OREGON STATE UNIVERSITY

Final Test Report

CS362 - Spring 2012

Rob Gibson

6/11/2012

Dominion is a card game with a foundation of simple rules, making it a promising candidate for a computer software implementation. From these simple rules flows a complex strategy game that presents hurdles to the development of a software version. In this test report, we will examine several implementations of the same components of a software *Dominion* game using testing techniques to discover faults and reveal their underlying bugs and other defects. Techniques used include random testing and static testing, along with coverage techniques and differential debugging. We will discuss the results of these tests and conclude with recommendations for how to proceed with both implementing and continuing to develop a more comprehensive testing suite for *Dominion*.

CONTENTS

INTRODUCTION

The goal of this testing effort was to determine if, given its interface, any particular implementation of the card game *Dominion* is correct, and if not, what the sources of the faults were. The implementations were tested with both valid and invalid input, and the output was checked in order to determine the correctness of the implementation.

Having the source code for the implementation available enabled additional testing capabilities, including checking for code coverage and attempting to gain as much branch coverage as well as the ability to run bounded model checking software against its implementation.

Another boon to this testing effort was the variety of implementations (of assigned functions) available for testing. This enabled differential debugging and testing whether the test suite will catch bugs by running it against several versions of the code.

EQUIPMENT USED

The testing equipment was two Windows 7 PCs, one a desktop and one a laptop with the following configurations:

HARDWARE

DESKTOP

Lenovo ThinkPad T420
AMD 4-core processor @ 3.1 GHz/core
8GB DDR3 RAM

LAPTOP

Intel 4-core processor @ 2.8 GHz/core
4GB DDR3 RAM

SOFTWARE

DESKTOP

Operating system: Windows 7 updated as of 5/9/2012
Primary development environment: Microsoft Visual Studio 2010 w/ Visual C++ compiler
Alternate compiler: GCC 4.7.0 running on Cygwin version 20120428

LAPTOP

Operating System: Windows 7 updated as of 6/7/2012
Primary development environment: Microsoft Visual Studio 2010 w/ Visual C++ compiler
Alternate compiler: GCC 4.7.0 running on Cygwin version 20120428

SUMMARY OF PROCEDURES

CHANGES MADE TO COMPILE IN VC++

All of the versions of dominion.c that were tested had the same issue when being compiled in VC++: the initialization of variable ‘x’ in the omnibus “cardEffect” switch statement. Once that had been pulled out, the only other change required to make the program compile was to edit the header files to include at the top:

```
#ifdef __cplusplus
extern "C" {
#endif
```

And at the bottom:

```
#endif
#ifdef __cplusplus
}
#endif
```

EARLY TEST CASES: BUYCARD

The earliest tests ran were designed to exercise several edge cases that were suspected of containing potential problems, and test for basic facts about the game state along the way:

TEST CASE 1: BUYING A CARD RANDOMLY, TWICE

Initialize a game and call buyCard on a random card, then compare the resulting state to a memcpy'd version, and run a member-wise comparison of the two states. The prior state is then used to store some of the information from the states, and buyCard called again, without having ended the turn. Having started a game with only one buy phase, this tests whether buyCard is only allowing the correct number of purchases. If there is something wrong, an assertion will catch it after the report about where the differences between expected and actual values are, so they could be investigated easily with a debugger.

TEST CASE 2: BUYING A CHEAP CARD TWICE

The game is then reinitialized and buyCard is called again, with the selected card as a copper. This time it checks to see if the game is actually buying the copper, and then calls it again to make sure it isn't able to buy it twice. Comparisons are reported to make it easier to narrow down where a problem might be, and assertions are made about where the copper should be after each buyCard call to ensure correctness.

TEST CASE 3: BUYING AN UNAFFORDABLE CARD

The game is reinitialized again, the players hand is set to all curses and a copy is made. Then it attempts to buy an adventurer, and asserts that the two are equal. While it is undefined behavior whether attempting to buy a card puts you in the buy phase, this tests that you can't buy a card that you shouldn't be able to.

RANDOM TESTING

RANDOM INITIALIZATION

The game state – effectively the program state, was the most important structure and component of the software being tested, and thus its proper initialization is required for the program to correctly execute. For the random testing component of the tests run, the game state was initialized to random values. In order to accomplish this, the game state object was treated as an array of characters and iterated over, setting each ‘character’ to a random value between -128 and 127. As part of the random test, most of the time the “initializeGame” method would be called on this randomly created game state, other times the test would proceed without initializing the game in order to test as many possible inputs as possible, not just valid ones.

RANDOM INPUT

Using the pseudo-random number generator, random inputs were generated for each execution of the test run, unless a “strategy” flag was set that would reduce the level of randomness in the test. The strategy flag is a way to toggle the system between randomly generating values from the full range of possible inputs (strategy off), or to generate random values only within the valid range of the inputs (strategy setting 1), or run a set strategy that would only use valid inputs that would intentionally progress a game towards its conclusion (strategy setting 2).

The strategy used was a variation of the “money” *Dominion* strategy: always buying money unless you can purchase a province, which case purchase a province. The variation to this strategy is to alternate buying either an adventurer or a gold if the player has >=6 and <8 currency to spend.

The reason for wanting different levels of random testing is to ensure that all components are fully exercised – that as many branches of execution are covered as possible. To determine the coverage of the test-suite runs, the gcov utility was used to count the number of executions of each statement in the source code.

RANDOM SERIES OF PROCEDURES

Again using the pseudo-random number generator, a series of gameplay actions are randomly chosen by generating a number between 0-2 inclusive and depending on the number, making the next call to the *Dominion* implementation either “buyCard”, “playCard” or “endTurn”. At the end of every turn, the test suite uses the method “isGameOver” to assess whether a game has been completed.

Even if “strategy” mode is turned 2, the order of actions played continues to use these random procedures, the only difference being the decision about which card is bought or which card is played is non-random.

CONSTRAINT-BASED TESTING

CBMC – CONSTRAINT BASED MODEL CHECKER

In order to accomplish this testing in a reasonable amount of time, some components were scaled down and unused code was commented out. The main areas changed in the source code to enable CBMC to work were scaling down the loop constraints to make it possible to solve the unwound loop structure in a reasonable amount of time. By default, MAX_HAND and MAX_DECK were set to 500, creating loops that would iterate far longer than ever likely, so by reducing these constants to 10, CBMC was able to solve for counter-examples in less than 5 minutes.

The main problem encountered when using CBMC was basically a Goldilocks problem: how many assumptions are absolutely necessary and what assumptions are unnecessary. Another potential problem with using CBMC by trimming down the loop counts and removing extraneous code that would lead to extremely long solve times is the removal of potentially buggy code or edge cases that would trigger bugs.

TEST RESULTS

For each test report, the battery of tests was run against the group's code. Members of the testing group involved were:
turcottm, gibsonro (author), luisramg, nolandr, junkerd, gormanv, genzg, bartoszk

TESTING CYCLE 1: SPECIFIC TEST CASES

TURCOTTM – FAILED:

Expected: state->supplyCount for target card one less than previous value if buyCard succeeded
Actual: state->supplyCount was not the expected value after buyCard for the target card

LUISRAMG – FAILED:

Expected: supplyCount for target card should be decremented by one; discard should have one more card in it
Actual: supplyCount, discard, discardCount, not the expected value after buyCard.

NOLANDR – FAILED:

Expected: state->coins remains unchanged when buying a copper, a 0 cost card.
Actual: state->coins in prestate was unequal with poststate after attempting to buy a copper.

JUNKERD – FAILED:

Expected: Freshly purchased copper will be in discard after buyCard is called.
Actual: Copper not properly discarded after being bought, along with other things not being right after buyCard-ing a copper

GORMANV – FAILED:

Expected: Calling buyCard on a target card twice with 1 buy action should not change discard or discardCount a second time.

Actual: After attempting to buy a card twice with only 1 buy action, the discard and discardCount have changed after the second buy.

GENZG – FAILED:

Expected: Freshly purchased copper will be in discard after buyCard is called.
Actual: buying a copper using buyCard did not result in the copper ending the buy phase in the discard

GIBSONRO – PASSED
BARTOSZK – PASSED

TESTING CYCLE 2: RANDOM TESTING

TURCOTTM - FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

LUISRAMG - FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

NOLANDR - FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

JUNKERD - FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

GORMANV - FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

GENZG – FAILED:

Expected: Playing the “feast” card finishes after drawing 3 cards.
Actual: Infinite loop when calling playCard on a feast.

Expected: Score will remain >= 0 as long as the player possesses no curses.
Actual: Score becomes negative without buying curses

Syntax error: buyCard declares "int who" twice: once at the beginning of the method, once at the end.

GIBSONRO – PASSED
BARTOSZK – PASSED

SUMMARY OF TEST RESULTS

With the second round of random testing, a coverage rate of 75% was achieved on all but turcottm's implementation. This means that a quarter of the code was either not reached, indicating a poor test, or the code was unreachable. It is likely that the test is incomplete, but in the single case of turcottm's test, it is more likely that the code itself is unreachable.

There was little improvement over time of the group's dominion code. Reports were sent to 3 students: turcottm, luisramg and genzg, though no replies were received. It also appears that their code does not reflect any fixes to the bugs reported.

Bartoszk's code was consistently good; they probably have had practice with dominion and have a test suite similar to my own to have gone without bugs that were easily detected in other peoples' code. Even some more obscure bugs, like the infinite loop in the cardEffect switch for the "feast" case, were stomped out by the time the test suite was run on Bartoszk's code.

Testing the implementations of *Dominion* revealed that it is riddled with errors, some more obvious than others. The infamous =! 2 if statement that breaks out of a loop comes to mind as a less obvious error that a student introduced into their code.

One difficult aspect of using assertions for testing purposes is that as soon as an assertion is violated, the program halts execution, preventing the tester from uncovering more bugs without either commenting out the assertion or fixing the underlying code. Because of this, the software "GinkoTest" written by Tyler McClung that has the feature of "nonbreaking asserts" was highly appealing, though in the end was not used.

Another difficulty that was encountered while testing *Dominion,* and probably would be apparent in any testing effort, was and would be a lack of a complete understanding of the requirements. Although there is little ambiguity with most of the rules, keeping all of the specifics of how a specific card or action should behave in mind is a difficult task. About half of the time developing the test suite was spent modifying it to match the actual requirements rather than imagined ones. For example, the initial test set for buyCard operated under the mistaken assumption that bought cards would go into the played cards before entering the discard, and the tests had to be modified to fix this faulty assumption.

ANOMALIES/BUGS

SYNTAX ERRORS

The most common failure was a failure to compile in G++ or VC++ due to the initialization the variable "x" in the "cardEffect" switch statement.

Another observed syntax error was the declaration of a variable twice in the same scope.

These errors, having the effect of preventing the compilation of the software, were also the most easily located and resolved.

LOGIC ERRORS

The most common errors of this type were incorrect usage of discard/played cards. Most of the time this bug would go unnoticed and not cause a fault in the game state, because it would only become a problem in rare cases, such as when a card is obtained through an action, then another action draws more cards than remain in the deck, reshuffling the improperly discarded card into the draw deck, and creating the possibility of an infinite loop.

There were many logic errors in the code that was not implemented during this term. While they were as common as errors in the code that was implemented this term, they were also universal and thus not easily discoverable using differential debugging. One such bug was in the "feast" card effect and would trigger an infinite loop.

Another logical bug that was discovered in the unaltered version of *Dominion* was allowing players to purchase more score cards than existed, eventually causing an overflow error and reducing player scores far below 0.

CONCLUSION AND RECOMMENDATIONS

It appears that the largest obstacle of testing and software development in general is obtaining a complete understanding of requirements. It does not matter if the correct implementation is simple or difficult when the requirements are misunderstood or ambiguous. In testing software, a complete understanding of the requirements is the most valuable knowledge, because without that understanding, any test suite conceived would be of little use.

"It's twice as hard to debug software as it is to write it, so anything that you are only just clever enough to understand is, by definition, too difficult for you to debug." This statement seemed to ring true whenever the problem was with the test and not with the code. With enough copy and pasting, similar segments of code blend together and become difficult to distinguish as problems, as with the double decrement operation that is featured below.
In future development, the practice of writing tests before implementing the software would appear to hold great promise, since it requires a foundation of understanding of the input and desired output before the developer even begins to write the implementation.

Using various methods of software testing revealed some of the benefits and flaws of each technique:

- Hand-made test cases are especially good at finding specific problems that can be imagined in advance, but poor at finding problems that are more subtle.
- Random testing is very good at testing the breadth of the software, but there is still the chance that edge cases or rare situations will not be encountered without being given an enormous amount of random testing time, or a very well designed random test.
- Model checking seems to be the best possible solution to the problem of software testing: based on supplied assumptions and constraints, it is possible to actually prove that code is correct. Problems still exist with this method though, in that sometimes one can assume too much and prevent the model checker from considering faulty code. The most major problem would be with any sufficiently large software, or any software that relies heavily on large looping structures. With software of this type, running CBMC may take a very long time. With a more thorough understanding of model checking and how to quantify how long an execution of the model checker will take, model checking would be an even more appealing tool in the testing arsenal. As it stands, however, model checking appears like it would be best utilized for unit testing smaller components than integration testing of many components.

IMPLEMENTATION RECOMMENDATIONS:

RECOMMENDATION 1: CODE REVIEW

In the process of testing these implementations of *Dominion*, the source code was visually inspected at multiple times. These visual inspections revealed an inconsistency of style and occasionally revealed bugs that would have been difficult to locate without having seen them directly in the code. In order to reduce the chance of misinterpretation of previously written code and ensure the consistency of future code, my first recommendation is that the software be subjected to a process of code review and that a consistent convention of variable names, line alignments, and other coding standards be applied. These standards would also have to apply to the header files containing the interface and basic data structures for this application, which themselves contain some inconsistencies. This code review process leads us to the second recommendation.

RECOMMENDATION 2: BREAK COMPONENTS DOWN

To facilitate the use of tools such as CBMC, a more modularized version of this application would be very useful. For example, the "cardEffect" method consists of a switch statement that contains the implementations of every single action card in the game all in a single method. In order to use CBMC to test this method, it became necessary to comment out almost all of the switch statement. In order to visually inspect the code for any particular card, one would have wade through dozens if not hundreds of lines to find the particular switch for that card. If the execution of each card had its own method, the switch statement could be greatly consolidated, increasing readability and enabling the tester to inspect the program flow with a greater degree of certainty as to what the code is doing in any particular section when following a stack trace in a debugger, or during calls from a test suite.

RECOMMENDATION 3: PROTECT THE GAME STATE

When the game state is freely accessible, it is tempting to constantly interact with it directly all the time. Introducing bugs is easier when you can freely change the game state. Although base C does not contain the idea of objects, it is still possible to use some encapsulation techniques to prevent unintentional and unwanted modification of the game state. The implementation provides some methods to help protect the game state, such as the whoseTurn, supplyCount, handCard, and numHandCards. These methods can be considered "getters" from the object oriented paradigm, whose goal is to keep you from directly interacting with the underlying state, and thus preventing you from introducing bugs when accessing the contents of the game state. Inside other methods that are useful at protecting the state, such as the discardCard method, however, these accessors are not used very often and thus present opportunities to corrupt the game state where it is not necessary.

RECOMMENDATION 4: SEPARATE ARRAY ACCESS FROM INDEX MODIFICATION

Another easy way to introduce complication and make source code harder to interpret is the modification of indexers into arrays as part of an assignment or access to an array item. By moving the indexer increment/decrement actions to outside of the access/assignment, visual inspection becomes a less convoluted task and fault localization can be made slightly easier. An example of a fault that might've been avoided in the original *Dominion* source provided:

```
state->deck[nextPlayer][state->deckCount[nextPlayer]--] = -1;
state->deckCount[nextPlayer]--;
```

At first glance, this may appear to set the last item in a player's deck to -1 and then reduce the counter for the deck. However, it is setting the card directly AFTER the last card in the players hand to -1, then decrementing their deck count by 2. This fault could have been more easily avoided if the style were always to put the increment/decrement of a variable outside of the array access, rather than a haphazard, mixed approach.