CS3050 Practical 2

200007413

1 Functionality Checklists

1.1 Exercise 1

1. ✓haveSameObj
2. ✓belongsTo
3. ✓shareDevelop
4. ✓printAllCom
5. ✓isCompound
6. ✓printAllAtomic
7. ✓hasPriorityCom
8. ✓hasHigherPriority

1.2 Exercise 2

- ✓validFormula
- ✓printFormula
- ✓calculateValues
- ✓isValue

1.3 Exercise 3

- ✓valid
- X isOnPossible
- X printAllPossible

2 Design

2.1 Exercise 1

The first functionality, *haveSameObj* required checking whether two software requirements were dependent on the same object.

I created facts for both element types: objective, and requirement. I then created a relationship objectiveDependency which modelled the dependency relationship between objective and requirement.

These facts allowed me to easily check that R1 and R2 were requirements, and that they both shared an objectiveDepndency relationship with the same variable O.

The next functionality, *belongsTo* required checking whether there was a dependency relationship between A and B where both variables could have any type. I decided to implement a recursive check, as any dependency of a dependency is still a dependency of the grandparent.

For this, I created a dependency clause, which recursively checks whether there are any of the four relationships, created: objectiveDependency, requirementDependency, subcomponent, and the functionalities component variable. It checks to see if there are dependency relationships between B and A. If not, then it goes up the dependency hierarchy to see if B is a dependency ancestor of A.

I decided to design the functionality type with three elements: the name of the functionality, the component it was matched to, and it's priority. This enabled me to get each value by using:

```
functionality(F, C, P).
```

This allows me to get the functionalities linked to each component, the functionalities linked to each priority, as well as the component and priority linked to each functionality.

For functionality *shareDevelop*, I again used the dependency clause. Here, I first checked that F1 and F2 were functionalities by using

```
functionality(F1,_)
```

This only returns true if F1 has been defined as a functionality. I then used

```
dependency(F1,X)
dependency(F2,X)
```

This checks to see if F1 and F2 have a shared ancestor dependency of any type.

For *printAllCom*, I created two helper clauses for printing lists.

```
printList([Head]):-
printList([Head|Tail]):-
```

These are reused in the other exercises. It works by splitting the list into the head and the rest of the list if the list contains more than 1 element (called Tail). It prints the head, then calls itself with the Tail.

I then used the *findall* command, which checks all dependency relationships where the object is an ancestor, and limits the result to just components. The result is a list, which is printed as described.

For *isCompound*, I could simply check that C is a compound, then check if there are any subcomponent relationships where C is the ancestor.

printAllAtomic used the same printList clause as described before as well as the isCompound clause. This finds all the dependencies of R which are components, then makes sure that they are also not compound. This gets a list which can then be printed using the printList clauses.

hasPriorityCom required making sure that C1 and C2 are components, then finding all of the functionality dependencies of both, which was done with the dependency clause, then limiting it's results to just funcionalities, this returned direct descendants and functionalities of subcomponents.

I then had to create a new set of clauses to determine the priority of each component. I decided to use the maximum priority out of all of the descendant functionalities as it makes sure that one very high priority task is always prioritised, no matter how many low priority tasks another component has.

To get the max, I used a similar method to the printList clause, but instead of printing, it checks the head of the functionalities list to see if it's priority is greater than the current maximum, and repeats with the rest of the list.

This gave me a maximum priority for both components, which could then be compared.

hasHigherPriority was relatively simple, I just got the priority from the functionality definition, then compared it to N.

2.2 Exercise 2

For exercise 2, I designed the valid formula to work recursively. First, I got the information about the current node.

As either P should be an operator or a value, I made sure that it was not void, and then first checked when P is an operator that the child nodes are valid sub-trees. This involved checking the operator to check which children were required (2 for and, or, implies; 1 for not), and then re-calling the validFormula clause. If all of the sub-trees are valid for an operator node, then that node is also valid.

I then added the check for leaf nodes, checking that a truth value existed for that proposition, and that the node did not have any children.

For printing the formula, I also used recursion to check if P is a proposition or operator, and either print the name of the proposition on it's own, or the name of the operator followed by an open bracket, then calling the printFormula on both child nodes before printing a close bracket.

I also made sure to check that the formula is valid first by calling the validFormula clause from before.

An issue I ran into is that my clause doesn't print a newline after printing the tree. This is minor, but given more time would have liked to improve this.

When a proposition node is found, the node truth value is checked, and then the parent node get's that value as the calculateValue asserts a truth value for the child node's ID dependent on it's children

or it's proposition value.

For when an operator is found, i created clauses for each of the operators which would call the calculateValues on the relevant child nodes, and then check the asserted truth values associated with the child nodes ID based upon the logic symbol.

The isValue clause was simple enough by calling the recursive calculateValues

2.3 Exercise 3

I failed to properly implement the isOnPossible clause. I began working on a backtracking algorithm which checks the requirements for PutOn in the given state, and then checks it's requirements in the previous states recursively.

I would have liked to complete this given more time. I feel the way to have done this would have been to create a diagram of the backtracking requirements to properly iron out the bugs.

3 Examples and Testing

3.1 Exercise 1

My testing for exercise 1 went well. I created an example model with two unconnected objectives, three requirements, four components (1 being a sub component), and 6 functionalities.

I tested each function with the first query having the expected yes or no outcomes for the input. For the second query in each test, I used as an opportunity to make sure that edge cases were identified. This helped me to spot problems with, for example, not making sure that F1 and F2 in shareDevelop are functionalities. Originally, this just checked for dependencies, but this meant that any objects passed in would return yes.

Another major issue overcame through testing was in attempting to print out the list. I found that I was having issues with the dependency clause. I realized through my tests that the clause was not checking functionalities in the correct way, and was instead of going up the hierarchy to check for the depending component, was checking for depending functionalities.

3.2 Exercise 2

I created three models for testing. They all used the same three truth values set: p, q, and r. Two of these were valid, and together used all of the operators correctly. I named the IDs 0-3, 10,13, and 20-23.

To make sure that the validFormula function worked, I tested my valid trees 0 and 20, but also tried the subtrees, as if the main tree is valid, so should all of the subtrees.

For my second query, I made sure that invalid trees were rejected. For this, i used the tree from index 10. However, I made sure to check that the sub-tree with id 11 passed because it on it's own is a valid

tree.

Then, I tested printing valid and invalid trees. I did not know of a way to create tests which checked output, but instead knew what it should look like and checked by eye. I did, however, make sure to check that the sub-trees from the first tree 0-3 also printed in a way that would make up the original tree.

For the second query, I checked the invalid tree to make sure printing returned no, and did not print anything.

Then, I tested the calculateValues clause, which uses the asserta function. To do so, I checked that the facts existed which matched the expected outcome of from the logic symbols. For the second test, I made sure that the invalid tree doesn't set any facts, unless the valid subtree is specified.

Finally, to check that isValue worked, I created tests for the invalid and valid trees. The first query checks that all of the values in the valid tree match those from calculateValues. Then, I made sure that the value check for invalid trees always fails. I then made sure that when checking the subtree values from the invalid tree, that it worked successfully.

4 Evaluation and Conclusions

In conclusion, I feel that I have developed my understanding of gprolog and automated reasoning. I successfully implemented most of the functionality, but lacked somewhat in my depth of testing.

I feel that my models were good in that they showed each aspect of functioning explicitly, making them more intuitively understandable .

Given more time, along with completing the required functionality,I would have made my solutions more interactive, by adding clauses which allowed creating valid values for each model.