Behavioural Concern Modelling for Software Change Tasks

Albert Lai and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall Vancouver B.C. Canada V6T 1Z4
{alai, murphy}@cs.ubc.ca

Abstract

Many modification tasks on an existing software system result in changes to code that crosscuts the system's structure. Making these changes is difficult because a developer must understand large parts of the system, and must reason about how the modification will interact with the existing behaviour. Typically, developers make these kinds of changes using an ad-hoc approach with tools that help in gaining some understanding of the existing system, but that do not provide any specific support for reasoning about, implementing, or analyzing just that part of the system related to the modification. In this paper, we present the Behavioural Concern Modelling (BCM) approach and tool that provide direct support for a systematic approach to modification tasks. This approach enables a developer to create a partial, abstract, grounded behavioural model of a concern(s). By grounded, we mean that the relationship between the model and the code is explicit: A developer describes which code contributes to which parts of the model. The examples we describe use a finite state machine as a model. We show how the approach can help a developer capture a concern, reason about design options, and implement modifications.

1 INTRODUCTION

All too often, modifications to an existing software system are made in an ad-hoc manner. A developer determines some parts of the code relevant to the modification and then starts to iteratively identify, understand, and change the code to perform the modification. When the points in the program related to the modification are well-localized, this approach can be effective. When the relevant points crosscut the system structure, this approach begins to fall apart: Developers have a difficult time estimating how long it will take to complete the modification task, the code added and changed as part of the modification introduces defects into seemingly unrelated parts of the system, amongst other problems. In this paper, we refer to the code related to a particular feature as a concern.

The ad-hoc approach to making a change seems to break down when the modification crosscuts the system because many of the subtasks the developer must perform to complete the modification task become harder. It is harder for the developer to identify relevant portions of the existing code because large parts of the system may need to be considered and understood. It is harder for the developer to evaluate options for the design of the modification because large parts of the existing design must be considered. It is harder for the developer to determine how the code implemented to satisfy the modification impacts other crosscutting concerns because those concerns are also typically implicit [3].

Existing tools can help the developer with some parts of some of these subtasks. Lexical searching tools, such as grep and Aspect Browser [7], can help identify relevant code. Structural analyzers, such as FEAT [14], flow analyzers, such as program slicers [17], and some reverse engineering tools, such as Shimba [16], can help a developer identify and build up an understanding of how relevant code works. These tools help a developer deal with the existing system, but they do not provide a developer any direct help with reasoning about, implementing, analyzing, or verifying the system with the modification.

We hypothesize that a developer can perform a modification task more systematically when the developer has access to a *behavioural* model of a concern (or concerns) relevant to the modification that is *partial*, *abstract*, and *grounded*. By *behavioural*, we mean a model that helps a developer reason about how the existing code works and how the modification might work. By *partial*, we mean that the model need represent only those parts of a concern relevant to the task at hand. By *abstract*, we mean that the model is of a size and complexity amenable for the developer to reason about. By *grounded*, we mean that there is a mapping between the model and the existing source. This mapping enables the model to be used to direct analysis on the code. For example, the mapping permits us to analyze whether the data- and control-flows in the system respect the model.

To investigate this hypothesis, we have developed an approach, called Behavioural Concern Modelling (BCM) (Section 2), and a supporting tool (Section 4). In the BCM approach, a developer iteratively posits all or part of a finite-state machine (FSM) representing the behaviour of a concern or concerns, and maps the pieces of that FSM to the source. The developer then uses the BCM tool to understand how data- and control-flows in the source relate to the posited state machine. Based on the flow information, the developer adjusts the FSM until a model suitable for reasoning about the modification is reached. The BCM tool builds on previous work in conceptual modules [2] (Section 4.1).

We have applied our approach to help assess a modification in two systems. First, we used our approach and tool to help assess a change to an FTP server. (Section 3). In this case, we chose the change task. We use this task to describe our approach and to demonstrate how the BCM approach can permit a more systematic evaluation of different ways to implement the change. Second, we applied our approach to assess an outstanding change to a public-domain web browser (Section 5). We present this task to demonstrate that the approach is viable for larger unfamiliar code bases.

The contribution of this paper is to propose, and demonstrate a viable approach for modelling the behaviour of a concern. This model can help a developer manage a concern during the performance of a software enhancement task. It may also help support the task of remodularizing the system (See Section 6.4). In this paper, we demonstrate the need to ground a model in the source to support such tasks.

2 BCM OVERVIEW

The BCM approach helps support a systematic modification process that consists of six steps.

1. The developer identifies concerns relevant to the modification. These concerns may be identified in a top-down fashion based on design information, or in a bottom-up fashion based on information in the code.

2. The developer iteratively builds an abstract, partial, behavioural model of the identified concerns, and grounds this model in the source.

3. The developer evaluates different design options by considering their impact and interaction with the behavioural concern model.

4. The developer selects a design.

5. The developer uses the behavioural concern model as a guide to implement the chosen design.

6. The developer uses the behavioural concern model to help verify the new implementation.

Our focus to date has been on the first five steps of the process. To clarify the BCM approach, we describe its use to plan a modification to an FTP server. We discuss the role of a behavioural model and BCM for the last step of the process in Section 6.3.

3 MODIFYING A FTP SERVER

jFTPd [1] is a FTP server written in Java (11 classes containing approximately 3000 lines of code) that supports basic FTP commands and anonymous login. For our case study, we chose to consider the addition of named-user logins to the jFTPd system.

3.1 Forming the Model

We began by identifying the concerns relevant to the modification at hand. We used our knowledge of existing jFTPd features to posit that the anonymous login concern would be impacted by the addition of named-user logins, and thus should be modelled.

To form the FSM behavioural model of the anonymous login concern we interchangeably performed two activities. First, we looked for code related to the concern, and based on our understanding of that code, we posited states and transitions in the FSM. Second, we used our belief of how the concern worked to extend the FSM, looking for support for the added states and transitions in the code. We describe this process in more detail.

Logging into an FTP server involves the `USER` FTP command. We began the formation of the model by looking for source that implemented this command. Using grep, we searched for the string "user" and found the `doUserCommand` method in the class `FTPConnection` (Figure 1). Some of the code in this method sets a `userName` field based on user input. To capture this behaviour, we introduced a *setUser* transition into the FSM for the anonymous login concern, and associated a subset of the code in the `doUserCommand` method as implementing that transition (Figure 2(a)). (A detailed description of how the FSM is recorded, and how code is associated with states and transitions of the FSM is provided in Section 4). We modelled this action as a transition because we had already started formulating a mental picture of the FSM as having authenticated and unauthenticated states.

Based on our experience of using FTP servers, we then considered the next behavioural step after determining the user, getting and checking the password, which involves the `PASS` command. Searching for the "pass" string, we identified the `doPassCommand` method. Examination of this method revealed that it is responsible for deciding whether a user has permission to log in, and whether to grant or deny access to the user. To model this behaviour, we added

[1] Available from `http://jftpd.prominic.org/1.3/index.html`.

```
protected boolean doUserCommand(String line) {
  if (line.length() <= 5)
    return false;
*  if (anonUser) {
    out.print("530 Can't change user ..." );
*  } else {
*    String user = line.substring(5);
*    userName = user;
    String userLower = user.toLowerCase();
    if (userLower.equals("ftp") ||
        userLower.equals("anonymous")) {
      out.print("331 Guest login ok, send ... ");
    } else {
      out.print("331 Password required for ... ");
    }
  }
  return true;
}
```

Figure 1. Code for SetUser Transition

a *handleAnonymousPass* transition, a *permitAnonymousLogin* state, an *authenticAnonymousUser* transition and a *rejectUser* transition (Figure 2(b)) to the anonymous login FSM. We also associated the relevant code with these states and transitions.

Next, we tried to identify the code connecting the *setUser* transition and the *handleAnonymousPass* transition. We knew there must be a means to connect these transitions because *setUser* must occur before *handleAnonymousPass* for a login to be successful. We expected there to be both a control-flow and a data-flow between the code associated with these transitions because of the ordering of the operations and because the input user name is passed between the transitions.

Instead of tracing these flows manually, which would be tedious, we used a *relatedness* query supported by the BCM tool. This query examines the data- and control-flows between two parts—states or transitions—of the FSM within a particular context of the program. The query returns information about the set of statements that comprise the flows, as well as providing class and method summaries for those statements. In this case, we performed a relatedness query between the *setUser* transition and *handleAnonymousPass* transition using all the classes from jFTPd as the context. This query returned the following class summary.

`FTPConnection` → `PassiveConnection`
`FTPConnection` → `WildcardFilter`
`FTPConnection` → `FTPHandler`
`FTPHandler` → `FTPConnection`

This summary informed us of overall connections between classes involved in providing the anonymous login behaviour. For example, the summary tells us that information may flow from objects of the `FTPConnection` to objects of the `PassiveConnection` class. As is often the case, this first query returned broad results. To narrow the

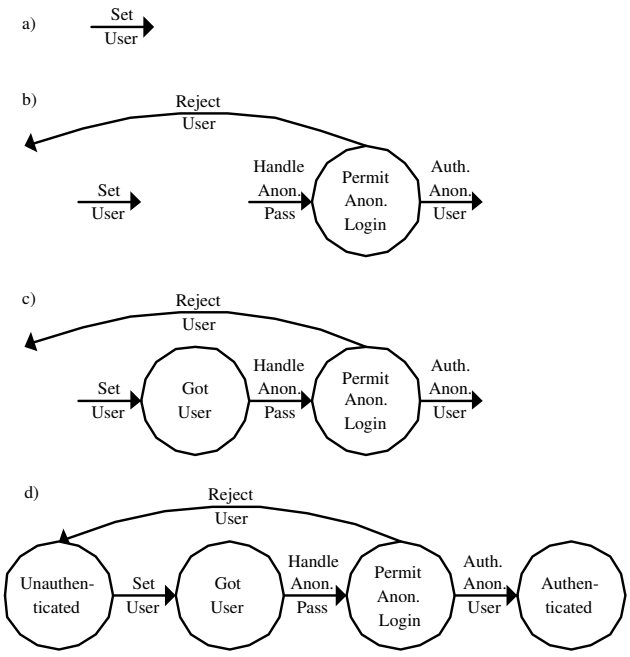

Figure 2. Steps in Building jFTPd Model

results, we refined the context of the query by removing the `WildcardFilter` class from consideration. We made this choice because we noticed that `WildcardFilter` extends `java.io.FilenameFilter` and we reasoned that it was unlikely that the filter was related to the *setUser* and *handlePass* transitions. To confirm this reasoning, we examined the source and found that `WildcardFilter` is used to implement the FTP LIST command, which is not related to user authentication.

To determine what specific code to investigate for the missing part of the FSM, we considered the method summary returned from the above refined query. Of the approximately twenty entries returned from the query, several of them were unrelated to *setUser* and *handlePass*; the flows these entries described handled other FTP commands. We refined our context to ignore flows in unrelated methods and the following method summary was returned.

`doPassCommand` → `printWelcome`
`doUserCommand` → `doPassCommand`
`doCommand` → `doUserCommand`
`doCommand` → `doPassCommand`
`run` → `doCommand`
`doCommand` → `run`
`doCommand` → `setBusy`
`doCommand` → `setLastCommandTime`

We examined the code in these methods and found that the `doCommand` method parses FTP commands and calls appropriate methods to handle the commands. We also de-

termined that the flow from `run` to `doCommand` was the result of a while-loop in `run` that reads FTP commands and passes those commands to `doCommand`. After the user is set via the *setUser* transition, jFTPd continues to process commands via the while-loop in `run`. Thus the same code also executes after jFTPd has determined the user. We associated code from `run` and `doCommand` with a *GotUser* state (Figure 2c).

We continued in this manner, expanding the model, and associating code with pieces of the model using the BCM tool, until we were satisfied that our model was sufficiently complete to reason about the modification task. Figure 2d shows the final model. To check that our model was sufficiently complete, we queried the tool for all of the control- and data-flows to and from all of the states and transitions in the FSM, and checked that none of the flows were pertinent to the model. No unexpected values were reported as flowing from the FSM, but unexpected values were reported as inputs. Specifically, the query reported two fields, `FTPConnection.anonUser` and `FTPConnection.userName` as being used by the FSM; these fields were defined outside of the code associated with the FSM. Since jFTPd uses the `anonUser` field to indicate whether the current user is an anonymous user, all uses of this field needed to be assessed as to whether they should be part of the model. A closer look at the query results indicated that we had not included a definition of the `anonUser` field from an instance initializer; we updated both the *Unauthenticated* and *Authenticated* states with that code as the states use that field to represent whether an anonymous user has logged in or has been authenticated. Similarly, the `userName` field is also defined in an instance initializer; we added an association of that statement to both the *Authenticated* and *Unauthenticated* states as well.

3.2 Considering Design Options

Based on the model we formed, we considered two different design options for implementing named-user authentication. One option was to generalize the existing mechanism, considering an anonymous login as a special case of named-user logins where the login name is "anonymous". The second option was to consider anonymous login as a separate case from named-user logins.

Figure 3(a) depicts the first option. This option requires minor modifications to the transitions and state associated with handling and authenticating passwords. In this option, only one path is needed from the unauthenticated state to the authenticated state. This option is conceptually simple, but it may be difficult to implement policies in which anonymous users need to be treated separately. For example, we might want to limit the number of anonymous logins as well as the total number of named-user logins.

Figure 3(b) depicts the second option. If we consider anonymous login and named-user login as separate mechanisms, anonymous login would be more explicitly represented in the source code, and it may be easier to implement such policies as described above.

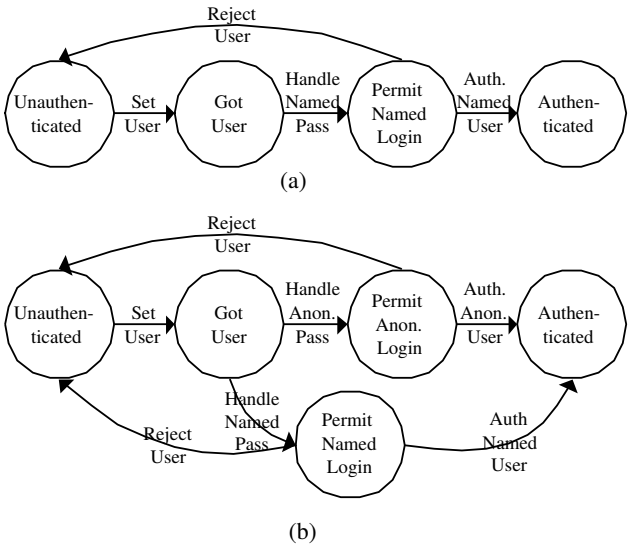

Figure 3. jFTPd Model Options

3.3 Implementing the Chosen Design

After we decide on a design, we must make the modifications to the code. The behavioural concern model can be used as a guide when performing the modifications as the model points to the specific code a developer must consider when making the change. For instance, if we choose the first option, we can determine from the model that we did not have to modify the code associated with the *Unauthenticated*, *GotUser*, and *Authenticated* states, or the *setUser* transition. The model can also help us determine that the *PermitAnonymousLogin* state and the *AuthenticateAnonymousUser* transition must be modified as shown in Figure 3.3. We compared the modifications to our model of the design option and found that modifications were well-aligned with our model elements.

4 THE BCM TOOL

The grounding of a behavioural concern model in the source is a critical part of our approach. Without this grounding, it is easy to create a model that overlooks important details about how the concern is implemented. The grounding also helps in the formation of the behavioural model. In this section, we describe the tool support we developed to help a developer create and investigate the grounding of the model. Since this tool builds on earlier work in conceptual modules, we begin with a description of conceptual modules.

```
  protected boolean doPassCommand(String line) {
    ...
    String password = line.substring(5);
    String userLower = userName.toLowerCase();
#   if (userLower.equals("ftp") ||
#       userLower.equals("anonymous")) {
      authorized = true;
      anonUser   = true;
      return true;
#   } else {
      try {
#       Properties access = new Properties();
#       access.load(new
#                   FileInputStream("access.lst"));
#       if (access.get(userName).equals(password)) {
*         authorized = true;
*         out.print("230 Logged in");
#       } else {
          out.print("530 Login incorrect.\n");
          userName = null;
#       }
      } catch (Exception e) {
        ...
      }
    }
    return true;
  }
```

Figure 4. Code forh PermitNamedLogin(#) and AuthenticateNamedUser(*) transitions

4.1 Conceptual Modules

The Conceptual Module (CM) approach supports the overlaying of logical structure on the structure of an existing system [2]. Each piece of the logical structure is modelled as a CM. A CM treats a collection of non-contiguous source code lines from multiple parts of the existing system as a logical unit. Tool support is provided to allow a developer to query about the control- and data-flows between CMs and between a CM and the existing source. For example, a developer can query about the interface to a CM. The tool will respond with a list of the inputs expected by the CM, the outputs the CM produces, and the control-flows emanating from the CM. Earlier work demonstrated the utility of the CM approach for reengineering C [2] programs.

In the work described in this paper, we use CMs to represent the states and transitions in a FSM. At its core, the BCM tool is a CM tool for Java. As we describe the BCM tool, we indicate where the basic CM concept and tool have been refined and extended to help support the software change process.

4.2 Tool Interface

The BCM tool provides three operations to a developer: CM creation, a relatedness query, and an interface query. Prior to running any of these operations, a developer must have specified a set of classes that form the *world* for the BCM tool. All subsequent analyses are conservative, within the limitations described in Section 4.4, with respect to this *world*.

CM Creation A developer creates a CM that represents a state or a transition by giving the CM a name and associating a set of source code lines from the *world* with the named CM. A developer can select individual lines, all of the lines within a method, or all of the lines within a class. A developer can also use set operations applied to existing CMs, such *AND*, and *OR*, to define new CMs consisting of the intersection or union of other CMs.

The original CM tool did not have the ability to create CMs from other CMs.

Relatedness Query A developer uses the relatedness query to determine the data- and control-flows that exist between two CMs. This query requires three inputs: a source CM, a target CM, and a context, which is also specified as a CM. The relatedness query identifies all data- and control-flows emanating from the source that reach the target through code included in the context. The query reports the flows in a method summary, and a class summary.

The original CM tool did not provide any hierarchical reporting of the flows between CMs, nor did it allow the filtering of results through a context.

Interface Query A developer can also perform an interface query on a defined CM to elucidate the data- and control-flows to and from the CM. The query determines which variables and fields, within the world, are possible *inputs* to the CM, which variables and fields are possible *outputs* to the CM, which *control transfers* might emanate from the CM. As described in Section 2, this query is helpful in gaining confidence that the appropriate code has been modelled as part of an FSM when using the BCM approach.

4.3 Tool Internals

The BCM tool works on Java bytecode using the Jikes Bytecode Toolkit (JikesBT [2]). To improve the efficiency of queries, BCM pre-processes the methods in the defined world, creating a method digest for each method. A method digest summarizes which fields and arguments a method might possibly use or define, and which methods it might possibly call.

To build a digest for a method, the tool first computes a control-flow graph for the method in which a vertex represents a bytecode and an edge represents a possible control-flow between bytecodes. This control-flow information is used to guide a simulation of the possible effects of the method on the Java Virtual Machine stack and on local variables. The BCM tool iterates over the methods until all method digests have reached a fix-point. At the end of this pre-processing, only the method digests are retained. Data- and control-flows within methods that are needed for particular queries are recalculated on demand.

[2] Available from `http://www.alphaworks.ibm.com/tech/jikesbt`.

The BCM tool takes a conservative approach to the analysis. For method-call bytecodes that have multiple possible targets, the BCM tool uses the union of the method digests of all possible targets. The possible targets are determined using JikesBT. The values of objects are not tracked, leading to additional imprecision in the results reported.

At any given time, the BCM tool retains only the data- and control-flows for the method it is currently analyzing. The performance of the tool is thus more dependent upon the complexity of a method's control-flow rather than the overall number of classes or methods in a given application. Using the Java 1.3 HotSpot VM running on a PIII 1Ghz machine, we were able to preprocess the 29000 lines-of-code XBrowser system in 10 minutes and the longest query completed in 15 minutes.

4.4 Tool Limitations

The current BCM tool does not support the analysis of native methods. The imprecision introduced by this limitation was not an issue in our analysis of jFTPd or XBrowser. Since most native methods are used for performance or for interfacing with external libraries, the control- and data-flows are often contained within the native code. One exception to this statement occurs with classes that extend `java.lang.Thread` and that override the `run` method. At some point, the native method `start` is called and it eventually calls the `run` method. This control-flow is not detected by BCM because it occurs in native code. We have worked around this problem by writing a temporary subclass of `Thread` with an overridden `start` method that explicitly calls `run`.

The BCM also does not perform any alias analysis, nor can it determine the results of calls using Java's reflection capabilities. Each of these limitations introduces imprecision into the control- and data-flow results.

5 EVALUATING BCM

For the BCM approach to be viable, it must be possible for developers to create useful models of a concern within a reasonable amount of time. To date, our focus has been on the first part of this statement within a specific context: is it possible to create a useful model of a concern for reasoning about a change? In this section, we describe a case study of applying the BCM approach to an outstanding change task on an unfamiliar system to provide additional evidence that model creation is possible. The model created for this change task provided a framework for introducing the desired behaviour and for examining how it would interact with the existing behaviour. Once the modifications to the model were complete, the existing mapping between the model and source code aided in identifying the structural units that needed modification.

5.1 XBrowser

The target of this study was the XBrowser system, which is a Web browser written in Java using Swing with features similar to Netscape Navigator version 3. [3] The code for XBrowser comprises 171 classes and approximately 29000 lines of code.

One of the outstanding feature enhancements for XBrowser was a request for Meta-Refresh support. In an HTML document, the META elements contain metadata such as a document's keywords and author. The META element may also be used to refresh a document window to another URL after a specified number of seconds.

For our study, we chose to apply the BCM approach to support the addition of Meta-Refresh feature to XBrowser. Little documentation about XBrowser was available; a situation that is all too common when evolving a software system.

5.2 Modelling the Existing Behaviour

We used our knowledge of the Meta-Refresh feature to predict what concerns in XBrowser would likely be impacted by the change, and thus which would be of interest to model. First, we predicted that the addition of Meta-Refresh would change the current URL, and thus would involve a navigation concern. Second, we predicted that the feature would require parsing the current document, and thus would involve a document parsing concern.

Similar to our description in Section 2, we grew partial models of each of these concerns by identifying code snippets of interest with grep, by positing model pieces and associating code with those pieces, and by using the BCM tool to perform relatedness queries to check that we had modelled the code of interest. As before, context was specified as part of the relatedness queries to make the output of queries feasible to read. Reducing the context was relatively easy; for instance, we filtered graphical user interface classes because these classes fell outside of our scope of interest.

Figure 5 shows the model resulting from this iterative process. The document parsing concern is captured by the *Parsing* state. Only one state is needed because a large portion of the HTML parsing is handled by Swing; the little parsing that is done by XBrowser is well-localized. The navigation concern is captured by the remaining states and transitions. The *Back Action* transition represents all of the actions, such as pressing the forward button in the browser, that may cause XBrowser to load a different document. The *Process HyperLink* state represents loading and display of an HTML document in a given frame target. The GUI thread interacts with the Page-Loading thread via the combined *Page Load Stop, Page Load Start* transition; in this combined transition, first the *Page Load Stop* event occurs,

[3] XBrowser is available from `http://xbrowser.sourceforge.net`.

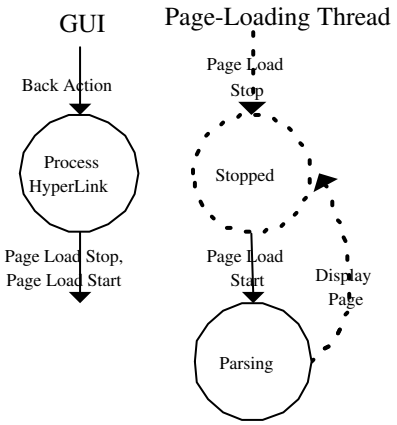

Figure 5. XBrowser Model

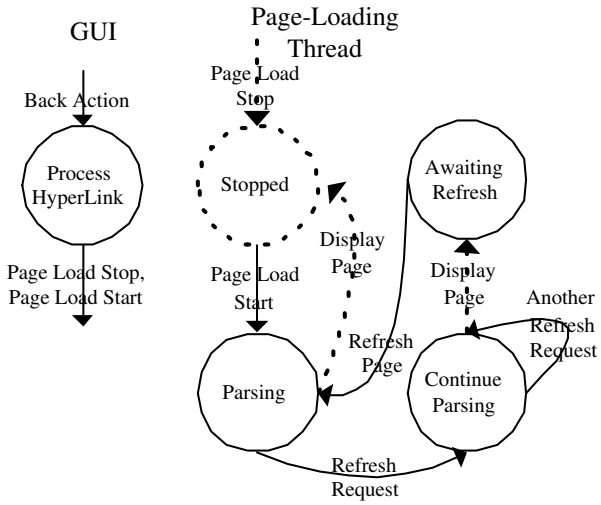

Figure 6. XBrowser Model with Meta-Refresh Feature as part of Page-Loading Thread

and then the *Page Load Start* event occurs. This transition stops the Page-Loading thread if it is running to prevent the thread from displaying previous requests. On receipt of a *Page Load Stop* request, the Page-Loading Thread enters the *Stopped* state. The *Page Load Start* transition causes the display of requested URL. Swing calls methods associated with the *Parsing* state to parse a HTML Document. After Swing has processed the page, it displays the page and the Page-Loading Thread returns to a stopped state.

This model has two interesting features: different fragments of state machines are used to represent behaviour in different threads, and some model elements, those that are dashed in Figure 5 and subsequent figures, do not have any code associated with them.

Each fragment of the model represents a part of the behaviour exhibited by a given thread. The two fragments of the state machine reflect the fact that the GUI thread and the Page-Loading thread may be in different states at different times. Had we merged the two, there would be a set of states representing the cross-product of the individual threads' states.

In the Page Loading thread *Stopped*, *Page Load Stop*, and *Display Page* are all model elements that do not have any associated code. Their behaviour is implemented by Java core libraries. Without these elements, the model would not accurately reflect the behaviour of the system. They provide context that helps developers make sense of the others elements.

5.3 Modelling the Meta-Refresh Feature

The model of the navigation and document parsing concerns provided a basis on which to consider approaches for implementing the Meta-Refresh feature. We considered two approaches: modelling the Meta-Refresh feature as part of the Page-Loading thread, and modelling the Meta-Refresh feature as a separate thread.

Figure 6 shows a model corresponding to the first approach. The Page-Loading thread notes any refresh META element it encounters while parsing, retaining only the last META element if multiple exist. This behaviour is modelled by the *Parsing*, and *Continue Parsing* states, and the *Refresh Request* and *Another Refresh Request* transitions. The *Parsing* state and *Continue Parsing* state need to be distinct because the *Continue Parsing* state represents the fact that we have encountered a META element. The *Another Refresh Request* handles HTML pages with multiple META elements. When parsing is complete, a *Display Page* transition is taken which causes Swing objects to display the document. If we had reached the *Awaiting Refresh* state, then we need to delay the appropriate number of seconds before taking the *Refresh Page* transition to start displaying the new page.

In the second approach, we create a new Meta-Refresh thread as Figure 7 shows. While parsing a document, the Page-Loading Thread may encounter a refresh META element. When this occurs, the *Refresh Stop, Refresh Start* transition is taken. This transition stops the Meta-Refresh thread if it is running and starts a new Meta-Refresh thread. Once the Meta-Refresh Thread receives a *Refresh Start* request, it enters the *Waiting* state and delays for the appropriate amount of time. The Refresh Thread then takes the *Display New Page* transition to start displaying the refresh page. Once the Refresh Thread reaches the *Process HyperLink* state, if the desired URL is not loading already, the Refresh Thread takes the *Page Load Stop, Page Load Start* transition. This transition instructs the Page-Loading thread to stop and display the new page refresh page.

Extending the model for the second approach helped us

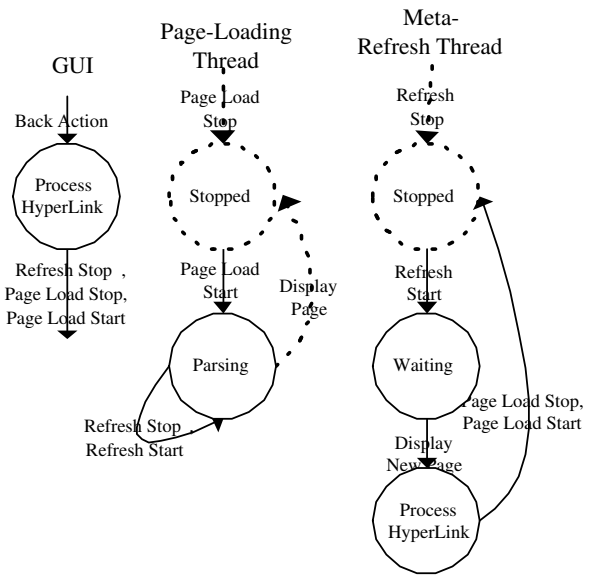

Figure 7. Meta-Refresh Feature as Separate Thread

consider subtle pieces of the Meta-Refresh feature. As one example, we had to consider the situation where a user decided to visit another page before a refresh had completed. Suppose a user visits a page with a refresh META element causing the browser to start a refresh thread. Suppose that before the refresh thread changes the current URL, the user visits a different page. The existing refresh thread is no longer relevant to the current page and should not change the current URL. We model this behaviour by modifying the GUI thread's *Page Load Stop, Page Load Start* transition. We rename it to *Refresh Stop, Page Load Stop, Page Load Start* to reflect the desired behaviour.

The two approaches both have their merits. The first approach does not require a new thread and thus is less complex and less prone to synchronization and threading bugs. The second approach is more flexible because the meta-refresh behaviour is implemented in a second thread. This enables more design choices for the developer such as whether or not the refresh should start when the META element is first encountered or when the document has finished loading.

5.4 Implementing the Meta-Refresh Feature

We used the changes to our model to guide us to the points in the source code we needed to modify or to augment. As one example, consider the the Refresh thread's *Page Load Stop, Page Load Start* transition. We need to ensure code exists in the system to perform this function. Since this transition is similar to the GUI thread's *Refresh Stop, Page Load Stop, Page Load Start* transition, we first consider the use of that code to perform the function. After examining the code and determining its suitablity, we can record it as an implementation appraoch in the model by grounding the Refresh's thread transition with that code.

6 Discussion

The BCM approach shows promise, but several questions remain. In this section, we discuss some of the choices we made in the definition of our approach and the implementation of our tool, describe extensions to the approach that would further help in systematizing the change process, and discuss how the approach might help in further modularization of a code base.

6.1 Form of the Model

We chose to use finite state machines (FSM) to model concern behaviour for several reasons. Their lightweight syntax and semantics allow developers to focus on describing the behaviour of a concern. Single elements in FSM's rely only on local knowledge, enabling developers to describe parts of a concern without knowledge of other parts of the concern.

The states in our models typically represent modes of computation; the transitions typically represent a possible change in modes. This interpretation may be confusing to developers who expect states to represent the potential values of fields, and transitions to represent changes in those fields, or flows of data. Further case study work is needed to determine if this interpretation is suitable for a wide range of change tasks, or if other model types, such as UML sequence or collaboration diagrams [9], may be more appropriate all, or some, of the time. In addition, the form of the model may be dependent not just on the change task, but also on the concerns involved. For example, sequence diagrams may be the best choice for modelling a transaction concern for a student enrollment system. Since the BCM tool is not currently sensitive to the form of model, the tool may be used to experiment with these different choices.

6.2 Filtering Relatedness Query Results

The query that returns information about how two elements of the FSM relate tends to produce a large number of results. The BCM tool can filter these results based on structural contexts described by classes, methods, or lines of code. Another possibility is to filter based on lexical information, such as variable and field names, or other structural information, such as inheritance relationships. Yet another possibility is to filter on a graph theoretic basis: A developer may only want results that are strongly-connected, or results that form the shortest path from the source to the target. Each of these filtering methods represents a tradeoff between returning too much information, and accidentally filtering out desired information. The situations in which these queries work best is an open question.

6.3 Verification

Our description of the uses of the BCM approach focused on five of six steps outlined as part of a systematic change process in Section 2. The sixth step involves verifying the implementation to determine whether the change has been made correctly. The BCM tool can also be used for this step to support simplistic data-flow based verification tasks. After mapping the changed source code to model elements, a developer can perform the relatedness query on a source and a target element to see if any unexpected flows may occur between the model elements. For example, consider the User Authentication concern from Section 3. If there is an unexpected flow from *Unauthenticated* to *Authenticated*, a malicious user might be able to gain unauthorized access to jFTPd.

This approach is a simplistic form of model checking [10]. Unlike model checking tools, the developer is responsible, using the BCM approach, to be systematic about the queries and is more limited in the queries that can be run. An advantage compared to existing source code model checking tools, such as Bandera [5], is that a higher-level model can be used. By higher-level, we mean that the states in a BCM model represent large pieces of processing, than a particular localized piece of state represented in Bandera. More investigation is needed to understand whether bounds can be placed on the uncertainty of the verification queries performed using the BCM tool. For example, is there a more precise way for a developer to know if they have captured enough of a concern to be confident about the correctness of a change.

6.4 Modularizing the Concern

In some cases, once a concern is identified in the code, it may be advantageous to capture that concern explicitly. Aspect-oriented approaches [11], such as AspectJ, [4] provide one means of modularizing and separating crosscutting code. Before creating an aspect, it is important to understand how the code contributing to a concern works. The BCM tool can help in this step but would need to be combined with other tools, such as refactoring tools, to help create the appropriate joinpoints between the concern and the core code.

7 RELATED WORK

In helping a developer during a change task, the BCM approach is similar to work on impact analysis. In creating a model of a program from existing artifacts, the BCM approach is similar to existing behavioural reverse engineering approaches. In helping to identify code related to a particular concern or feature, the BCM approach is similar to feature-finding tools.

[4] http://www.aspectj.org

7.1 Impact Analysis

To understand the scope of a software enhancement, a developer may prior to making a change, perform an impact analysis (e.g., [1]). Given a point in the system involved in the enhancement, an impact analysis determines others points in the system that are transitively dependent upon the seed point. This set of points is intended to help a developer estimate the effort required to make the change, determine the code that may need to be examined to make the change correctly, amongst other tasks.

In allowing a developer to investigate control- and data-dependencies in the code, BCM is similar to code-based impact analysis techniques. The BCM approach differs from these techniques in two ways. First, using BCM, a developer need only form a model and link it to the code when analyis is needed that involves that behaviour. Other impact analysis techniques, such as Sneed's [15], rely on traceability between artifacts, such as between use cases and the code, to be in place for the whole system. A developer using BCM thus pays the cost of the traceability only when the impact analysis is needed.

Second, using the BCM approach, a developer need not compute nor analyze a larger scope of impact in the system than necessary as the analysis is directed by the behavioural model. In comparison to slicing approaches to impact analysis [6], the BCM approach may thus scale better.

7.2 Reverse Engineering Tools

Reverse engineering tools help a developer build models of an existing system from source code and other artifacts. We limit out comparison to tools that produce behavioural models, models which describe how (part of) a system works.

Several researchers have developed techniques for generating state machine representations, such as UML statecharts, from scenario diagrams or message sequence charts [12, 18]. These tools build a state machine representation per class, and thus cannot model the crosscutting behaviour that is possible in the BCM approach.

The Shimba [16] tool can generate sequence diagrams that model crosscutting behaviour at the class and class member level. In contrast, the BCM approach can model crosscutting behaviour at the statement level, producing more abstract and compact models by summarizing a collection of statements (possibly across classes) into a single element in a BCM model.

Di Lucca and colleagues has considered the recovery of use case models from object-oriented code [13]. In this approach, a developer identifies statements which correspond to user-level input or output events, and then a tool automatically identifies the portions of the code base that correspond to these potential use cases. This approach is limited to identifying code that corresponds to externally visible behaviour; the BCM approach does not have this limitation.

7.3 Feature-finding Tools

To help support software change, a number of tools have been built to aid in the identification of the parts of a code base related to a particular feature, or concern.

Wilde and Wong each use an approach based on comparing execution information collected for test cases exercising a feature with execution information collected for test cases that do not exercise the feature [19, 20]. Chen and Rajlich advocate the determination of code related to a feature through a systematic exploration of an abstract program dependence graph representation of a system [4]. The Aspect Browser tool helps a developer identify a concern through the use of lexical queries [7]. The Aspect Mining tool (AMT) [8] augments the queries of Aspect Browser by supporting lexical queries over expressions that include type information. The FEAT tool supports concern identification through the use of structural queries [14].

These feature-finding tools complement the BCM approach by helping identify code of interest when planning a change. The identified code can be used as a basis to help build a BCM model. The BCM model can then be used to help reason about different design possibilities, and to help implement and verify the modification.

8 Summary

When performing modification tasks, developers often encounter crosscutting concerns. It is difficult for developers to understand how modifications interact with these concerns. Current tools help a developer analyze the existing code, but do not help the developer reason about, implement, or analyze a modification.

In this paper, we have presented the Behavioural Concern Modelling approach and tool that provide direct support for a systematic approach to modification tasks. The tool helps a developer model concerns pertinent to a modification and supports the querying of source through a created model. A developer may then use the model to reason about design choices and may use the model as a guide to performing the modification.

9 Acknowledgements

We thank Jonathan Sillito for comments on an earlier draft of this paper. This research was funded in part by an IBM University Partnership Research Grant, and in part by a University of British Columbia graduate fellowship.

References

[1] R. Arnold and S. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] E. Baniassad and G. Murphy. Conceptual module querying for software reengineering. In *Proc. of Int'l Conf. on Soft. Eng.*, pages 64–73. IEEE Comp. Soc. Press, 1998.

[3] E. Baniassad, G. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *Proc. of the 1st Conf. on Aspect-oriented Software Development*, pages 120–126. ACM Press, 2002.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proc. of the 8th Int'l Workshop on Program Comprehension*, pages 241–247. IEEE Comp. Soc. Press, 2000.

[5] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. of Int'l Conf. on Soft. Eng.*, pages 439–448. IEEE Comp. Soc. Press, 2000.

[6] K. Gallagher and J. Lyle. Using program slicing in soft. maint. *IEEE Trans. on Soft. Eng.*, 17(8):751–761, 1991.

[7] W. Griswold, J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of Int'l Conf. on Soft. Eng.*, pages 265–274. IEEE Comp. Soc. Press, 2001.

[8] J. Hanneman and G. Kiczales. Overcoming the prevalent decomposition of legacy code. Workshop on Advanced Separation of Concerns at the Int'l Conf. on Soft. Eng., 2001.

[9] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computing*, 98(2):142–170, 1992.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, pages 220–242. Springer, 1997.

[12] K. Koskimies, T. Syst, J. Tuomi, and T. Mnnist. Automated support for modeling OO software. *IEEE Software*, 15(1):87–94, 1998.

[13] G. A. D. Lucca, A. R. Fasolino, and U. D. C. i. Recovering use case models from object-oriented code: a thread-base d approach. In *Working Conf. on Reverse Engineering*, pages 108–117. IEEE Comp. Soc. Press, 2000.

[14] M. Robillard and G. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the Int'l Conf. on Soft. Eng.*, pages 406–416. IEEE Comp. Soc. Press, 2002.

[15] H. Sneed. Impact analysis of maintenance tasks for a distributed object-oriented system. In *Proc. of Int'l Conf. on Soft. Maint.*, pages 180–189. IEEE Comp. Soc. Press, 2001.

[16] T. Systa, K. Koskimies, and H. Muller. Shimba – an environment for reverse engineering java softare systems. In *Soft.: Pract. and Exp.*, volume 31, 4, pages 371–394, 2001.

[17] M. Weiser. Program slicing. In *Proc of the 5th Int'l Conf. on Soft. Eng.*, pages 439–449. IEEE Comp. Soc. Press, 1981.

[18] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of Int'l Conf. on Soft. Eng.*, pages 314–323. IEEE Comp. Soc. Press, 2000.

[19] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proc. of Int'l Conf. on Soft. Maint.*, pages 200–205. IEEE Comp. Soc. Press, 1992.

[20] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *Proc. of the Symp. on App.-Specific Sys. and Soft. Eng. and Tech. (ASSET)*, pages 192–203. IEEE Comp. Soc. Press, 1999.