Kaouter BELAOUCHI
Shubham Dhingra
Firdaws Bouzegza

Software Analysis Project

Implementation of A Lifting Model in Groove

This document is a report that presents the work done by our team. You will find there, our solution to the problem, the stages of our development process, the application and finally the result. then in appendix we put the beginning of our work for the continuation which will aim to complete the solution..

Software Analysis Project

Report

Implementation of A Lifting Model in Groove

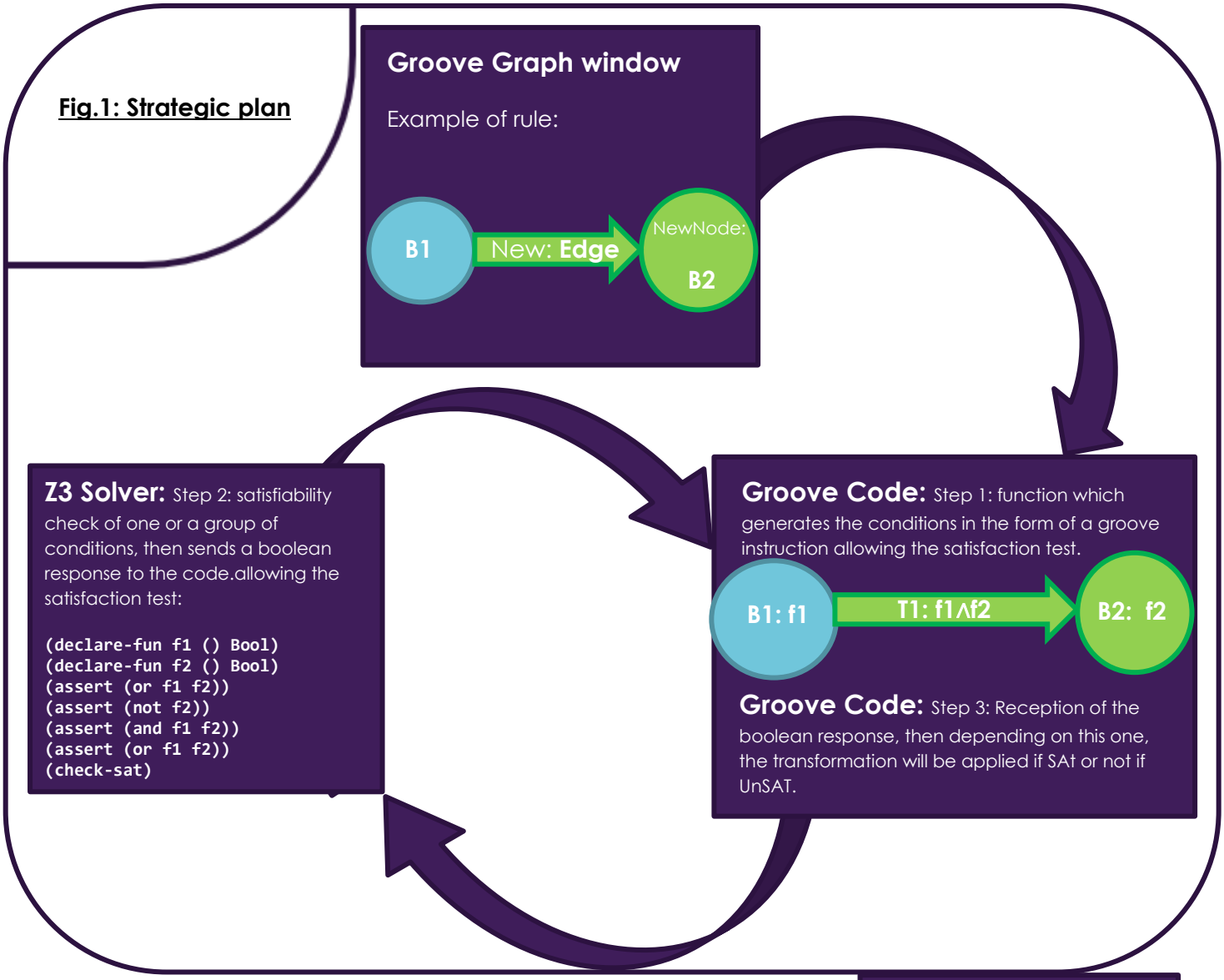
Documentation

First of all we start by documenting ourselves on the subject. Our work refers directly to the research paper: ***Lifting Model Transformations to Product Lines***, *Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, Marsha Chechik, Department of Computer Science, University of Toronto, Toronto, Canada, {rsalay, famelis, mjulia, adisandro, chechik}@cs.toronto.edu.* We then used as a basis the model presented in the research paper to start our work, and also, we are based on the algorithm fig.6 of the document to implement our solution in Groove.

Understanding the logic of the solution

The logic of the solution is that we can impose a condition on the transformations of the model due to the rule Rf. For this it will be necessary to go through a Sat Solver, we opt for Z3 which, depending on the validity of the condition, which is in this model a combination of features, will allow or not the transformation of the model according to the rule put in place. The following diagram explains the logic of a somewhat complex process:

Fig.1: Strategic plan



The logic being to schematize, we will divide the **fig.1** into two. That is to say divide the Z3 part and the groove code to focus first on the Groove development part.

The goal will therefore be to start with a very simple model, to which we will apply the rule of the figure above, then to generate the feature conditions for each of the nodes / edges, to finally transfer the whole to a csv file, while being as faithful as possible to the syntax of Z3.

The implementation of the code which allows the association of features with the different nodes and edges that we will obtain in the model, requires several steps that will be explained in the form of a function / class in java:

- **Class Box:** will be the type that will store each node / edge with its feature condition. Every Box has a Name, Feature, and a Contents List. Only the node B1(fig.1) will store in his Contents list all the other components of the graph which are nodes and edges that the rule will create.
- **addBox(Box b):** is the function of the Box class which allows the addition of a Box object in the Contents list of node B1.
- **[Class Box] generatefeatures():**is a function that automatically generates features and associates them with corresponding nodes / edges.
- **[Class Box] csvWriter():**this function allows the transcription of nodes / edges with their respective features in a csv file, while respecting the syntax of Z3, allowing the adding of processing by Z3 later, faster and more efficient.

Implementation of the chosen strategy

Here is the so-called "technical" part of the report, accompanied by screenshots and some explanations.

We must find in the Groove code the part managing the transformations. This class is **RuleApplication Class**. It is composed of a multitude of functions, but we will only be interested in the function: **applyDelta (DeltaTarget target)** which manages the creation / deletion of nodes and edges.
This function is composed of several functions, only two of them are of particular interest to us:

- **addNodes (RuleEffect record, DeltaTarget target)**
- **and addEdges (RuleEffect record, DeltaTarget target)**

then in these functions we will set up the instantiation of our Box type objects, to allow their creation, at the same time as the nodes / edges generated during the application of the rule on the graph.

```
private void addNodes(RuleEffect record, DeltaTarget target) {
    if (record.hasAddedNodes()) {
        B1.addBox(new Box("b"));
        for (HostNode node : record.getAddedNodes()) {
            target.addNode(node);
            if (node instanceof ValueNode) {
                registerAddedValueNode((ValueNode) node);
            }
        }
    }
}
```

```
private void addEdges(RuleEffect record, DeltaTarget target) {
    if (record.hasAddedEdges()) {

        for (HostEdge edge : record.getAddedEdges()) {
            B1.addBox(new Box("t"));
            HostNode targetNode = edge.target();
            if (targetNode instanceof ValueNode) {
                ValueNode valueNode = (ValueNode) targetNode;
                if (this.source.containsNode(targetNode)) {
                    unregisterIsolatedValueNode(valueNode);
                } else if (registerAddedValueNode(valueNode)) {
                    target.addNode(targetNode);
                }
            }
            target.addEdge(edge);
        }
```

Then just set up the functions mentioned above for the generation of features and transcription, according to the syntax of Z3 in a csv file. here are the functions and Class Box in screenshots:

```
// allocate features to every box type i.e node/edge
    public void generatefeatures() throws FileNotFoundException, IOException {
        this.setfeature("f1 ");
        String addition = "";
// for each node(box)/edge(box) in the list, it associate a condition
        for (Box b : this.contents) {
            if (b == this.getBoxes().get(0)) {
                this.getBoxes().get(1).setfeature("(and f1 f2) ");

            } else if (b == this.getBoxes().get(1)) {
                this.getBoxes().get(0).setfeature("f2 ");
                addition = "(and " + this.getfeature() + this.getBoxes().get(0).getfeature()
                        + this.getBoxes().get(1).getfeature();
            } else {

                if (z > 2) {
                    addition += addition + ")";
                    z = 1;
                } else
                    z++;
                b.setfeature(addition + ")");
            }

        }
        this.csvWriter();

    }
```

```
// this function write directly in the csv file to register "node/edge feature"
    public void csvWriter() throws FileNotFoundException, IOException {
        String ret = this.Name + " ; " + this.feature + "\n";
        for (Box b : this.contents) {
            ret += b.Name + " ; " + b.feature + "\n";
        }
        System.out.println(ret);
        List<String> lignes = Arrays.asList(ret);
        Path fichier = Paths.get("mapping.csv");
        //To write after the file, use the following command
        Files.write(fichier,lignes, Charset.forName("UTF-8"));
    }
```

```java
public class Box {
// This class generate features associated with nodes/edges
//and write it in a csv file
    private ArrayList<Box> contents;
    private String Name;
    static int x = 1;
    private String feature = "";
    static String fiapply = "f1";
    static int y = 1;
    static int z = 1;
//  private ArrayList<String> csvcontents;

// constructor for edge "t" and nodes "b"
    public Box(String type) {
        this.contents = new ArrayList<Box>();
        switch (type) {
        case "t":
            int y =x - 2;
            this.Name = type + y + " ";
            break;

        default:
            this.Name = type + x + " ";
            //if (x == 1) {
                //this.contents.add(new Box("t"));
            //}
            x += 1;
            break;
        }
    }
```

Appendix

Here we have the **WashingMachine** graph which is the Model on which we want to apply the lifting, and the **Rf** rule that we want to apply to it. Of course, **Rf** can only work if the lifting is operational. Here are some screenshots: