Project Report
Seungtack Baek
CS6305.501
Apr/26/2012

Residue Number System Implementation
with Java Applet

1. Objective

The Objective of this project is to design and write a Java applet that mimics Residue Number System (RNS) operation. This applet will take a set of numbers as moduli, two operands (in decimal[1]) and the operation. As the output of this applet, the result of the operation (in decimal[1]) and the detailed description of each operation will be shown in the log, as shown in the figure 1 below. As an additional feature, users can turn off the log message pane, if they desire to do so. The more details of this applet will be discussed in the future sections.

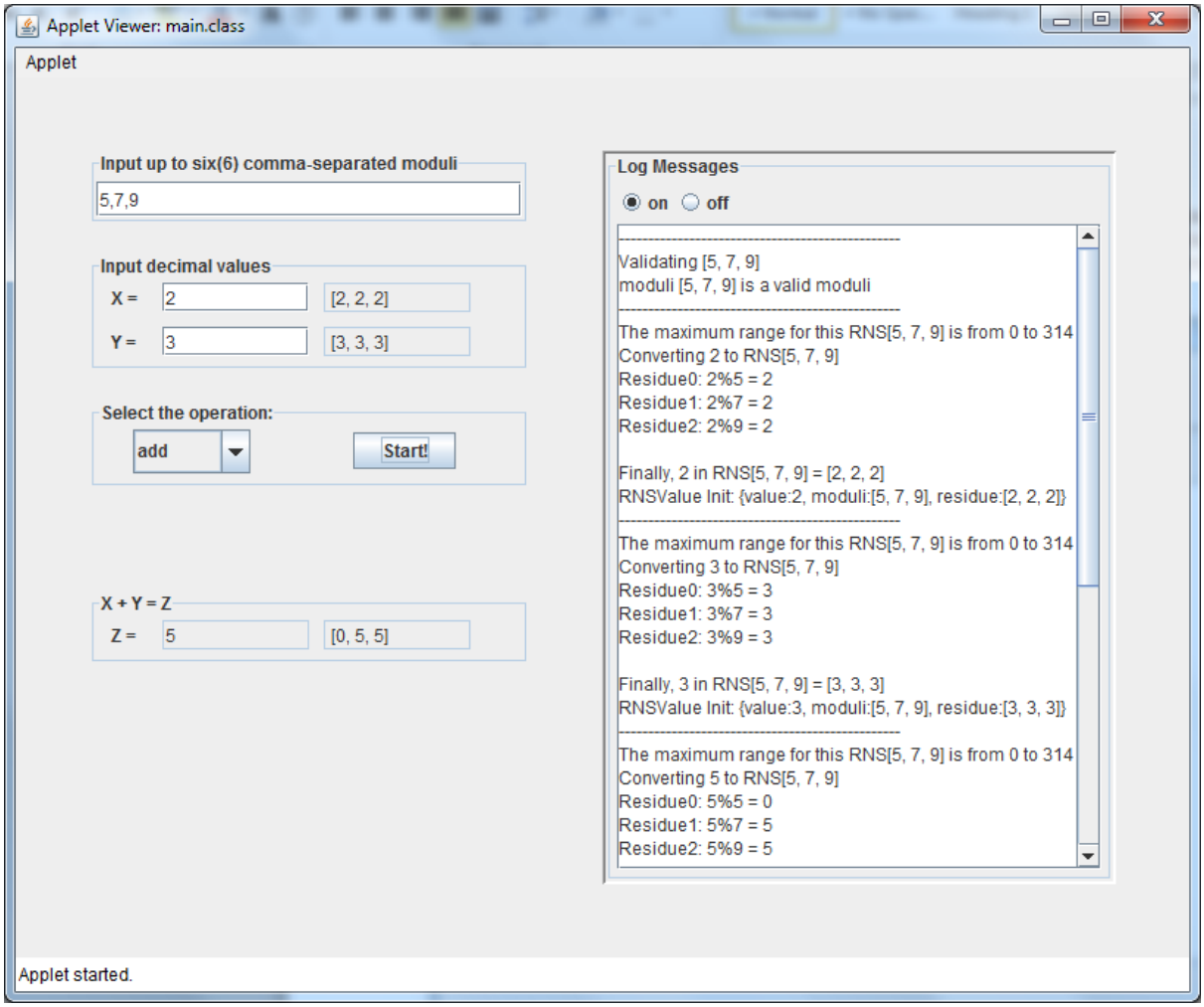

Figure 1 The screenshot of running applet example

1. *Although the project requirements states to receive "binary" values, decimal representation is used for user friendliness and ease of implementation with the instructor's consent.*

2. Development

In order to implement this system, Java is used as a development language, along with javax.swing components to create the GUI for this applet. The following UML diagram shows the simplified structure of this project.

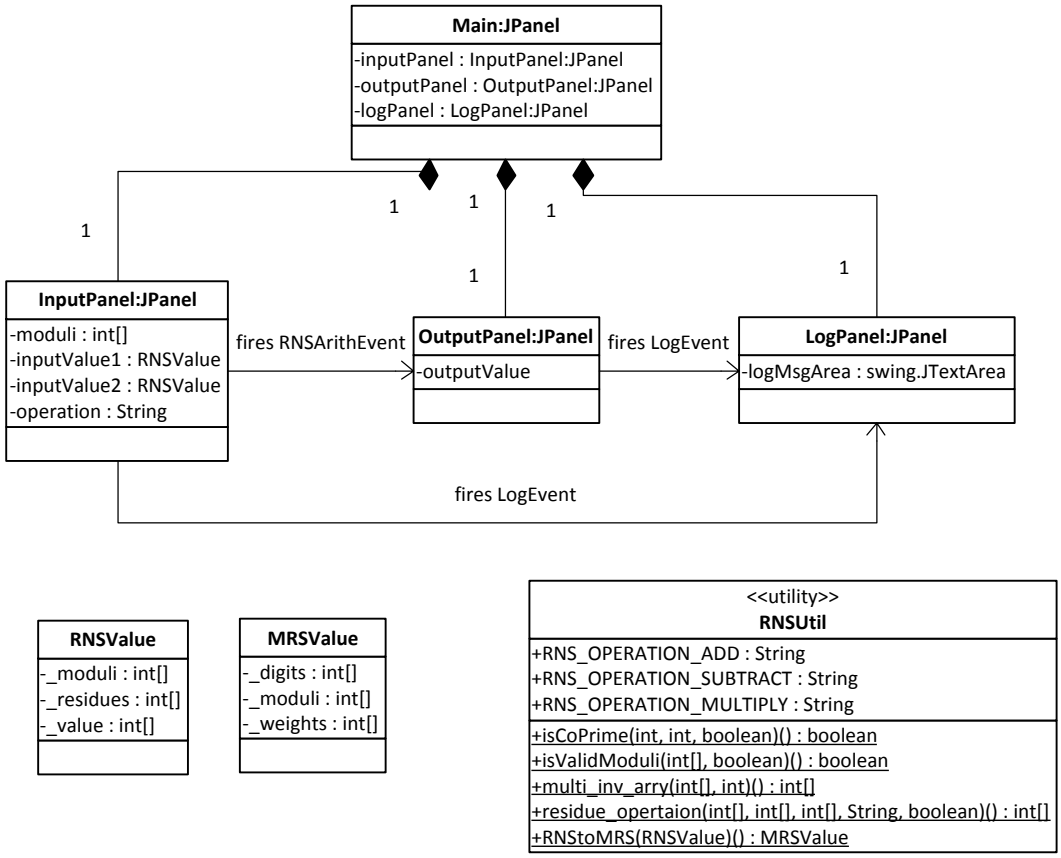

Figure 2 Simplified UML Diagram of this project

As seen above, it is contains 3 main views: InputPanel, OutputPanel and LogPanel. InputPanel receives all 4 required inputs (two operands, moduli and operation) and sends it to OutputPanel, while logging necessary information to LogPanel. The example of necessary information is conversion steps from "binary" -decimal in our case – to RNS value. Upon receiving inputs from InputPanel, OutputPanel carries out the specified operation and convert the result back to MRS (then to decimal, so user will see).

3. Applet In Action (How to Use It?)

Using this applet consists of six steps after launching the applet itself. Please refer to figure 3 below to see the corresponding components. The six steps to use this applet is as the following:

1. Input up to six, mutually co-prime moduli, separated by comma. If not separated by comma, it will not be able to parse the input and error message will be prompted. Also, if a user puts more than six numbers, all number after sixth moduli will be disregarded. Lastly, they have to put set of numbers that are mutually co-prime of each other. If not, corresponding error message will be shown as well.
2. Then input 2 integers to be carry operation upon in decimal format. If numbers of wrong format are entered, such as 3d1a or 3.51, it will show corresponding error message. Also, be noted that I assumed

only positive numbers in this system. So if moduli are {2, 3}, then the range is {0, 5}, not {-3, 2}.

3. Now, select the desired operation to be carried out from dropdown menu.
4. Finally, click "Start!" button and see the result.
5. The operation result will be shown in the OutputPanel and the label "X+Y=Z" will change accordingly. For an example, if a user wants to carry out multiplication, the label will now be "X*Y=Z." Also, on the right field of the decimal result, the RNS result is also shown.
6. Throughout the whole process, the conversion steps, both from decimal to RNS and RNS back to decimal (MRS actually, but converting MRS to decimal is easy), is logged. If a user desired to turn off the log message, one can do so by clicking "off" radio button, and vice versa.

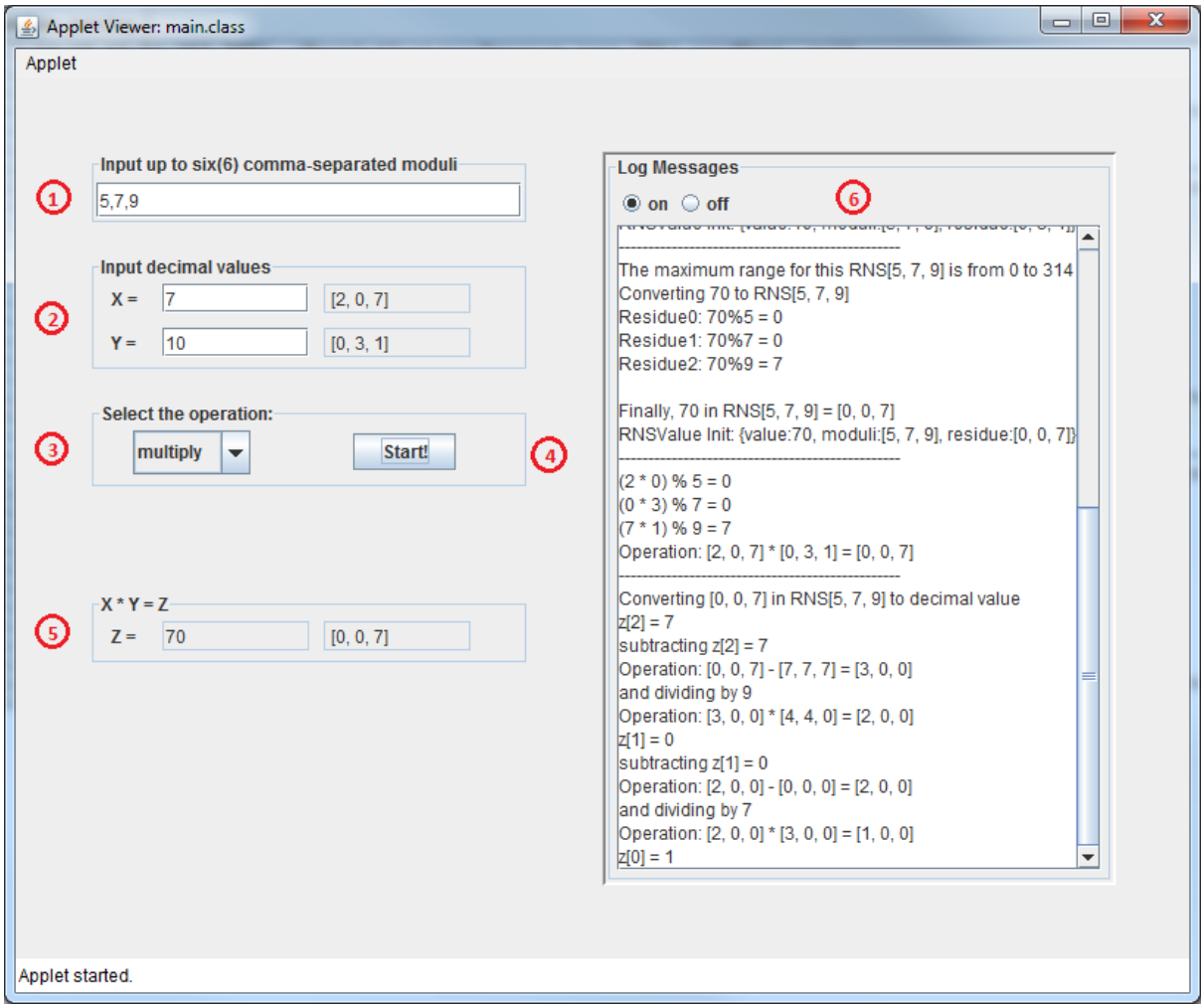

Figure 3 Steps to Run the Applet

4. Testing the Applet

To verify the correctness and completeness of this applet, testing the correctness of the RNS operation was crucial. To test the RNS operation, which is carried out in RNSUtil class, I have applied random testing to RNSUtil. In my test suite, I created a function that generates random set of moduli. The random set of moduli has a set of one to six mutually co-prime numbers. However, since the range of the RNS itself could lead to integer overflow, the range of the each modulus was restricted between 1 and 100, which was statistically small enough to not worry about the overflow. (yet, $\sqrt[6]{2^{32}} \approx 2^5$ should be the correct value to ensure to keep

overflow from happening). Then I also generated two operands between the range [1, $range \times 1.5$]. I chose 50% more than the range just to test if RNSUtil can catch the overflow. Lastly, since there are only 3 operations, I tried all three operations for each generated inputs.

To cut to the point, RNSUtil passed the random number testing on 10,000 generated inputs, which consists of 30,000 test cases. For all cases, the RNSUtil showed expected results.

5. The "Take-Away"

First of all, from this project, I gained more comprehensive knowledge on how RNS works. The basic of RNS operations is as following:

$$x \circ y \bmod m = (x \bmod .m \circ y \bmod m) \bmod m$$
$$\text{,where } \circ \in \{+,.-,\times\}$$

So, any basic operation, with an exception of division, can be carried out between the residues, instead of the numbers itself. The advantage we get from this technique is the speedup. For carefully chosen moduli, we can expect logarithmic time operation. Yet, the cost of this speedup is much less efficiency. We represent less numbers with same number of bits, since some numbers are unrealistic (such as 4 in mod 3 system).

Second of all, it became clearer to me how to convert the RNS value back to the decimal value via MRS. I only had some basic idea on how to convert RNS to MRS. Yet, having implemented the actual procedure, I now understand the workflow even better.

The last thing that I've learned from this project was on the implementation itself: the Java Applet. I have not built too much applet until today. In building this applet, I have learned a lot about how applet works and the concept behind event-driven programming. Building this applet, I can perceive how other applets work and apply this technique to solve other problems.

What "we" can learn from this project is the detailed description of conversion steps, both from decimal to RNS and back to decimal. I implemented this with a user who is not too much familiar with RNS system in mind. So, one can be more familiar with the RNS operations by looking at the detailed descriptions on conversion steps.

6. Exploration & Future Improvement

While developing this applet, there were a lot of challenging problems to solve. Yet, because of limited time and resource, I had to simplify this applet. In this future, this applet can be improved even further. Here is the list of possible improvements in the future.

1. The applet will receive more than six moduli and it will be more resilient against faulty inputs, such as 3asb or 3.015. Even possibly try to estimate the users' intent so that if they put 3.0, take it as 3.
2. As stated in the footnote on page1, for simplicity of implementation, I used decimal numbers. Yet, it will not too much work to change that to receive binary string as well. This is desired from the specification in the project description sheet.
3. Also moduli auto-correction is desired. So, if a user inputs non-mutually co-prime moduli, it will try to solve the conflict. For an example, if 3, 5, 9 given, the applet will recognize that 3 and 9 are not co-prime. So, it can either modify 9 so it will be co-prime with 3 (can be done easily by incrementing or decrementing 1) or simply drop 3 or 9.

4. Another desired feature will be dynamic range. Currently this applet assumes positive number and fixed range. Yet, if I implement dynamic range, it will cover more ranges than now and the selection of input will be even broader.
5. One interesting thing to implement is to also mimic what other adders/multipliers do and compare the performance of those two (along with comparison of the efficiency as well). Although the result of this might not reflect that between actual hardware implementations, it might give some idea on possible speedup by using RNS adders over plain binary adders (such as CLA, RCA, etc.). Combination of this with improvement 2 above might make it even more interesting.