

Firstly we must note some technical points on these figures. All the XXM's given do not make use of machines embedded in states; this allows the examples to be more straight forward to understand. In fig. 5 there is further work to be done to fully specify this XXM. Both the searchAuthor and nextAuthor functions are complex, and require the specification of a sub XXM. Finally we have not presented XXM to handle the data in the DPM, these are presented informally below.

6.4 Derived FFD and DPM

The FFD in this system are fairly simple as they would be in most systems. The following is an example for getUserInput:

– pre-conditions: (prompt:String);
– print prompt;
– read input;
– post-conditions: (input:String);

The DPM for borrowers is constant in definition across the iterations whilst books is altered. The definitions are given below.

Iteration 1:
Borrowers

– UID : Integer
– userExists(uid) / true if user exists
– user(uid) / returns a record

Books

- BID : Integer
- bookExists(BID) / true if book exists
- book(BID) / returns a record

Iteration 2:
Books

- BID : Integer
- Author : String
- Title : String
- bookExists(BID) / true if book exists
- book(BID) / returns a record

We note here that despite adding extra data to the system, we only had to revise the methods in the XXM and in this specification that were directly related. There would be no knock on effects elsewhere in the system.

6.5 Testing

Given the definition of the XXM given above it is possible to create test sets based on the SXM theory of testing. To do this we would collect the input sets required to achieve path coverage such that every loop is executed at least once and every state is visited. This would ensure that we translate the XXM into a minimal SXM, in effect we collapse the transition space such that every function is triggered by an input symbol. In this case iteration two collapses to a four state machine! As this example is fairly simple and every function in the original XXM and the SXM constructed has a unique output (in terms of memory state which we assume is observable) based on its location only a single transition following a state is required to confirm that the state was visited. This means that only a fairly small test set is required.

In order to test that no additional functionality was added we need to ensure that for each of the functions of the XXM where input is collected (and every transition in the manufactured SXM) that no other input causes the state to change. This can be achieved by using an input sequence that should reach the state we wish to test and applying all invalid inputs before applying an valid input. The memory should not change (noting that if the XXM defines how a input should be handled, then this must be a valid input) until the valid input is read.

7 Conclusions and Future Work

We have shown that a simple method can be used to model story cards as commonly used by the XP community. This is flexible and simple enough to embrace change, however it does require the use of a more formal notation. As a formal method it allows the user to test their applications reliably based on the SXM method of testing, whilst using a notation which is closer to a programming language.

Whilst the SXM method of testing is useful it is not well adapted to change and hierarchical features. We propose to investigate this further. Firstly to confirm that hierarchical features can be tested in isolation. And secondly to see if it is possible only to test the areas of the specification that have changed, whilst retaining confidence in the testing. The second of these is probably the most challenging and may require some qualification of how reliable a restricted test set would be.

In order to make this method useful it is also necessary to build a tool, such that the story cards, XXM models and the code can be tied together in a flexible fashion. This should help to ease the problem of maintainability in a rapidly changing program.

Having shown that there is some use in presenting the problem in this way we wish to develop the formal definitions further. Therefore in a future paper we hope to present a precise formal definition of this methodology and XXM in particular.

In order to demonstrate how these methods can be applied effectively we hope to be able to use them with in some industrial projects similar to those already studied. In so doing we hope also to be able to better define how the changes in these projects effect their progress.

8 Acknowlagements

We would like to thank Kirill Bogdanov and Marian Gheorghe in the department and the referees of this paper for their help and advice.

References

1. Beck, K.: Extreme programming explained: embrace change. Addison Wesley Longman, Reading Massachusetts, (2000)
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. Fowler, M.: Refactoring: Improving the design of existing code. Addison Wesley, (2000)
9. Medcalf, A: Evolving Databases. Undergraduate project Dissertation, Department of Computer Science, University of Sheffield, June (2001)
10. Kemmerer, R. A.: Testing Formal Specifications to Detect Design Errors. IEEE Transactions on Software Engineering, Vol. SE-11, No.1, January (1985)

Appendix: Extreme X-Machines

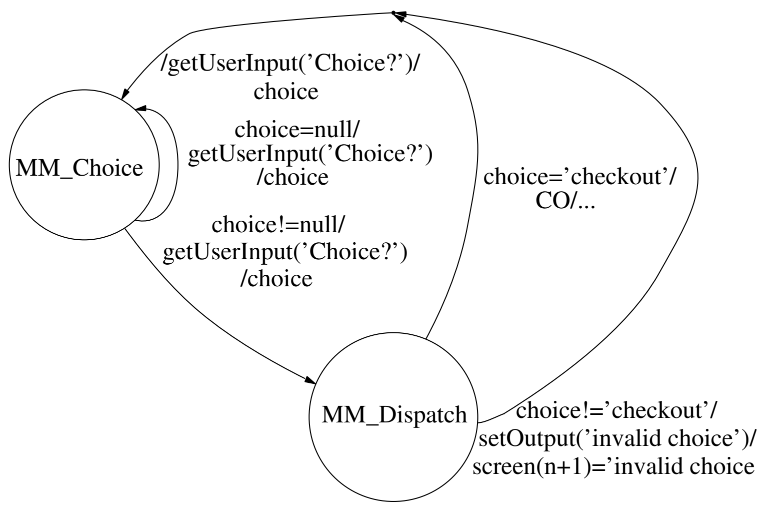

Fig. 1. Iteration 1: Menu XXM

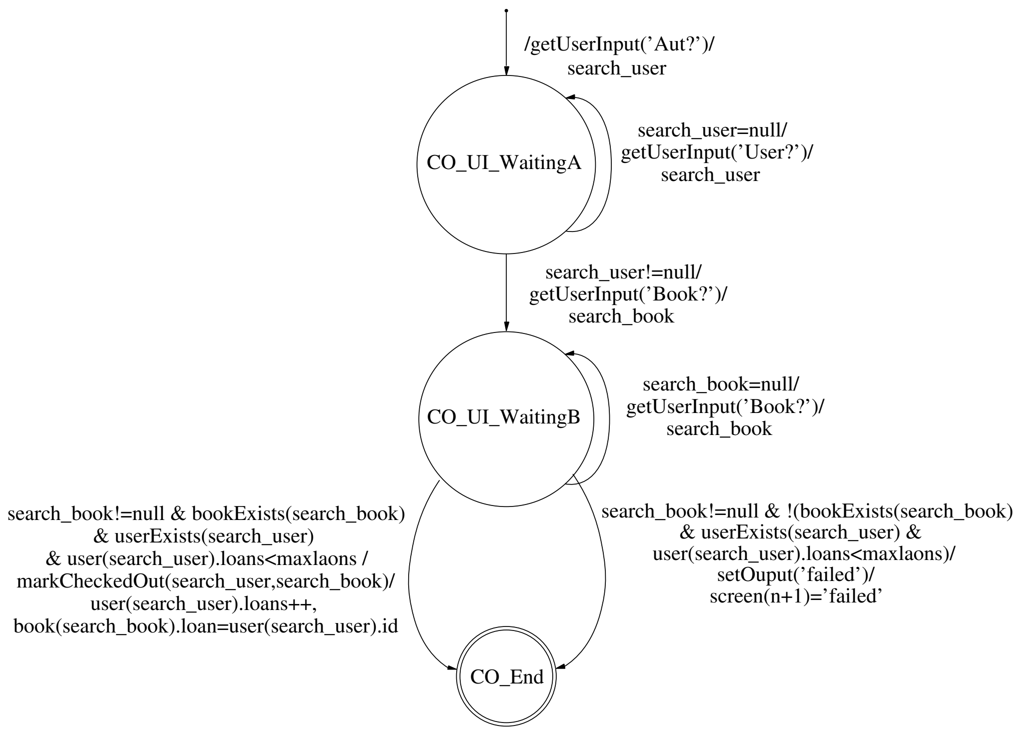
/getUserInput('Aut?')/
search_user
CO_UI_WaitingA
search_user=null/
getUserInput('User?')/
search_user
search_user!=null/
getUserInput('Book?')/
search_book
CO_UI_WaitingB
search_book=null/
getUserInput('Book?')/
search_book
search_book!=null & bookExists(search_book)
& userExists(search_user)
& user(search_user).loans<maxlaons /
markCheckedOut(search_user,search_book)/
user(search_user).loans++,
book(search_book).loan=user(search_user).id
search_book!=null & !(bookExists(search_book)
& userExists(search_user) &
user(search_user).loans<maxlaons)/
setOuput('failed')/
screen(n+1)='failed'
CO_End

Fig. 2. Iteration 1: Check out XXM

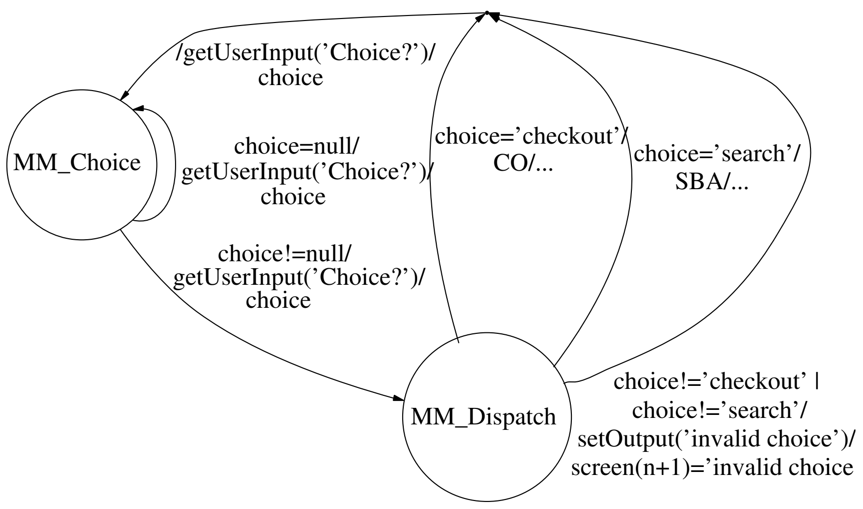

Fig. 3. Iteration 2: Menu XXM

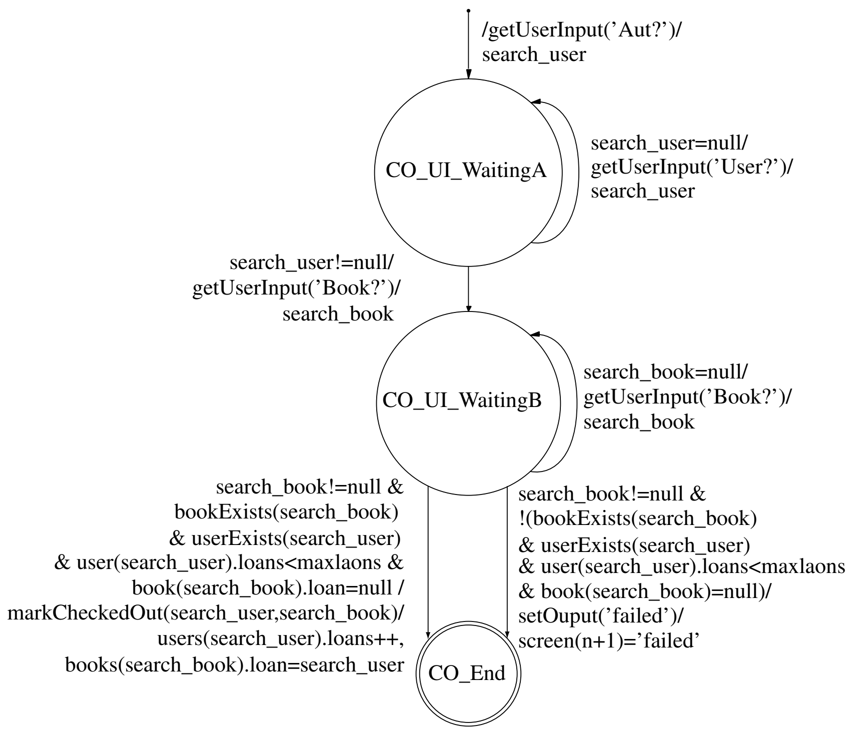

Fig. 4. Iteration 2: Check out XXM

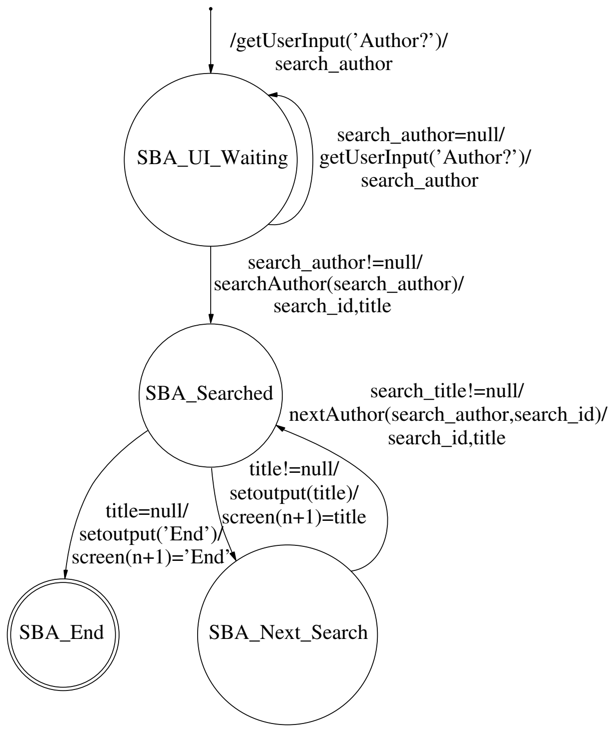

Fig. 5. Iteration 2: Search by author XXM