МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

01.04.02 Прикладная математика и информатика
(код и наименование направления подготовки)

Математическое моделирование
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Автоматизированное модельное тестирование бизнес-процессов с использованием сетей Петри»

Студент Д. Б. Сабиров
(И.О. Фамилия) (личная подпись)

Научный руководитель канд. пед. наук., доцент, О.М. Гущина
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

ОГЛАВЛЕНИЕ

Введение

Современные информационные системы активно развиваются. Все больше бизнес-процессов подвергаются автоматизации (BPA). BPA могут быть реализованы в различных сферах бизнеса, включая маркетинг, продажи и рабочий процесс. Наборы инструментов различаются по сложности, но растет тенденция к использованию технологий искусственного интеллекта, которые могут понимать естественный язык и неструктурированные наборы данных, взаимодействовать с людьми и адаптироваться к новым типам задач без обучения под руководством человека. Провайдеры BPA, как правило, сосредотачиваются на разных отраслях промышленности, но их основной подход, как правило, схож в том, что они будут пытаться обеспечить кратчайший путь к автоматизации, используя пользовательский интерфейс, а не углубляться в код приложения или базы данных, стоящие за ними. Они также упрощают свой собственный интерфейс до такой степени, что эти инструменты могут использоваться напрямую неквалифицированным персоналом. Таким образом, основным преимуществом этих наборов инструментов является их скорость развертывания, а недостатком является то, что они привлекают в организацию еще одного поставщика ИТ.

Так как такие системы находят широкое применение и по своей сущности с каждым днем становятся все сложнее. Возникает проблема того, как контролировать корректность выполнения автоматизированных бизнес-процессов. Число специалистов по тестированию требуется все больше и больше. Затраты компаний на тестирование программного обеспечения (ПО) могут доходить до 20% от его стоимости и с увеличением сложности системы затраты только растут.

Для решения этой проблемы на текущий момент компании активно ищут специалистов по тестированию ПО, обучают их самостоятельно или возлагают

эти обязанности на разработчиков ПО. Такой подход к решению проблемы все же не может длиться вечно, так как не все компании готовы тратить столько средств на тестирование своего продукта, и мы все больше замечаем снижение качества ПО.

Тема является актуальной так, как ПО становятся все сложнее, а требования к их надежности все время растут. Требуется аппарат способный автоматически тестировать систему, генерировать различные тестовые сценарии, о которых разработчик даже мог и не предполагать. Также необходим легкий способ составления самих тестов и снижения требований к разработчику тестов.

В текущей работе предлагается решение проблемы тестирования бизнес-процессов. Для этого будет описана математическая модель тестирующей системы, основанной на сетях Петри, а также будут описаны некоторые её прикладные части.

Научная новизна текущей работы заключается в новом подходе к тестированию программного обеспечения, а именно в использовании различных шаблонов описания бизнес-процессов и выстраивания их в иерархию.

Целью этого исследования является разработка математической и программной модели системы автоматического тестирования бизнес-процессов.

Объектом исследования является процесс тестирования бизнес-приложений.

Предметом исследования являются сети Петри и различные их расширения.

Поставленная цель будет достигнута если будут решены следующие задачи:

- Произведен анализ методики модельного тестирования ПО;
- Исследованы сети Петри и их модификации;
- На основе полученных знаний построена методика тестирования бизнес-процессов с использованием сетей Петри;
- Проведен анализ прикладного применения разработанной модели.

Перед формированием поставленных целей и задач был проведен анализ литературы по текущему направлению исследования.

Одной из самых близки по тематике работ является статья Testing concurrent user behavior of synchronous web applications with Petri nets авторы Jeff Offutt и Sunitha Thummala. В этой работе выполнялось тестирование веб приложений не со стороны сервера, а со стороны пользователя. Такой подход был выбран так, как современные веб приложения не обрабатывают запросы от одного пользователя одним и тем же контроллером. В связи с эти при тестировании на стороне сервера упускаются многие факторы параллельного взаимодействия. Основной упор в работе был сделан на поиск ошибок в параллельном взаимодействии нескольких пользователей и максимальном покрытии тестами.

Авторы старались автоматизировать как можно большее количество вещей. Такие как создание ребер, создание состояний, создание условий. В конечном итоге приходится все равно корректировать модель, но это значительно проще чем создавать её с нуля. Тесты генерируются и исполняются автоматически.

Также сети Петри применяются в кандидатской диссертации А.Н. Савина «Формирование тестирующих программ с использованием сетей Петри-Маркова». В этой работе также поднимается вопрос улучшение надежности программных продуктов. Так же приводится факт того, что тестирующая систем может быть намного сложнее разрабатываемой системы. Автор приводит утверждения необходимости покрытия 80% системных процессов. В ходе работы автор разработал систему для оценки времени работы системы. Эта оценка выполняется на основе математической модели, описанной сетями Петри-Маркова.

Основной упор в работе был сделан на представление и анализ параллельных процессов с помощью сетей Петри-Маркова. Сама разработанная система не выполняет тестирование кода она лишь оценивает время работы и возможные параллельные взаимодействия в системе.

Основные этапы выполнения работы:

– В ходе первого этапа было произведено исследование проблемы тестирования систем автоматизации бизнес-процессов. Были выявлены основные недостатки имеющихся решений;

– На втором этапе работы был произведен поиск решений выявленных на первом этапе проблем. Решение заключается в создании модели системы с использование сетей Петри, а в дальнейшем применять подход MBT тестирования. Также была произведена разработка самой системы автоматического модельного тестирования бизнес-процессов.

– На третьем этапе был произведен анализ разработанной системы и её практическое применение.

Особенность этой работы заключается в реализации, которая тестирует ПО не только на каком-то одном уровне (на стороне сервера, на стороне клиента, на уровне модели), а тесно связывает сторону клиента и сторону сервера. Реализация позволяет выполнить запрос от клиента и проверить правильность работы серверных механизмов до ответа клиенту. В связи такой особенность реализации открываются новые возможности применения сетей Петри для тестирования ПО.

Предложенный в работе подход к тестированию программного обеспечения позволяет автоматически тестировать большие BPA системы. Это позволяет не только существенно сэкономить на затратах на тестирования, но и в принципе позволяет поддерживать возможность тестировать такие системы.

Работа изложена на 71 страницах, содержит 30 рисунков.

Глава 1 Сети Петри и тестирование программного обеспечения

1.1 Тестирования как процесс разработки программного обеспечения

Тестирование – важнейший этап разработки программного обеспечения (ПО). На этом этапе проверяется соответствие написанных скриптов программы с исходными бизнес-требованиями. Также проверяется и сама программа тестирования.

Более формальное определение тестирования программного обеспечения:

Процесс планирования, подготовки, исполнения и анализа, направленный на установление характеристик информационной системы и выявление разницы между фактическим и требуемым состоянием [20].

Тестирование имеет особое значение в разработке крупных проектов. Важно правильно спланировать этап тестирования и сделать это нужно вовремя. План и программа тестирования должны быть выполнены в деталях при составлении бизнес– и функциональных спецификаций. Если сценарий тестирования был составлен на данной стадии, это даст бизнесу и разработчикам более четкое понимание требований бизнеса [15].

Для дальнейшего понимания специфики тестирования с использованием сетей Петри важно перечислить некоторые виды тестирования:

По уровню тестирования:

– Модульное тестирование (Unit Testing)

Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

– Интеграционное тестирование (Integration Testing). Проверяется взаимодействие между компонентами системы после проведения модульного тестирования.

– Системное тестирование (System Testing). Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т. д.

В настоящее время большую часть тестов составляют модульные тесты. Интеграционные тесты сложны в написании и в поддержки. Для написания интеграционного теста нужно составить сценарий работы [17]. Такой тест способен покрывать только конкретные заранее просчитанные сценарии.

Далее мы рассмотрим такой математический аппарат как сети Петри и определим то какие процессы ими можно моделировать.

1.2 Одноуровневые сети Петри

1.2.1 Основные определения сетей

Сеть $Net = (P_N, T_N, F_N)$ является двудольным ориентированным графом с конечным множеством вершин $P_N \cup T_N$, где $P_N \cap T_N = \emptyset$, и множеством дуг $F_N \subseteq (P_N \times T_N) \cup (T_N \times P_N)$.

Для простоты обозначения далее вместо P_N, T_N, F_N будем записывать P, T, F. Множество $P = \{p, \dots\}$ содержит в себе все позиции в сети, множество $T = \{t, \dots\}$ содержит все переходы в сети, множество $F = \{f, \dots\}$ ссодержит в себе все ориентированные ребра в сети [5]. На схеме позиции изображают в виде кружков, переходы изображают в виде прямоугольников, а ребра изображаю в виде

стрелочек, соединяющих позиции и переходы. Для обозначения ребра будем использовать следующие обозначение (u, v), u и v элементы соединённые эти ребром, а также где u входной элемент для v, а v выходной элемент для u. Множество всех входных ребер для u будем обозначать как u_{in}, а множество всех выходных ребер для u будем обозначать как u_{out}.

1.2.2 Обыкновенные сети Петри

Обыкновенной сетью Петри называется набор $PNet = (Net, W)$, где Net уже известный нам набор (P, T, F), а $W: F \to N$ определяет кратность ребер.

Основываясь на этом, запишем функцию инцидентности (формула 1).

$$F(x, y) = \begin{cases} n, & \text{если } xFy \wedge \mathrm{W(x, y)} = \mathrm{n} \\ 0, & \text{если } \neg(xFy) \end{cases} \quad (1)$$

Пара $(PNet, M_0)$ где $M: P \to N$ называется маркировочной сетью. Маркировочная сеть задает состояние сети Петри и определяет расположение фишек в вершинах этой сети. Схематически фишки обозначаются как черные кружки внутри элемента, обозначающего вершину.

Дадим определение поведению сети Петри.

Пусть $PNet = (Net, W)$ – обыкновенная сеть Петри. Тогда переход $t \in T$ является активным при разметке M, если для любой позиции из $p \in t_{in}: M(p) \geq F(p, t)$.

Дадим определение переходу в сети Петри. Пусть переход t является активным в состоянии M. Тогда состояние M' такое, что $M'(p) = M(p) - F(p, t)$ для всех $p \in t_{in}, M'(p) = M(p) + F(t, p)$ для $p \in t_{out}$, и $M'(p) = M(p)$ для $p \notin (t_{in} \cup t_{out})$, называется результатом срабатывания перехода t при разметке M. Набор (M, t, M') называется шагом срабатывания и обозначается как $M[t\rangle M'$.

Последовательностью срабатывания называется последовательность шагов в $PNet$: $M_0[t_1\rangle M_1[t_2\rangle M_2$ ..., где M_0 начальная разметка сети [5].

Далее приведем пример работы обыкновенной сети Петри. Предположим, для выполнение некой заявки нужно предоставить два документа типа A и один документ типа B. Для представления этого процесса в сети Петри нам необходимы три состояния и один переход. Первое состояние отображает наличие документа типа A, второе состояние отображает наличие документа типа B, третье состояние отображает выполненную заявку. Два ребра с кратностью 2 и 1 соединяют два первых состояние с переходом, который отвечает за выполнение заявки, а также одно ребро с кратностью 1 соединяет переход с третьим состоянием. Переход не будет активирован до тех пор, пока в состоянии один и два не будет достаточного количества фишек. На рисунке 1 изображено срабатывание такой сети Петри.

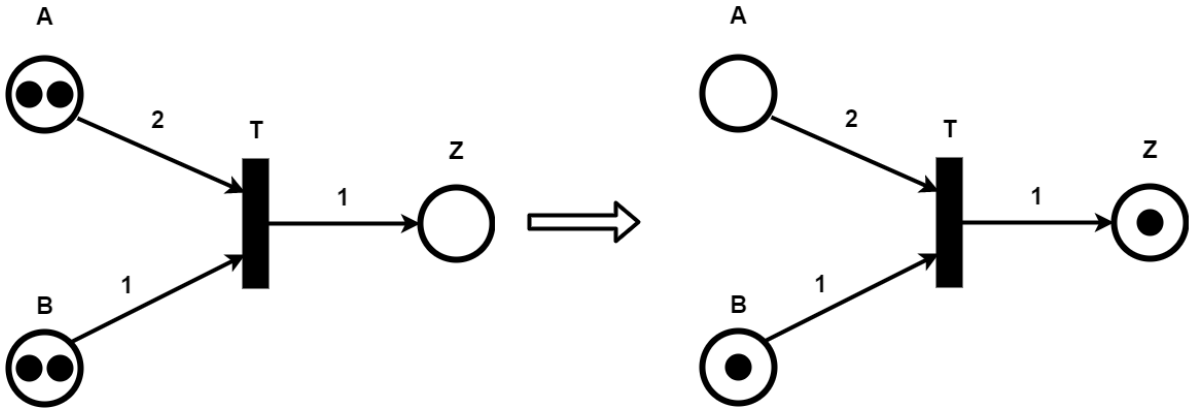

Рисунок 1 – Пример результата срабатывания сети Петри

Как видно при прохождении через переход поглощается столько фишек какова кратность ребер соединяющую позицию и переход. Также в результате срабатывания перехода в выходной позиции оказывается столько фишек какова кратность ребра, выходящего из перехода.

1.2.3 Элементарные сети Петри

Элементарные сети Петри имеют некоторые ограничения по сравнению с обыкновенными. В них каждая позиция должна содержать не более одной фишки. Следовательно, каждое ребро такой сети должно иметь кратность 1, так как в вершине не может присутствовать больше, чем одна фишка, а значит и удалить из неё больше, чем одну фишку не представляется возможным. Аналогично мы не можем поместить в вершину больше, чем одну фишку при срабатывании перехода.

В элементарной сети Петри каждая позиция подразумевает некое условие, которое может принимать значение истина или ложь. С помощью элементарных сетей Петри можно задавать некие процессы, состояния которых принимают только два значения.

Главным их отличием от обыкновенных сетей Петри это то, что на активацию перехода влияет не только входная разметка, но и выходная, так как мы не можем переместить фишку в вершину пока в ней находится другая фишка.

Для примера приведем статус заказа в некем интернет-магазине. На рисунке 2 изображены статусы заказа и его состояние определяется разметкой сети. Фишка находится в активном состоянии заказа [2].

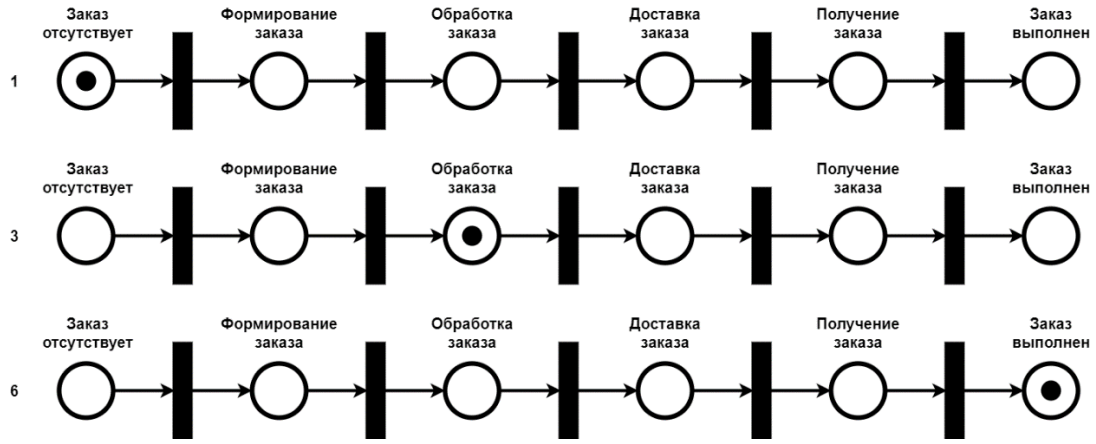

Рисунок 2 – Различные статусы заказа в элементарной сети Петри

Как видно элементарная сеть Петри позволяет удобно описать последовательные операции.

1.2.4 Сети позиций/переходов

Сети позиций/переходов являются теми же сетями Петри только с тем лишь ограничением, что для каждой позиции существует ограничение на максимальное число фишек в ней. Таким образом можно сделать вывод, что элементарные сети Петри являются сетями позиций/переходов, где на каждое состояние задано максимальное количество фишек равное единице.

Сетью позиций/переходов называется набор $PTNet = (Net, K, W, M_0)$, где $N = (P, T, F)$ есть конечная сеть, $K: P \to N \setminus \{0\}, W: F \to N \setminus \{0\}$ – функции, задающие емкость каждой позиции и кратность ребер, M_0 – функция, задающая начальную разметку сети позиций/переходов [1].

Очевидно, что для такой сети действует то же правило, что и для элементарных сетей Петри. Переход может активировать только тогда, когда во всех входных позициях достаточное количество фишек и во всех выходных позиций выполняется условие $c + e \leq mx$, где c – количество фишек в вершине, e – кратность выходного ребра, mx – максимальное число фишек, которое может находиться в выходном состоянии.

1.2.5 Раскрашенные сети Петри

Сети Петри, описанные выше, являются одноуровневыми сетями Петри. Раскрашенные (цветные) сети Петри является сетями высокого уровня. Перечислим основные отличия сетей высокого уровня от одноуровневых сетей:

- Все фишки в такой сети отличаются между собой;
- Позиции могут содержать несколько индивидуальных наборов фишек;
- Переходы могут срабатывать на различные фишки удаляя их из входных позиций и создавать новые в выходных.

В раскрашенных сетях Петри каждая фишка относится к некоему типу (цвету). Также и для каждой позиции задается свой цвет. Цвет позиции определяет фишки какого цвета может хранить текущая позиция. Приведем определение раскрашенной сети Петри [1].

Раскрашенной (цветной) сетью Петри называется набор $CPN = (\Omega, N, C, W, G, M_0)$, где

– Ω – конечное непустое множество цветов (типов);

– $N = (P, T, F)$ – конечная сеть с множеством позиций P, множеством переходов T и отношений инцидентности F;

– $C: P \to \Omega^*$ - функция, задающая раскраску каждой позиции. Она присваивает каждой позиции $p \in P$ цвет $C(p)$;

– W – функция, приписывающая дугам сети N выражение языка Σ;

– $G: T \to \Sigma$ – функция охраны, приписывающая каждому переходу $t \in T$ некоторое выражение булевского типа;

– M_0 – функция, задающая начальную разметку сети.

На основе описанных выше разновидностей сетей Петри будет строится модель системы бизнес-процессов тестируемой системы. Также будут произведены модификации этих сетей так, чтобы перенести некоторые особенности объектно-ориентированных языков программирования на поведение сетей Петри.

1.3 Элементарные структуры сетей Петри

Для построения сетей Петри используют некие элементарные структуры. На рисунке 3 изображена структура, которая называется последовательность. Структура последовательности представляет собой последовательное выполнение действий при выполнении условия. После срабатывания перехода

разрешается срабатывание другого перехода. Метка на месте $P0$ разрешает переход $t0$. Срабатывание перехода $t0$ включает переход $t1$ и т.д.

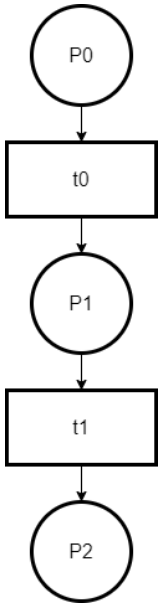

Рисунок 3 – Структура последовательность

На рисунке 4 показан пример структуры вилки, которая позволяет создавать параллельные процессы. Срабатывание перехода $t0$ запускает два параллельных процесса $P1$ и $P2$.

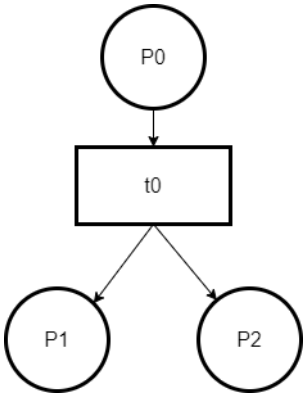

Рисунок 4 – Структура вилка

Как правило, одновременные действия необходимо синхронизировать друг с другом. Сеть объединения на рисунке 5 объединяет две или более цепи, позволяя другому процессу продолжить выполнение только после завершения предшествующих процессов.

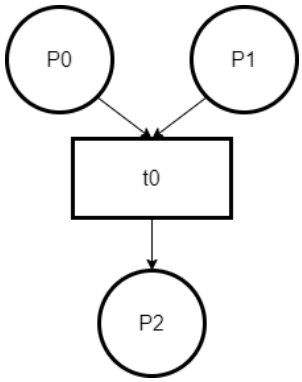

Рисунок 5 – Структура объединения процессов

На рисунке 6 изображена структура выбора, в которой срабатывание перехода $t0$ отключает переход $t1$. Этот строительный блок подходит, например, для моделирования оператора if-then-else.

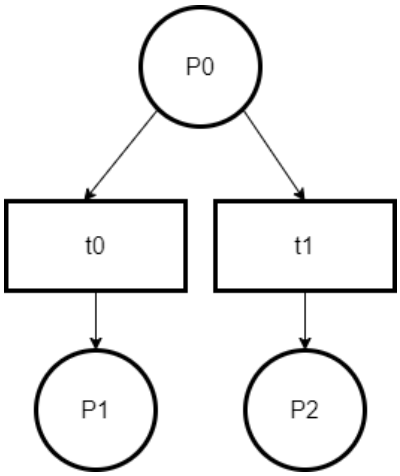

Рисунок 6 – Структура выбора

Слияние - это элементарная сеть, которая позволяет задействовать один и тот же переход для двух или более процессов [8]. На рисунке 7 показана сеть с двумя независимыми переходами $t0$ и $t1$, которые имеют общее место вывода $P2$. Следовательно, при срабатывании любого из этих двух переходов создается условие $P2$, которое позволяет запускать другой переход.

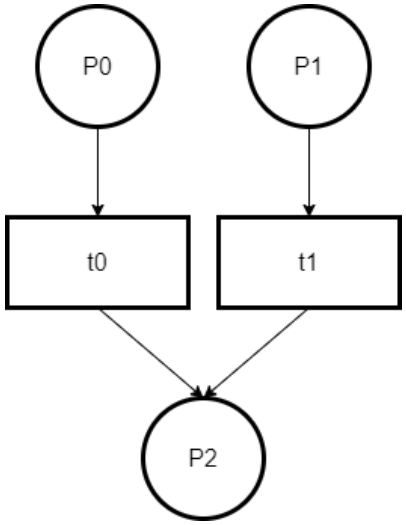

Рисунок 7 – Структура слияния

Благодаря таким простым структурам можно создавать более сложные модели сетей Петри [25].

1.3 Тестирование на основе моделей

Тестирование на основе моделей (MBT) – это метод тестирования программного обеспечения, при котором поведение тестируемого программного обеспечения во время выполнения проверяется на соответствие прогнозам, сделанным моделью.

Модель – это описание поведения системы. Поведение можно описать в терминах входных последовательностей, действий, условий, выходных данных и

потока данных от входа к выходу [7]. Она должен быть практически понятным и многоразовой и должно иметь точное описание тестируемой системы.

Создание модели - это часть жизненного цикла разработки программного обеспечения, в отличие от разработки независимого тестового сценария [24]. Вся команда должна сосредоточиться на создании тестируемого продукта и моделей, описывающих реальный пользовательский опыт.

Тестирование на основе моделей должно стать неотъемлемой частью проектирования продукта еще на стадии разработки требований. И разработчики, и тестировщики могут сосредоточиться на моделях, созданных только для удовлетворения системных требований, и с самого начала создать тестируемое приложение. В результате команда может сократить объем обслуживания набора тестов и сгенерировать больше тестов с использованием различных алгоритмов.

Доступно множество моделей, которые описывают различные аспекты поведения системы [6].

Примеры модели: Поток данных, Поток управления, Графики зависимостей, Таблицы решений, Конечные автоматы.

Тестирование на основе модели описывает, как система ведет себя в ответ на действие (определяемое моделью). Достаточно выполнить некое действие и посмотреть, отреагирует ли система в соответствии с ожиданиями.

1.4 Проблемы и преимущества модельного тестирования

Развертывание MBT в каждой организации, очевидно, требует больших вложений и усилий.

Ниже приведены недостатки MBT в разработке программного обеспечения:

– Новые необходимые навыки специалиста по тестированию,
– Увеличенное время обучения,
– Сложности в понимании модели.

Преимущества тестирования на основе моделей:

– Простое обслуживание тестового набора данных,
– Снижение стоимости,
– Улучшенное покрытие тестов,
– Возможность выполнять тесты параллельно,
– Раннее обнаружение дефектов,
– Экономия времени,
– Повышение удовлетворенности работой тестировщика.

Хотя тестирование на основе моделей является рентабельным и прибыльным для бизнеса в долгосрочной перспективе, внедрение этого подхода в устоявшиеся процессы компании может оказаться сложной задачей.

Перед внедрением MBT мышление и бизнес-культура должны сместиться в сторону моделирования. Это означает, что организации следует полностью пересмотреть свой подход к разработке и тестированию. Модели должны стать частью рабочего процесса разработки, который вносит изменения в инфраструктуру [18]. Выбор инструмента MBT также может быть проблемой. Он должен быть масштабируемым, обеспечивать надежное тестовое покрытие и позволять строить сложные модели. Поиск и внедрение такого инструмента займет некоторое время, но как только вы найдете инструмент, который вам подходит, вы получите рентабельное тестирование с меньшими затратами на обслуживание [16].

Также разработчики программного обеспечения должны знать, как разработать тестируемый продукт, а знание парадигм программирования должно дополнять навыки тестирования. Тестировщики, в свою очередь, должны использовать новый подход к тестированию на основе моделей вместо традиционных методов тестирования. Как только все это решено, подход снимает некоторые задачи, которые бизнес-аналитики, разработчики и тестировщики пытаются оптимизировать.

Глава 2 Автоматизированное модельное тестирование с использованием сетей Петри

2.1 Проблемы тестирования с использованием цепей Маркова

Цепи Маркова - эффективный способ проведения тестирования на основе моделей. Тестовые модели, реализованные с помощью цепей Маркова, можно понимать, как модель использования. Она называется тестированием на основе использования статистической модели. Модели использования, то есть цепи Маркова, в основном состоят из двух артефактов: конечного автомата (FSM), который представляет все возможные сценарии использования тестируемой системы, и рабочих профилей (OP), которые квалифицируют FSM для представления того, как система работает или будет использоваться статистически. FSM помогает узнать, что может быть или было протестировано, а OP помогает получить операционные тестовые примеры. Тестирование на основе использования статистической модели начинается с того факта, что невозможно полностью протестировать систему и что сбой может возникать с очень низкой частотой. Этот подход предлагает прагматичный способ статического получения тестовых примеров, которые направлены на повышение надежности тестируемой системы. Тестирование на основе использования статистической модели недавно было расширено для применения во встроенных программных системах [12].

Основная целью FSM на основе тестирования является сравнение уменьшенной FSM модели M с n состояниями с реализацией модели FSM, которая, как предполагается, будет представлены неизвестными FSM I. Когда сокращенный автомат M , представляющий правильную версию указанного конечного автомата, сравнивается с конечным автоматом I, могут быть выявлены следующие ошибки:

– Ошибка вывода: I и M отличаются выводом перехода;

– Ошибка переноса: I и M различаются по конечному состоянию. Когда конечное состояние, достигнутое в обоих автоматах, отличается после применения тестового примера;

– Ошибка перехода: это общий термин для ошибки вывода или переноса;

– Отсутствующие состояния: у I меньше состояний, чем у M;

– Дополнительные состояния: у I больше состояний, чем у M.

Было предложено несколько методов для создания набора тестовых примеров, которые гарантируют, что реализация не содержит ни одну из перечисленных выше ошибок, то есть полные в отношении этих ошибок. Полнота сгенерированного тестового примера подтверждается демонстрацией того, что никакая неисправная реализация FSM не пройдет тестовые примеры. Основными различиями между ними являются стоимость создания последовательностей и эффективность (способность тестовых примеров обнаруживать ошибки). Одним из первых предложенных методов был метод W, который считается предшественником area, поскольку на нем основано большинство следующих методов: UIO, UIOv, Wp, HIS и SPY [22].

2.2 Проблема применения конечного автомата для тестирования ПО

Применять конечный автомат для тестирования программного обеспечения возможно, но у него есть один существенный недостаток. С помощью конечного автомата очень сложно моделировать ситуации с большим количеством состояний [19]. Для примера приведем некий интернет-магазин, в котором пользователь может совершать покупки неких товаров. На рисунке 8 изображен то какие состояния могут быть.

Как видно у нас есть 3 возможных состояния:

– Когда пользователь не зарегистрирован, а следовательно, у него нет ещё покупок;
– Когда пользователь зарегистрировался, но ещё не сделал покупку;
– Когда пользователь зарегистрирован и уже совершил покупку.

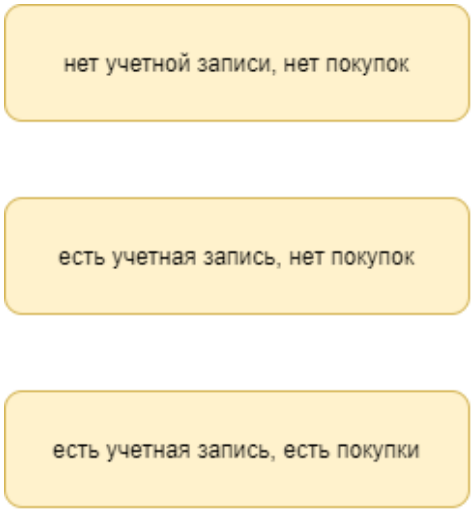

Рисунок 8 – Возможные состояния интернет-магазина

На данный момент количество состояний не так велико и в них достаточно просто разобраться, но все же можно заметить, что одно состояние конечного автомата хранит в себе полное состояние системы, а не отдельный её аспект.

Теперь изобразим переходы, которые будут указывать на действия, которые нужно совершить, чтобы сменить состояние системы (рисунок 9).

Как видно по рисунку 9 пользователь может:

– из состояния «нет учетной записи, нет покупок» перейти в состояние «есть учетная запись, нет покупок» посредством создания учетной записи пользователя;

– из состояния «есть учетная запись, нет покупок» вернуться на предыдущее состояние посредством удаления совей учетной записи или перейти

в следующее состояние «есть учетная запись, есть покупки», совершив первую покупку;

– из состояния «есть учетная запись, есть покупки» вернуться на предыдущее состояние посредством удаления всех своих покупок или остаться в том же состоянии, совершив ещё одну покупку.

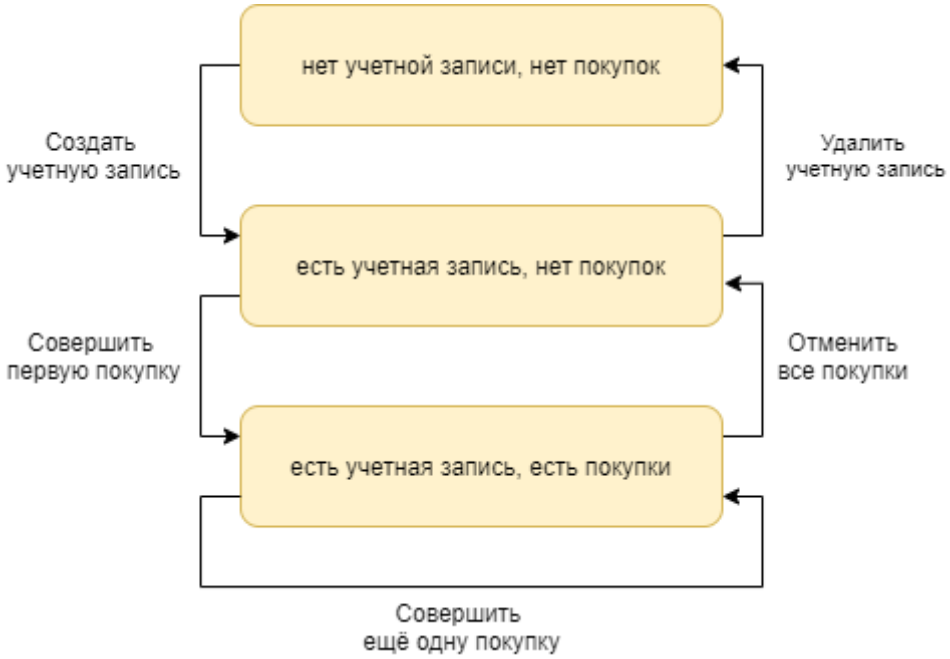

Рисунок 9 – Возможные переходы между состояниями

Изображенная выше система достаточно проста. Если же попробовать усложнить текущую модель и добавить наличие у пользователя купона, то модель значительно усложниться. Модель, которая контролирует наличие купона у пользователя изображена на рисунке 10.

Можно заметить, как значительно увеличилась модель системы и как снизилась её читаемость. Также на этой модели, для компактности, отсутствуют некоторые переходы такие как переходы из любого состояния в первое.

Пользователь может удалить учетную запись не только находясь в состоянии «есть учетная запись, нет покупок, нет купонов».

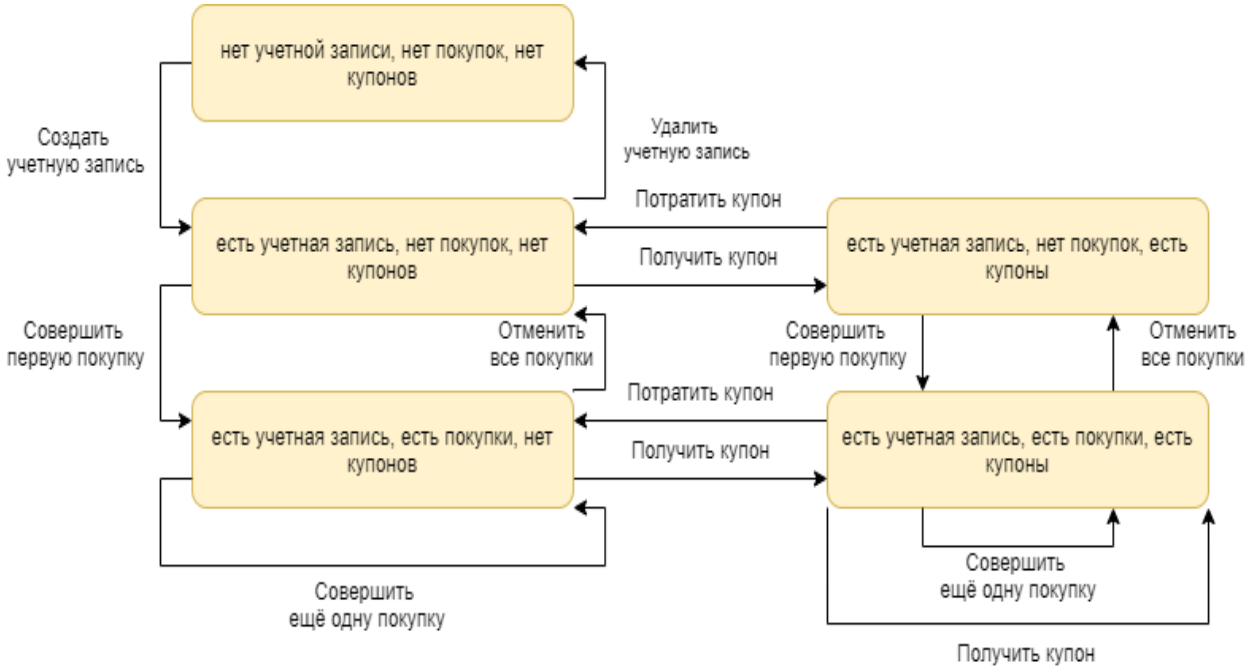

Рисунок 10 – Модель с возможностью получать купоны

Выведем следующие недостатки конечного автомата:

– дублирование переходов и состояний,

– отсутствие возможности разделить состояния.

– Попробуем теперь изобразить эту же модель, но уже используя сети Петри. Для этого смоделируем начальное состояние системы (рисунок 11).

На рисунке 11 изображено начально состояние системы. При моделировании сетями Петри мы не храним общее состояние системы в одном узле. Общее состояние системы описывается токенами находящимися в различных вершинах сетей Петри.

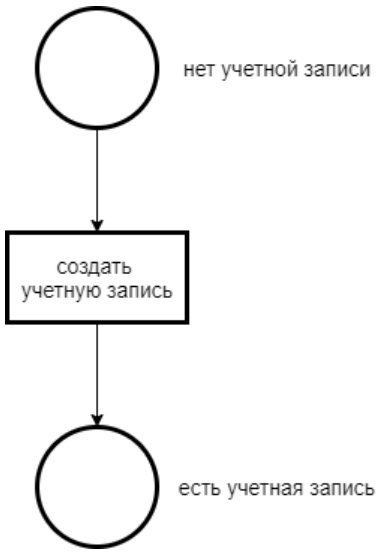

Рисунок 11 – Описание одного состояния сетью Петри

На рисунке 12 изображено то, как в сетях Петри отображается система в состоянии «нет учетной записи» и «есть учетная запись».

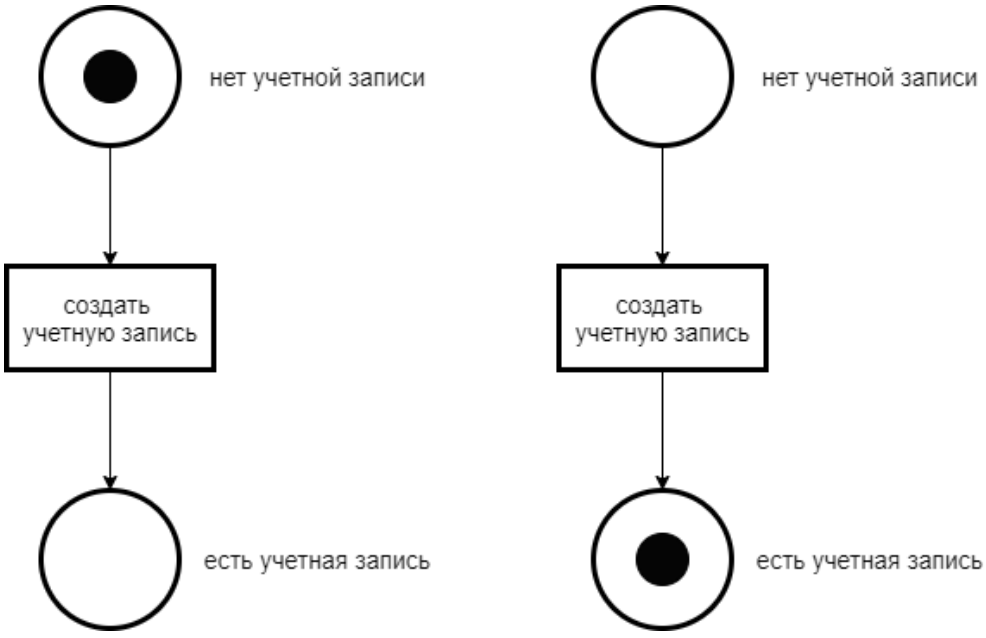

Рисунок 12 – Моделирование одного параметра системы с помощью сетей Петри

Далее поэтапно смоделируем все состояния системы с помощью сетей Петри. Для этого создадим три позиции:

– учетная запись отсутствует
– учетная запись создана
– у пользователя есть покупки

На рисунке 13 изображены три, описанных выше состояния.

Рисунок 13 – Состояния системы в сетях Петри

Далее необходимо добавить возможные переходы системы. Здесь можно заметить основное отличие сетей Петри от конечного автомата. В сетях Петри нет дублирования переходов, то есть независимо от числа состояний системы в ней будет требоваться дополнительно моделировать повторяющиеся действия.

На рисунке 14 изображена модель системы в сетях Петри.

Как можно заметить модель получилась простая и читаемая. Двойная стрелочка между состоянием «есть учетная запись» и переходом «совершить покупку» означает, что токен не будет удаляться из при выполнении перехода, а будет скопирован. Далее на рисунке 15 приведен пример начального состояния системы.

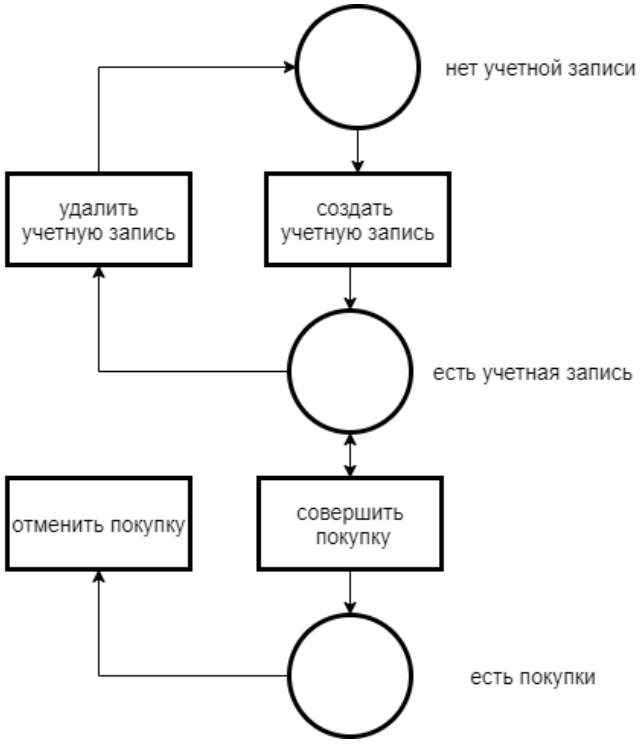

Рисунок 14 – Модель системы в сетях Петри

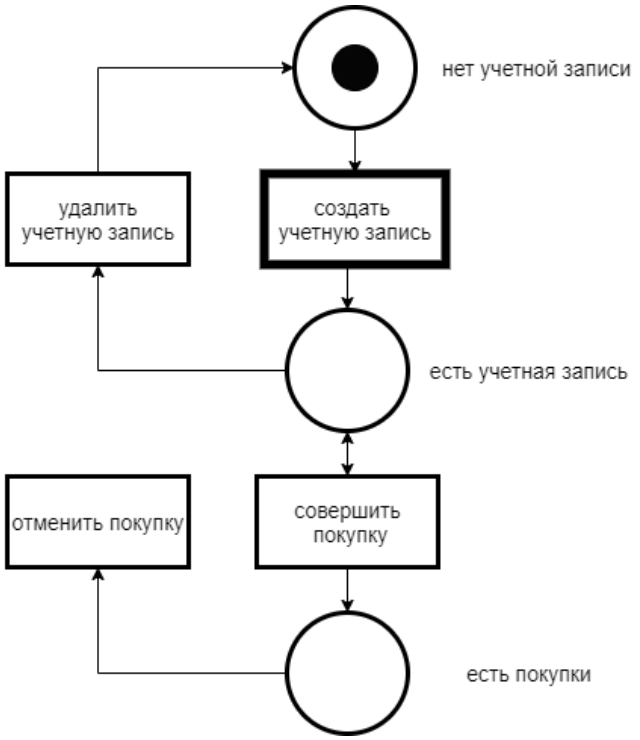

Рисунок 15 – Начальное состояние системы

Жирным контуром отмечена доступные переход. Очевидно, в начале для пользователя доступна только регистрация. Выполним переход и посмотрим на состояние системы (рисунок 16).

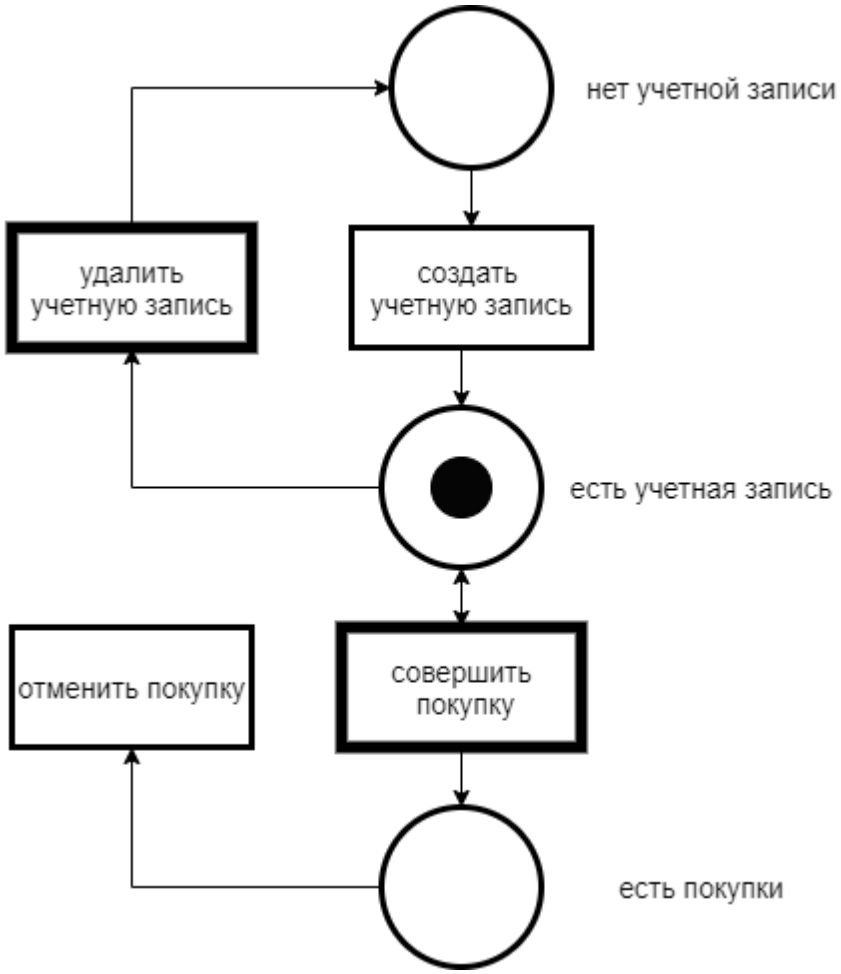

Рисунок 16 Состояние системы после регистрации пользователя

Теперь активно системе доступны переходы «удалить учетную запись» и «совершить покупку». Также можно заметить, что токен полностью описывает состояние системы. У пользователя создана учетная запись и нет покупок. Далее выполним переход «совершить покупку» (рисунок 17).

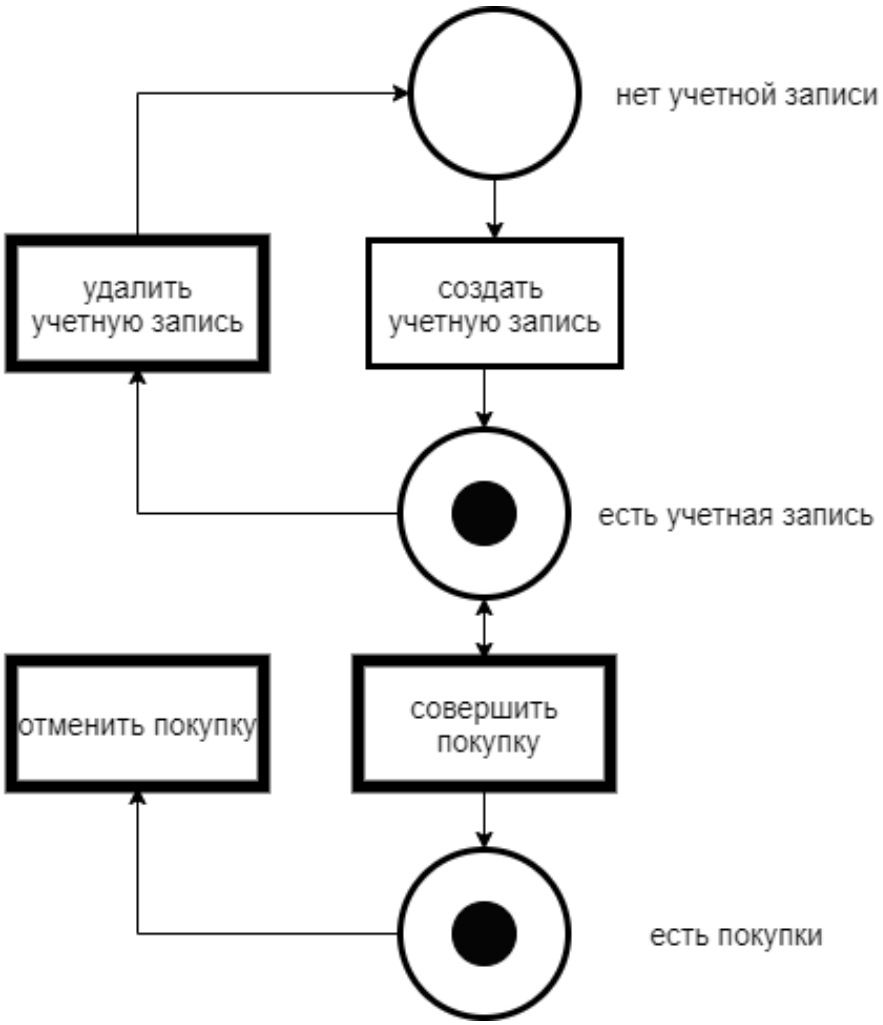

Рисунок 17 – Состояние системы после совершения покупки

Благодаря тому, что токены могут дублироваться, сеть Петри позволяет указывать на наличие сразу нескольких параметров системы. Так в текущем состоянии у пользователя есть учетная запись и есть одна покупка. Также существует возможность повторно совершить покупку и тогда число токенов в состоянии «есть покупка» увеличится (рисунок 18).

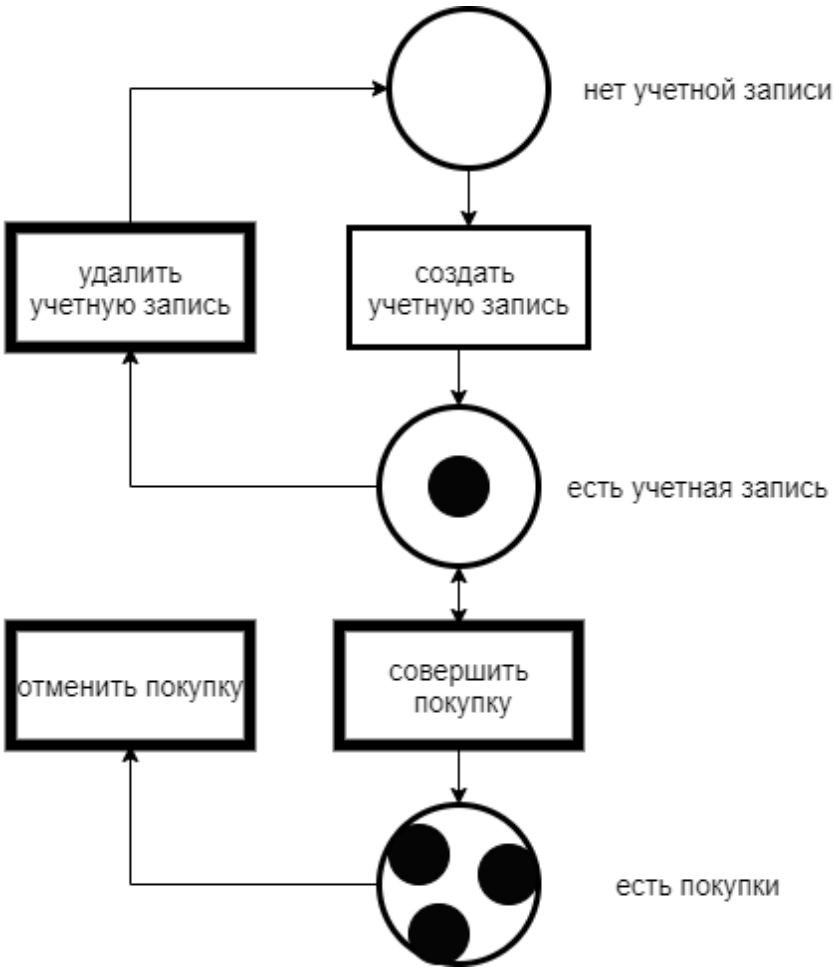

Рисунок 18 – Пример состояния системы после совершения нескольких покупок

Теперь, если же требуется расширить модель системы и добавить в неё хранение купонов, то с моделью, описанной сетями Петри, это делает достаточно просто.

Для этого необходимо создать новую позицию в сети Петри. Позиция должна описывать наличие у пользователя купона. Также необходимо добавить переходы, которые будут обозначать действие по получению купона и по его удалению. На рисунке 19 приведена расширенная модель системы.

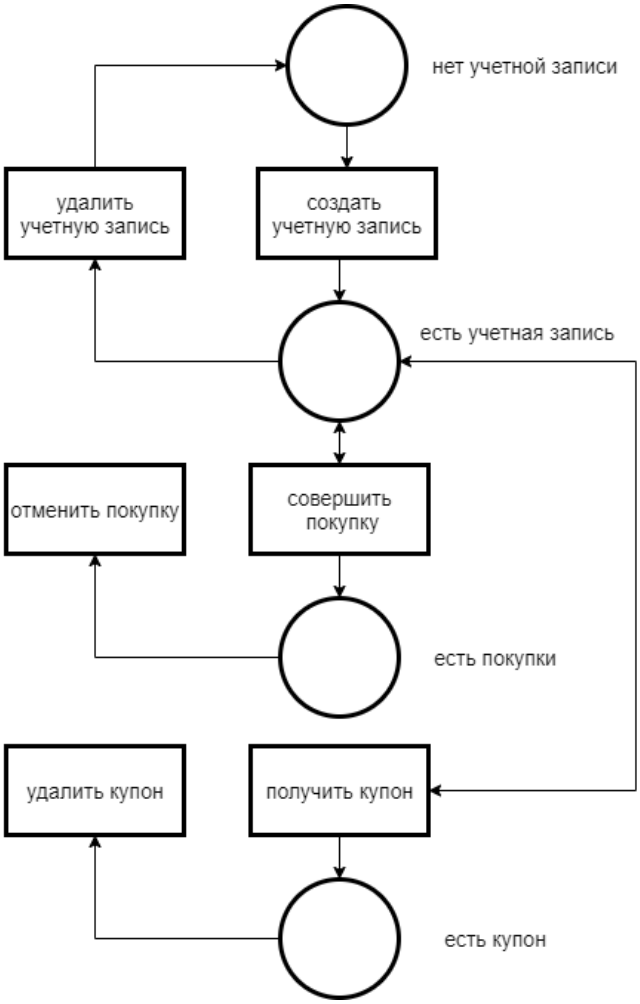

Рисунок 19 – Расширенная модель системы, описанная сетями Петри

Как можно заметить в модели не произошло существенного увеличения числа состояний или переходов. Это и есть существенное преимущество сетей Петри над конечным автоматом. Модель системы, описанная сетями Петри, продолжает оставаться понятной и компактной.

2.3 Проблема многопользовательского тестирования веб-приложений

Веб-приложения теперь используются во всех аспектах нашей жизни для управления работой, предоставления продуктов и услуг, чтения электронной

почты и обеспечения развлечениями. Программные технологии, используемые для создания веб-приложений, предоставляют функции, которые помогают разработчикам разрабатывать гибкую функциональность, но такие приложения сложно моделировать и тестировать.

В частности, сетевая модель запрос-ответ означает, что веб-приложения по своей природе являются «без состояния» и неявно параллельными. Они не имеют состояния, потому что новое сетевое соединение устанавливается для каждого запроса (например, когда пользователь нажимает кнопку отправки).

Таким образом, сервер по умолчанию не распознает несколько запросов от одного и того же пользователя. Веб-приложения также работают параллельно, потому что несколько пользователей могут одновременно использовать одно и то же веб-приложение, создавая конкуренцию за одни и те же ресурсы [9].

К сожалению, большинство веб тестов приложения не дает адекватной оценки этим аспектам веб-приложения, оставляя много ошибок программного обеспечения в развернутом веб-приложении. Часть этой проблемы заключается в том, что большинство традиционных инструментов моделирования программного обеспечения (таких как UML) не имеют встроенной поддержки параллельных и не связанных с состоянием аспектов веб-приложений.

В этом исследовательском проекте используется новая модель, основанная на сетях Петри для описания определенных аспектов поведения веб-приложений.

Это является новой техникой для разработки тестов из модели, которые явно тестируют параллелизм в веб-приложениях. Для тестирования используется критерии покрытия, которые определены в модели сети Петри [23].

Многие веб-приложения не работают из-за ошибок параллелизма. Параллелизм возникает, когда может быть запущено более одной задачи одновременно, которые могут взаимодействовать между собой. Из-за способа развертывания веб-приложений на серверы и доступ к ним через Интернет, они неявно параллельны.

Сложность параллелизма усугубляется тем, что многие дизайнеры и разработчики не знают достаточно для разработки и создания высококачественных параллельных программ, и многие тестировщики работают, не имея опыта тестирования на проблемы с параллелизмом.

Веб-приложение – это больше, чем статический веб-сайт – это программа, которая развернута через Интернет. Пользователи получают доступ к программе через веб-браузеры, и большая часть программного обеспечения работает на удаленных серверах, с компьютерами пользователей. Веб-браузеры также позволяют пользователям явно создавать параллелизм с использованием нескольких вкладок в одном браузерном окне, несколько окон в одном экземпляре браузера, или окна в разных браузерах. Эти варианты могут привести к разному поведению [10].

2.4 Иерархические сети Петри

Сложный бизнес-процесс может включать в себя множество других подпроцессов, каждый из которых имеет свои особенности и структуру. Процесс, который описан полностью одной моделью сети Петри будет достаточно сложен в понимании и поддержке. Такой процесс в каком-то виде даже будет избыточен, так как не во всех случаях нужны все возможности цветных сетей Петри и возможно какую часть процесса было бы проще описать с помощью элементарных сетей Петри. Также разбив модель на части, она становится более структурированной и читаемой. Процессы могут отличаться между собой, но какие-то их части совпадать тогда деление на модели процесса на подмодели позволит переиспользовать повторяющиеся части. Организовать такую структуру, где сеть Петри состоит из других сетей Петри разного уровня позволяет теория вложенных сетей Петри [2].

Иерархические сети Петри представляют из себя сеть Петри с не мгновенными переходами. То есть переход окончательно срабатывает только тогда, когда выполнится сеть более низкого уровня. Та сеть в свою очередь тоже может содержать не мгновенные переходы в виде других сетей Петри.

Таким образом мы можем задавать модели сложных бизнес-процессов, а также объединять их.

2.5 Метод описания бизнес-процесса с помощью иерархических сетей Петри

При построении модели бизнес-процесса с помощью иерархических сетей Петри важно разделить процесс на логические части. Каждый отдельный подпроцесс должен быть описан одним из видов сетей Петри. Так допустим, как отмечалось раннее, последовательные процессы или простые состояния, которые принимают значение истина или ложь лучше моделировать с помощью элементарных сетей Петри. После того как получена модель подпроцесса их можно объединять. В таком случае для сети net_i уровня l_k сеть net_j уровня l_{k+1} будет является переходом, который сперва должен выполниться.

Так как сети Петри моделируют асинхронные процессы в них нет четкого порядка активации переходов. Каждый переход может сработать равновероятно. Для тестирования бизнес-процессов такое поведение не дает нужной эффективности, и чтобы повысить качество тестирования их покрытие важно контролировать очередность выполнения переходов. Ранее уже был предложен метод добавления некой вероятности срабатывания перехода. Эта вероятность снижалась каждый раз при срабатывании перехода. Таким образом увеличивается частота срабатывания более отдаленных от начальной позиции переходов. С применением иерархических сетей Петри этого подхода недостаточно. Теперь уменьшая вероятность срабатывания перехода в сети уровня l_k мы будем

уменьшать вероятность срабатывания всех сетей уровня l_{k+1} ... l_n, являющимися потомками этого перехода. Для решения этой проблемы предлагается использовать следующий механизм [9].

Так как алгоритм выбора срабатывания очередного перехода не понимает на каком уровне расположен переход. Мы будем при построении начальной разметки сети Петри присваивать вероятность срабатывания перехода для сети уровня l_k вероятность, вычисляемую по рекурсивной формуле 2.

$$p_{i+1} = 2 * p_i \tag{2}$$

Таким образом каждый переход уровня $i + 1$ будет иметь в два раза большую вероятность срабатывания чем переход уровня i. Очевидно, что любую иерархическую сеть Петри можно представить в виде обычной сети Петри, но только при условии, если каждый уровень придерживается определенному заранее типу сетей Петри. Применение иерархического представления сетей Петри значительно облегчает построение модели бизнес-процесса, а также уменьшает объем самой модели за счет переиспользования повторяющихся элементов. В дальнейшем при активном применении модельного тестирования с использованием сетей Петри позволит накопить базу готовых моделей и из них строить новые бизнес-процессы и следовать методологии TDD [21].

2.6 Свойства сетей Петри

После того как получена модель бизнес-процесса она уже может быть использована в качестве модели для тестирования. Однако основным преимуществом сетей Петри является их четкая математическая модель, которую можно исследовать. Благодаря этому можно уже на этапе проектирования бизнес-процесса находить в нем ошибки и точки оптимизации [14].

Перечислим свойства сетей Петри:

– ограниченность – это свойство указывает на ограничение на число фишек в одной позиции. Так позиция p_i называется k-ограниченной, число фишек в ней не больше k. Сеть Петри будет называться ограниченной только, если все позиции в ней ограничены;

– безопасность – это свойство указывает на то, что в сети Петри общее число фишек не превышает единицы. Так мы сможем проверить правильность построения сети Петри, если нам необходимо смоделировать последовательные смены состояний. Такая сеть должно обладать свойством безопасности;

– сохраняемость – это свойство указывает на то, что в сети Петри суммарное число фишек остается постоянным в любой момент времени. Таким свойством должны обладать сети, в которых фишки являются ресурсами и логично, что их число должно оставаться постоянным;

– живость – это свойство относится к переходу в сети Петри. Переход обладает этим свойством, если существует такая разметка сети Петри, где этот переход активен;

– достижимость – это одно из основных свойств разметки сети Петри. Оно указывает на возможно получения определенной разметки M_k из разметки M_0.

Таким образом можно изначально определить какими свойствами должна обладать сеть Петри и проведя анализ сети можно определить правильно ли построена модель [4].

2.7 Дерево достижимости

Дерево достижимости – это ориентированное корневое дерево, вершинами которого, соответствуют возможные маркировки, а дугам – переходы. В таком дереве корнем будет являться начальная маркировка M_0. Из корня выходят дуги,

соответствующие разрешенным переходам в этой сети. Далее из всех полученных маркировок аналогично выходят дуги для перехода в другие доступные разметки. Дерево представляет все возможные последовательности активации переходов. Так всякий путь, выходящий из корня дерева, соответствует доступной последовательности переходов [9].

Так, анализируя дерево достижимости, можно определить все достижимые позиции в сети Петри. Это позволяет находить ошибки в проектировании бизнес-процессов в отличии от привычной нотации BPMN.

2.8 Подход к созданию MBT тестов

При использовании термина MBT предполагается, что модели могут быть неформальные, то есть не иметь формальной семантики. Таким образом, модель не может быть выполнен напрямую. В MBT модель впервые используется чтобы создавать абстрактные тесты, которые впоследствии уточняются для создания конкретных тестов [26]. Абстрактный тест определяется в терминах модели, и использует только имена, которые появляются в модели. Например, абстрактный тест может быть что-то вроде: [Enter State1, переход в State2, переход в State5, переход в State2, переход в State6]. Конкретный тест должен включать команды и входы, которые даны и понятны реализацией.

В зависимости от модели, конкретные тесты могут быть созданы автоматически из абстрактных тестов. Это может быть относительно легко или может быть довольно сложным и требовать значительные знания о реализации. Предполагается, что конкретные тесты должны быть созданы тестером, хотя это возможно в будущем хотя бы частично автоматизировать [29].

Модели предлагают несколько преимуществ для анализа и тестирования программного обеспечения. Они позволяют тестировщикам выполнять большую часть своей проектной работы. на абстрактном уровне, где они не теряются в

деталях исходного кода. Они также позволяют тестировщикам понять и сосредоточиться на ключевых функциональных поведениях программного обеспечения.

2.9 Построение модели сети Петри для веб-приложений

Теперь определим метод использования сетей Петри для моделирования параллельного поведение веб-приложения. Генерация модели в основном автоматизировано, но включает в себя дополнительное руководство. Во-первых, используется утилита, которая была разработана для этого исследования. Она используется для извлечения навигационной структуры веб-приложения.

Она выполняет следующее:

– Извлекает каждый компонент (HTML, сервлет или JSP) в позицию в сети Петри. Модель и усилия по тестированию, фокусируется на уровне представления, поэтому не моделируются Java-классы в бизнес-логике и в слоях данных.

– Извлекает ссылки, вызовы методов или событий, которые вызывают переходы между компонентами, чтобы сформировать переходы сети Петри.

– Извлекает логические выражения для формирования условий на переходах.

– Строит на основе полученных позиций, переходов и условий модель сети Петри.

Если модель сети Петри получилась очень большая или сложная, тестировщик имеет возможность вручную разложить сеть Петри, созданную в утилите на отдельные сети Петри. Это разложение может помочь процессу генерации тестового набора или разделить задачи. Ожидается, что это будет более необходимо в больших приложениях. Также важно отметить, что разложение

может потребовать значительного времени от тестировщика, а также знание системы и сетей Петри [28].

2.10 Критерии испытаний для моделей сети Петри веб приложения

Модельное тестирование (MBT) используется для получения тестов из абстрактной модели программного обеспечения. Модель на основе тестирования начинается с модели, которая описывает всю или часть ожидаемого поведения системы и ее окружения. Как сказано ранее предполагается, что модели не обязательно являются исполняемыми (то есть формальное описание не обязательно).

Критерии испытаний используются для выбора конечного набора тестов, которые охватывают определенные элементы модели (например, посещение каждой позиции или каждого перехода в графической модели).

Критерий теста – это набор инженерных правил для того, как генерировать тесты из какого-то тестового артефакта. Критерии теста могут быть определены в исходном коде, модели или в функциональных требованиях. Критерии теста создают тестовые требования, которые являются конкретными вещами, которые должны быть проверены или покрыты [11]. Например, критерий покрытия состояний требует, чтобы каждое утверждение было проверено хотя бы один раз, таким образом каждое утверждение представляет собой требование теста. В MBT, тестовые требования приведены с точки зрения модели, то есть узлов и ребер в графе или мест и переходов в сети Петри.

Каждое требование к тесту удовлетворяется одним или несколькими тестовыми случаями.

Тестовый набор состоит из пары входных и ожидаемых выходных значения модели плюс другие входные данные (например, предварительное условие и последующие значения условий), необходимые для запуска входных значений.

В MBT тестовые случаи сначала рассматриваются как абстрактные тесты, используя имена из модели. Затем они переводятся или уточняются, к конкретным тестам, которые содержат входные значения, которые могут быть заданы в реализации.

В текущей работе используется модель сети Петри веб-приложения для генерации тестов, чтобы охватить каждый переход и место в сети Петри.

Тестирование всех возможных последовательностей выполнения в моделях сети Петри невозможно, поэтому используется критерии покрытия для определения тестовых случаев, которые направлены на конкретные проблемы. Утилита генерирует модель сети Петри непосредственно из реализации, и генерирует абстрактные тесты автоматически. Затем уточняется сценарий испытания на конкретные испытания вручную.

После того как получена модель бизнес-процесса она уже может быть использована в качестве модели для тестирования. Однако основным преимуществом сетей Петри является их четкая математическая модель, которую можно исследовать. Благодаря этому можно уже на этапе проектирования бизнес-процесса находить в нем ошибки и точки оптимизации.

2.11 Проверка соответствия диаграммы UML бизнес-процессу описанного сетью Петри

Процесс выполнения (также известный как экземпляр процесса или сценарий) представляется как последовательность действий, называемых трассировкой. Журнал событий представляет собой мультимножество путей, представляющих поведение

Определим подход к применению соответствия проверка на наличие моделей бизнес-процессов, ориентированных на артефакты в UML, который учитывает не только последовательность задач, но также контекстно-зависимые

ограничения. Это достигается переводом оригинальной модели UML в сеть Петри. Такой способ включает в себя знания из контекстно зависимых ограничения. Далее лог существующего приложения сравнивается с логом, генерируемым сетью Петри.

Определяются модели UML в Visual Paradigm, чтобы можно было автоматически получить файл XML с его спецификацией. Можно использовать и другие инструменты при условии, что они способны генерировать требуемый XML. Затем модель UML внедряется в систему, которая должна иметь возможность генерировать соответствующие журналы для проверки соответствия.

Затем во время выполнения, когда процессы развернуты и система уже сгенерировала журналы событий, мы используем инструменты в платформе для проверки соответствия моделей UML. Эта структура включает в себя несколько плагинов (не показаны на рисунке), которые позволяют преобразовать файл XML, кодирующий модель UML, в сеть Петри, а затем выполнить проверку соответствия из данного журнала событий и сгенерированной сети Петри. Затем эти результаты могут быть применены обратно к исходной спецификации UML для обновления модели или внесения изменений в развернутую систему [27].

Проверка соответствия появилась как область процесса майнинга, который сравнивает существующие модели процессов с фактическими наблюдениями выполнения процесса, представленных в виде событий, фиксируемых в логах.

Соответствие на основе выравнивания считается современными методами и был адаптирован к конкретным сценариям, таким как большие процессы, генетические алгоритмы, мультиперспектива и частичный порядок. Большая часть методов проверки соответствия сосредоточены на потоке управления процессами.

Несмотря на это, методы, которые принимают во внимание дальнейшие измерения, такие как данные или ресурсы появились недавно, что может стать первым шагом к улучшению проверки соответствия.

С точки зрения представления бизнес-процессов, различаются процессно-ориентированные и артефактно-ориентированные подходы. Процессно-ориентированные подходы фокусируются на потоке управления процессом, который, как правило, использует такие языки, как BPMN, YAWL, диаграммы UMLActivity или Workflownets для представления процессов.

В случае использования BPMN и UML диаграмм, существуют методы для перевода этих моделей в сети Петри. С другой стороны, YAWL и Work-Flow сети уже основаны на сетях Петри [23].

Как можно заметить из этой главы сети Петри являются очень мощным и гибким инструментом при моделировании различных систем.

Благодаря совей расширяемости сети Петри могут удобно подстраиваться под решение различных задач математического моделирования. Так они отлично подходят при моделировании бизнес-процессов и значительно снижают объем модели.

Глава 3 Разработка системы тестирования ПО с использование сети Петри

3.1 Общая концепция тестирующей системы

Тестирующая система будет состоять из нескольких модулей:

1. Среда проектирования и выполнения сети Петри – это среда, в которой выполняется проектирование сети Петри, её валидация и выполнение без привязки к тестируемой системе.
2. Модуль симуляции – этот модуль принимает валидную сеть Петри и запускает симуляцию передавая сигналы модулю тестирования системы.
3. Модуль тестирования системы – этот модуль напрямую связан с тестируемой системы и может выполнять манипуляции и проверять состояния системы. Этот модуль также содержит в себе модули, отвечающие за различные интеграции с системой такие как WEB, GUI, БД и т.д.

Схема тестирующей системы изображена на рисунке 20

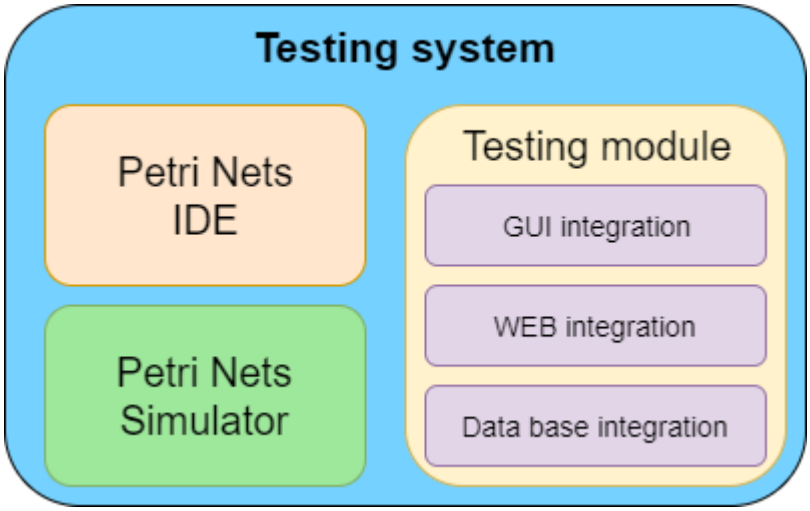

Рисунок 20- Схема тестирующей системы

Рассмотрим теперь более подробно каждый модуль.

На текущей момент в роли среды проектирования сетей Петри выполняет программа CPN Tools. Эта программа позволяет проектировать и выполнять сети Петри, а также конвертировать их в формат XML. В дальнейшем планируется заменить эту программу дополнить другой которая будет иметь интеграцию с кодом и позволит производить его анализ и за счет подсказок увеличит скорость разработки. На рисунке 21 изображено рабочее окно программы CPN Tools.

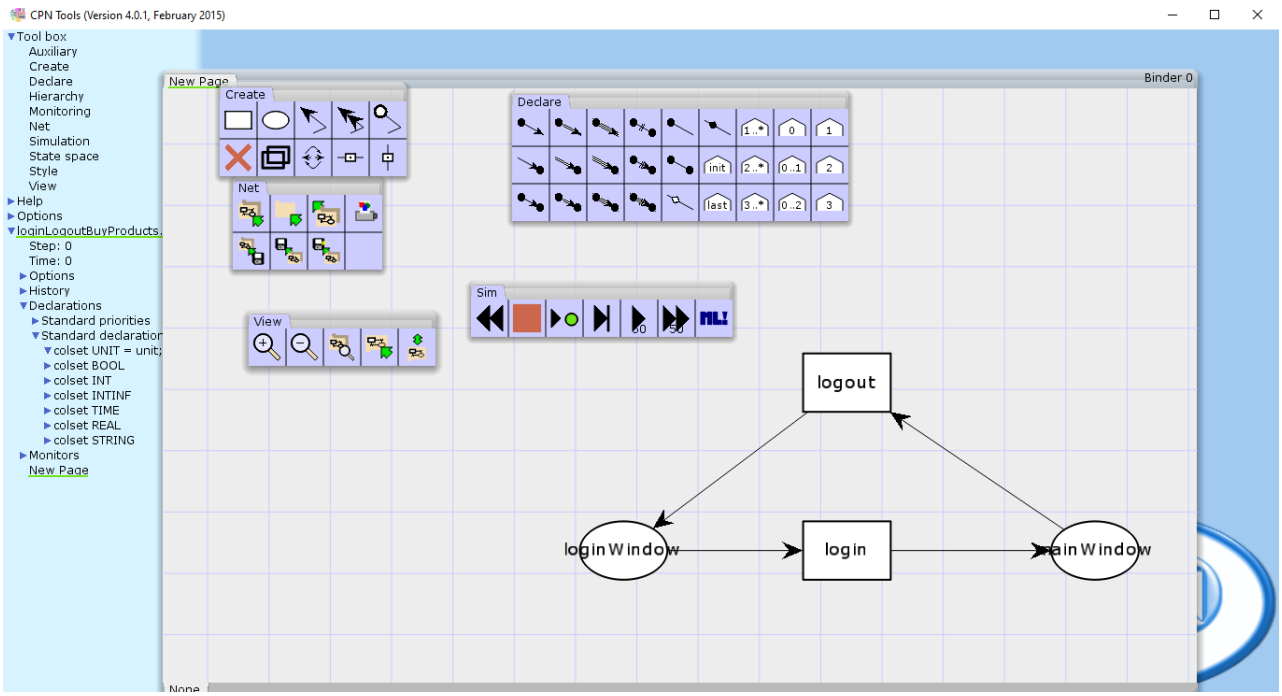

Рисунок 21 - Рабочее окно программы CPN Tools

Система тестирования не требует использования именно этой программы для создания сети Петри ей лишь необходим валидный файл формата .cpn.

Модуль симуляции принимает на вход XML файл формата .cpn который содержит описание сети Петри. При запуске модуля симуляции он будет моделировать поведение сети Петри и отправлять управляющие команды модулю тестирования.

Модуль тестирования содержит в себе различные модули интеграции с системой. С помощь этих модулей выполняется проверка системы, то есть выполнения неких действий в ней и проверка соответствия ожидаемого и получившегося состояния системы. Весь код, который пишет тестировщик записывается в модуле тестирования. Также модуль тестирования должен определять момент завершения тестирования, а также формировать отчет о неправильном поведении системы и вести журнал исполнения. В листинге 1 приведен пример на языке программирования Java демонстрирующий привязку тестирующего кода из модуля тестирования к модулю симуляции.

Листинг 1 – Пример привязки кода тестирования к симулятору сети Петри

```
    //Select model file
        PetriNetContext context = PnmlLoader.load(Paths.get("PATH"),
"PAGE_ID");
        //Install token
        Token<Integer> token = new TokenImpl<>();
        token.setColor("val");
        token.setData(123);
        context.addToken(Integer.class, "start", token);
        //Add checkers
        context.addPlaceStateChecker("start", () -> {
            System.out.println("Test state start");
            return true;
        });
        context.addTransitionAction("do", () -> {
            System.out.println("My action");
        });
        context.addPlaceStateChecker("state1", () -> {
            System.out.println("Test state state1");
            return true;
```

```
});
context.addPlaceStateChecker("state2", () -> {
    System.out.println("Test state state2");
    return true;
});
context.addPlaceStateChecker("state3", () -> {
    System.out.println("Test state state3");
    return true;
});
context.addTransitionAction("doUp", () -> {
    System.out.println("My action doUp");
});
context.addTransitionAction("doDown", () -> {
    System.out.println("My action doDown");
});
context.addPlaceStateChecker("finish", () -> {
    System.out.println("Test state finish");
    return true;
});
//start engine
Engine engine = new Engine(context);
engine.start(10);
```

Благодаря такой схеме проектирования системы тестирования она остается очень гибкой с возможность заменять модули под необходимые конфигурации.

Код таких сущностей, как токен, ребро, переход и позиция представлены в приложении а.

Ключевым звеном всей системы является компонент манипулирования симуляцией. Именно в нем задаются правила работы разработанной модели. В листинге 2 приведен пример компонента способного симулировать простые сети Петри.

Листинг 2 – Пример простого компонента симуляции сети Петри

```
public class Engine {
    private long ticks = 0;
    private long maxTicks = 0;
    private Map<String, Place> placeMap;
    private Map<String, Transition> transitionMap;
    private Map<String, Token> tokenMap;
    private Random random;
    public Engine(PetriNetContext context) {
        placeMap = context.getPlaceMap();
        transitionMap = context.getTransitionMap();
        tokenMap = context.getTokenMap();
        random = new Random(System.currentTimeMillis());
    }
    public void start(int maxTicks) {
        this.maxTicks = maxTicks;
        loop();
    }
    private void loop() {
        testStartState();
        while (ticks < maxTicks) {
            List<Transition> transitions = getAvailableTransition();
            if (transitions.isEmpty()) {
                //check paths
                return;
            }
            Transition transition = selectRandom(transitions);
            move(transition);
            checkPlaceStates(transition);
            ticks++;
        }
    }
```

```
    private void testStartState() {
        for (Place place: placeMap.values()) {
            if (!place.getTokenContext().isEmpty() && !place.check())
{
                throw   new   IllegalStateException("Illegal   check
state!");
            }         }     }
    private void checkPlaceStates(Transition transition) {
        for (Edge edge: transition.getOutputEdges()) {
            if (!edge.getPlace().check()) {
                throw   new   IllegalStateException("Illegal   check
state!");
            }         }     }
    private void move(Transition transition) {
        List<Edge> edges = transition.getInputEdges();
        Token                          token                          =
edges.get(0).getPlace().getTokenContext().getToken(Integer.class,
"val");
        removeTokens(edges);
        transition.execute();
        installTokens(transition.getOutputEdges(), token);
    }
    private void removeTokens(List<Edge> edges) {
        for (Edge edge: edges) {

edge.getPlace().getTokenContext().removeToken(Integer.class, "val");
        }     }
    private void installTokens(List<Edge> edges, Token token) {
        for (Edge edge: edges) {

edge.getPlace().getTokenContext().addToken(Integer.class,    "val",
token);
        }     }
```

```
    private List<Transition> getAvailableTransition() {
        List<Transition> res = new ArrayList<>();
        for (Transition transition: transitionMap.values()) {
            if (checkPlaces(transition)) {
                res.add(transition);
            }         }
        return res;
    }
    private Boolean checkPlaces(Transition transition) {
        for (Edge edge: transition.getInputEdges()) {
            Token<Integer>                  token                  =
edge.getPlace().getTokenContext().getToken(Integer.class, "val");
            if (token == null) {
                return false;
            }         }
        return true;
    }
    private <T> T selectRandom(List<T> container) {
        if (container.isEmpty()) {
            throw new IllegalArgumentException("Container is empty");
        }
        return      container.get(Math.abs(random.nextInt())      %
container.size());
    }
     private  static  void  dfs(Node  node,  List<PairNodeId>  path,
List<List<PairNodeId>> paths) {
        String nodeName = node.getNodeName();
        if (components.containsKey(nodeName)) {
            String id = getAttributeValue(node, "id");
            if (id == null && nodeName.equals("button")) {
                id = getAttributeValue(node, "action");
            }
            if (id != null) {
```

```
                PairNodeId pairNodeId = new PairNodeId();
                pairNodeId.id = id;
                pairNodeId.node = nodeName;
                path.add(pairNodeId);
                paths.add(path);
            }
            NodeList childList = node.getChildNodes();
            if (childList.getLength() != 0) {
                for (int i = 0; i < childList.getLength(); i++) {
                    dfs(childList.item(i),  new  ArrayList<>(path),
paths);
                }             }         }     } }
```

Как видно для хранения всего текущего состояния сети Петри система активно использует контекст (приложение б). Он выполнен независимо от всего модуля и доступ к нему возможен из любой точки симулятора. Благодаря такому решению появляется возможность расширять систему дополнительными обработчиками и наблюдателями состояния сети Петри. На этой идеи базируется интеграция тестирующей системы с другими системами. Подробнее об это говориться в следующем разделе.

На текущий момент система способна моделировать цветные сети Петри, но как отмечалось ранее система проектирования сетей не связана с кодом и проектировать такую сеть достаточно сложно. Поэтому дальнейшее развитие тестирующей системы будет нацелено на упрощение проектирования сетей Петри.

3.2 Интеграция симулятора сетей Петри с внешними системами

Цветные сети Петри (CPN) обеспечивают полезный формализм для описания параллельных систем, таких как сетевые протоколы и системы

документооборота. Инструменты CPN предоставляет среду для редактирования и моделирования моделей CPN, а также для проверки их правильности с помощью анализа пространства состояний [3]. Однако иногда этого не всегда бывает недостаточно.

Поскольку инструменты CPN по своей сути являются графическими, они не могут управляться внешними приложениями, поэтому трудно использовать инструменты CPN в настройках, которые выходят за рамки интерактивного использования одним пользователем. Такие случаи включают повторное моделирование на нескольких серверах в сети, описывающей сложную процедуру принятия решений в виде модели CPN в которой пользователям, необходимо устанавливать параметры модели с помощью пользовательского интерфейса [12].

Из-за архитектуры симулятора и инструмента пространства состояний в инструментах CPN также трудно реализовать новые методы анализа модели.

Инструменты CPN в основном состоят из двух компонентов графического редактора и симулятора. Редактор позволяет пользователям интерактивно строить модель CPN, которая передается симулятору, который проверяет ее на наличие синтаксических ошибок и генерирует специфичный для модели код для моделирования модели CPN. Редактор вызывает сгенерированный код симулятора и отображает результаты графически.

Редактор может загружать и сохранять модели в формате XML. Редактор накладывает большинство ограничений на использование инструментов CPN, упомянутых выше. Заменив редактор собственным приложением, можно снять ограничения, наложенные редактором.

Предложенное решение страдает от двух проблем, затрудняющих такое взаимодействие. Во-первых, используемый протокол, так как связь между редактором и симулятором является низкоуровневой и сложной в реализации. Во-вторых, симулятор CPN оптимизирован для моделирования и инкрементной

генерации кода, что затрудняет его использование для других целей. Для решения этой проблемы предлагается создать некий программный интерфейс между редактором и симулятором. Интерфейсы реализованы на языке программирования Java [25].

Интерфейс верхнего уровня состоит из объектно-ориентированного представления моделей CPN, возможности передачи этого представления в симулятор и выполнения моделирования и проверки текущее состояние в симуляторе. Кроме того, он включает в себя загрузчик, который может импортировать модели, созданные с помощью инструментов CPN. Интерфейс низкого уровня инкапсулирует структуры данных, используемые в симуляторе, и предоставляет интерфейс к модели CPN, облегчающий быстрое моделирование, полезный для эффективного анализа и других приложений, выполняющих переходы с небольшим или полным отсутствием взаимодействия с пользователем. Другими словами, этот интерфейс организует единую точку управления симулятором для того, чтобы дать пользователю возможность настраивать его поведение.

В качестве редактора используется CPN Tools, как наиболее удобный инструмент, но существует возможность использовать любой другой редактор.

CPN Tools - это инструмент для редактирования, моделирования и анализа цветных сетей Петри. Графический интерфейс пользователя основан на передовых методах взаимодействия. Средства обратной связи предоставляют контекстные сообщения об ошибках и указывают отношения зависимости между элементами сети. Инструмент имеет инкрементную проверку синтаксиса и генерацию кода, которые происходят во время построения сети. Быстрый симулятор эффективно справляется как с несвязанными, так и с синхронизированными сетями. Полные и частичные пространства состояний могут быть сгенерированы и проанализированы, а стандартный отчет о пространстве состояний содержит такую информацию, как свойства

ограниченности и свойства живучести. Функциональные возможности механизма моделирования и возможностей пространства состояний аналогичны соответствующим компонентам Design/CPN, который является широко распространенным инструментом для цветных сетей Петри [13].

3.3 Интерфейс низкого уровня

Целью интерфейса низкого уровня является обеспечение эффективного доступа к симулятору CPN, в частности с целью внедрения новых и более эффективных методов анализа. Чтобы поддержать это, интерфейс предоставляет доступ к состоянию модели CPN и выполнения разрешенных переходов.

Интерфейс разработан с учетом анализа пространства состояний, но может использоваться и для других целей. Он предназначен для того, чтобы быть независимым от фактического формализма, что позволяет разрабатывать инструменты, которые задают свое формальное описание модели.

Интерфейс нижнего уровня предоставляет следующие точки взаимодействия:

- getInitialStates - возвращает список начальных состояний.
- nextStates - принимает состояние и событие и возвращает итоговые состояния.
- executeSequence – выполняет выбранное количество итераций сети Петри.

В дополнение к реализации сигнатуры модели интерфейс низкого уровня также предоставляет различные служебные функции. Кроме того, предусмотрен интерфейс для проверки модели, позволяющий пользователям создавать функции, специфичные для заданной формальной модели.

Чтобы создать экземпляр интерфейса для моделей CPN, необходимо определить типы состояние и событие. Чтобы иметь совместимость с пространством состояний CPN Tools, определим структуру *Token* с типами

данных и функциями для манипулирования состояниями. тип отражал иерархическую структуру модели CPN.

Состояния определяются типом данных. Оно отображает текущее положение фишек в сети, а также её раскраску.

События определяются типом данных. Определим конструктор для каждого перехода, названного в честь модуля, на котором он находится, и имени перехода. Каждый конструктор представляет собой пару номера экземпляра и записи со всеми переменными перехода [29].

3.4 Интерфейс высокого уровня

Интерфейс высокого уровня предоставляет из себя надстройку над интерфейсом низкого уровня и является основным механизмом интеграции с CPN системой.

Предоставляется доступ к следующим инструментам:

– Информация о всех состояниях (цвет, число токенов, описание),
– Информация о всех переходах (команда, требуемое число токенов),
– Информация о всех токенах (цвет, расположение),
– Информация о *n* предыдущих переходах,
– Управление параметрами симулятора,
– Управление ходом выполнения симулятора,
– Получение списка возможных состояний системы.

Благодаря этому интерфейсу открывается возможность наблюдать прямо в редакторе то, как происходит процесс тестирования. Также нет необходимости использовать какой-то определенный редактор. Благодаря этому интерфейсу можно интегрировать симулятор с любой другой системой либо разработать собственный интерфейс мониторинга (рисунок 22).

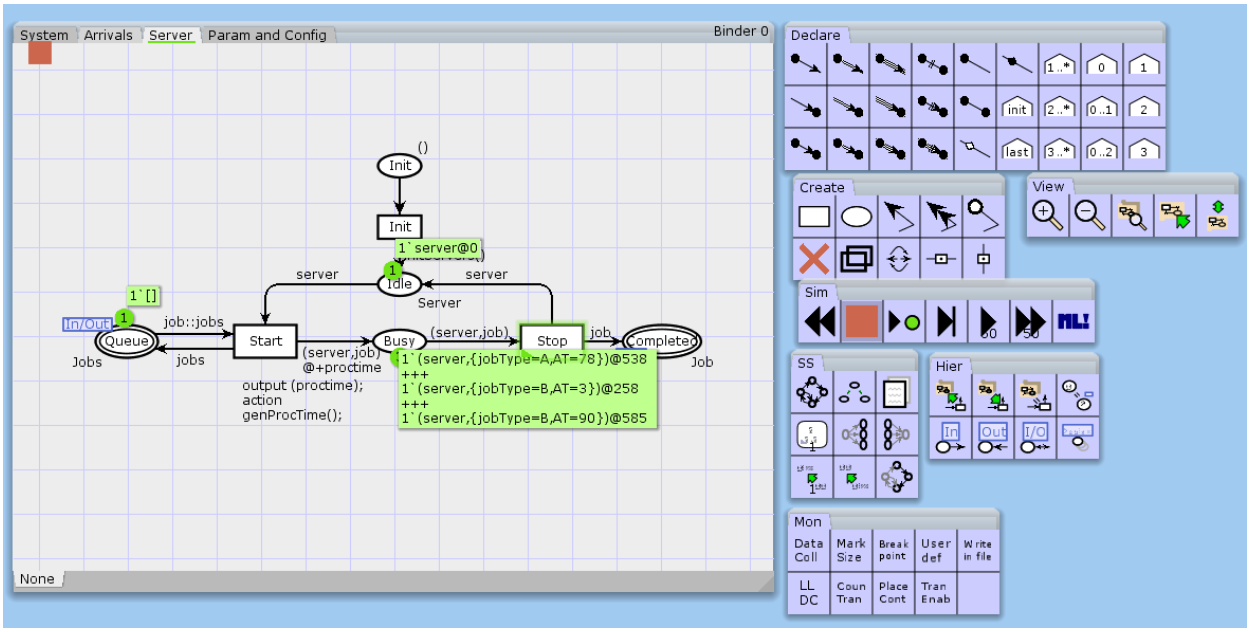

Рисунок 22 – Пример отображения состояния реальной системы в редакторе CPN Tools

Реализация протокола. Редактор инструментов CPN взаимодействует с симулятором с помощью проприетарного протокола, который представляет собой реализацию системы удаленного вызова процедур (RPC). Обмен данными происходит по протоколу TCP/IP в формате JSON.

Многие приложения могут извлечь выгоду из тесной интеграции с симулятором CPN. За счет того, что интерфейс написан на языке программирования Java открываются большие возможности использования готовых фреймворков, разработанных для этого языка. Также довольно легко найти разработчиков, владеющих данным языком.

Глава 4 Практическое применение разработанной методики тестирования

4.1 Тестирование на реальной системе

В первую очередь рассмотрим предполагаемый сценарий использования системы. Имеется некий WEB ресурс доступ, к котором осуществляется через заполнение формы с логином и паролем. Необходимо протестировать выполняется ли вход в систему при заполнении верных данных в форму и выполняется ли выход из неё.

Для выполнения тестирования этого функционала необходимо составить сеть Петри. Она будет состоять из двух состояний и двух переходов. Первое состояние – это состояние, когда у пользователя открыто окно входа в систему и ещё никакие данные не введены. Второе состояние – это состояние, когда пользователь вошел в систему и наблюдает её главное окно. Переход между первым состоянием – это выполнение действия по заполнению данных и нажатие кнопки входа в систему. Переход между вторым состоянием и первым это нажатие кнопки выхода из системы. Изображение получившейся сети Петри изображено на рисунке 23.

Сама сеть Петри не подразумевает под собой привязку к какой-либо реализации. Это просто описание абстрактных действий и состояний.

Далее нам необходимо загрузить сеть в модуль тестирования и привязать для каждого перехода и состояния код тестирования. Так как производится тестирование WEB-приложения мы будем использовать Selenium WebDriver. Он будет имитировать поведение пользователя и проверять текущее состояние страницы.

Следующим этапом будет написание кода тестирования и привязки его к модели.

Далее необходимо выбрать условия прекращения тестирования и можно запускать выполнение теста. Тест либо выполниться успешно, либо выведется отчет о том какая последовательность действий привела к ошибке.

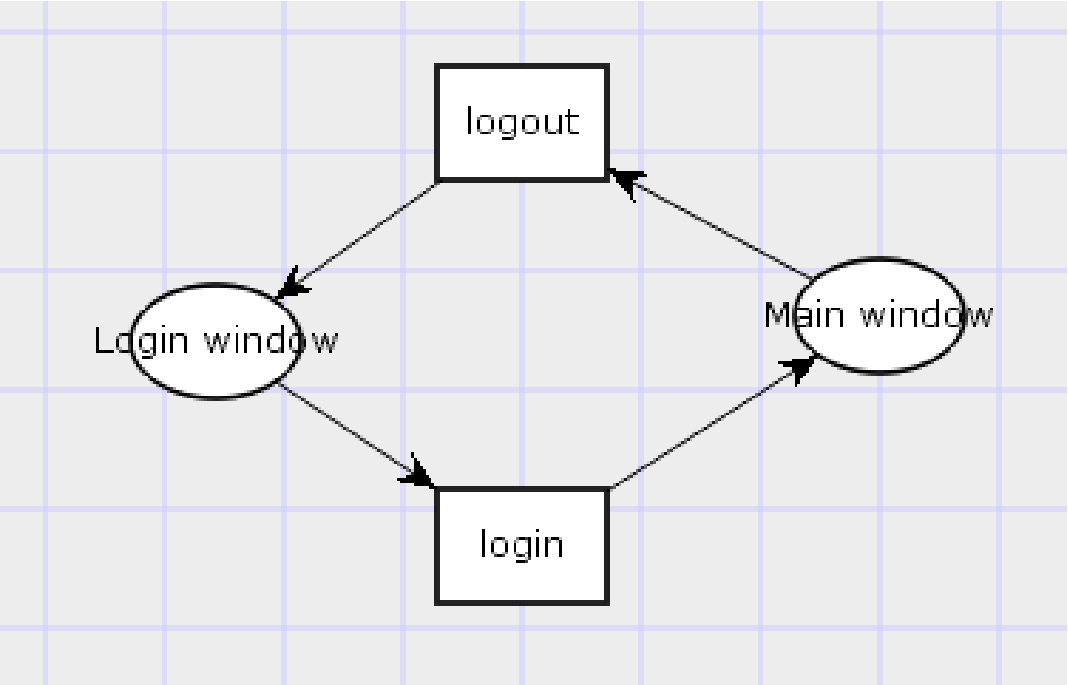

Рисунок 23 - Сеть Петри

В дальнейшем если потребуется расширить покрываемость системы тестами, достаточно будет изменить сеть Петри и привязать к новым состояниям и переходам новый код. Весь ранее написанный код останется и не претерпит изменений. Приведем пример.

В нашей системе необходимо теперь проверять то, что при не правильном вводе данных в систему выводится сообщение об ошибке и пользователь не попадает на экран. Для этого необходимо взять текущую сеть Петри и добавить состояние, в котором мы наблюдаем на экране сообщение об ошибке и два перехода. Один переход – это ввод неверных данных, а второй переход – это закрытие окна сообщения. Код во второй переход даже можно не писать, если

сообщение появляется в виде уведомления и не требует действия от пользователя. На рисунке 24 изображена обновленная сеть Петри.

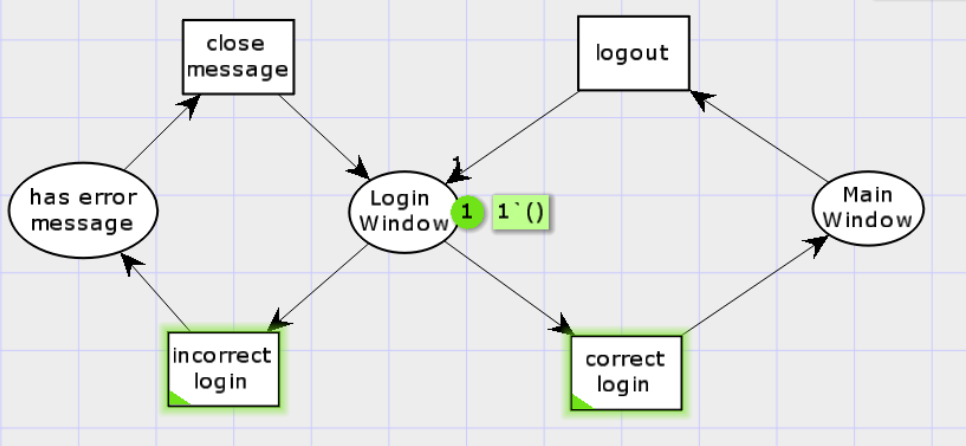

Рисунок 24 – Обновленная сеть Петри

Такой подход значительно сокращает время на разработку и не требует от разработчика дополнительного времени на составления нового сценария. За счет того, что код привязывается к модели поиск кода, который требуется обновить в случае изменения реализации достаточно прост. Блоки тестирования четко структурированы и имеют разграничения на действия и проверку.

4.2 Измерение стоимости тестирования

Измерение стоимости тестирования - важный шаг к оправданию любой инициативы по тестированию программного обеспечения. Все затраты на тестирование оправдываются путем сравнения полученных выгод с затратами. Преимущества могут заключаться в измерении качества и предотвращении отказов или более раннем обнаружении ошибок и т. д.

Прежде чем менеджеры по тестированию начинают оценивать стоимость затрат на тестирование, они должные ответить на следующие вопросы:

– Сколько нужно потратить на тестирование программного обеспечения?

– Сколько следует выделить физических ресурсов для тестирования?

– Какие уровни тестирования программного обеспечения разумны?

– В какой момент стоимость тестирования программного обеспечения становится слишком высокой?

– В какой момент стоимость тестирования программного обеспечения становятся слишком низкими?

Ответ на поставленные выше вопросы заключается в определении того, сколько сейчас организация тратит на тестирование программного обеспечения. Дело в том, что у большинства менеджеров такой информации нет. Они думают, что знают, потому что система управления проектами сообщает о расходах по каждой фазе проекта. Такие системы детализируют расходы, часто по отдельным задачам, для каждой фазы жизненного цикла разработки. Однако стоимость этапа тестирования - это не стоимость тестирования проекта.

Некоторая работа по тестированию выполняется на других этапах таких, как тестирование проекта, модульное тестирование и т. д. Много работы выполняется во время тестирования системы, которые не является тестированием, например, документация, отладка, анализ и устранение недостатков. Тщательный анализ показывает, что фактические затраты на тестирование программного обеспечения обычно составляют от 15 до 25 процентов от общей стоимости проекта. Многие менеджеры по тестированию считают, что затраты на тестирование программного обеспечения намного выше и составляют 50 процентов от стоимости проекта. Они ошибочно рассматривают все расходы по проекту после программирования как затраты на тестирование. Хотя верно, что для многих проектов хороший способ оценить окончательную общую стоимость - это взять фактические расходы на этапе программирования и удвоить их, но все

же не все затраты после программирования является результатом тестирования программного обеспечения.

Определение затрат на тестирование программного обеспечения - важный первый шаг к планированию любых инициатив по улучшению и оправданию инвестиций. Оценка суммы, затрачиваемой на тестирование и измерение качества, а также стоимости доработки или исправлений имеет фундаментальное значение. Большинство организаций тратят слишком много времени, беспокоясь о том, как подробно рассчитать эти затраты. Оценивать затраты на фактическую стоимость тестирования необходимо консервативно не боясь завысить стоимость принятия решения о необходимости тестирования ПО, так как если посчитать убытки от отказа, они будут куда больше [16, 21].

В процентах от общего объема разработки прямые затраты на тестирование программного обеспечения приблизятся к 25 процентам. Затраты на косвенное тестирование или затраты на плохое тестирование обычно как минимум в два раза превышают прямые затраты и могут быть значительно выше. Общая стоимость тестирования программного обеспечения в большинстве организаций достаточно велика, чтобы привлечь внимание практически любого менеджера. Как только будет оценена стоимость плохого тестирования, проблемы с обоснованием исчезнут. Руководству следует ожидать окупаемости инвестиций в улучшение тестирования. Выгоды можно отследить, а результаты станут заметны. Стоимость тестирования программного обеспечения включает в себя прямые затраты на измерения плюс косвенные затраты на отказы и исправления [30].

4.3 Анализ полученных результатов

Тестирование проводилось на четырех различных бизнес-процессах с привязанными к ним системами. На рисунке 25 изображен первый бизнес-процесс оплаты билета на самолет и бронирование отеля.

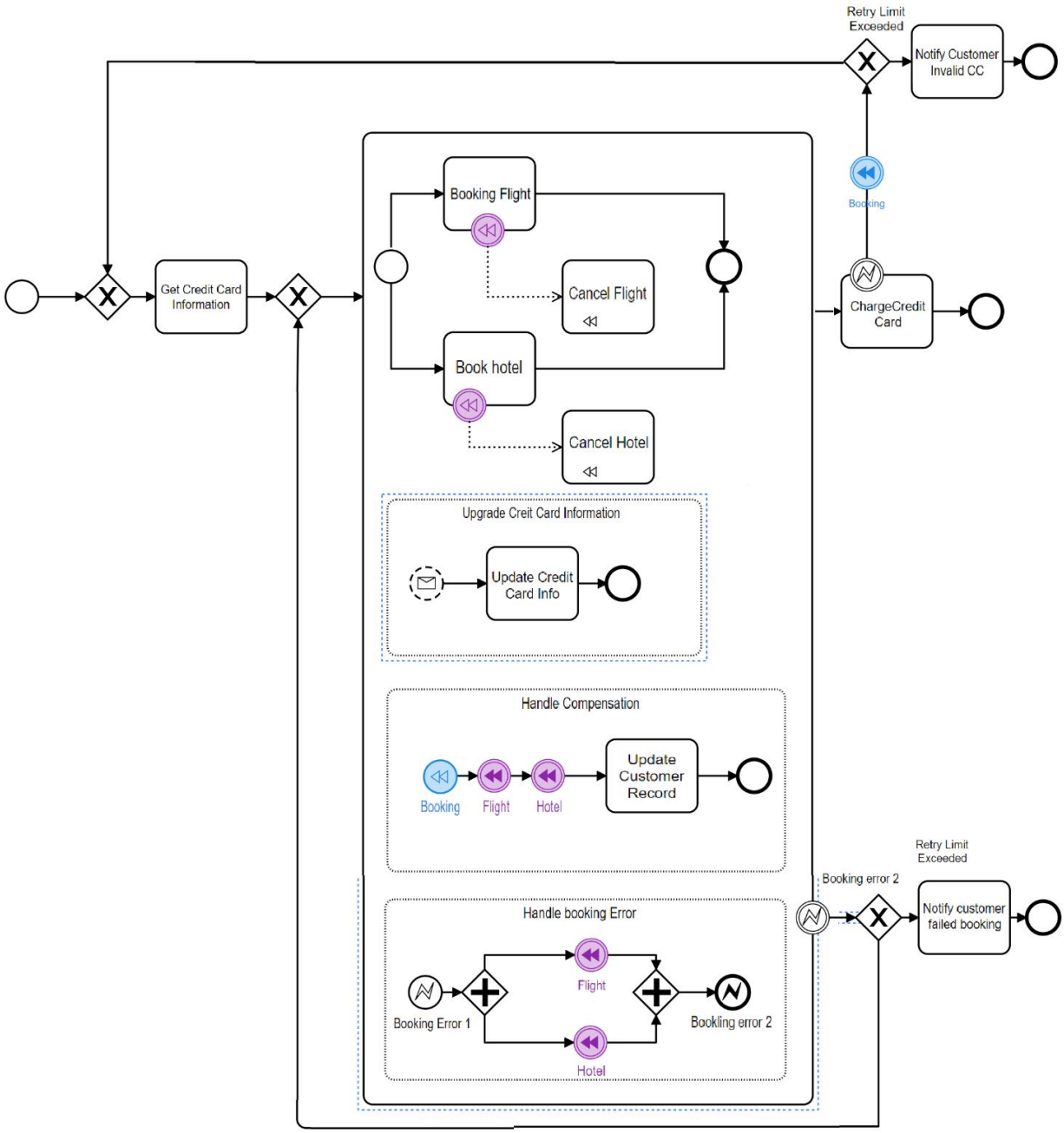

Рисунок 25 – BPMN модель покупки авиабилета и бронирование отеля

Вторая тестируемая система – это процесс подготовки товара к отгрузке со склада (рисунок 26).

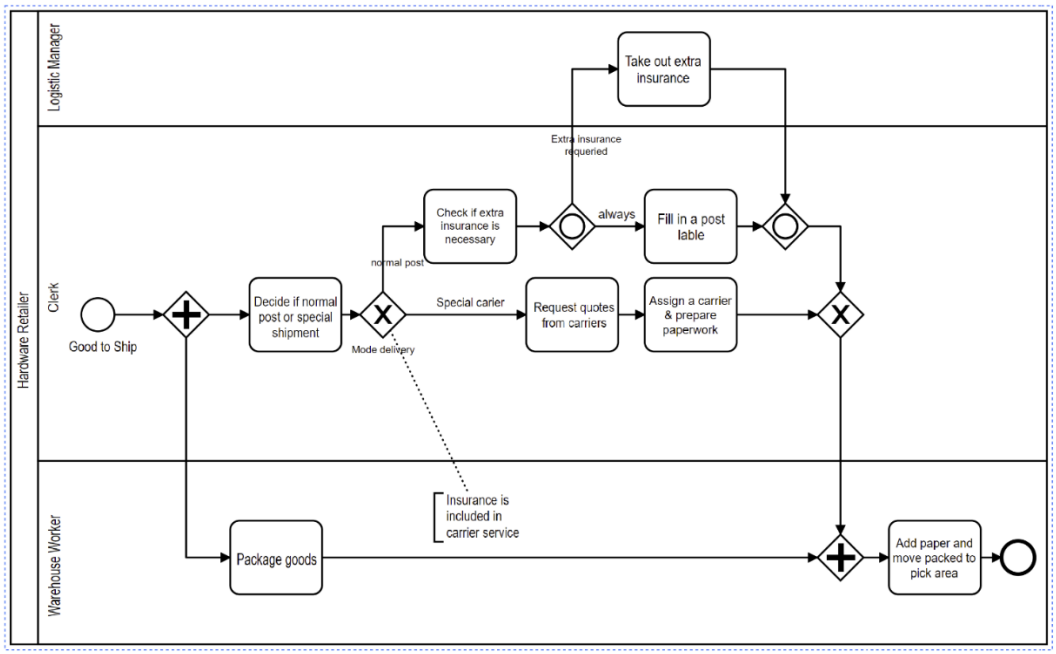

Рисунок 26 – BPMN модель процесса отгрузки товара со склада

Третья тестируемая система – это процесса заказа пиццы. На рисунке 27 изображена схема этого процесса.

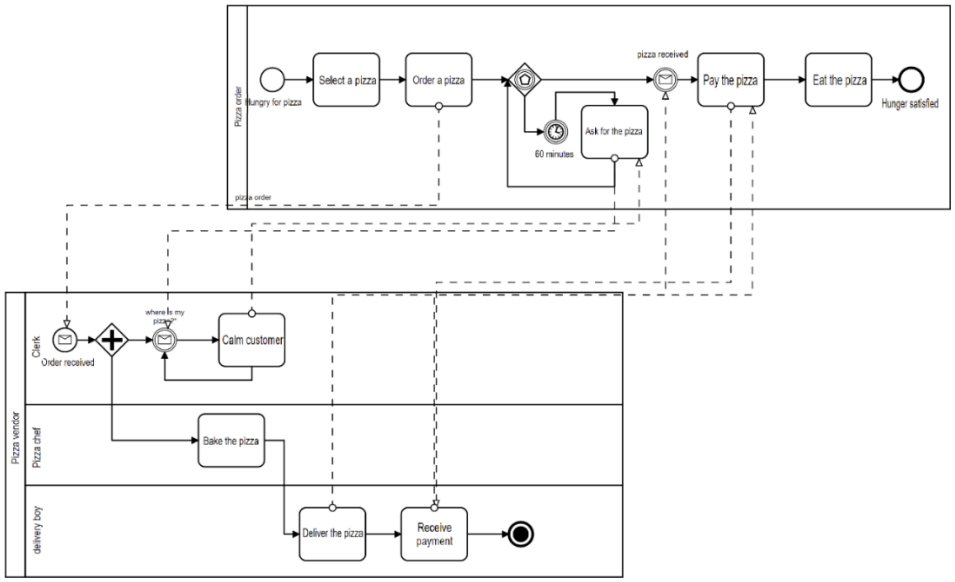

Рисунок 27 – BPMN модель процесса заказа пиццы

Четвертая тестируемая система – это система, которая позволяет купить и оформить доставку товара (рисунок 28).

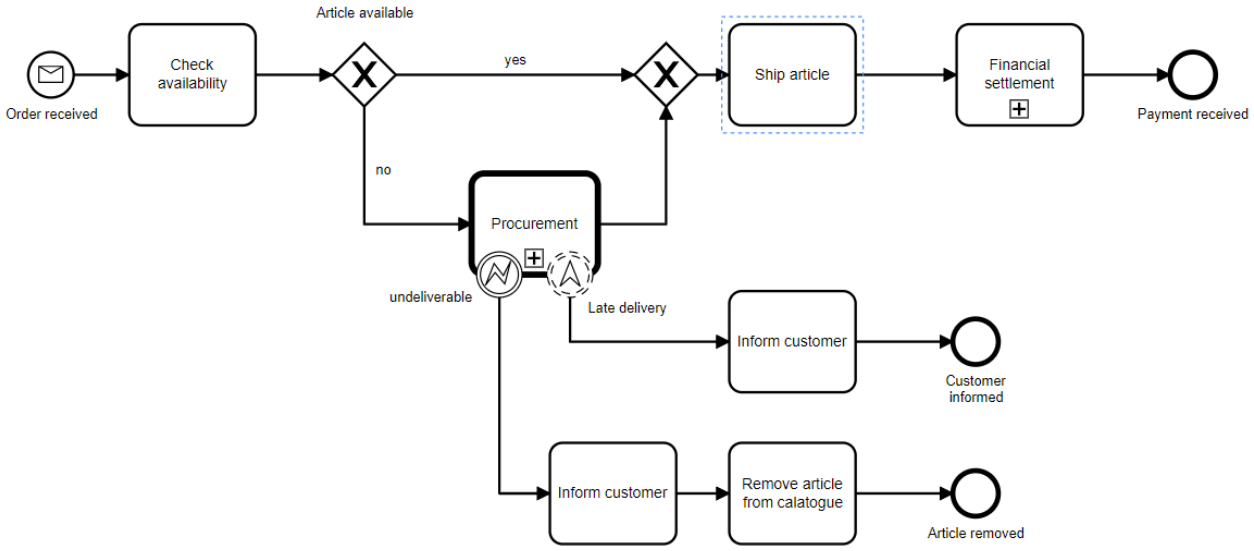

Рисунок 28 – BPMN модель процесса заказа товара

Большинство из этих моделей имеют строго прямое продвижение по бизнес-процессу, а, следовательно, имеют ограниченное количество путей в графе состояний. Такие системы можно полностью покрыть ручными тестами, но в случае изменения бизнес-процесса стоимость покрытия нового сценария будет существенно дороже чем при использовании тестов на основе модели сети Петри [13]. Так для сети Петри потребуется некое константное время для описания нового состояние и перехода, а для ручных тестов все будет зависеть числа связанных с новым состоянием тестов. Так на рисунке 29 приведена средняя стоимость покрытия тестами нового узла для каждой системы с ограниченном числом тестовых сценариев.

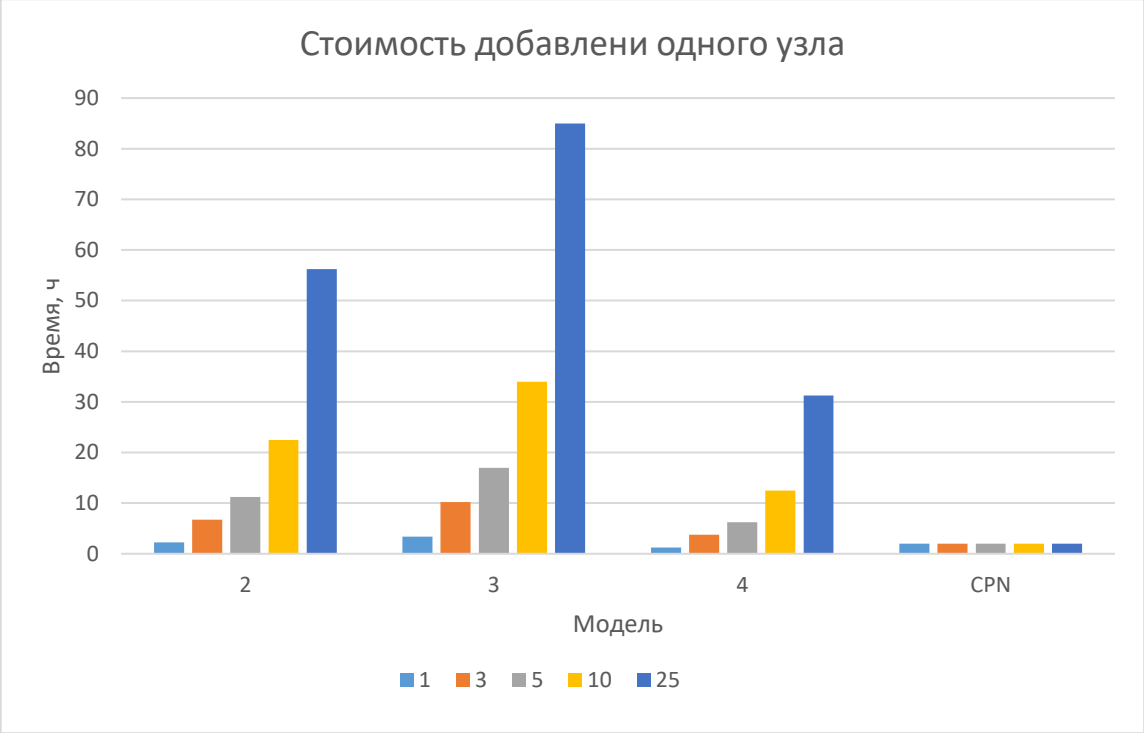

Рисунок 29 – Стоимость добавления одного узла в BPMN модель

Как видно при изменении бизнес-процесса стоимость внесения изменений в тестовые сценарии существенно возрастает для ручных тестов, но для модели на основе сети Петри – это значение постоянно.

Рассмотрим первую модель, где число сценариев имеет неограниченное количество. Такие процессы самые распространенные. На рисунке 30 приведен расчёт того сколько необходимо времени на написание определенного числа тестов.

Как видно на рисунке 30 время на разработку сети Петри имеет некоторое константное значение, и оно существенно выше времени на разработку небольшого числа коротких тестов. В таких случаях использование сетей Петри можно только оправдать возможным изменением бизнес-процесса. В ином же случае лучше будет обойтись более дешевыми ручными тестами. С другой стороны, начиная с определенного момента можно заметить, что с определенного

момента сети Петри дают ощутимую выгоду, так как они не зависят от длины тестового сценария и не зависят от количества необходимых тестов.

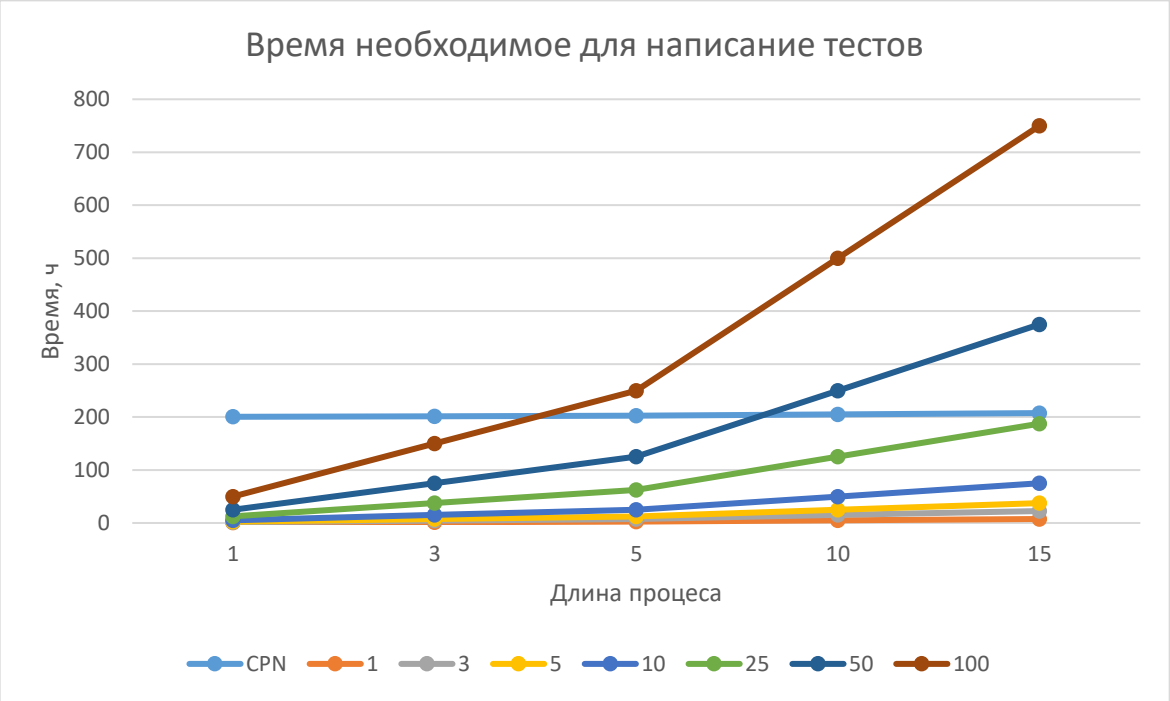

Рисунок 30 – Зависимость времени от количества узлов в тестовом сценарии

Также следует не забывать о том, что сети Петри позволяют находить такие сценарии, о которых специалист по тестированию мог и не задумываться. Также сети Петри не ограничивают в написании ручных сценарием. Потому что выбор переход в сети Петри определяется случайно, но его можно четко настроить.

Из всего этого можно сделать вывод, что применение сетей Петри в больших и часто изменяющихся системах – это отличная возможность поддерживать систему в рабочем состоянии и позволяет существенно сэкономить на ресурсах.

Заключение

Сети Петри являются удобным инструментом для моделирования любых асинхронных процессов. В частности, для моделирования бизнес-процессов. Такая модель поддается математическому анализу и позволяет находить в ней ошибки проектирования задолго до реализации этой модели. Так на основе сетей Петри была построена модель тестирующей системы, которая выполняет проверку соответствия модели, описанной сетями Петри и реальной системе. Сети Петри очень гибкий инструмент и как видно позволяет добавлять свои правила обработки переходов. Это позволяет постоянно модифицировать и подстраивать тестирующую систему для ускорения выполнения тестов и повышения покрываемости тестами системы.

Предложенная модель позволяет не только в значительной степени снизить затраты на тестирование больших систем BPA, но и сделать его в принципе возможным. В результате проведенного анализа затрат, в среднем применение сетей Петри для тестирования BPA систем может сократить затраты на тестирования в 2 – 2.5 раза. Такая экономия заключается в том, что нет необходимости тратить время специалистов по тестированию на поддержание, изменение и актуализацию тестовых сценариев. В тестирующей системе на основе сетей Петри тестовые сценарии отсутствуют в принципе. Они генерируются из модели автоматически. Следовательно, ресурсы специалистов по тестированию сконцентрированы на поддержании математической модели тестируемой системы в актуальном состоянии.

Благодаря подходу MBT тестирования, сети Петри открывают возможность для создания прогроммного обеспечения по контролю модели системы. Так в текущей работе была описана модель взаимодействия между моделью тестируемой системы и самой тестирующей системой. Такое взаимодействие позволяет наблюдать отображение реальной системы на её модели и наоборот

контролировать реальную систему через её модель. Обнаружение ошибок при таком взаимодействии становится более оперативным. Современные системы являются параллельными и в них могут возникать такие ситуации как взаимные блокировки и гонки за ресурсы. Сети Петри предназначены для моделирования таких систем и благодаря отображению реальной системы на модель, можно наблюдать в реальном времени возникшие проблемы в системе и анализировать их возникновение.

В дальнейшем разработанная модель может иметь множество модификаций. Так может быть разработана модель, позволяющая в автоматически генерировать сети Петри по уже имеющимся бизнес-процессам, описанных в нотации BPMN. Также при написании тестовых сценариев присутствует достаточное большое количество шаблонного кода и имеется возможность сделать его автоматическую генерацию. Как вариант решения этой проблемы использовать подход метапрограммирования. Одним из языков поддерживающих метапрограммирование является Groovy. С его помощью можно писать тесты для тестирующей системы без их компиляции, а также это позволит создавать некие шаблоны тестов и на их основе выполнять генерацию тестовых условий. Следует одной весьма значимой модернизацией является улучшение интерфейса взаимодействия с тестирующей системой. Улучшив понимание и удобство управления системой, можно привлекать менее квалифицированных специалистов по тестированию, что ещё позволит уменьшить затраты на тестирование.

Список используемой литературы

1. Аалст Вил ван дер, Хей ван Кейс. Управление потоками работ: модели, методы и системы. - М.: Физматлит, 2007. - 320 с.
2. Башкин В.А., Ломазова И.А. Подобие обобщенных ресурсов в сетях Петри // Труды МСО-2005. - М.: МГУ, 2005. - С. 330-336.
3. Башкин В.А., Ломазова И.А. Эквивалентность ресурсов в сетях Петри. - М.: Научный мир, 2008. - 208 с.
4. Доррер М.Г. Алгоритм преобразования моделей бизнес-процессов в одноцветные сети Петри // Моделирование и анализ информационных систем. - 2010. - №2. - С. 5-16.
5. Захаров В.А. Проверка эквивалентности программ при помощи двухленточных автоматов // Кибернетика и системный анализ. - 2010. - №4. - С. 39-48.
6. Калянов Г.Н. Теория и практика реорганизации бизнес-процессов. - М.: СИНТЕГ, 2000. - 212 с.
7. Коннов И. В. Применение ослабленных отношений симуляции в методе сетевых инвариантов для верификации параметризованных асинхронных моделей // Моделирование и анализ информационных систем. - 2010. - №3. - С. 3-13.
8. Котов В.Е. Сети Петри. - М.: Наука, 1984. - 160 с.
9. Ломазова И.А. Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой. - М.: Научный мир, 2004. - 208 с.
10. Методология разработки тестовых случаев на основе сценариев использования [Электронный ресурс] // software-testing.ru URL: https://software-testing.ru/library/5-testing/78-2008-09-29-07-33-51 (дата обращения: 12.10.2019).

11. Орел А.А. Шаблоны проектирования бизнес-процессов с помощью сетей Петри // Математика. Механика. - 2009. - №11. - С. 45-48.

12. Рябухин С.И. Вестник НГУ. Серия: Информационные технологии // Применение сетей Петри для моделирования событийно-процессных цепей и построения структур базы данных. - 2013. - №4. - С. 92-101.

13. Сорокин В.Е. Применение иерархических сетей Петри для моделирования телекоммуникационных протоколов // Сборник научных трудов Донецкого института железнодорожного транспорта. - Донецк: Донецкий институт железнодорожного транспорта, 2007. - С. 91-96.

14. Сэм Канер, Джек Фолк, Енг Кек Нгуен Тестирование программного обеспечения. - Киев: Диа Софт, 2001. - 544 с.

15. Тестирование. Фундаментальная теория [Электронный ресурс] // Habr URL: https://habr.com/ru/post/279535/ (дата обращения: 11.10.2019).

16. Управление бизнес-процессами. Практическое руководство по успешной реализации проектов [Электронный ресурс] // ВикиЧтение URL: https://econ.wikireading.ru/63736/ (дата обращения: 23.10.2019).

17. Шеер А.В. Бизнес-процессы. Основные понятия. Теория. Методы: Пер. с англ.. - 2-е изд. - М.: Просветитель, 1999. - 152 с.

18. Bashkin V. A. Approximating bisimulation in one-counter nets // Proc. of “Program semantics, specification and verification: theory and applications. - St.Petersburg: 2011. - С. 10-17.

19. Hesuan Hu, MengChu Zhou, ZhiWu Li Algebraic Synthesis of Timed Supervisor for Automated Manufacturing Systems Using Petri Nets // Automation Science and Engineering IEEE Transactions. - 2010. - №3. - С. 549-557.

20. How do we measure the Costs of Software Testing [Электронный ресурс] // Software Testing Genius URL: https://www.softwaretestinggenius.com/how-do-we-measure-the-costs-of-software-testing/ (дата обращения: 30.05.2021).

21. Hyun-Jung Kim, Jun-Ho Lee, Tae-Eog Lee Time-Feasible Reachability Tree for Noncyclic Scheduling of Timed Petri Nets // Automation Science and Engineering IEEE Transactions. - 2015. - №3. - С. 1007-1016.

22. Javier Esparza, Martin Leucker, Maximilian Schlund Learning Workflow Petri Nets // Fundamenta Informaticae. - 2011. - №113. - С. 1-24.

23. Juan-Ignacio Latorre-Biel and Emilio Jimenez-Macias Petri Net Models Optimized for Simulation, Simulation Modelling Practice and Theory. [Электронный ресурс] // IntechOpen URL: https://www.intechopen.com/books/simulation-modelling-practice-and-theory/petri-net-models-optimized-for-simulation (дата обращения: 07.09.2019).

24. Marius Kloetzer, Cristian Mahulea, Calin Belta, Manuel Silva An Automated Framework for Formal Verification of Timed Continuous Petri Nets // Industrial Informatics IEEE Transactions. - 2010. - №3. - С. 460-471.

25. Modeling Workflow with Petri-Nets. [Электронный ресурс] // Universita di Pisa URL: http://pages.di.unipi.it/ferrari/CORSI/SISD/Lezioni/WFModel.pdf (дата обращения: 09.09.2019).

26. N. Hagge B. Wagner Java code patterns for Petri net based behavioral models // Industrial Informatics. - Perth: 2005. - С. 450-455.

27. Prasun Hajra, Ranjan Dasgupta Modelling of a Multi-Track Railway Level Crossing System Using Timed Petri Net // World Academy of Science, Engineering and Technology. - 2012. - №11. - С. 1426-1432.

28. Remigiusz Wiśniewski, Andrei Karatkevich, Marian Adamski, Anikó Costa, Luís Gomes Prototyping of Concurrent Control Systems With Application of Petri Nets and Comparability Graphs // Control Systems Technology IEEE Transactions. - 2018. - №2. - С. 575-586.

29. Test Maturity Model: как тестировщику оценить проект и спланировать процессы [Электронный ресурс] // Habr URL: https://habr.com/ru/company/provectus/blog/448458/ (дата обращения: 20.10.2019).

30. Wei Zheng, J.R. Mueller, R. Slovak, E. Schnieder Function modeling and risk analysis of automated level crossing based on national statistical data // Proc. Informatics 2nd in Control, Automation and Robotics (CAR). - Wuhan: IEEE, 2010. - C. 281-284.

Приложение А

Базовые сущности симулятора сетей Петри

```
public class EdgeImpl implements Edge {

    private TypeEdge typeEdge;

    private Place place;

    private Transition transition;

    private Supplier<Boolean> condition = () -> true;

    public EdgeImpl(TypeEdge typeEdge) {
        this.typeEdge = typeEdge;
    }

    public EdgeImpl(TypeEdge typeEdge, Supplier<Boolean> condition)
{
        this.typeEdge = typeEdge;
        this.condition = condition;
    }

    @Override
    public void setCondition(Supplier<Boolean> condition) {
        this.condition = condition;
    }

    @Override
    public boolean getCondition() {
        return condition.get();
    }

    @Override
    public Place getPlace() {
        return place;
    }

    @Override
    public void setPlace(Place place) {
        this.place = place;
    }

    @Override
    public Transition getTransition() {
        return transition;
    }

    @Override
```

```
    public void setTransition(Transition transition) {
        this.transition = transition;
    }

    @Override
    public TypeEdge getTypeEdge() {
        return typeEdge;
    }
}

public class PlaceImpl implements Place {

    private String name;

    private List<Edge> outputEdges = new ArrayList<>();

    private Supplier<Boolean> checkState = () -> true;

    private TokenContext tokenContext = new TokenContextImpl();

    private double probability = 1.0;

    @Override
    public void setCheckState(Supplier<Boolean> checkState) {
        this.checkState = checkState;
    }

    @Override
    public Boolean check() {
        return checkState.get();
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public TokenContext getTokenContext() {
        return tokenContext;
    }

    @Override
```

```
    public void setTokenContext(TokenContext tokenContext) {
        this.tokenContext = tokenContext;
    }

    @Override
    public List<Edge> getOutputEdges() {
        return outputEdges;
    }

    @Override
    public void setOutputEdges(List<Edge> outputEdges) {
        this.outputEdges = outputEdges;
    }

    @Override
    public void setProbability(double probability) {
        this.probability = probability;
    }

    @Override
    public double getProbability() {
        return probability;
    }
}

public class PlaceImpl implements Place {

    private String name;

    private List<Edge> outputEdges = new ArrayList<>();

    private Supplier<Boolean> checkState = () -> true;

    private TokenContext tokenContext = new TokenContextImpl();

    private double probability = 1.0;

    @Override
    public void setCheckState(Supplier<Boolean> checkState) {
        this.checkState = checkState;
    }

    @Override
    public Boolean check() {
        return checkState.get();
    }

    @Override
```

```
    public String getName() {
        return name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public TokenContext getTokenContext() {
        return tokenContext;
    }

    @Override
    public void setTokenContext(TokenContext tokenContext) {
        this.tokenContext = tokenContext;
    }

    @Override
    public List<Edge> getOutputEdges() {
        return outputEdges;
    }

    @Override
    public void setOutputEdges(List<Edge> outputEdges) {
        this.outputEdges = outputEdges;
    }

    @Override
    public void setProbability(double probability) {
        this.probability = probability;
    }

    @Override
    public double getProbability() {
        return probability;
    }
}

public class TransitionImpl implements Transition {

    private String name;

    private List<Edge> inputEdges;

    private List<Edge> outputEdges;
```

```
    private Action action = () -> System.out.println("Empty action");

    public TransitionImpl() {
        inputEdges = new ArrayList<>();
        outputEdges = new ArrayList<>();
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void execute() {
        action.execute();
    }

    @Override
    public void setAction(Action action) {
        this.action = action;
    }

    @Override
    public void addInputEdge(Edge edge) {
        inputEdges.add(edge);
    }

    @Override
    public void addOutputEdge(Edge edge) {
        outputEdges.add(edge);
    }

    @Override
    public List<Edge> getInputEdges() {
        return inputEdges;
    }

    @Override
    public List<Edge> getOutputEdges() {
        return outputEdges;
    }
}
```

```
public class PetriNetMatrix {

//    transition ->
//    p
//    l
//    a
//    s
//    e
//    ||
//    \/

    private int[][] matrix;
    public PetriNetMatrix(int cntPlace, int cntTransition) {
        matrix = new int[cntPlace][cntTransition];
    }

    public void addOutputEdge(int placeIndex, int transitionIndex,
int rate) {
        if (rate <= 0) {
            throw new IllegalArgumentException("Rate should be
positive");
        }
        matrix[placeIndex][transitionIndex] -= rate;
    }

    public void addInputEdge(int placeIndex, int transitionIndex,
int rate) {
        if (rate <= 0) {
            throw new IllegalArgumentException("Rate should be
positive");
        }
        matrix[placeIndex][transitionIndex] += rate;
    }
}
```

Приложение Б

Контекст состояния сети Петри

```
public class TokenContainerImpl<T> implements TokenContainer<T> {

    private Map<String, Token<T>> container = new HashMap<>();

    @Override
    public Token<T> addToken(String color, Token<T> token) {
        return container.put(color, token);
    }

    @Override
    public Token<T> getToken(String color) {
        return container.get(color);
    }

    @Override
    public Token<T> removeToken(String color) {
        return container.remove(color);
    }

    @Override
    public int getSize() {
        return container.size();
    }

    @Override
    public boolean isEmpty() {
        return container.isEmpty();
    }
}

public class TokenContextImpl implements TokenContext {

    private  Map<Class, TokenContainer> context = new HashMap<>();

    @Override
    public <T> Token<T> addToken(Class<T> c, String color, Token<T>
token) {
        TokenContainer<T> container = getTokenContainer(c);
        return container.addToken(color, token);
    }

    @Override
    public <T> Token<T> getToken(Class<T> c, String color) {
        TokenContainer<T> container = getTokenContainer(c);
        return container.getToken(color);
    }
```

```
    @Override
    public <T> Token<T> removeToken(Class<T> c, String color) {
        TokenContainer<T> container = getTokenContainer(c);
        return container.removeToken(color);
    }

    @Override
    public <T> TokenContainer<T> getTokenContainer(Class<T> c) {
        TokenContainer<T> container = context.get(c);
        if (container == null) {
            context.put(c, new TokenContainerImpl<T>());
            container = context.get(c);
        }
        return container;
    }

    @Override
    public int getSize() {
        int res = 0;
        for (TokenContainer tokenContainer: context.values()) {
            res += tokenContainer.getSize();
        }
        return res;
    }

    @Override
    public boolean isEmpty() {
        return getSize() == 0 ? true: false;
    }
}

public class PetriNetContextImpl implements PetriNetContext {

    private Map<String, Place> placeMap = new HashMap<>();

    private Map<String, Transition> transitionMap = new
HashMap<>();

    private Map<String, Token> tokenMap = new HashMap<>();

    public PetriNetContextImpl(Map<String, Place> placeMap,
Map<String, Transition> transitionMap) {
        for (Place place: placeMap.values()) {
            this.placeMap.put(place.getName(), place);
        }

        for (Transition transition: transitionMap.values()) {
```

```
            this.transitionMap.put(transition.getName(),
transition);
        }
    }

    @Override
    public <T> void addToken(Class<T> c, String placeName, Token<T>
token) {
        placeMap.get(placeName).getTokenContext().addToken(c,
"val", token);
        tokenMap.put(token.getColor(), token);
    }
    @Override
    public Map<String, Place> getPlaceMap() {
        return placeMap;
    }

    @Override
    public void setPlaceMap(Map<String, Place> placeMap) {
        this.placeMap = placeMap;
    }

    @Override
    public Map<String, Transition> getTransitionMap() {
        return transitionMap;
    }

    @Override
    public void setTransitionMap(Map<String, Transition>
transitionMap) {
        this.transitionMap = transitionMap;
    }
    @Override
    public void addPlaceStateChecker(String name, Supplier<Boolean>
supplier) {
        placeMap.get(name).setCheckState(supplier);
    }

    @Override
    public void addTransitionAction(String name, Action action) {
        transitionMap.get(name).setAction(action);
    }
    @Override
    public Map<String, Token> getTokenMap() {
        return tokenMap;
    }
}
```