The document describes the solution to an exercise we tried on 27th May'15. It is very similar to the Key assignment at https://courses.cs.ttu.ee/w/images/1/12/ITI0130_Lab12_Key_assignment.zip. As discussed in the session, this is one possible solution among others.

I would recommend this especially to those who were not present in the session on 27th May and also to those who defended on 20th May, because of some doubts that we encountered with the "assignable" clause. Based on some tests that I tried by declaring the assignable clause at the beginning and end of the specs, I found some impact on the behaviour of the Key tool. I have described one scenario towards the end of the document. In this solution, we declared the assignable clause at the end of the spec as it is done in some standard examples provided with the tool.

Regardless of any differences in the solution, those who have already defended may not submit any further updates of the assignment.

The exercise is a simple class modelling two basic operation of a Bank – "addAccount" and "suspendAccount". The method "find" is identical to the one in the KeY assignment.

We start with the "find" function. In the assignment file we need to comment the JML specification of the other functions as shown to allow the Key tool to load the file.

```
/*@ //normal_behaviour
@
@
@
@
@
@   //assignable account[*];
@*/
```

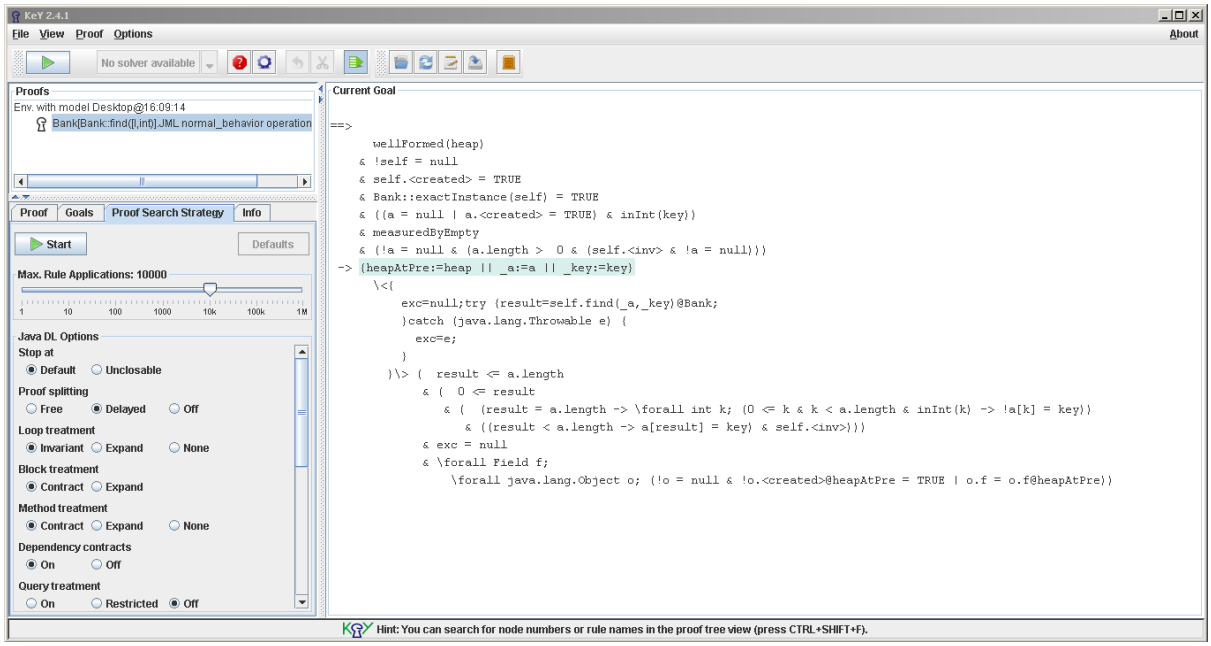
The first set of JML specifications for the "find" function is below. It is based on the pre-conditions and post-conditions provided in the comment.

```
/*
 * The function finds a number (key) in an int array.
 * It returns the length of the array if key is not found
 * else returns the index of key.
 *
 * Pre-Condition -
 *     Array must not be null.
 *     Array must not be empty.
 *
 * Post-Condition -
 *     Return value must not be greater than array size
 *     Return array size if the key is not found
 *     Return array index of the key if it is found.
 */

/*@ normal_behaviour
@      requires a != null;
@      requires a.length > 0;
@      ensures \result <= a.length;
@      ensures \result == a.length ==>
@                  (\forall int k; 0 <= k && k < a.length; a[k] != \old(key));
@      ensures \result < a.length ==> a[\result] == \old(key);
@      assignable \nothing;
@*/
```

```
protected int find(int[] a, int key) {

	int i = 0;

	/*@ loop_invariant
	@	(0 <= i && i <= a.length) &&
	@	(\forall int k; 0 <= k && k < i; a[k] != key);
	@	assignable i;
	@	decreases a.length - i;
	@*/
	while(i < a.length && a[i] != key)
		i++;

	return i;
}
```

The pre-conditions are self-explanatory but post-conditions can be formulated in other ways as well.

Load the file with the “find” specs in Key tool and please ensure the “Proof Search Strategy” is set to default. The default button should be greyed out as shown in the screenshot:



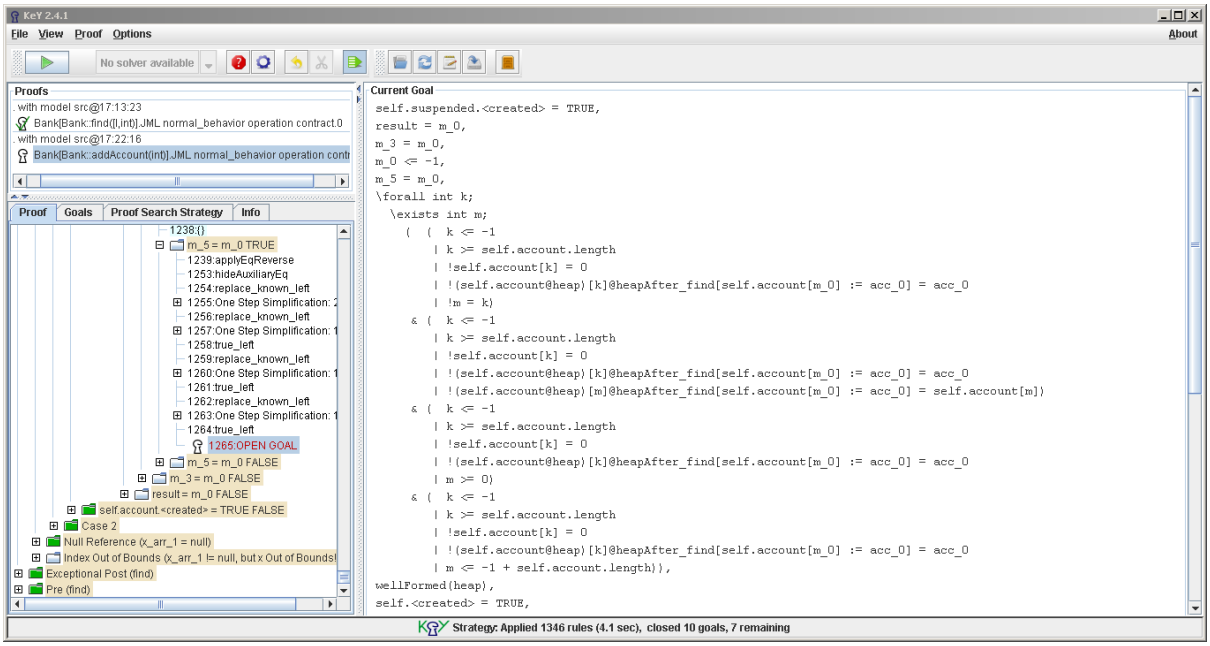
Please run (“start”) the proof-search. The tool will prove the goal and display statistics window.

The specification used for “addAccount” function is as shown below. We have a nested quantifier for post-condition. We can formulate the logic in other ways as well. The one shown here ensures that only one available cell of the array is changed and the others remain same as the old value.

```
/*
 * The function accepts the ID of the account as argument and stores it in
 * the account array.
 *
 * Pre-Condition -
 *	Account ID must not be zero.
 *	Account space must not be full.
```

```
 *     Account must not already exist.
 *
 * Post-Condition -
 *     Account data is stored in an empty cell.
 *     Data of other accounts must not changed
 */

/*@ normal_behaviour
@    requires (acc != 0);
@    requires (\exists int k; 0 <= k && k < account.length;
@                    account[k] == 0);
@    requires (\forall int k; 0 <= k && k < account.length;
@                    account[k] != acc);
@    ensures (\exists int k; 0 <= k && k < account.length;
@                    \old(account[k]) == 0 && account[k] == \old(acc) &&
@                    (\forall int m; 0 <= m && m < account.length && m != k;
@                    \old(account[m]) == account[m]));
@    assignable account[*];
@*/
public void addAccount(int acc) {
       account[find(account,0)] = acc;
}
```

When we try to prove the “addAccount” function with the specs above, the tool shows open goals. One of them is “Index Out of Bounds” as shown below. We identify this case in both “addAccount” and “suspendAccount”.



In the source code of the method, only one array “account” is being accessed and the index is provided by “find” method. In order to ensure the lower bound of the return value in the contracts, we add another post-condition to the “find” method as shown:

```
/*@ normal_behaviour
@      requires a != null;
@      requires a.length > 0;
@      ensures \result >= 0;
@      ensures \result <= a.length;
```

```
@      ensures \result == a.length ==>
@                  (\forall int k; 0 <= k && k < a.length; a[k] != \old(key));
@      ensures \result < a.length ==> a[\result] == \old(key);
@      assignable \nothing;
@*/
```

Please reload both “find” and “addAccount” method and start the proof-search for each respectively. The tool completes the proof-search and displays statistics.

The specs for “suspendAccount” are as shown below. The quantifier logic is similar to that of “addAccount”.

```
/*
 * The function accepts account ID as arguments
 * and marks the account as suspended.
 *
 * Pre-Condition -
 *     account ID must not be zero.
 *     account must exist.
 *     account must not be suspended already.
 *
 *
 * Post-Condition -
 *     The suspended status of account is marked as true
 *     The suspended status of other accounts must not change.
 *
 */

/*@ normal_behaviour
@      requires (acc != 0);
@      requires (\exists int k; 0 <= k && k < account.length;
@                  account[k] == acc && !suspended[k]);
@      ensures (\exists int k; 0 <= k && k < account.length;
@                  account[k] == \old(acc) && suspended[k] &&
@                  (\forall int m; 0 <= m && m < suspended.length && m != k;
@                  \old(suspended[m]) == suspended[m]));
@      assignable suspended[*];
@*/

public void suspendAccount(int acc) {
       suspended[find(account,acc)] = true;
}
```

The proof-search of “`suspendAccount`” fails with the above specifications. We identify the “Index Out of Bounds” case as in “addAccount”. In the source code of the method “`suspendAccount`”, two arrays are used. The index of array “suspended”, accessed in the method, depends on the size of the array “account”.

We add the pre-condition that sizes of these two arrays are equal, thus ensuring, given the other conditions, that the array index is valid. The new specs are as follows.

```
/*@ normal_behaviour
@      requires (acc != 0);
@      requires (account.length == suspended.length);
@      requires (\exists int k; 0 <= k && k < account.length;
@                  account[k] == acc && !suspended[k]);
@      ensures (\exists int k; 0 <= k && k < account.length;
```

```
@                   account[k] == \old(acc) && suspended[k] &&
@                   (\forall int m; 0 <= m && m < suspended.length && m != k;
@                   \old(suspended[m]) == suspended[m]));
@      assignable suspended[*];
@*/
```

Please reload the "suspendAccount" method and start the proof-search. The tool completes the proof-search with statistics.

As mentioned at the beginning of the document, I have tried a scenario which shows an impact of the position of assignable clause on the behaviour of Key tool. In the tests the Key tool splits the contracts when the assignable clause is declared at the beginning.

For example in the "addAccount" function, if we declare the assignable clause at the beginning as shown below-

```
/*@ normal_behaviour
@   assignable account[*];
@   requires (acc != 0);
@   requires (\exists int k; 0 <= k && k < account.length;
@                   account[k] == 0);
@   requires (\forall int k; 0 <= k && k < account.length;
@                   account[k] != acc);
@   ensures (\exists int k; 0 <= k && k < account.length;
@                   \old(account[k]) == 0 && account[k] == \old(acc) &&
@                   (\forall int m; 0 <= m && m < account.length && m != k;
@                   \old(account[m]) == account[m]));
@*/
```

The Key tool splits the contract as shown in the screenshot. As already mentioned, in order to avoid any issues that the position may cause, in this solution we declared the clause at the end of the spec, as it is done in some standard examples.