Объединение сценариев с возможностью минимизации на разных уровнях

Пример 1

Пусть у нас есть две иерархии паттернов, частично похожие:

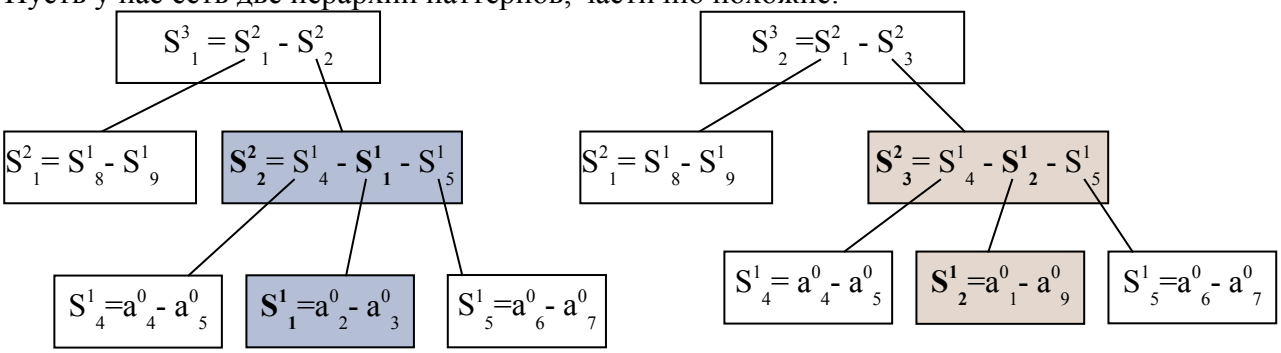
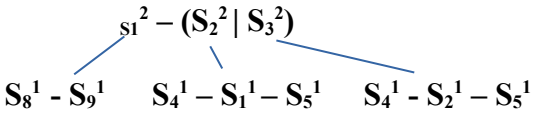


Рис.1

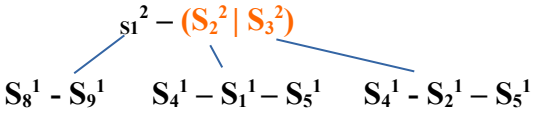
И мы хотим их соединить в один сценарий, используя ИЛИ. Склеиваем на верхнем уровне:



Если мы захотим это ИЛИ перенести на более низкий уровень и будем сверху вниз все преобразовывать, то иерархия уйдет и будет только одноуровневый паттерн.

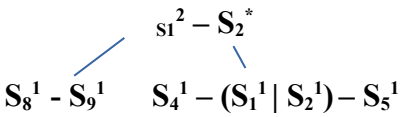
Переносить на нижние уровни можно только части иерархии. Дадим возможность пользователю выбирать ту часть верхнего уровня, которую он хочет детализировать с верхнего уровня на нижний.

Ниже оранжевым показано, что пользователь выбрал.



теперь пользователь выбирает скобку (то, что он хочет минимизировать) и нажимает кнопку — минимизировать

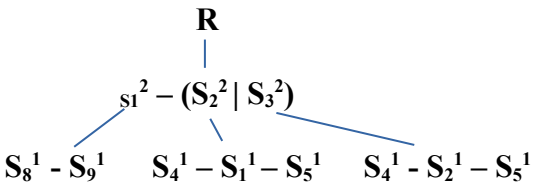
И получит он автоматически следующее:

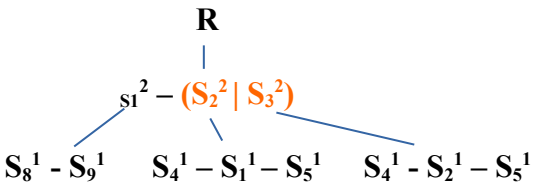


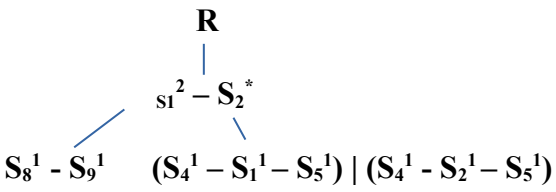
Вопрос: зачем при объединении нужно делать разные варианты?

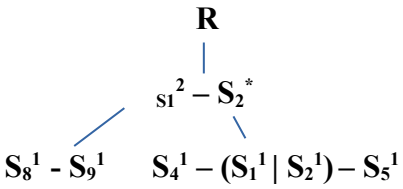
Сами паттерны, которые мы получаем на основе машинного обучения, нужны для детектирования. Мы сценарии из паттернов строим для того, чтобы эксперт мог интерпретировать то, что получилось. Просматривая разные варианты сценариев, ему будет легче увидеть смысл полученных зависимостей.

По ходу рассуждений оказалось, что сценарии у нас неправильно выглядят. Не хватает корня. Поэтому еще сверху добавляем корень.

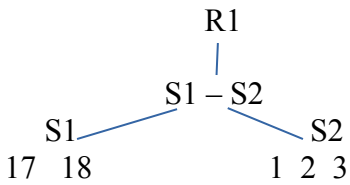


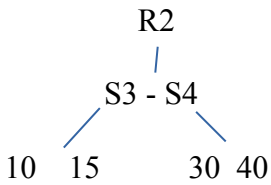




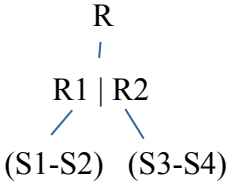


Пример 2 (где ничего не упрощается)

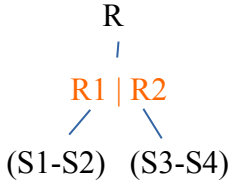




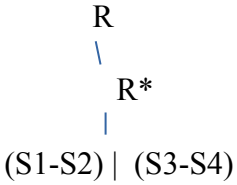
Сначала склеиваем на верхнем уровне:



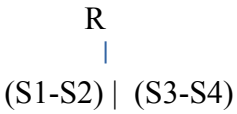
Пусть эксперт хочет перенести склеивание на нижний уровень — выделяет:



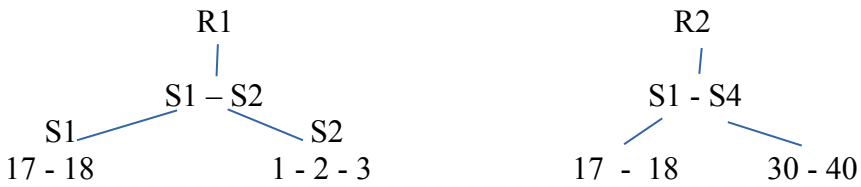
И автоматически получает:

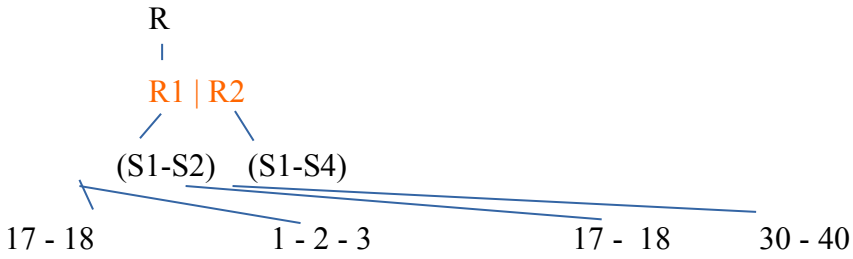


Однако два подряд уровня с одиночными регулярными выражениями не имеют смысла, оставляем один:

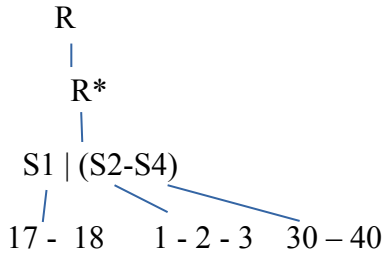


Пример 3 — здесь уже минимизация будет, т. к. есть общие символы в выражениях

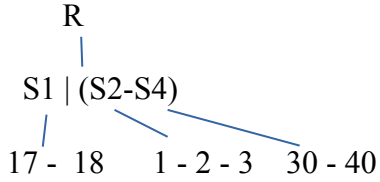




Получаем на втором и нижнем уровне избыточность — повторяются символы и даже повторяются фрагменты регулярных выражений на нижнем уровне. Поэтому пользователь выбрал фрагмент, который хочет детализировать:



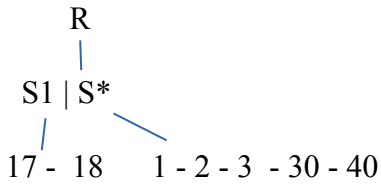
Убираем один одиночный символ:



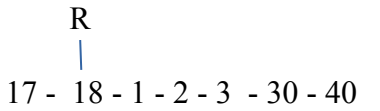
Получили более короткую запись, которую эксперт, может быть, сможет легче интерпретировать.

Может пользователь и дальше выделить, например, (S2-S4). Тогда это заменится на новый

символ, а его расшифровка будет 1 - 2 - 3 - 30 – 40



Если пользователь выделит весь второй уровень, то получится, что верхний и второй уровень будут иметь одинокие символы. Склеим их в один, получим:



Метод объединения набора разноуровневых паттернов в иерархию сценариев с возможностью минимизации на разных уровнях

1) Каждый паттерн преобразуем в отдельную иерархию сценариев, добавляя операцию — (конкатенации) между символами на каждом уровне, а также добавляя корни R1, R2, …, Rn, где n – количество полученных паттернов.
2) Объединяем регулярные выражения верхних уровней объединяемых сценариев с помощью операции | (ИЛИ).
3) Если для интерпретации иерархии сценариев требуется минимизация части или всего верхнего уровня на нижних уровнях, то в автоматизированном режиме выполняются последующие шаги 4)-9) итерационно.
4) Пользователь выделяет часть регулярного выражения верхнего уровня (или весь уровень), которую нужно минимизировать на нижнем уровне.
5) Происходит обработка только выделенного выражения, оно автоматически заменяется на новый символ (не встречавшихся ранее в регулярных выражениях иерархии сценариев).
6) На нижнем уровне автоматически появляется выделенное выражение с заменой символов на их расшифровку из следующего уровня
7) Полученное регулярное выражение автоматически минимизируется
8) Пользователь получает возможность переименовать любой символ (кроме нижнего уровня), на любой другой (в частности, осмысленный — например, Читать файл)
9) Если на двух рядом стоящих уровнях при этих манипуляциях получились одинокие символы, то эти уровни склеиваем в один.

При реализации у пользователя 2 кнопки: Объединить и Минимизировать. Нажав на кнопку Объединить пользователь выбирает, какие паттерны он хочет объединять. Теперь я уже думаю, что, может быть, оставить лес иерархий сценариев. Потому что какая-то часть некоторого класса вредоносов может иметь одно поведение, а другая часть — совсем другое. Тогда их объединение не даст никакой пользы для интерпретации. Хотя тогда эксперт увидит, что нижние уровни совсем разные и не будет минимизировать на нижних уровнях. Т.е. ИЛИ останется на верхнем уровне — в этом классе одно поведение или совсем другое. Все-таки лучше одна иерархия сценариев. И пользователь при Объединении ничего не выбирает — автоматически объединяются все паттерны.
Нажав на кнопку Минимизировать пользователь выделяет выражение, которое он хочет минимизировать на нижнем уровне. Это полезно в 2х случаях: 1) Когда есть | (ИЛИ) на верхнем уровне, а на более низком уровне символы, участвующие в операции ИЛИ детализируются так, что на нижнем уровне есть повторяемость (пример 3, первая минимизация); в этом случае можно будет на нижнем уровне провести минимизацию регулярного выражения, и иерархия сценариев получится более компактной. 2) Когда на верхнем уровне есть — (конкатенация символов), и имеет смысл для интерпретации

соединить с помощью конкатенации элементы нижнего уровня (пример 3, вторая минимизация).

Итак, цель этого метода — дать возможность эксперту получить более наглядную визуализацию полученных зависимостей.

Далее я расскажу о части работы, которая была проделана и опишу реализацию.

В заданные сроки выполнить задание целиком (с интерфейсом пользователя) было проблематично, поэтому было решено разделить задачу на две части:

1. Реализация математической модели
2. Разработка пользовательского интерфейса

В данной работе приводится реализация и описание первой части.

Итак, о модели.

Была разработана система классов на языке C++ для описания многоуровневых паттернов, а также реализованы алгоритмы «склейки» двух многоуровневых паттернов в один (с помощью операции «или») и минимизации узла, склеенного операцией «или».

Многоуровневые паттерны по большому счету являются деревьями, поэтому для них подойдет много стандартных подходов, применяемых для деревьев.

Посмотрим на задание и поймем, что существует 3 типа узлов дерева (паттерна):

- Простое выражение – паттерн, содержащий в себе последовательность листовых значений или простых выражений (в случае паттерна более высокого уровня)
- Листья – значения на самом нижнем уровне дерева. Также являются простыми выражениями.
- Бинарное выражение – будем представлять так скобку – пару паттернов, склеенных операцией «или».

Для удобства реализации объединим все типы выражений под общим интерфейсом, назовем его, скажем, `IExpression`. Какие общие методы мы можем вынести в этот интерфейс и заставить всех наследников класса реализовывать их по-своему? Вспомним, что нам придется реализовывать алгоритм минимизации, поэтому нужен выяснять, равны ли два выражения. Получаем первый виртуальный метод:

```
virtual bool isEqual(const IExpression* other) const = 0;
```

Далее вспоминаем, что у выражений неплохо бы спрашивать операнды, с которыми они оперируют – для «скобки» это будут паттерны, стоящие по обе стороны от операции «или», а для простого выражения это будет оно само. Получаем еще один метод:

```
virtual std::vector<const IExpression*> getOperands() const = 0;
```

Ну и дадим возможность клонировать выражения, это всяко удобно:

```
virtual std::shared_ptr<IExpression> clone() const = 0;
```

Рассмотрим по очереди классы, реализующие этот интерфейс.

Начнем с простого выражения. Напомню, это наш «обычный» паттерн, который состоит из некоторого набора листовых значений или других паттернов. Поэтому к реализации виртуальных методов интерфейса добавим метод, позволяющий получить структуру паттерна.

```
class SimpleExpression : public IExpression

{
public:
	SimpleExpression(const std::vector<std::shared_ptr<IExpression>>& structure = {});
	const std::vector<std::shared_ptr<IExpression>>& getStructure() const;
	bool isEqual(const IExpression* other) const override;
	std::vector<const IExpression*> getOperands() const override;
	std::shared_ptr<IExpression> clone() const override;

private:
	std::vector<std::shared_ptr<IExpression>> mStructure;
};
```

Далее рассмотрим представление листового значения: нам нужно уметь создавать его из какого-то примитивного типа, а потом это же значение и спрашивать.

```
class LeafNode : public SimpleExpression

{
public:
	using DataType = char;
	LeafNode(DataType data);
	DataType getValue() const;
	bool isEqual(const IExpression* other) const override;
	std::shared_ptr<IExpression> clone() const override;

private:
	DataType mValue;
};
```

Ну и представление бинарного выражения (в нашем случае это операция «или»):

```
class BinaryExpression : public IExpression {

public:
	BinaryExpression(std::shared_ptr<IExpression> left,
	                 std::shared_ptr<IExpression> right);
	bool isEqual(const IExpression* other) const override;
	std::vector<const IExpression*> getOperands() const override;
	std::shared_ptr<IExpression> clone() const override;

private:
	std::shared_ptr<IExpression> mLeft;
	std::shared_ptr<IExpression> mRight;
};
```

Далее напишем функции для склейки и минимизации: splice и minimize соответственно.

Splice имеет следующую сигнатуру:

```
std::shared_ptr<Model::BinaryExpression> splice(std::shared_ptr<Model::IExpression> left,
                                                std::shared_ptr<Model::IExpression> right);
```

И реализовывается очень просто – мы просто вернем бинарное выражение, состоящее из левого и правого операндов:

```
return std::make_shared<Model::BinaryExpression>(left->clone(), right->clone());
```

Minimize тоже выглядит довольно просто – если операнды состоят из одинакового количества паттернов более низкого уровня, пробежимся по ним и из двух одинаковых оставим всего один, несовпадающие же заменим на операцию «или».

```
std::shared_ptr<Model::SimpleExpression> minimize(std::shared_ptr<Model::BinaryExpression> expr) {
    auto left = expr->getOperands()[0];
    auto right = expr->getOperands()[1];

    const Model::SimpleExpression* exprLeft = dynamic_cast<const Model::SimpleExpression*>(left);
    const Model::SimpleExpression* exprRight = dynamic_cast<const Model::SimpleExpression*>(right);

    if(!exprLeft || !exprRight)
        return addDummyParent(expr);

    auto structLeft = exprLeft->getStructure();
    auto structRight = exprRight->getStructure();
    if(structLeft.size() != structRight.size())
        return addDummyParent(expr);

    std::vector<std::shared_ptr<Model::IExpression>> structure;
    for(size_t i = 0; i < structLeft.size(); ++i)
    {
        const auto leftItem = structLeft[i].get();
        const auto rightItem = structRight[i].get();
        std::shared_ptr<Model::IExpression> newItem;

        if(leftItem->isEqual(rightItem))
        {
            newItem = leftItem->clone();
        }
        else
        {
            newItem = std::make_shared<Model::BinaryExpression>(leftItem->clone(), rightItem->clone());
        }

        structure.push_back(newItem);
    }

    return std::make_shared<Model::SimpleExpression>(structure);
}
```