Behavioural Concern Modelling for Software Change Tasks

by

Albert Yee-Hang Lai

B.Sc., University of British Columbia, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

December 2001

© Albert Yee-Hang Lai, 2001

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

December 6, 2001

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. Most of the time, developers attempt to make a change use an ad-hoc process with tools that help in gaining some understanding of the existing system, but which do not provide any specific support for reasoning about, implementing, or analyzing the modification.

This thesis presents the Behavioural Concern Modelling (BCM) approach and tool that provide direct support for a systematic approach to modification tasks. This approach helps a developer create a partial, abstract, grounded behavioural model of a concern or concerns. The model is grounded in that the relationship between the model and the code is explicit: A developer describes which code contributes to each part of the model. The examples described use a finite state machine as a model and show how the approach can help a developer capture a concern, reason about design options, and implement modifications.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Gail Murphy, for her advice and encouragement. I would also like to thank my parents for their patience and support.

ALBERT YEE-HANG LAI

The University of British Columbia
December 2001

Chapter 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 span, or crosscut, multiple modules, this approach begins to fall apart [2]: 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.

The ad-hoc process 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, or the underlying concerns, 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 modification's code impacts other crosscutting concerns because those concerns are also implicit.

Existing tools can help the developer with some parts of some of these subtasks. Lexical searching tools, such as grep and Aspect Browser [10], can help identify relevant code. Structural analyzers, such as FEAT [18], flow analyzers, such as program slicers [21],

and some reverse engineering tools, such as Shimba [19], 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 help a developer reason about, implement, analyze, or verify the modification because they focus on the existing system, not the system with the modification.

The thesis of this research is 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*. The *behavioural* characteristic of the model helps a developer reason about how the existing code works and how the modification might work. The *partial* characteristic enables a developer to model only those parts of a concern relevant to the task at hand. The *abstract* characteristic ensures that the model is of a size and complexity amenable for the developer to reason about. The *grounded* characteristic maintains 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 enables a developer to analyze whether the data- and control-flows in the system respect the model.

To investigate the use of such models, an approach called Behavioural Concern Modelling (BCM) and a supporting tool have been developed. In the BCM approach, a developer posits all or part of a finite-state machine (FSM) representing the behaviour of a concern or concerns, and then uses the BCM tool to determine how data- and control-flows relate to the posited state machine. The BCM tool builds on previous work in conceptual modules [1]. A conceptual module (CM) is a logical module, consisting of a set of possibly non-contiguous lines in the source, that can be overlaid on an existing system. Relationships between CM's can be established based on flow analysis between the lines of code mapped to different CM's. The BCM tool supports CM's for Java [9] and enables developers to represent the states and transitions of a FSM with CM's.

1.1 A Sample Modification

To clarify the BCM approach, I describe the use of the approach to assess a change to a FTP server. The approach supports five of six steps in a systematic modification process (Figure 1.1).

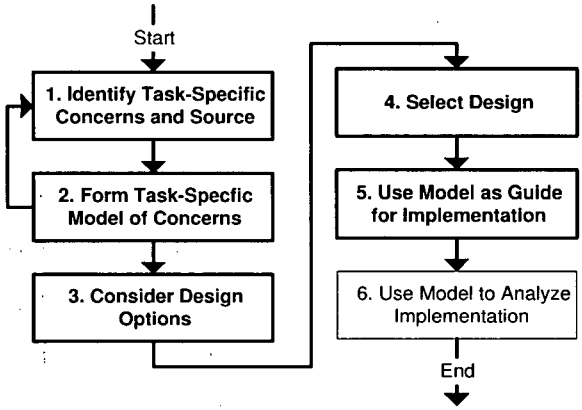

Figure 1.1: Modification Process

1. The developer identifies concerns, and their associated source code, in the existing system that are relevant to the modification.

2. The developer forms a grounded, abstract, partial, behavioural model of the concerns pertinent to the given modification task.

3. The developer considers several different design options and models the ones worth further consideration.

4. The developer selects a design.

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

6. The developer uses the model to help analyze the implementation.

The role of the behavioural model and BCM in the last step of the process is discussed in Section 5.1.4. The posited six step modification does not directly correspond to any documented existing process.

1.1.1 Modifying a FTP Server

jFTPd[1] is a FTP server written in Java that supports basic FTP commands and anonymous login. The modification task in consideration is the addition of named-user logins to the jFTPd system. This modification will either build on or impact the anonymous login concern.

The first step in the process requires the identification of the code (and concerns) relevant to the anonymous login concern. The second step requires the formation of the modification-specific, behavioural model. Although these two steps could be performed separately, a developer may find it useful to iteratively build the model as the developer identifies the relevant code. Furthermore, the developer can use the model to help identify code of interest.

To start, the source code implementing the `USER` FTP command is identified: This command must exist if anonymous login is supported. The `doUserCommand` method in the `FTPConnection` class is identified using grep. This method supports this command by setting the `userName` field based on user input. A subset of the `doUserCommand` is modelled as a *setUser* transition (Figure 1.4a). In the BCM tool, a CM is created to represent this transition. Four lines of code marked with a asterisk in Figure 1.2 are associated with that CM.

This fragment of a model must be expanded to support reasoning about the change. Because FTP user authentication involves the two commands `USER` and `PASS`, it is important to identify the code that implements the `PASS` command. The `doPassCommand` method is identified using grep. Examination of this method reveals that it is responsible for several different functions: it decides whether a user has permission to log in, and it

[1] Available from http://jftpd.prominic.org/1.3/index.html. The code for jFTPd comprises 11 classes and approximately 3000 lines of code.

```
protected boolean doUserCommand(String line) {
  if (line.length() <= 5)
    return false;
*  if (anonUser) {
    out.print("530 Can't change user from guest login.");
*  } 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 your complete e-mail address.");
    } else {
      out.print("331 Password required for "+user+".");
    }
*  }
  return true;
}
```

Figure 1.2: Code Associated with SetUser Transition

decides whether to grant or deny access to the user. Several model elements are created to model this behaviour: a *handleAnonPass* transition, a *permitAnonymousLogin* state, an *authenticateAnonymousUser* transition and a *rejectUser* transition (Figure 1.4b). Each model element is associated with code as illustrated in Figure 1.3.

```
  protected boolean doPassCommand(String line) {
1   if (userName == null) {
      out.print("503 Login with USER first.\\n");
      return true;
    }
2   String userLower = userName.toLowerCase();
2   if (userLower.equals("ftp") || userLower.equals("anonymous")) {
      printWelcome(line);
3     authorized = true;
3     anonUser   = true;
      return true;
2   } else {
      out.print("530 Login incorrect.\\n");
4     userName = null;
      return true;
2   }
  }
```

1 HandleAnonPass Transition
2 PermitAnonymousLogin state
3 AuthenticateAnonymousUser transition
4 RejectUser

Figure 1.3: Code Associated with Various States and Transitions

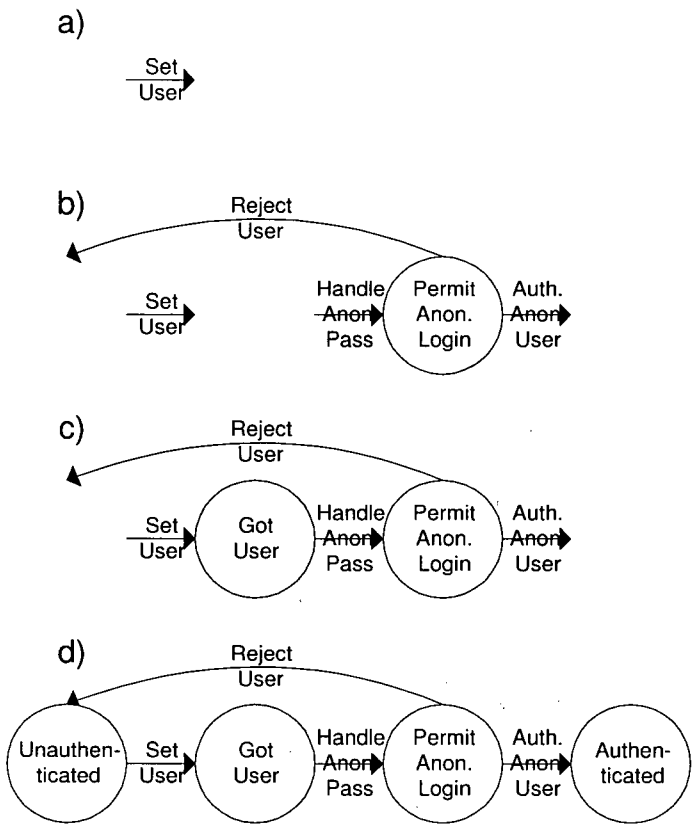

Figure 1.4: Steps in Building a Model for jFTPd

In the FTP user-authentication mechanisms, the user name is passed from the implementation of the `USER` command to the implementation of the `PASS` command. Thus there must be a data-flow between the *setUser* transition and *handleAnonPass* transition. Because the `USER` command must be issued before the `PASS` command in order for the login to be successful, there must exist a control-flow between the *setUser* and *handleAnonPass*. Both the data-flow and the control-flow form part of the state between the transitions.

The manual tracing of these flows is a tedious task that can be avoided with the use of a *relatedness* query supported by the BCM tool. This query examines the data- and control-flows between two CM's within a given context as specified by a third, *context*, CM. The query responds with the set of statements that comprise the flows, as well as class and method summary information for those statements. A relatedness query between *setUser*

and *handleAnonPass* with all the jFTPd classes as the context returns the following class summary.

`FTPConnection` → `PassiveConnection`

`FTPConnection` → `WildcardFilter`

`FTPConnection` → `FTPHandler`

`FTPHandler` → `FTPConnection`

As is often the case, a first query returns results that are very broad. The query context is refined by removing `WildcardFilter`. Since this class extends `java.io.FilenameFilter`, it is unlikely to be related to the *setUser* and *handlePass* transitions.[2]

The method summary reports several methods that implement FTP commands unrelated to *setUser* and *handlePass*. Some of these methods include `doListCommand`, `doCwdCommand`, and `doDeleCommand`. The commands implement functionality unrelated to the user authentication concern and thus they are removed from the query context.

With the refinements, the query between *setUser* and *handlePass* returns the following method summary:

`doPassCommand` → `printWelcome`

`doUserCommand` → `doPassCommand`

`doCommand` → `doUserCommand`

`doCommand` → `doPassCommand`

`run` → `doCommand`

`doCommand` → `run`

`doCommand` → `setBusy`

`doCommand` → `setLastCommandTime`

Examination of `doCommand` reveals that it parses FTP commands and calls the appropriate methods to handle the commands. Examination of `run` reveals that it reads

[2]The `WildcardFilter` is used to implement the LIST command and is not related to user authentication.

FTP commands and passes them to `doCommand`. A *GotUser* state, containing the loop in `run` and the parsing statements in `doCommand`, is created as shown in Figure 1.4c.

As noted previously, `doCommand` parses commands and dispatches them to other methods. The commands that `doCommand` parses includes some that are typically used when the system is in an unauthenticated state, such as `USER`, and some that are used when the system is in an authenticated state, such as `LIST`. Because the parsing statements execute when the system is in both of these states, the statements not only belong to the *GotUser* state, but also to an *Unauthenticated* and an *Authenticated* state. Other statements also execute while the system is in those states, but those statements are not relevant to the change task. For example, one might consider whether or not to model the individual FTP command handlers because some of the corresponding FTP commands are only valid when a user is authenticated. However, a model including the command handlers would not aid in the addition of named-user logins and thus they are not included in the model.

Steps one and two of the modification process are repeated until the model is complete enough to reason about the modification task. Queries introduced as part of the original CM tool help check that the model is complete by reporting the interface to a logical module represented by a CM. These queries elucidate the data- and control-flows to and from the interface. A developer can examine the query results to verify that none of the interface flows are pertinent to the model and thus the model is sufficiently complete. A new CM is created that consists of all the state and transitions, and their associated cross-cutting code. The interface query determines which variables and fields are *inputs* to this CM, which variables and fields are *outputs*, and which *control transfers* emanate from the CM.

No unexpected values are reported as part of the *outputs* list or the *control transfer* list, but unexpected values are reported as *inputs* to the CM. Specifically, the query reports two fields, `FTPConnection.anonUser` and `FTPConnection.userName`, as inputs. jFTPd uses the `anonUser` field to indicate whether the current user is an anonymous user, and thus all definitions of this field should be part of the model. The input query

reports that an instance initializer for `anonUser` exists outside of CM; this statement is added to the *Unauthenticated* state. Similarly, the `userName` field is also defined in an instance initializer and that is also added to the *Unauthenticated* state. At this point, the model is sufficiently complete (Figure 1.4d).

For step three, there are two different design options for implementing named-user authentication. One option is to consider anonymous login as an instance of named-user logins where the anonymous user has the name "anonymous". The second option is to consider anonymous login to be a special case that is separate from named-user logins.

In the first option, as Figure 1.5 shows, only slight adjustments need to be made to the model, renaming transitions and states to reflect the change from anonymous to named-user authentication. Only one path from the unauthenticated state to the authenticated state is needed. This option is conceptually simple, but it may be difficult to implement policies in which it is useful to treat anonymous users as a separate class of users. For example, there may be a need to limit the number of anonymous logins as well as the total number of named-user logins.

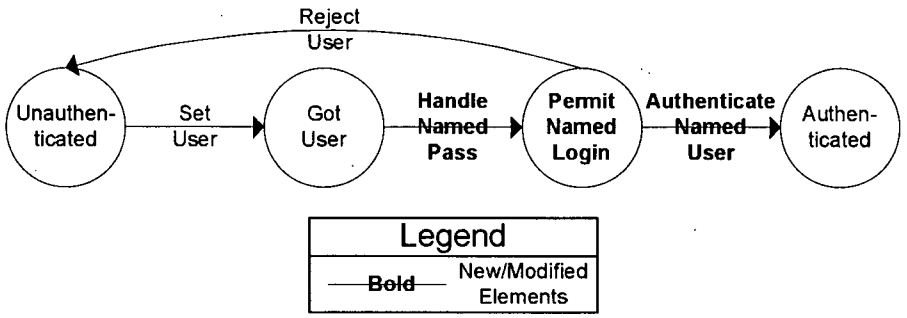

Figure 1.5: jFTPd Model with Single Login Path

In the second option, anonymous login and named-user login are separate mechanisms, as Figure 1.6 shows. Anonymous logins would be more explicitly represented in the source code, and it may be easier to implement such policies as described above.

After the design options are considered, a design is selected as part of step four. For step five, the model can serve as a guide to aid developers in locating points in the

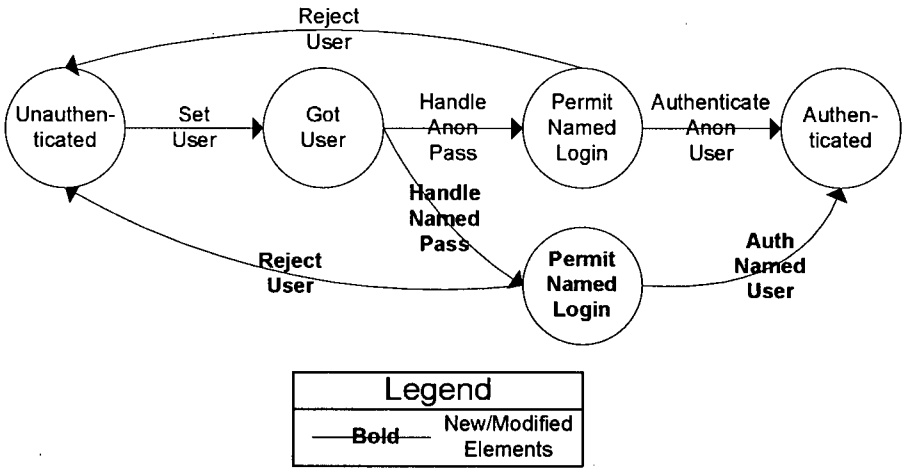

Figure 1.6: jFTPd Model with Separate Named Login Path

code that need to be modified. For instance, if the first design option was selected, the model could be used to determine that no modifications were required for the *Unauthenticated*, *GotUser*, and *Authenticated* states or the *setUser* transition, amongst others. The model could aid in determining that the *PermitLogin* state must be modified, and an *AuthenticateNamedUser* transition added. In addition to providing design information about the modification, because it is grounded, the model can point a developer to specific code that must be considered. For example, consider the first option for named-user support was chosen. The change to the *PermitAnonymousLogin* transition to *PermitNamedLogin* would indicate that the some of the lines in the method `doPassCommand` in Figure 1.3 would need to be changed.

1.2 Thesis Outline

The contribution of this thesis is an approach for modelling the behaviour of a concern. This model can help to manage a concern during a software enhancement task. This thesis also demonstrates a need to ground a model in the source to support such tasks. Additional uses of the model are discussed in Section 5.1.

This chapter presented an overview of the approach, and provided an example of

how the approach might be used to perform a modification task. The remaining chapters of the thesis are organized as follows. Chapter 2 describes the BCM tool interface and tool internals including how it calculates data- and control- flows. Chapter 3 presents a case study on an outstanding change request to a public-domain web browser. Chapter4 compares this work with other related work, including reverse engineering approaches and concern identification tools. Chapter 5 summarizes this work.

Chapter 2

The BCM Tool

The grounding of a behavioural concern model in the source is a critical part of the BCM approach. Without this grounding, it is easy to create a model that overlooks important details about how the concern is implemented. Since the approach is partial, there may be details that a developer does choose to leave out because they are not relevant to the task at hand. However, by grounding the model, a the developer must explicitly choose which details are relevant and which can be safely ignored or delayed. This section describes the tool support provided to help a developer create and investigate the grounding of the model.

Section 2.1 describes the architecture of the BCM tool. Section 2.2 provides an overview of the interface between the user of the tool and the tool itself. The rest of this chapter describes some of the details of the tool's structure. Section2.3.1 describes the data structure used to store information on methods. Sections 2.3.2- 2.4 provide details about the data-flow analyzer. Section 2.5 discusses possibilites for tool extensions to support finer-grained queries.

2.1 Tool Architecture

Figure 2.1 presents an architectural overview of the BCM tool. BCM uses Jikes Bytecode Toolkit (JikesBT[1]) to read Java class files [16]. Within BCM a data-flow analyzer constructs a control-flow graph for the bytecodes in a given method and performs data-flow analysis

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

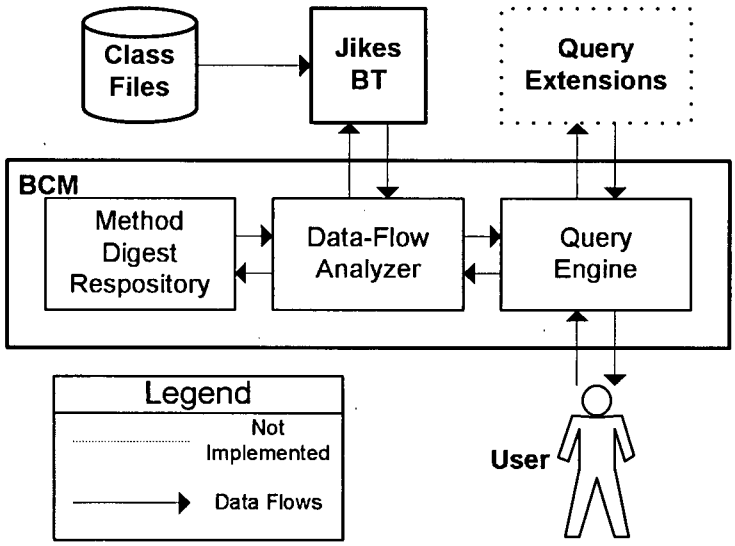

Figure 2.1: BCM Architecture

on that method. To reduce the memory requirements for BCM, the data-flow analyzer summaries the side effects of methods and stores them as method digests in the method digest repository. Data- and control-flows for methods are not retained between calculations and are only recalculated when such information is required. This typically occurs when a user has a query about the flows through a particular method. As a given query expands over several methods, the flows within those methods are recalculated on demand.

2.2 Tool Interface

The BCM tool enables a developer to construct a finite state machine (FSM) model and to map source code to the state and transition model elements. A developer can perform queries about how the model elements relate to each other and how they relate to the source code. These features of BCM are built on a Conceptual Modules representation of model elements.

A Conceptual Module (CM) captures a collection of source code lines and treats them as a logical unit. The source code lines need not be contiguous nor do they need to be restricted to the lines from a single method or class. A developer creates a CM from a set of lines by specifying the names and signatures of the lines' enclosing methods and their line numbers within those methods. Lines are selected from a developer-specified set of classes. This set of classes represents the *world* for CM's. All CM data- and control-flow analyses are conservative with respect to this world.

Currently, states and transitions are simply named CM's. BCM does not provide a graphical interface for drawing FSM's. A developer must depend on naming conventions to distinguish between states and transitions and to identify the connections between them.

Once the user has a partial model containing some states or transitions, the user can perform a relatedness query. This query requires three inputs: a source, a target and a context. The source and target can be any states or transitions. Like the states and transitions, the context is a CM and is specified by a set of lines[2]. The query starts from the lines identified by the source and follows any data- or control-flow connections within the context. Once the query has identified all the flows from the source, it filters out the flows that do not reach the target. The query reports the remaining flows, a method summary, and a class summary in a textual interface.

The user can also perform an interface query on a CM as described in Section1.1.1.

2.3 Tool Internals

BCM uses JikesBT to process the classes specified for the world. BCM pre-processes the classes to improve the efficiency of further analyses. For each bytecode in a method, BCM calculates data- and control-flows by simulating the operations performed on the Java Virtual Machine (JVM) stack and local variables (Figure2.2). BCM uses the method simulation results to generate a method digest that summarizes the method's side effects.

Using a fixpoint algorithm, BCM iterates over all the methods in the world until

[2]For an example of the inputs and outputs to the query, see Appendix A

all the method digests have reached a fix point. If dependencies between two methods are cyclic, the algorithm does not try to order them. If depencencies between two methods are acyclic, the algorithm iterates over the dependent method later. This reduces the number of iterations until a fix point is reached[3].

As the algorithm iterates over all the methods, it keeps track of all the side effects it has encountered and adds new side effects as it encounters them. Thus the number of noted side effects monotonicly increases over the algorithm's iterations. Since the total number of side effects in a given set of methods is fixed, the algorithm will eventually run to completion.

Once the algorithm completes, only the method digests are retained. Data- and control-flows within methods are re-calculated on demand.

```
- build control flow graph for method
- add the first bytecode in the method to a bytecode queue
- loop while there are bytecodes in the queue
  - get the next bytecode in the queue
  - if the current bytecode throws an exception that is caught
    by the current method, add the exception handler bytecode to
    the bytecode queue
  - simulate the bytecode
    - simulate local variable loads
    - simulate local variable stores
    - simulate stack pops
    - simulate stack pushes
    - if the current bytecode is a method call, look up the
      corresponding method digest and note method side effects
  - if the results of the simulation have not been noted before,
    propagate the results to immediate bytecodes in the control-flow
    and add those bytecodes to the bytecode queue
```

Figure 2.2: Pseudo-code for Method Digest Calculation

2.3.1 Method Digests

A method digest contains information on which arguments are used and defined in a given method. Since the Java VM stores method arguments in a local variable array, the algorithm keeps track of which local variable entries are method arguments. When the algorithm

[3]For a brief analysis of the run-time complexity of this algorithm see Section 2.3.4.

simulates a bytecode that defines or uses a method argument, it notes this in the method digest for the method.

During method simulation, the algorithm identifies the bytecodes that define and use fields, local variables and method arguments. It also performs data-flow analysis on anything else that can be placed on the VM stack including constants, bytecode address references, and computation results not stored in local variables. The algorithm does not maintain actual values of objects, but it keeps track of the bytecode index at which a reference was defined or used. The algorithm also keeps track of the class of a particular reference to more accurately simulate exception throwing bytecodes. The throwing of an exception requires at least two bytecodes: one to place an instance of an exception on the stack and one to get the value from the stack and throw it.

2.3.2 Data flow

The algorithm generates a method digest by simulating the operations performed for each of the method's bytecodes. Each bytecode performs a series of operations possibly involving the Java VM's stack, local variable array and a variety of data types. For example, the stack for a single method call may contain values of base types such as `int`, objects such as `HashMap` and method references such as `java.util.HashMap.<init>(int)`. Henceforth, these types are collected referred to as *references*. In BCM, there is a hierarchy of classes used to represent the references that a Java VM can manipulate. Each type of reference has several pieces of associated information:

- the bytecode index at which this reference was defined
- depending on the type of reference:
  - the object's class (e.g., `int`, `java.util.HashMap$Entry`)
  - an address (used in *Jump Sub-Routine* bytecode)
  - a local variable index to keep track of local variable a reference came from
  - a field reference (e.g., `java.lang.System.out`)
  - a method reference

The algorithm uses these references to perform data-flow analysis by identifying where the references are defined and used. When the algorithm simulates a bytecode that defines a reference, the reference's bytecode index is set to the current bytecode index. For example, in Figure 2.3, bytecode 2 defines a reference on the stack. This reference has a bytecode index field with a value of 2, the bytecode index at which the reference was defined. Furthermore, this reference is on the stack prior to simulating bytecode 3 and is used by bytecode 3. The algorithm does not keep track of the actual value of reference. For example, if the algorithm encountered `i = 10`, the algorithm would not store the value `10`, but would keep track of which bytecode defined `i`.

References can be defined by a number of bytecodes including:

- load constant (e.g., 3.14159, "Hello World")
- arithmetic, bit and logic operations (e.g., multiply, xor, ==)
- method/constructor calls
- assignment to an array (e.g., `array[i] = 0`)
- assignment to a field
- return statement (e.g., `return 123`)

When the algorithm simulates the following bytecodes, references are used and their usage is noted:

- method/constructor calls with arguments
- arithmetic, bit, and logic operations
- assignment to an array (via an `aastore` bytecode)
- reading a value from an array (via an `aaload` bytecode)
- field access

When the algorithm simulates an instance or interface method call, it uses class hierarchy analysis to calculate a conservative digest for all methods that can execute as a result of the call. For example if several classes implement a `doit(List arg)` method and only some of the classes modify `arg`, then the calculated digest reports `arg` is defined by `doIt`.

2.3.3 Control Flow

For each method, the algorithm builds a control-flow graph (CFG) where each vertex is a bytecode and an edge is a control flow between bytecodes. For some bytecodes, control-flow proceeds with the next instruction, but *goto*'s, *if*'s, *switch*'s, and *try/catch/finally* blocks require special handling.

The control-flow graph does not have edges between method calls and method targets. Instead, the algorithm uses method digests to simulate the effects of a target method. JikesBT determines the method target for method-call bytecodes that only have one possible target, such as `InvokeStatic`. For bytecodes that have multiple possible targets, BCM uses JikesBT to determining all the methods that override or implement a given method. BCM uses those methods to calculate a conservative digest for simulation.

While building a method's CFG, the algorithm checks the method for an *exception table*. In Java bytecode, exception tables describe which exceptions are locally handled by what code within the same method. Each entry in the table describes the range of bytecodes handled, the type of exception handled and the bytecode index of the handler. The exception may also contain entries for `finally` blocks. The only difference with these entries is that the type of exception handled is stated as *any*. In the example in Figure2.3, if any exception is thrown between bytecode 4 (inclusive) and bytecode 14 (exclusive), the control-flow will proceed to bytecode 18.

For each declared exception in the exception table, the algorithm examines the bytecodes specified by the exception entry. If the algorithm encounters a method call, it uses JikesBT to determine if the corresponding method can throw the exception specified by the

exception entry or a sub-class of that exception. If the method can throw such an exception, the algorithm adds a control flow edge from the method call to the exception's handler. For other *throwable*'s, the algorithm uses a conservative approximation: all bytecodes specified by an exception table entry can throw the exception specified by the entry. This creates an edge between each of the bytecodes specified and the corresponding handler.

A `finally` block in Java is typically compiled into a subroutine. The subroutine can be called with the *Jump Sub-Routine* bytecode and when the subroutine completes, the *Return* bytecode returns control back to the bytecode following the *Jump*. The control-flows for *Jump* bytecodes are determined when most other control-flows are determined, prior to simulating the method. However, the control-flows for *Return* bytecodes are determined during simulation of the method because different callers to the subroutine will have different return points.

Execution Contexts

Consider the bytecodes in Figure 2.3 on page 23. When the Jump Sub-Routine bytecode at index 14 is simulated, the return bytecode index of 17 is pushed on the stack. Execution then continues at bytecode 24 where the return index is stored in a local variable. When the sub-routine returns at bytecode 35, it uses the bytecode index in local variable 4 to return to the correct bytecode index. Similarly, when the Jump Sub-Routine bytecode at index 19 is simulated, the return bytecode index of 22 is pushed on the stack. Eventually, the sub-routine returns at bytecode 35, but this time it returns to bytecode index 22. Note that there are two separate control-flow paths that share a common sub-routine.

To accurately simulate this behaviour, each bytecode needs to keep track of the stack and local variables for each control-flow that enters the bytecode. This provides a context for each of the different ways a bytecode can be executed.

2.3.4 Computing Resources

In general, the number of classes or methods in a given application does not significantly affect the computing resources required by BCM. At any given time, BCM retains only the data- and control-flows for the method it is currently analyzing. The more complex a method's control-flows are, the more CPU and memory BCM requires to analyze the method.

The data-flow analysis first performs a topological sort on the methods. The sort uses call relationships between methods to determine which method in a pair of methods is independent and thus should be ordered earlier in the sort. Methods with cycles between them are ordered arbitrarily. The topological sort is an optimization to reduce the number of fix point iterations over all the methods. A topological sort runs in time $O(|methods| + |methodCalls|)$ [5].

In an ideal situation, there are no cycles between methods and the algorithm completes after iteration because the topological sort has ordered them in order of dependence. The worst case situation arises when all the methods are all dependent on each other. In this case when one method digest is updated, potentially all the other method digests need to be updated. However, each method digest stores the side effects of the methods it calls. Thus the algorithm is bounded by $O(|methods|^2)$ iterations.

For each method, data-flow information between bytecodes is propagated similar to how method digests are progated between methods. Thus the algorithm iterates over a maximum of $O(|bytecodes|^2)$ bytecodes per method. Thus in the ideal situation, the algorithm completes in $O(|methods||bytecodes|^2)$ and in the worst case $O(|methods|^2|bytecodes|^2)$.

Without statistical analysis of a large percentage of the Java code in existance, it is almost impossible to determine the characteristics of the average case situation. However, BCM has processed several system in several minutes. For example, BCM has processed a system called XBrowser (Chapter 3) consisting of 29000 lines of Java source code in 10 minutes on a PIII 1Ghz with 512MB of RAM running Java 1.3 HotSpot VM.

2.4 Tool Limitations

BCM does not support analysis of native methods. A particular case of this problem 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 between `start` and `run` is not detected by BCM. One workaround is to write a temporary subclass of `Thread` with an overridden `start` method that explicitly calls `run`.

To a lesser extent, native methods also cause problems in the analysis of code that uses reflection. The methods that implement reflection are native and can not be analyzed by BCM. To complicate matters, the actual value of variables is sometimes required for analysis, as in `Class.forName(myClass)` where `myClass` is a `String`.

BCM also does not support alias analysis. Thus some data-flow dependencies between lines of source code may not be identified.

2.5 Tool Extensions

Section 1.1.1 described the use of a context CM to refine query results. BCM has another feature that provides users with finer-grained control over query results. Users can write plug-ins in Java that specify which data- and control-flows to consider as part of the query and which flows to report as part of the query results.

The plug-in feature uses a variation of the visitor pattern [8] to perform frontier exploration. A developer's plug-in does not have to visit all bytecodes in a given set of classes. The plug-in feature provides a queue that contains the current frontier and data- and control-flow information about each bytecode. The developer is only responsible for deciding which bytecodes should initially be on the queue, what flows to report as query results and what new flows need to be added when new bytecodes are encountered.

2.6 Alternative Flow Analyzer

During the development of BCM, the Soot [20] project was developed. Soot was designed for bytecode optimization using data-flow analysis. There is potential for BCM to use Soot as it's data-flow analyzer, but Soot provides more detailed analysis than BCM requires. Soot uses constraint-based type analysis which is often not necessary in BCM because most of the time only a given dependence is important and not the actual type of the dependence. For example, if a bytecode uses a value defined by another bytecode, only this dependence is important and not the actual type of the valued used. Although BCM could benefit from alias analysis, Soot does not support it. Also, BCM's problems with native methods can not be alleviated through the use of Soot because Soot also performs its analysis on bytecodes.

```
public class MethodSum {
    public static void main(String[] args) {
        int i = 0, j = 1;
        try {
            args[i++] = "Hello";
            j = 2;
        } finally {
            System.out.println(i + j);
        }
    }
}

Method void main(java.lang.String[])
   0 iconst_0
   1 istore_1
   2 iconst_1
   3 istore_2
   4 aload_0
   5 iload_1
   6 iinc 1 1
   9 ldc #1 <String "Hello">
  11 aastore
  12 iconst_2
  13 istore_2
  14 jsr 24
  17 return
  18 astore_3
  19 jsr 24
  22 aload_3
  23 athrow
  24 astore 4
  26 getstatic #7 <Field java.io.PrintStream out>
  29 iload_1
  30 iload_2
  31 iadd
  32 invokevirtual #8 <Method void println(int)>
  35 ret 4
Exception table:
   from   to  target type
     4    14    18   any

MethodSum.main(java.lang.String[]):
  defs:
    bytecode index: 11
    type:           java.lang.String[]
    local var:      0
  uses:
    bytecode index: 4
    type:           java.lang.String[]
    local var:      0
```

Figure 2.3: Bytecode and Method Summary Example

Chapter 3

Evaluating BCM

For the BCM approach to be viable, it has to be possible for developers to create useful models of a concern within a reasonable amount of time. To date, the focus of this work 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? To provide additional evidence that model creation is possible, this section describes a case study of applying the BCM approach to an outstanding change task on a system. 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.

3.1 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.[1] The code for XBrowser comprises 171 classes and approximately 29000 lines of code.

One of the outstanding feature enhancements for XBrowser is a request for Meta-Refresh support. In any HTML document, the HEAD element may contain any number of META elements. Each of these META elements provides metadata such as a document's

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

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 this study, the BCM approach was applied to support the addition of Meta-Refresh feature to XBrowser. Little technical documentation about XBrowser was available, providing a "worst-case" type of situation for applying BCM. Even JavaDoc HTML pages were unavailable.

3.1.1 Modelling the Existing Behaviour

Knowledge of the Meta-Refresh feature helps to identify the XBrowser concerns that are impacted by the change and therefore are of interest to model. First, the addition of Meta-Refresh changes the current URL, and thus there is a navigation concern. Second, the feature requires parsing the current document, and thus there is a document parsing concern.

Similar to the description in Section 1.1.1, partial models of each of these concerns were *grown* through a combination of identifying code snippets of interest with grep, positing model pieces and associating code with those pieces, and using relatedness queries to check that code of interest had been modelled. 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 because the graphical user interface classes fell outside the scope of interest and they were easily filtered out of the queries.

Parts of the navigation concern were identified first. XBrowser supports history navigation using "Back" and "Forward" buttons. Grep was used to search for the string `Back`. The code in figure 3.1 was identified and captured as a *Back* transition in Figure3.2a.

Grep identified a `BackAction` class containing an `actionPerformed` method. This method calls a `showPreviousPage` method which in turn calls a `processXHyperLink` method. The body of the `processXHyperLink` method calls two methods that collectively set the current page to a specified URL. The first method stops the previous page-loading thread, and the second starts a new thread and loads the new page. These methods are captured as a *Process HyperLink* state and the actual method bodies are cap-

```
class BackAction ... {
  void actionPerformed(...) {
    getActiveRenderer().showPreviousPage();
  }
}

class XCustomRenderer ... {
  void showPreviousPage() {
    if( hasBackwardHistory() )
      processXHyperLink(...);
  }
}
```

Figure 3.1: Code Associated with Back Transition

tured as a *Page Load Stop, Page Load Start* transition (Figure 3.2b) [2].

Use of the relatedness query helped check that all of the pertinent code between *Back* and *Page Load Stop, Page Load Start* has been captured. The initial query used *Back* as the source, *Page Load Stop, Page Load Start* as the target, and all the classes in XBrowser as the context. The class summary reported a large number of user interface classes. For example, `XURLComboBox` was reported because the URL it displays must change when the user navigates to a different URL. These classes were removed from the query context. The class summary also reported classes that implement URL history. These classes were also irrelevant to the modification and were removed from the context. Re-running the query reduced the number of classes that needed to be examined. The new class summary only reported one additional unexpected class, `XHTMLEditorKit`. The corresponding method summary contained a call from `XCustomRenderer.destroying()` to `XHTMLEditorKit.destroyAllApplets()`. The method `destroying` contained part of the *Page Load Stop, Page Load Start* transition and terminated any running threads, including Java Applets, associated with a web page. The remaining entries in the method summary were methods that had already been examined while building the model.

The GUI thread's *Page Load Stop, Page Load Start* transition controls the state of the page-loading thread. This behaviour was modelled in the page-loading thread with a *Page Load Stop* transition that enters a *Stopped* state and a *Page Load Start* transition that

[2]This transition represents a sequence of events. First *Page Load Stop* occurs and then *Page Load Start*.

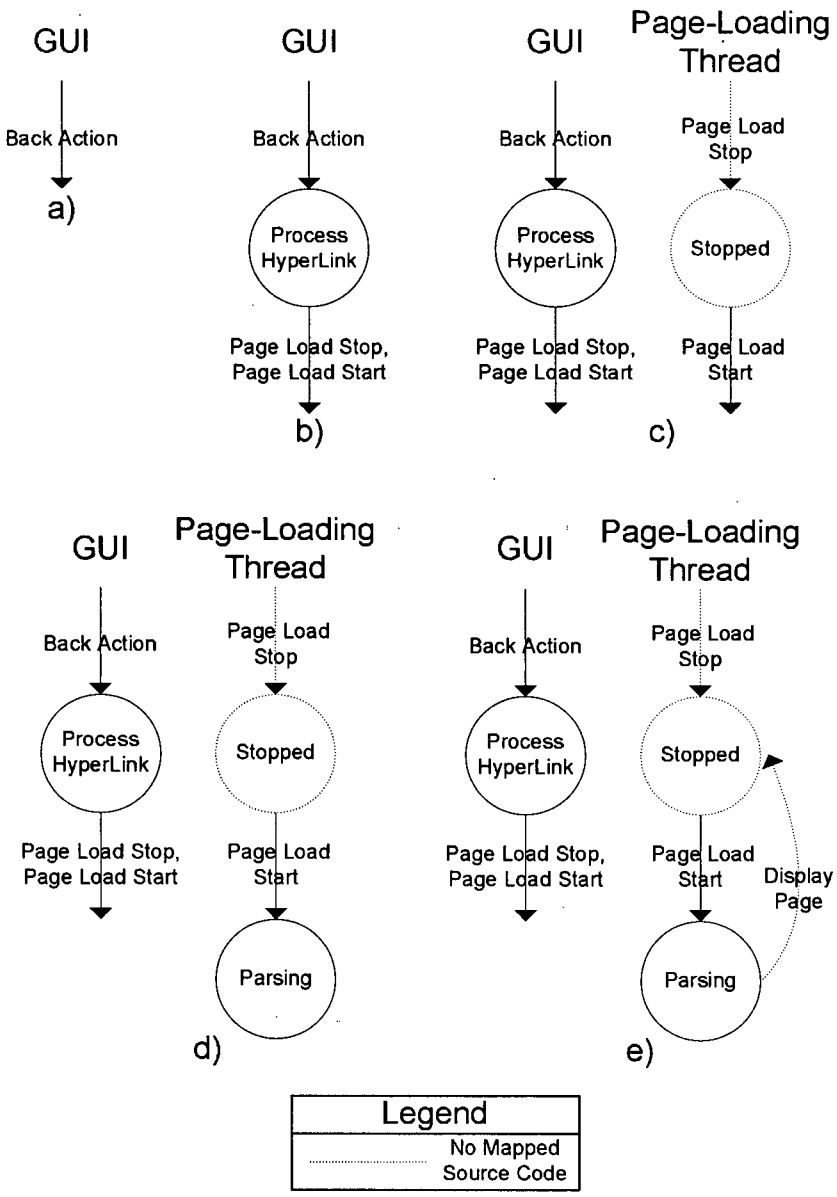

Figure 3.2: Steps in Building XBrowser Model

leaves from the *Stopped* state (Figure 3.2c). These new model elements were isolated from the previous elements to show that the state of the page-loading thread is independent of the other thread.

Under normal circumstances, every state and transition has some associated code.

In this case, only *Page Load Start* has some associated code, a call to `JEditorPane.setPage(...)` that sets the URL for the current page. For *Page Load Stop* and *Stopped*, there is no associated code because their behaviour is completely implemented by `java.lang.Thread`. Although it is possible to obtain source code for `java.lang.Thread`, it is considered to be library code and external to XBrowser. Nonetheless, the new state and transitions served to describe behaviour pertinent to the modification task.

Parts of the document parsing concern were identified next. Knowledge about Swing and experience with using XBrowser helped in identifying the document parsing concern. In Swing, the page displayed in a browser window is set by a call to the `JEditorPane.setPage(...)` method. Since XBrowser supports JavaScript, a call to this method will result in the parsing of the page to execute any JavaScript present in the document. Thus there is a control-flow between `setPage` and the code implementing the document parsing concern. This control-flow must be implemented by both Swing code and XBrowser code since `setPage` is a Swing method and JavaScript is not directly supported by Swing. Of the code that implements the control-flow, only the portions implemented by XBrowser were relevant to the modification task as changes to Swing were not an option. Examination of Swing documentation and the code for the class `XCustomRenderer` revealed that a static initializer registers the class `XHTMLEditorKit` with Swing's `JEditorPane` to handle HTML content. Further examination reveals that the control-flow extends through the methods `XHTMLEditorKit.createDefaultDocument` and `XHTMLDocument.getReader`. The `getReader` method returns an instance of the class `XHTMLReader`. This class contains methods for handling different kinds of HTML tags such as start, end, text and comments tags including JavaScript. The `XHTMLReader` class is captured as a *Parsing* state.

Although a large part of the control flow is in Swing, it is still useful to capture the connection between `setPage` and `XHTMLReader`. The connection is modelled by setting *Parsing* as the target state for the *Page Load Start* transition as in Figure3.2d.

After using Swing to display the page for a URL, the page-loading thread returns to

the *Stopped* state. This behaviour is modelled with a *Display Page* transition from *Parsing* to *Stopped* (Figure 3.2e). The *Display Page* transition does not have any associated code, but serves to describe behaviour pertinent to the modification task.

The *input*, *output* and *control transfer* queries were used to confirm that the model has captured the pertinent portions of the navigation and document parsing concerns. These queries help elucidate the model boundaries when performed on a CM consisting of all the states and transitions. The queries on the XBrowser model reported nothing unexpected.

The Resulting Model

Figure 3.3 shows the model resulting from this iterative process. Most of the model elements describe the behaviour of the navigation concern. Only the *Parsing* state describes the document parsing concern in XBrowser. A large portion of the HTML parsing is handled by Swing and what little parsing code exists in XBrowser is well localized.

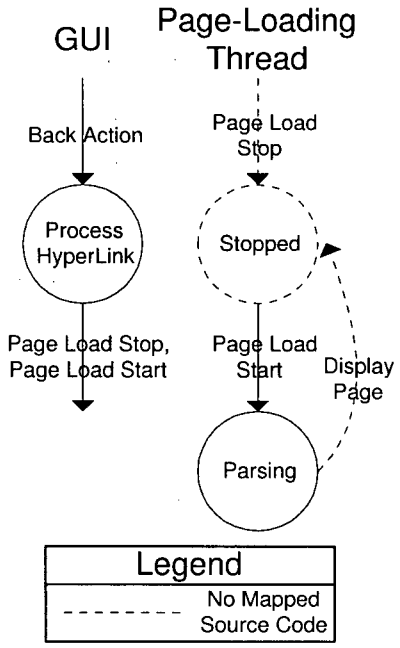

Figure 3.3: Complete XBrowser Model

This model has two interesting features: different fragments of state machines are

used to represent behaviour in different threads, and some model elements 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 the two thread been merged, there would be a set of states representing the cross-product of the individual threads' states. The cross-product of a large number of threads each with a significant number of states would lead to a state explosion. This is an issue with the form of the model and is discussed in Section 5.1.1.

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 Swing and 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.

3.1.2 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. After some deliberation, the approach in Figure 3.4 was selected.

The Refresh behaviour is modelled as a separate thread consisting of three states: *Stopped*, *Waiting*, and *Process HyperLink*. One detail worth noting is the *Process HyperLink* state. Although the behaviour described by this state is similar to the behaviour of the GUI thread's *Process HyperLink* state, the two states execute in different thread contexts. The distinction between the two states is consistent with the rest of the model where different states and transitions are isolated based on the executing thread.

Extension of the model aided in the consideration of subtle pieces of the Meta-Refresh feature. One example is the case of documents with multiple refresh META elements. This was handled by the addition of a *Refresh Stop, Refresh Start* transition from

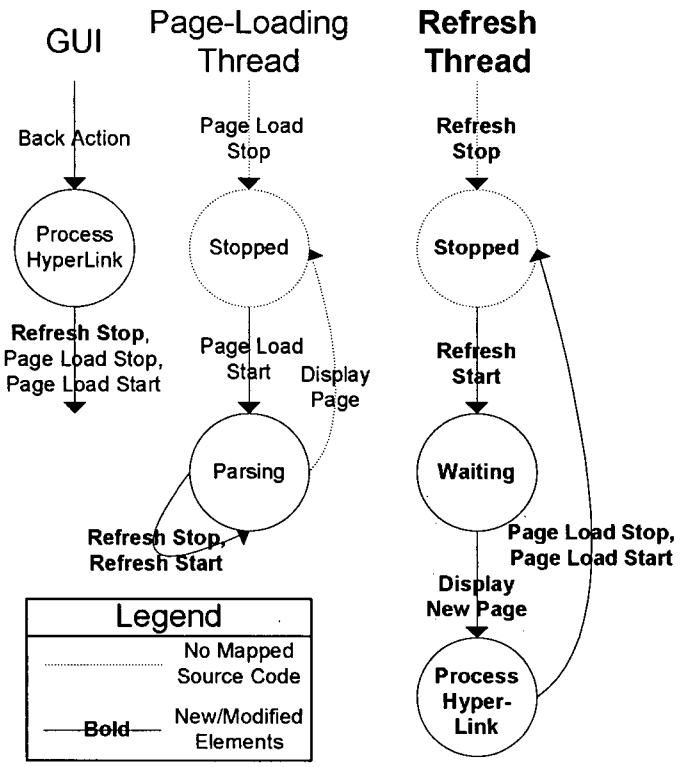

Figure 3.4: XBrowser Model with Meta-Refresh Feature

Parsing to *Parsing*. This transition ensures any existing refresh thread is stopped, and is thus prevented from changing the current URL, before a new refresh is started for the next META element.

Another subtle piece of the Meta-Refresh feature involves changing the current URL before the refresh thread runs to completion. Suppose a user visits a page with a refresh META element. The browser parses the element and starts 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. This behaviour is modelled by modifying the GUI thread's *Page Load Stop, Page Load Start* transition. It is renamed to *Refresh Stop, Page Load Stop, Page Load Start* to reflect the desired behaviour.

3.1.3 Implementing the Meta-Refresh Feature

The model served as a guide to indicate the points in the source that needed modifying or augmenting.

As one example, consider the Refresh thread's *Page Load Stop, Page Load Start* transition. There needs to be code that 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, the reuse of that code is considered. Examination of the code reveals that it is appropriate and it is used to ground the Refresh thread's transition. Also, the code for the GUI thread's implementation of *Refresh Stop* needs to be implemented. The *Page Load Stop, Page Load Start* transition from the GUI thread prior to the model modification maps to the method `destroying`. This method relinquishes resources required by the current page and is a good location to which to add *Refresh Stop*.

As another example, the model served as a guide for selecting an implementation option for the Refresh thread. One option was to implement the thread in a new class, say `XRefresher`. This approach would isolate the refresh behaviour from other existing behaviours. However, this choice would result in a mutual dependence between the Refresh thread and the Renderer, leading to a high degree of coupling. As a result, the mutual dependence was removed by implementing the Refresh thread behaviour as part of an existing class.

It is worth noting that without analyzing the Swing and Java core libraries, it is impossible to determine the exact nature of the interactions between those libraries and XBrowser. In particular, a developer must rely on library documentation to avoid problems with unexpected interactions including thread interactions.

3.2 Summary

BCM models describe existing behaviour and present a framework for considering modifications to the behaviour. These models facilitate the integration of desired and existing

behaviour, and aid developers in considering subtle but important behavioural details. Once a developer has modelled the desired behaviour, the mapping provided by these models supports the developer in identifying sections of code that need modification or augmentation.

Chapter 4

Related Work

In the creation of a model of a program from existing artifacts, the BCM approach is similar to existing reverse engineering and reengineering approaches. In the identification of code related to a concern, the BCM approach is similar to existing concern finding tools. The following sections compare the BCM approach and tool to these two bodies of existing work. In the case of reverse and reengineering tools, the comparisons are limited to those tools that provide either direct support of reengineering or the reverse engineering of behavioural, rather than structural, models.

4.1 Reverse Engineering and Reengineering Tools

Reverse engineering is defined as the analysis of software components and their interrelationships in order to obtain a description of the software a high level of abstraction β]. Unlike most reverse engineering tools, BCM enables developers to abstract concerns that crosscut system structure. This gives developers the freedom to capture concepts without being restricted by existing structures. Reengineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [3]. Unlike other reengineering tools, BCM supports the formation of a user-defined behavioural model through the use of data-flow analysis queries.

4.1.1 Shimba

Shimba is a reverse engineering environment that uses dynamic event traces and static program analysis to automatically generate scenario and state diagrams [19]. Shimba generates such diagrams in the context of existing classes and thus cannot generate such models for concerns that crosscut a system.

Shimba can abstract interactions between objects by recognizing user-specified behavioural patterns, but such abstractions are limited to one of two types: repetition constructs or subscenarios. The repetition construct allows developers to capture repetition of behavioural patterns found in `while`, `for` and `do-while` structures. Developers are limited to abstractions defined by the context of those structures. Subscenarios describe a specific sequence of events common to multiple scenarios. Subscenarios limit developers to abstractions that align along sequences of events. Unlike the abstraction mechanisms in Shimba, BCM allows developers to create abstractions based on any set of possibly non-contiguous lines of source.

Shimba allows developers to filter event traces based on class-level entities such as methods, variables, and interfaces. For example, this allows developers to collect event traces for a specific set of classes. BCM provides additional capabilities by enabling users to filter results based on source code lines while simultaneously providing users with data- and control- flow information.

4.1.2 Rational Rose

Rational Rose is a development environment that supports a team of developers by providing a common modelling language, known as UML, for expressing and sharing design concepts [6]. Rose supports developers throughout the entire software development lifecycle including analysis, design, implementation and back to analysis again. Rose manages specifications, designs written in UML and source code developed in a variety of integrated development environments. Rose also supports reverse engineering of software from source code to UML sequence diagrams.

Rose limits a developer to working with existing system structures. This makes it difficult for users to identify and remember which pieces of structure, such as a set of lines within a method, contribute to a concern or feature.

4.1.3 Womble

Womble is a static analysis tool that extracts object models from Java bytecode [12]. It creates models automatically without user intervention. The basic unit in Womble models represents classes. Arcs between classes describe arity and inheritance relationships. Womble differs from other structural extractors in two ways. It analyzes how fields are used to determine the arity of relationships between classes. It also abstracts the use of collections, such as arrays and hashtables, into names of relationships. For example, if a *Company* class has an array of *Person*'s called *employees*, the model would represent this with an arc named *employees* between the *Company* class and the *Person* class. There would be no mention of the array implementation of *employees* in the model.

Womble presents a structural description of existing software while BCM enables developers to describe system behaviour. Womble is not suited to capturing or manipulating crosscutting concerns. Work on Womble describes its use in the formation of object models, but the work does not describe how the models can be used in a modification task or how the model affects the modification process.

4.1.4 Use Case Model Recovery

Lucca, Fasolino and Carlini describe a reverse engineering approach to recover use case models from object-oriented code [17]. In this approach, developers identify statements that form input events and output events. A tool then automatically identifies code corresponding to potential uses cases. The mapping between a given use case to its corresponding code supports developers in program understanding and maintenance impact analysis. One problem with this approach is that developers still need to isolate the relevant use cases from all the use cases returned by the tool. This problem is exacerbated by the fact that

current tools for this approach lack any kind of filtering mechanism.

Use cases provide a description of the externally visible behaviour of a system. This description is typically presented from a system user's perspective. Concerns differ in that they are implementations of concepts in a system's code. Typically the details of concerns are only apparent to software designers and developers.

4.1.5 Conceptual Modules

Baniassad and Murphy introduced the Conceptual Modules [1] approach to help software developers performing software reengineering tasks. This approach enables a developer to overlay a desired structure on an existing structure and to query about the relationships between those two structures. As described earlier, a conceptual module (CM) is a logical module that consists of a set of possibly non-contiguous lines in the source. The BCM approach extends the CM approach in two ways. First, it uses the logical modules to represent pieces of a behavioural model, rather than using them to represent static modules. Second, the BCM tool permits the overlay of CM's on object-oriented Java code rather than the procedural C code supported by the earlier tool. When compared to the earlier tool, BCM has additional support for analysis of exceptions, fields and polymorphic method calls.

4.2 Concern Identification Tools

Aspect Browser, Concern Graph and AMT are some of the existing concern identification tools. The tools' approach complement the BCM approach. Each can be used to help identify the code related to a concern. Each can be used to help systematize the actual process of making a change once the change is decided upon. The BCM approach extends these approaches by helping to systematize how the change should be made: the behavioural model built of the concern code as part of the BCM approach provides the basis on which to reason about different approaches to the change task.

4.2.1 Aspect Browser

The Aspect Browser tool supports the identification of a concern using lexical queries [10]. The results of queries are shown using a map representation similar to the view presented in the Seesoft tool [7]. The map metaphor helps developers navigate through an identified concern, but it has no inherent structure. The metaphor also serves as a guide to developers performing maintenance by highlighting the points in code that potentially need modification.

4.2.2 Aspect Mining Tool

The Aspect Mining tool (AMT) [11] supports concern identification by supporting a combination of lexical and structural queries. A developer may perform a lexical query over expressions that combines type information. Similar to the Aspect Browser, AMT shows the results of queries through a Seesoft-like view. The concern identified has no inherent structure.

4.2.3 Concern Graphs

A Concern Graph provides an abstracted representation of the code related to a concern. The representation consists of structural items, such as classes and particular calls within methods, that comprise the concern. The FEAT tool supports the identification of concern code in a Java system through structural queries and supports the representation of the identified code as a Concern Graph. A Concern Graph representation of the concern code can be used to reason about the concern and can be used as a basis for identifying the dependences—calls and uses—between code in the Concern Graph and the rest of the code base.

Chapter 5

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.

This thesis has discussed a systematic approach to modification tasks supported by the Behavioural Concern Modelling approach and tool. 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.

5.1 Discussion

The BCM approach shows promise, but several questions remain. This section presents some of the choices made in the definition of the approach and the implementation of our tool. It also describes extensions to the approach that would further help in systematizing the change process, and discusses how the approach might help in further modularization of a code base.

5.1.1 Form of the Model

Finite state machines (FSM) are suited to modelling concern behaviour in several ways. 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.

One potential problem with FSM's is state explosion. This occurs when a large number of states is required to model the behaviour of a given system. In such situations, abstracting parts of the model may help reduce the number of states that need to be considered at a given time.

The states in BCM 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 [13], may be more appropriate some, or all, 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.

5.1.2 Models as Long-Term Documentation

In the BCM approach, models serve as documentation for a specific modification task. These models could serve as long-term documentation that span multiple tasks if several issues were addressed. Developers would need to define more precise semantics for their models so that every developer would share the same understanding of the model's behaviour. Tool support would have to enable developers to examine different views for different tasks. As one example, a tool that enables the model to be viewed at different

levels of detail would be helpful. For example, if a modification task required a developer to change how XBrowser parsed JavaScript, one would not use the *Parsing* state from Figure 3.2. One would expand *Parsing* to several states representing whether the parser is in the `BODY` of the HTML page or whether it has encountered a `SCRIPT` tag.

5.1.3 Filtering Relatedness Query Results

The query that returns information about how two CM's relate tends to produce a large number of results. Currently, the BCM tool filters the results based on structural contexts described by classes, methods, and lines. Another possibility is to enable filtering based on lexical information, such as variable and field names, or inheritance relationships. Yet another possibility is to filter on a graph theoretic basis: A developer may only want the results that are well-connected, or the 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 still an open question.

5.1.4 Analysis Using Behavioural Models and BCM

This thesis describes how the BCM approach applies to five of the six steps outlined as part of a systematic change process in Section 1.1. The sixth step involves analyzing the implementation to determine whether the change has been made correctly. The BCM approach can also be used for this step. 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 1.1.1. 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 similar to model checking [14]. Unlike model checking tools, the developer is more limited in the queries that can be run. An advantage compared to existing

source code model checking tools, such as Bandera [4], is that a "higher-level" model can be used. That is the "states" in the model represent large pieces of processing, rather than a particular localized piece of state.

5.1.5 "Aspectualizing" the Concern

In some cases it may be advantageous to capture a concern's code explicitly as, for instance, an aspect in AspectJ [15]. Understanding how the concern's code *works* is an important step before trying to separate the code. 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 interface, or joinpoints, between the existing code and the aspect code. Concern finding tools, as discussed in Section 4.2, may also be more effective than the BCM tool at elucidating pertinent code.

There are benefits to using the BCM approach in forming aspects. BCM models may indicate that a certain piece of behaviour must occur before or after another piece of behaviour. This relationship between behaviours is explicitly support by some of the modularization mechanisms in AspectJ. Another benefit is that developers can use queries to help determine how difficult it might be to refactor a piece of behaviour as an aspect without actually performing the remodularization. For example, a large number of data- or control- flows between one CM and another may indicate that it will be difficult to create an aspect from either of the CM's.

Bibliography

[1] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *International Conference on Software Engineering*, pages 64–73. IEEE Computer Society Press, 1998.

[2] Elisa L.A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. Technical Report UBC-CS-TR-2001-16, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, October 2001.

[3] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448. IEEE Computer Society Press, 2000.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] R. Corporation. Rational rose. http://www.rational.com/products/rose/index.jsp [16 November 2001].

[7] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[10] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of International Conference on Software Engineering*, pages 265–274. IEEE Computer Society Press, 2001.

[11] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In *Workshop on Advanced Separation of Concerns at International Conference on Software Engineering*. IEEE Computer Society Press, May 2001.

[12] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, pages 194–202. IEEE Computer Society Press, 1999.

[13] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[14] 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.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[17] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Ugo De Carlini. Recovering use case models from object-oriented code: a thread-based approach. In *Working Conference on Reverse Engineering*, pages 108–117. IEEE Computer Society Press, 2000.

[18] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. Technical Report UBC-CS-TR-2001-13, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, September 2001.

[19] Tarja Systa. Understanding the behavior of Java programs. In *Working Conference on Reverse Engineering*, pages 214–223. IEEE Computer Society Press, 2000.

[20] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135. IBM Canada Ltd., 1999.

[21] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

Appendix A

Relatedness Query Example

```
Query Inputs:
    Query Source: (setUser transition)
        FTPConnection.doUserCommand(String), line 534
        FTPConnection.doUserCommand(String), line 537
        FTPConnection.doUserCommand(String), line 538
        FTPConnection.doUserCommand(String), line 539
        FTPConnection.doUserCommand(String), line 540
        FTPConnection.doUserCommand(String), line 542
    Query Target: (handleAnonPass transition)
        FTPConnection:doPassCommand(String), line 560
    Query Context:
        All Classes except WildcardFilter

Query Outputs: (gotUser state)
    Class Summary:
        FTPConnection -> PassiveConnection
        FTPConnection -> FTPHandler
        FTPHandler -> FTPConnection
    Method Summary:
        doPassCommand -> printWelcome
        doUserCommand -> doPassCommand
        doCommand -> doUserCommand
        doCommand -> doPassCommand
        run -> doCommand
        doCommand -> run
        doCommand -> setBusy
        doCommand -> setLastCommandTime
```

Figure A.1: Query Example