Содержание

Введение

Среди большого числа понятий, которые возникли и исследуются в информатике и кибернетике, одним из наиболее важных является понятие алгоритма. Алгоритм представляет процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов. Каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего. Понятие алгоритма тесно связано с понятием конечного автомата, для них свойственен одинаковый способ функционирования: система переходит из состояния в состояние в соответствии с заданной функцией переходов и осуществляет очередной (последовательный) шаг алгоритма.

По мере усложнения решаемых задач все большее внимание привлекают "неалгоритмические"параллельные системы с недетерминированным поведением, в которых отдельные компоненты функционируют, в основном, независимо, иногда взаимодействуя друг с другом. Примером могут служить такие системы параллельной обработки информации, как многопроцессорные вычислительные машины, параллельные программы, моделирующие параллельные дискретные системы и их функционирование, мультипрограммные операционные системы и т.п.

Системы с параллельно функционирующими и асинхронно взаимодействующими компонентами не описываются адекватно в терминах классической теории автоматов. Такие фундаментальные понятия, как состояние автомата и глобальная функция перехода не удобны для наглядной и экономичной характеризации недетменированной динамики поведения систем с локальными связями между независимыми параллельными процессами.

Среди многих существующих методов описания и анализа дискретных параллельных систем выделился подход, который основан на использовании сетевых моделей, восходящий к сетям особого вида, предложенным Карлом Петри для моделирования асинхронных информационных потоков в системах преобразования данных.

Часто алгоритм функционирования сложных систем представляют в виде диаграммы деятельности UML, отличающейся от классиче-

ской блок-схемы наличием элементов для представления многопоточной обработки. При этом в многопоточных системах встают вопросы о достижимости состояний и наличии блокировок. Для решения этих вопросов часто диаграмму деятельности представляют в виде простой сети Петри, при этом моделируется лишь процесс выполнения без привязки к данным. В отличие от простых сетей Петри, в раскрашенных сетях немаловажную роль играет типизация данных, основанная на понятии множества цветов, которое аналогично типу в декларативных языках программирования. Таким образом, представив диаграмму в виде раскрашенной сети Петри можно провести моделирование работы с учетом типов входных данных, что максимально приблизит процесс моделирования к реальному функционированию процесса.

Целью работы является исследование, разработка и реализация метода представления диаграммы деятельности в виде раскрашенной сети Петри, позволяющего выявить блокировки и недостижимые состояния.

Для достижения поставленной цели необходимо решить следующие задачи:

1. классифицировать существующие методы анализа диаграмм деятельности;

2. разработать метод представления диаграммы деятельности в виде раскрашеннои˘ сети Петри;

3. программно реализовать метод представления диаграммы деятельности в виде раскрашеннои˘ сети Петри;

4. исследовать факторы, влияющие на появление блокировок;

4. исследовать корректность построения раскраски сети.

1 Аналитический раздел

1.1 Анализ решений

Диаграмма деятельности — это, по существу, блок-схема, которая показывает, как поток управления переходит от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Результат может привести к изменению состояния системы или возвращению некоторого значения. Она отличается от традиционной блок-схемы более высоким уровнем абстракции, возможностью представления с помощью диаграмм деятельности управления параллельными потоками наряду с последовательным управлением.

Обычную блок-схему можно представить в виде конечного автомата, так как в любой момент времени из одной вершины поток выполнения переходит строго в одну другую. Теория автоматов имеет широкое распространение и для них реализовано множество методов, что делает задачу анализа весьма тривиальной задачей. Но с помощью автомата нельзя представить схемы с параллельными вычислениями, поэтому возникает необходимость использовать другой математический аппарат для исследования их поведения.

Таким аппаратом стали сети Петри, созданные специально для моделирования дискретных динамических систем. Сеть Петри представляет собой двудольный ориентированный граф, состоящий из вершин двух типов — позиций и переходов, соединённых между собой дугами. Вершины одного типа не могут быть соединены непосредственно. В позициях могут размещаться метки (маркеры), способные перемещаться по сети. Для анализа сетей Петри существует два метода: матричный метод и метод построения дерева достижимости. Оба метода имеют свои недостатки, но, применимо к данной проблематике лучшие результаты будет давать метод, основанный на деревьях достижимости.

1.2 Обзор существующих решений

1.2.1 CPN Tools

CPN Tools — это специальная моделирующая система, которая использует язык сетей Петри для описания моделей. Система была разработана в Университете Орхуса в Дании и свободно распространяется для некоммерческих организаций (http://cpntools.org/). CPN Tools предлагает очень мощный класс сетей Петри для описания моделей. Согласно стандартной классификации такие сети называют иерархическими временными раскрашенными сетями Петри. Было доказано, что они эквивалентны машине Тьюринга и составляет универсальную алгоритмическую систему. Таким образом, произвольный объект может быть описан с помощью этого класса сетей.

Простейшая концепция раскрашенной сети Петри использует различные типы фишек, где тип фишки определен натуральным числом и визуально представлен как цвет. Концепция раскрашенной сети Петри CPN Tools более сложная. Такие сети часто называют обобщенными раскрашенными сетями, так как тип фишки представлен абстрактным типом данных, как в языках программирования. Термин «раскрашенная» сохраняется исторически, но теперь очень трудно представить такие «цвета» визуально.

Временные сети Петри используют понятие модельного времени для описания продолжительности действий в реальных объектах. В отличие от классических сетей Петри, где срабатывание перехода происходит мгновенно, срабатывание перехода во временной сети связано с определенной продолжительностью или временной задержкой. Это позволяет анализировать временные характеристики реальных объектов, например, время отклика как характеристику качества обслуживания сети.

Иерархические сети обеспечивают построение сложных моделей. В таких сетях элемент может быть представлен другой сетью. В CPN Tools переход может быть замещен дополнительной сетью. Таким образом, получается вложенная конструкция: сеть внутри сети. Количество

уровней иерархии не имеет принципиальных ограничений. Отметим, что, такой подход широко распространен в языках программирования, где процедуры используются для управления сложностью.

Для достаточно простых моделей возможна генерация полного пространства состояний (графа достижимости). Это — лучший способ для верификации, например, телекоммуникационных протоколов. CPN Tools обеспечивает построение пространства состояний и автоматическую генерацию по нему отчета, который содержит выводы о стандартных свойствах сетей Петри, таких как ограниченность и живость. Кроме того, предусмотрен специальный язык на основе языка CPN ML для описания запросов о нестандартных свойствах пространства состояний, которые важны для пользователя. К сожалению, для сложных моделей пространство состояний может быть слишком большим, и его построение не представляется возможным.

Единственный способ для анализа сложных моделей — это имитация их поведения. CPN Tools предусматривает пошаговую имитацию для поиска и устранения ошибок в разрабатываемой модели, а также автоматическое выполнение определенного количества шагов. Имитация на больших временных интервалах – это путь для статистического анализа поведения модели. Такой подход применяется для оценки характеристик телекоммуникационных сетей, например, пропускной способности и качества обслуживания.

1.2.2 UniMod

UniMod является разработкой кафедры компьютерных технологий Санкт-Петербургского университета информационных технологий, механики и оптики. Долгосрочная цель проекта заключается в создании единой методологии для разработки и дальнейшего развития, что позволит сократить разрыв между проектированием и разработкой.

Проект UniMod (http://unimod.sourceforge.net) с открытым исходным кодом содержит набор инструментов, позволяющих визуально проектировать и реализовывать программы. При этом первоначально строится схема связей, состоящая, как и в *switch-технологии*, из источников

событий, системы управления и объектов управления, в которых реализованы вызываемые из автоматов выходные воздействия и опрашиваемые автоматами входные переменные. В данном случае система управления — это система взаимосвязанных автоматов.

Такой подход к проектированию систем акцентирует внимание разработчиков на изучении предметной области, выделении объектов управления и поставщиков событий. Затем проектируется один или несколько автоматов, для каждого из которых создается схема связей и граф переходов. Такой подход позволяет на ранних стадиях проектирования выявить и устранить множество возникающих неясностей в постановке задачи, а также предусмотреть весьма не очевидные детали поведения системы.

Сначала формируются события, обрабатываемые одним из автоматов в соответствии с графом переходов. Автомат при поступлении события может проверить различные логические условия и в результате выбрать необходимый переход в новое состояние. С переходом может быть ассоциирован некий набор выходных воздействий на объекты управления. Эти воздействия выполняются при выборе данного перехода. В общем случае действия могут выполняться не только на переходах. При поступлении определенного события автомат может перейти в конечное состояние, завершив тем самым работу приложения.

Одна из сильных сторон пакета UniMod — это возможность визуального конструирования программ. В отличие от распространенного подхода, когда вспомогательные картинки и UML-диаграммы рисуются с надеждой на улучшение документации и продуктивности труда, разработанные с помощью инструментального средства UniMod диаграммы и вручную написанные классы в целом формируют работающее приложение.

1.2.3 Язык программирования ДРАКОН

Наиболее близким и точным аналогом диаграмм деятельности являются математически строгие дракон-схемы визуального алгоритмиче-

ского языка ДРАКОН. Более отдаленным аналогом являются схемы алгоритмов по ГОСТ 19.701-90.

ДРАКОН (Дружелюбный Русский Алгоритмический язык, Который Обеспечивает Наглядность) — визуальный алгоритмический язык программирования. Был разработан в рамках космической программы «Буран». Основной задачей разработчиков было создание единого универсального языка программирования, который своей доступностью и мощностью был бы способен заменить специализированные языки ПРОЛ2 (для разработки бортовых комплексных программ Бурана), ДИПОЛЬ (для создания наземных программ Бурана) и ЛАКС (для моделирования). В качестве аксиоматики для ДРАКОНа были выбраны устремлённые графы (специальный класс циклических орграфов). Такое двумерное структурное программирование годится для доказательного построения алгоритмов методом Дейкстры. В отличие от блок-схем, дракон-схемы имеют средства для описания работы в реальном времени.

Некоторые ученые считают, что существующие способы записи алгоритмов и программ (принятые во всем мире) слишком трудны для понимания и требуют неоправданно больших трудозатрат. Это обстоятельство ставит непреодолимый барьер для многих специалистов, работа которых связана с алгоритмами, но которые не имеют резерва времени, чтобы научиться выражать свои профессиональные знания в форме алгоритмов и программ. Язык ДРАКОН использует новую эргономичную нотацию (дракон-схемы) и за счет этого существенно облегчает алгоритмизацию и программирование. Благодаря использованию дракон-схем алгоритмы и программы становятся более понятными, доходчивыми, ясными, прозрачными.

ДРАКОН — графический (визуальный) язык, в котором используются два типа элементов:

— графические фигуры (иконы);

— текстовые надписи, расположенные внутри или снаружи икон (текстоэлементы).

Поэтому язык ДРАКОН имеет не один, а два синтаксиса: графический и текстовый. Графический (визуальный) синтаксис охватывает алфавит икон, правила их размещения в поле чертежа и правила связи икон с помощью соединительных линий. Текстовый синтаксис задает алфавит символов, правила их комбинирования и привязку к иконам. (Привязка необходима потому, что внутри разных икон используются разные типы выражений). Императивная (процедурная) часть языка Дракон может описываться на основных языках программирования (c, c#, pascal и т.д.).

Система получается как результат процедурной декомпозиции деятельности на два и более алгопроцессов связанных отношением вызова и/или взаимодействия, т.е. решение сложной задачи мы разбиваем на подзадачи. В алгоритме решения при этом выделяем основной алгоритм и вспомогательные («вставки» в терминах языка ДРАКОН).

ДРАКОН поддерживает декомпозицию алгоритма выделением вспомогательных алгоритмов-вставок («предопределённых процессов» в терминах блок-схем по ГОСТ 19.701-90).

Представляемые схемами модели процессы могут находиться либо в отношении «главный-подчинённый» (иерархическая, или ранговая модель), либо в отношении «партнёров» (одноранговая, или диспозитивная модель). По порядку же возникновения всегда существует первичный процесс, который для другого данного процесса бывает:

— вызывающий — когда данный процесс был вставкой (во вставку во вставку и т. д. — если уровней вызова много) в другой процесс;

— «родительский» — когда данный процесс порождён как часть системы т. н. совместно протекающих взаимодействующих процессов (асинхронных или параллельных).

В ранговой модели существует только одна рабочая точка. Она последовательно проходит процессы, начиная с первичного вызывающего. Там, где указана вставка другого процесса, совершается переход на его схему. Когда эта схема пройдена до конца — переход обратно на место

указания вставки. В совокупности эти переходы образуют т. н. переход с возвратом. Первичный процесс здесь понимается как головной.

Суть асинхронности (параллелизма) — в допущении более чем одной «рабочей точки» для системы процессов. Каждая точка развёртывает свою схему, и при необходимости процессы взаимодействуют. Первичный процесс в этом случае понимается как базовый; он может контролировать ход порождённых им процессов и при необходимости «снимать» их — досрочно прекращать исполнение.

Переход от текстового (табличного) представления к графовому и называется визуализацией. ДРАКОН визуализирует структуру маршрутов для любого текстового языка программирования — и вообще для представления формальных информационных моделей, программно или алгоритмически строгих. Второе подразумевает построение модели, допускающей интерпретацию по Тьюрингу/Посту, Черчу/Клини, Маркову. [1] Первое следует понимать в смысле структуризации программы по Т. Бадду и Н. Вирту.

Дракон-схема реализует представление маршрутов алгоритма в классе устремлённых циклических ориентированных графов с дополнительно наложенным ограничением планарности (укладки на плоскости без пересечений), как это было показано Ермаковым и Жигуненко. [2] Вершины дракон-схемы представляют операторы и псевдооператоры при условии заполнения их текстом.

По предложению Паронджанова [1], для укладки содержание схемы разделяется на части-ветки так, чтобы из каждой пары пересекающихся цепей одна оказывалась связью между ветками. А эти связи укладываются в особую структуру — петлю силуэта — где ветки разделяются соединителями. Тем самым в знаковом (человекочитаемом) представлении цепочки следования вершин и их группы (образующие как линейную, так и нелинейную структуру) упорядочены на сцене и снабжены метками-именами веток. То и другое повышает удобство чтения.

ДРАКОН не относится к языкам «визуальным» в смысле, по-видимому, наиболее распространённом. Точнее будет называть его «графическим» (граф-языком) алгоритмизации (программирования). Термин был

предложен А. А. Тюгашевым в его работе [3] по программированию для систем реального времени. По-простому говоря, разницу можно показать следующим образом:

— «визуальный» язык — такой, что использует не чисто текстовое представление внешней формы предмета описания как основу для описания содержания предмета (определения его модели).

— «графический», граф-язык — такой, что представляет непосредственно содержание предмета описания, его модель, используя графику.

Так, ДРАКОН представляет свой предмет формализации — маршрутную структуру процесса (в частности, программной процедуры) непосредственно. И именно её «визуализирует» в виде графа. Тогда как «визуальный» язык типа Delphi представляет экранные формы программы как результат её работы. И уже с ними связывает процедуры как предмет формализации программистом. Причём в общем случае существуют и процедуры, не работающие непосредственно с экранными формами, и содержание процедур представляется текстом.

В ввиду большой наглядности языка ДРАКОН он получил широкое распространение расчетах космических и военно-морских программах. Распространение языка ДРАКОН можно разделить на два этапа.

На первом этапе сфера применения ДРАКОНа была ограничена ракетно-космической техникой. Язык применялся и применяется в Пилюгинском центре при разработке программ для бортового компьютера «Бисер», установленного на борту ракет-носителей и разгонных блоков космических аппаратов.

На втором этапе возникла необходимость приспособить инструментальные средства языка ДРАКОН для гражданских нужд широкого применения, для эксплуатации на персональных компьютерах (в том числе ноутбуках).

В результате сфера применения языка стала постепенно расширяться. Началось использование дракон-схем за рамками ракетно-космической техники — для решения задач в различных предметных областях.

1.3 Анализ UML диаграмм

В последнее время наблюдается общее повышение интереса ко всем аспектам, связанным с разработкой сложных программных приложений. Для многих компаний корпоративное программное обеспечения и базы данных (БД) представляют стратегическую ценность. Существует высокая заинтересованность в разработке и верификации методов и подходов, позволяющих автоматизировать создание сложных программных информационных систем (ИС). Известно, что систематическое использование таких методов позволяет значительно улучшить качество, сократить стоимость и время поставки ИС.

Визуальные модели широко используются в существующих технологиях управления проектированием систем, сложность, масштабы и функциональность которых постоянно возрастают. В практике эксплуатации ИС постоянно приходится решать такие задачи как: физическое перераспределение вычислений и данных, обеспечение параллелизма вычислений, репликация БД, обеспечение безопасности доступа к ИС, оптимизация балансировки нагрузки ИС, устойчивость к сбоям и т.п.

Построение модели корпоративной ИС до ее программной разработки или до начала проведения архитектурной реконструкции столь же необходимо, как наличие проектных чертежей перед строительством большого здания. Хорошие модели ИС позволяют наладить плодотворное взаимодействие между заказчиками, пользователями и командой разработчиков. Визуальные модели обеспечивают ясность представления выбранных архитектурных решений и позволяют понять разрабатываемую систему во всей ее полноте. Сложность разрабатываемых систем продолжает увеличиваться, и поэтому возрастает актуальность использования «хороших» методов моделирования ИС. Язык моделирования, как правило, включает в себя:

— элементы модели — фундаментальные концепции моделирования и их семантику;

— нотацию — визуальное предоставление элементов моделирования;

— принципы использования — правила применения элементов в рамках построения тех или иных типов моделей ИС.

Технология визуального моделирования, позволяет работать со сложными и очень сложными системами и проектами. И не важно, преобладает ли в проекте техническая сложность (статическая) или динамическая сложность управления. Сложность программных систем возрастает по мере создания новых версий. И в какой-то момент наступает «эффект критической массы», когда дальнейшее развитие ИС становиться невозможным, поскольку уже никто не представляет в целом "что и почему происходит". Происходит потеря управлением проектом. Внешней причиной или толчком возникновения этого неприятного эффекта может послужить, например, увольнение ведущего программиста или системного аналитика.

Сам по себе язык UML является языком графического описания для объектного моделирования в области разработки программного обеспечения, созданного консорциумом OMG (object managment group). UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но в средствах выполнения UML-моделей как интерпретируемого кода возможна кодогенерация. Большинство редакторов UML диаграмм используют свои собственные нотации для текстового описания полученных диаграмм, учитывающие особенности самой системы.

1.3.1 Диаграмма деятельности

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций.

Одно из основных направлений использования диаграмм деятельности — отображение внутрисистемной точки зрения на прецедент. Диаграммы деятельности применяют для описания шагов, которые должна предпринять система после того, как инициирован прецедент.

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на этих диаграммах также присутствуют обозначения состояний и переходов. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние выполняется только при завершении этой операции. Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Они позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения.

В контексте языка UML деятельность (activity) представляет собой совокупность отдельных вычислений, выполняемых автоматом, приводящих к некоторому результату или действию (action). На диаграмме деятельности отображается логика и последовательность переходов от одной деятельности к другой, а внимание аналитика фокусируется на результатах. Результат деятельности может привести к изменению состояния системы или возвращению некоторого значения.

Диаграмму деятельности можно представить как $AD = \{N, E\}$, где N — вершины, E — переходы. Переходами является отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе состояние. Вершины представляют собой состояния и могут быть одного из типов:

— **Начальное и конечное состояние**.

— **Деятельность** — состояние, которое представляет вычисление атомарного действия, как правило — вызов операции. Состояния действия не могут быть подвергнуты декомпозиции. Они атомарные, то есть внутри них могут происходить различные события, но выполняемая в состоянии действия работа не может быть прервана. И наконец, обычно предполагается, что длительность одного состояния действия занимает неощутимо малое время.

— **Условие** — описывает различные пути выполнения в зависимости от значения некоторого булевского выражения. Графически точка ветвления представляется ромбом. В точку ветвления может входить ровно один переход, а выходить — два или более. Для каждого исходящего перехода задается булевское выражение, которое вычисляется только один раз при входе в точку ветвления. Ни для каких двух исходящих переходов сторожевые условия не должны одновременно принимать значение «истина», иначе поток управления окажется неоднозначным. Но эти условия должны покрывать все возможные варианты, иначе поток остановится.

— **Разделение и слияние** — разделение потока выполнения на несколько параллельных процессов.

1.3.2 Представление UML диаграмм

Основным стандартом описания UML диаграмм является XMI (XML metadata interchange) — стандарт OMG для обмена метаданными с помощью языка XML. XMI может использоваться для любых метаданных, если их метамодель может быть выражена с помощью MOF (meta-object facility). С точки зрения OMG всю информацию можно разделить на абстрактные и реальные модели. Абстрактные модели предоставляют общую семантическую информацию в виде произвольного описания MOF, в то время как реальные модели должны представлять сами UML диаграммы. XMI не имеет жесткой стандартизации, а его описание содержит лишь предложения для реализации. В настоящий момент несколько крупных производителей имеют свои собственные реализации представления XMI, лишь косвенно основанных на основном стандарте, что

делает невозможным переносимость файлов между различными редакторами UML.

Для гибкости модели в XMI не описываются напрямую связи между вершинами. Вместо этого для каждой вершины приписывается список входных и выходных переходов. Каждый переход содержит информацию о исходящей и входящей вершине и логическое условие для срабатывания перехода. Таким образом, для поиска исходящих вершин мы проходим по всем переходам и уже из них получаем вершины. Стоит отметить, что у вершины может быть несколько входных и выходных переходов, но каждый переход имеет лишь одну родительскую вершину и одну целевую.

1.4 Сети Петри

Сети Петри разрабатывались специально для моделирования таких систем, которые содержат взаимодействующие параллельные компоненты. В своей докторской диссертации Связь автоматов Карл Адам Петри сформулировал основные понятия теории связи асинхронных компонент вычислительной системы. В частности, он подробно рассмотрел описание причинных связей между событиями. Его диссертация посвящена главным образом теоретической разработке основных понятий, с которых начали развитие сети Петри. [4]

Реальные дискретные системы состоят из разнообразных компонентов, различающихся физическими свойствами, функциональным назначением, сложностью внутренней структуры. Для того, чтобы спроектировать адекватный математический аппарат, предназначенный для моделирования систем, необходимо установить круг вопросов, которые должные решаться с помощью моделей и осуществить переход от физических сущностей к их абстракциям, сначала в форме некоторого набора концептуальных понятий, затем — в точных математических терминах.

Часто при проектировании системы необходимо узнать выполняет ли система те функции, для которых она предназначен; функционирует ли она эффективно; могут ли в ней возникнуть ошибки и аварийные ситуации; имеются ли в ней узкие места.

Первый шаг на пути к построению модели дискретной системы — это абстрагирование от конкретных физических и функциональных особенностей ее компонентов. Компоненты системы и их действия представляются абстрактными событиями, каковыми могут быть, например, исполнение оператора программы, переход триггера из состояния в состояние, прерывание в операционной системе и т.п.

Событие может произойти один раз, повториться многократно, порождая конкретные действия, или не произойти ни разу. Совокупность действий, возникающих как реализация событий при функционировании дискретной системы, образует процесс, порождаемый этой системой. В общем случае одна и та же система может функционировать в одних и тех же условиях по-разному, порождая некоторое множество процессов, т.е. функционировать недетерменированно.

Реальная система функционирует во времени, события происходят в некоторые моменты времени и длятся некоторое время. В синхронных моделях дискретных систем события явно привязаны к определенным моментам или интервалам времени, в которые происходит одновременное изменение состояний всех компонентов системы, трактуемое как изменение общего состояния системы. Смена состояний происходит последовательно. Этот подход к моделированию больших параллельных систем имеет ряд недостатков.

Во-первых, в большой системе приходится учитывать состояние всех компонентов при каждой смене ее общего состояния, что делает модель громоздкой, особенно в тех случаях, когда локальные изменения касаются небольшого фрагмента системы.

Во-вторых, при таком подходе исчезает информация о причинно-следственных связях между событиями в системе. Например, если два события при функционировании системы произошли одновременно, то мы не знаем, произошло ли это случайно или в этом факте скрыт какой-то функциональный смысл. Такие понятия, как конфликты между компонентами системы (из-за ресурсов) или ожидание одним из компонентов результатов работы других компонентов, трудно выражаются в терминах смены состояний системы.

В-третьих, в так называемых асинхронных системах события могут происходить внутри неопределенно больших интервалов времени, заранее трудно или нельзя указать более точно время их начала, конца и длительность.

Выходом может служить отказ от введения в модели дискретных систем времени и тактированных последовательностей изменений состояний, а замена их причинно-следственными связями между событиями. Модели такого типа (в том числе сети Петри) называют асинхронными (если возникает необходимость осуществить привязку ко времени, то моменты или интервалы времени представляют как события). Таким образом, существенно синхронные системы могут описываться в терминах асинхронных моделей. Замена временных связей причинно-следственными дает возможность более наглядно описать структурные особенности функционирования систем.

Отказ от времени приводит к тому, что события в асинхронной модели рассматриваются или как элементарные (неделимые, «мгновенные»), или как составные, имеющие некоторую внутреннюю структуру, образованную из «подсобытий».

Взаимодействие событий в больших асинхронных системах имеет, как правило, сложную динамическую структуру. Эти взаимодействия описываются более просто, если указывать не непосредственные связи между событиями, а те ситуации, при которых данное событие может реализоваться. При этом глобальные ситуации в системе формируются с помощью локальных операций, называемых условиями реализации событий. Условие имеет емкость: условие не выполнено (емкость равна 0), условие выполнено (емкость больше 0), условие выполнено с n-кратным запасом (емкость равна n, где n - целое положительное число). Условие соответствует таким ситуациям в моделируемой системе. как наличие данного для операции в программе, состояние некоторого регистра в устройстве ЭВМ, наличие деталей на конвейере и т.п. Определенные сочетания условий разрешают реализоваться некоторому событию (предусловия события), а реализация события изменяет некоторые условия (посту-

словия события), т.е. события взаимодействуют с условиями, а условия — с событиями.

Таким образом, предполагается, что для решения основных задач, озвученных ранее, достаточно представлять дискретные системы как структуры, образованные из элементов двух типов — событий и условий.

1.4.1 Моделирующие способности сетей Петри

К базовым элементам сетей Петри можно отнести позиции и переходы между ними. Переходы в сетях Петри изначально предназначены для представления событий, а позиции отражают условия для возможности их срабатывания. Маркер сети Петри может быть остановлен только в позиции, но не на переходе, так как переходы представляют собой атомарные действия.

Два наиболее простых подкласса сетей Петри образуются за счет наложения строгих топологических ограничений на структуру сети, т.е. ограничения на отношение инцидентности F. Сеть называется автоматной, если каждый переход сети имеет ровно одно входное и одно выходное место (рис. 1.1). Сеть Петри со множеством мест P называется синхронизационным графом, если в каждое место сети входит ровно одна дуга и из каждого места выходит ровно одна дуга (рис. 1.2). Из определения автоматной сети следует, что граф связен и при своем срабатывании любой переход t_i изымает ровно одну фишку из своего входного места p_1 и помещает ровно одну выходную фишку в выходное место p_2. Так как автоматная сеть конечна, то граф ее разметок конечен, и, следовательно, в классе автоматных сетей разрешимы проблемы достижимости в общем виде. Оба подкласса способны моделировать только простые дискретные системы.

Простые сети Петри по выразительной мощности превосходят такие классы моделей, как конечные автоматы, но все же не все системы можно моделировать с их помощью. Это заставило искать такие обобщения сетей, которые увеличивали бы их выразительную мощность.

Это приводит к универсальным моделирующим системам, порождающим рекурсивно-перечислимые классы языков, т.е. к системам, равномощным машинам Тьюринга. Для дальнейшего исследования воспользуемся тем фактом, что машина Тьюринга равномощна счетчиковому автомату (модификация машины Минского). В [4] приводится доказательство того, что любой рекурсивно перечислимый язык порождается некоторым счетчиковым автоматом.

Счетчиковый автомат состоит из конечного множества счетчиков $\{x_i : 1 \leqslant i \leqslant N\}$, алфавита и программы автомата. Программа представляет из себя связный ориентированный граф с одной начальной вершиной без входных дуг, с одной заключительной вершиной без выходных дуг. Из остальных вершин выходит одна или две дуги. Вершинам приписаны операторы одного из шести типов:

а) начальной вершине приписан «старт», единственный в программе;

б) заключительной вершине приписан «стоп», единственный в программе;

в) оператор прибавления единицы $x_i = x_i + 1$, может быть приписан вершине с одной входной дугой;

г) оператор печати символа из алфавита автомата;

д) оператор недетерминированного перехода;

е) оператор условного вычитания единицы if $x_i \neq 0$ then $x_i = x_i - 1$.

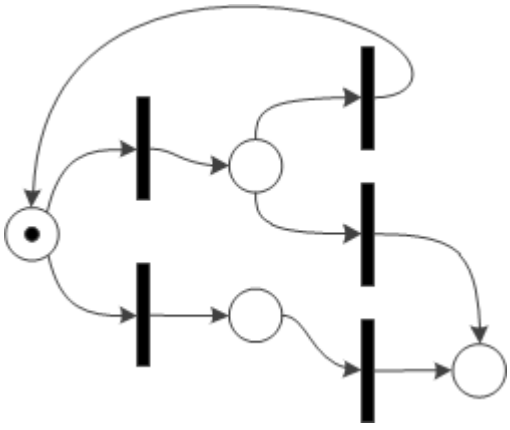


Рисунок 1.1 — Автоматная сеть.

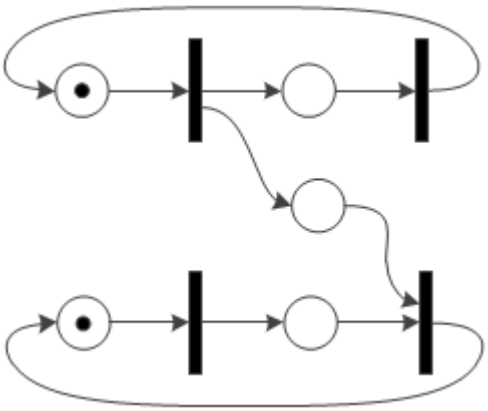


Рисунок 1.2 — Синхронизационный граф.

Все операторы, кроме условного вычитания единицы можно промоделировать с помощью сети Петри. Причина этого состоит в том, что в сети Петри можно заметить (отметить это срабатыванием некоторого перехода) тот факт, что место сети изменило разметку с нулевой на ненулевую, но нельзя отметить факт изменения разметки с ненулевой на нулевую. Таким образом, из двух альтернатив ($x_i \neq 0$ и $x_i = 0$), содержащихся в операторе условного вычитания единицы, в сети Петри можно представить только первую, но нельзя отобразить проверку на ноль, т.к. сеть не может непосредственно среагировать на отсутствие фишки в месте (если место не ограничено).[5] Таким образом, для решения этих проблем вводится понятие ингибиторной сети (рис. 1.3) и сети с приоритетами (1.4), которая строго мощнее класса сетей Петри.

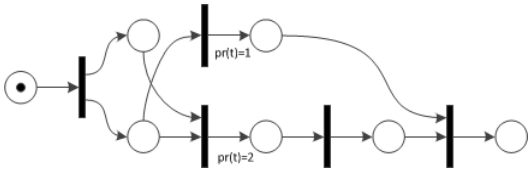


Рисунок 1.3 — Ингибиторная сеть.

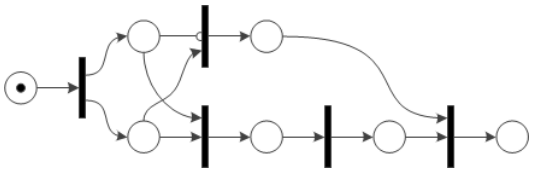


Рисунок 1.4 — Сеть с приоритетами.

При моделировании сетями Петри дискретных систем фишки часто соответствуют объектам, передаваемым от компонента к компоненту системы. Зачастую эти объекты имеют дополнительные атрибуты, позволяющие различать их и использовать эти различия для управления функционированием системы. Таким образом, мы приходим к модели сети Петри, в которой фишкам приписаны некоторые признаки, например цвет, а условие срабатывания переходов определяется таблицей, учитывающей цвета фишек. Такая сеть называется раскрашенной и является сетью, строго более мощной чем сеть Петри. Выразительная мощность раскрашенной сети Петри зависит от количества признаков. Класс сетей с конечным множеством признаком эквивалентен классу сетей Петри [4] [6], хотя при преобразовании могут значительно увеличится размеры сети.

Логичной модификацией сети Петри так же является разделение ее работы на такты. Такая сеть называется синхронной. В начале

такта выясняется, какие переходы могут сработать, из них выбирается максимальное множество взаимной неконфликтных переходов, затем это множество переходов срабатывает обычным образом меняя разметку сети. Необходимо отметить, что внутри такта выбранное множество срабатываемых переходов не меняется. Эта идея дает решение для проблемы недетерменированности переходов сети, т.е. для каждого перехода вводится его приоритет, и порядок срабатывания определяется старшинством приоритетов.

В сетях с приоритетами и синхронных сетях наметился отход от чисто локального принципа управления функционированием сети, принятого в сетях Петри, и переход к привлечению более глобальной информации о состоянии сети. Такой подход используется и в самомодифицирующихся сетях, в которых каждой дуге приписана модифицируемая кратность. Если кратность дуги — число, то она имеет смысл, что и кратность в сетях Петри. Но в качестве кратности может выступать символ q некоторого места, в этом случае кратность дуги переменна и равна текущей разметке M(q) места q. Поскольку разметка места q может динамически меняться в процессе работы, то и кратность тех дуг, которым сопоставлен символ q, динамически меняется. Эта модификация интересна тем, что позволяет достаточно просто моделировать ингибиторные сети (рис. 1.5) и сети с приоритетами (рис. 1.6), т.е. сеть равномощна машине Тьюринга.

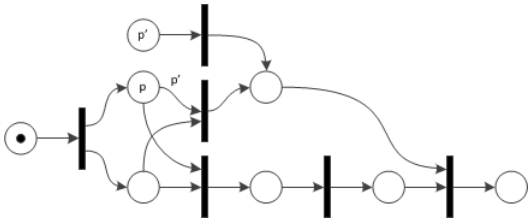


Рисунок 1.5 — Самомодифицирующаяся сеть, эквивалентная ингибиторной сети на рис. 1.3.

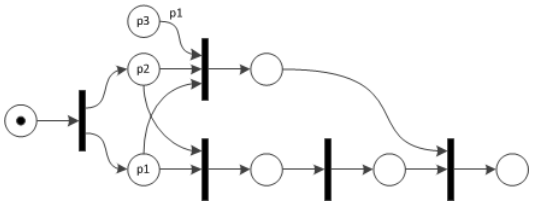


Рисунок 1.6 — Самомодифицирующаяся сеть, эквивалентная сети с приоритетами на рис. 1.4.

Ниже в таблице 1.1 приведено краткое сравнение выразительных мощностей модификаций сетей Петри. Типы сетей расположены в порядке возрастания их выразительной мощности.

Таблица 1.1 — Сравнение выразительной мощности сетей.

	Тип сети	Выразительная мощность
1	Автоматная сеть	Сохраняющая и дерево достижимости для нее является конечным. Из-за этого ограничения эта разновидность сетей неприменима для моделирования систем с разделяемым доступом.
2	Обыкновенная сеть Петри	Используется для моделирования дискретных систем с разделяемым доступом.
3	Раскрашенная сеть Петри	Такая модификация сети используется для упрощения сети за счет учета особенностей функционирования моделируемого объекта. Если множество признаком конечно, то сеть эквивалентна простой сети Петри.
4	Ингибиторыне сети, сети с приоритетами и самомодифицирующиеся сети	Сеть описывает рекурсивно-перечислимые классы языков, равномощные машинам Тьюринга. В связи с этим в [5] формулируется теорема о том, что проблема ограниченности, достижимости, эквивалентности по префиксным и терминальным языкам, пустоты этих языков неразрешима для этих сетей

Продолжение на след. стр.

Продолжение таблицы 1.1

5	Иерархические сети Петри	Иерархические сети являются обобщением регулярных сетей и служат для моделирования иерархических систем, которые на ряду с неделимыми, атомарными компонентами, содержат составные компоненты, сами представляющие собой систему. Иерархическая сеть функционирует, переходя от разметки к разметке, как и регулярная сеть, но правила функционирования иерархической сети отличаются от соответствующих правил для регулярной сети. Эти различия вызваны наличием составных переходов, срабатывание которых является не мгновенным событием, как в сетях Петри, а составными действием. [4] Иерархическая сеть обладает такими же моделирующими способностями, что и машина Тьюринга.

1.4.2 Раскрашенные сети Петри

Теория раскрашенных сетей Петри разрабатывается более 20 лет рабочей группой (CPN Group) университета г.Орхуса (University of Aarhus, Denmark) под руководством профессора Курта Йенсена (Kurt Jensen). Этой группой разработана основная модель, включающая использование типов данных и иерархических конструкций, определены концепции динамических свойств, развивается теория методов анализа.

Раскрашенная сеть Петри — это графоориентированный язык для проектирования, описания, имитации и контроля распределенных и параллельных систем. Графическими примитивами показывается течение процесса, а конструкциями специального языка имитируется необходимая обработка данных. Сеть представляет собой направленный граф с двумя типами вершин – позициями и переходами, при этом дуги не могут соединять вершины одного типа, т.е. граф является двудольным.

Множество позиций (обозначаются эллипсом) описывают состояния системы. Переходы (обозначают прямоугольниками) описывают условия изменения состояний. Позиции называются входными для конкретного перехода, если направление дуги, указывает на переход. Позиции называются выходными для перехода, если дуга ведет от перехода к позиции.

В отличие от простых сетей Петри, в раскрашенных немаловажную роль играет типизация данных, основанная на понятии множества цветов, которое аналогично типу в декларативных языках программирования. Соответственно, для манипуляции цветом применяют переменные, функции и другие элементы, известные из языков программирования. Ключевой элемент сети — позиция — имеет определенное значение из множества цветов. [7]

Для отражения динамических свойств в сеть Петри введено понятие разметки сети, которая реализуется с помощью, так называемых фишек, размещаемых в позициях. Цвет позиции определяет тип фишек, которые могут там находиться. Конкретизация фишки, находящейся в данной позиции, определяется инициализирующим выражением начальной разметки или формируется в результате правильного выполнения шага итерации сети Петри.

Сеть представляет собой асинхронную систему, в которой фишки перемещаются по позициям через переходы. Переход может сработать (т.е. переместить фишку из входной позиции в выходную для данного перехода), если во всех входных позициях для данного перехода присутствует хотя бы одна фишка и выполнено логическое выражение, ограничивающее переход (спусковая функция).

Дуги могут иметь пометки в виде выражений (переменных, констант или функций), определенных для множества цветов, и использоваться либо для «вычленения» компонентов сложного цвета фишек при определении условия срабатывания перехода, либо для изменения цвета фишки следующей позиции после срабатывания перехода. [8]

1.5 Анализ сетей Петри

С помощью Сетей Петри можно моделировать широкий класс систем, представляя должным образом взаимодействие различных процессов. Наиболее мощны сети Петри в моделировании систем, включающих параллельные действия, причем параллельность моделируется естественным образом. Однако одно моделирование малополезно, необходимо провести анализ моделируемой системы. Для анализа используются два подхода: дерево достижимости и матричный метод.

В ходе анализа рассматриваются вопросы безопасности, ограниченности, сохранения, активности и достижимости в сети.

Безопасность и ограниченность

Сеть является ограниченной, если число фишек в каждой позиции не превышает N. Безопасность — это частный случай ограниченности. Безопасность позволяет представить позицию триггером, но в более общем случае можно использовать счетчик. Однако любой аппаратно-реализованный счетчик ограничен по максимальному числу, которое он может представить.

Позиция $p_i \epsilon P$ сети Петри $C = (P, T, I, O)$ с начальной маркировкой μ является k-безопасной, если $\mu'(p_i) \leqslant k$ для всех $\mu' \epsilon R(C, \mu)$.

Сохранение

Сеть является сохраняющей, если общее число фишек в системе не изменяется. Данное условие крайне важно в случае, если фишки представляют собой некоторые системные ресурсы.

Активность

Сеть Петри является активной, если в ней не могут возникнуть тупики. Тупик в сети Петри — это переход (множество переходов), которые не могут быть запущены. Переход называется активным, если он не тупиковый. Существует пять уровней активности:

а) Переход t_i обладает ***активностью уровня 0***, если он никогда не может быть запущен.

б) Переход t_i обладает ***активностью уровня 1***, если он потенциально запустим, т.е. существует такая $\mu' \epsilon R(C, \mu)$, что t_i разрешен в μ' .

в) Переход t_i обладает ***активностью уровня 2***, если для всякого целого n существует последовательность запусков, в которых t_i присутствует, по крайней мере, n раз.

г) Переход t_i обладает ***активностью уровня 3***, если существует бесконечная последовательность запусков, в которых t_i присутствует неограниченно часто.

д) Переход t_i обладает ***активностью уровня 4***, если для всякой $\mu' \epsilon R(C, \mu)$ существует такая последовательность запусков σ, что t_i разрешен в $\delta(\mu', \sigma)$.

Переход, обладающий активность уровня 0 называется пассивным. Переход, обладающий активностью уровня 4, называется активным.

Достижимость

Для данной сети Петри C с маркировкой μ определить $\mu' \epsilon R(C, \mu)$. Это основанная задача анализа сетей, т.к. остальные задачи сводятся к задаче достижимости.

1.5.1 Дерево достижимости

Дерево достижимости представляет множество достижимости Сети Петри. Каждая i-я вершина дерева связывается с расширенной разметкой $\mu(i)$. В расширенной разметке число меток в позиции может быть либо неотрицательным целым, либо бесконечно большим. Бесконечное число меток обозначим символом ω. Каждая вершина классифицируется или как граничная, терминальная, дублирующая вершина, или как внутренняя. Граничными являются вершины, которые еще не обработаны алгоритмом. После обработки граничные вершины становятся либо

терминальными, либо дублирующими, либо внутренними. Терминальные (пассивные) маркировки — это маркировки в которых нет разрешенных переходов. Дублирующие маркировки — это маркировки, ранее встречающиеся в ранее встречающиеся в дереве.

Дерево достижимости можно использовать для решения задач безопасности, ограниченности и сохранения. Его нельзя использовать для решения задач достижимости и активности в общем случае. Решение этих задач ограничено существованием символа ω. Символ ω означает потерю информации; конкретные количества фишек отбрасываются, учитывается только существование их большого числа. [5] Вместе с тем, в отдельных конкретных случаях дерево достижимости позволяет судить о свойствах достижимости и активности. Например, сеть, дерево достижимости которой содержит терминальную вершину, не активна. Аналогично искомая маркировка μ в задаче достижимости может встретиться в дереве достижимости, что означает ее достижимость. Кроме того, если маркировка не покрывается некоторой вершиной дерева достижимости, то она недостижима.

1.5.2 Матричный способ

Второй подход к анализу сетей Петри основан на матричном представлении сетей Петри. Альтернативный по отношению к определению сети Петри в виде (P, T, I, O) является определение двух матриц D^+ и D^-, представляющих входную и выходную функцию. Каждая матрица имеет m сток (по одной на переход) и n столбцов (по одному на позицию).

Матричная форма эквивалентна стандартной форме, но позволяет дать определение в терминах векторов и матриц. Пусть $e[j]$ — m-вектор, содержащий нули везде, кроме j-ой компоненты. Переход t_j представляется m-вектором $e[j]$. Переход в t_j в маркировке μ разрешен, если $\mu \geqslant e[j] * D^-$, а результат запуска перехода t_j в маркировке μ записывается как:

$$\delta(\mu, t_j) = \mu - e[j] * D^- + e[j] * D^+ = \mu + e[j] * (D^+ - D^-) = \mu + e[j] * D, \quad (1.1)$$

где $D = D^+ - D^-$ — основная матрица изменений.

Маркировка μ' достижима из маркировки μ, если существует последовательность (возможно, пустая) запусков переходов σ, которая приводит из μ' в μ. Следовательно, получаем, что μ' достижима из μ, если

$$\mu' = \mu + x * D \tag{1.2}$$

имеет решение в неотрицательных целых.

Матричный подход к анализу сетей Петри очень перспективен, но имеет некоторые трудности. Прежде всего, матрица D не полностью отражает структуру сети: переходы, имеющие как входы, так и выходы из одной позиции (петли), представляются соответствующими элементами матриц D^+ и D^-, но затем уничтожаются в матрице $D = D^+ - D^-$.

Основная проблема заключается в том, что решение уравнения 2.1 является необходимым для достижимости, но не достаточным.

2 Конструкторский раздел

2.1 Этапы решения поставленной задачи

Исходными данными для задачи является описание диаграммы деятельности с помощью XMI. По этому описанию строится UML диаграмма и преобразуется в простую сеть Петри. Для построения раскрашенной сети из описания диаграммы выделяются используемые переменные, и строится предварительное описание типов. После задания пользователем начальной разметки на основе предварительных данных о типах переменных строится окончательное множество типов и раскрасок. И с учетом видимости переменных выстраивается раскраска сети. Общий процесс функционирования представлен на рис. 2.1.

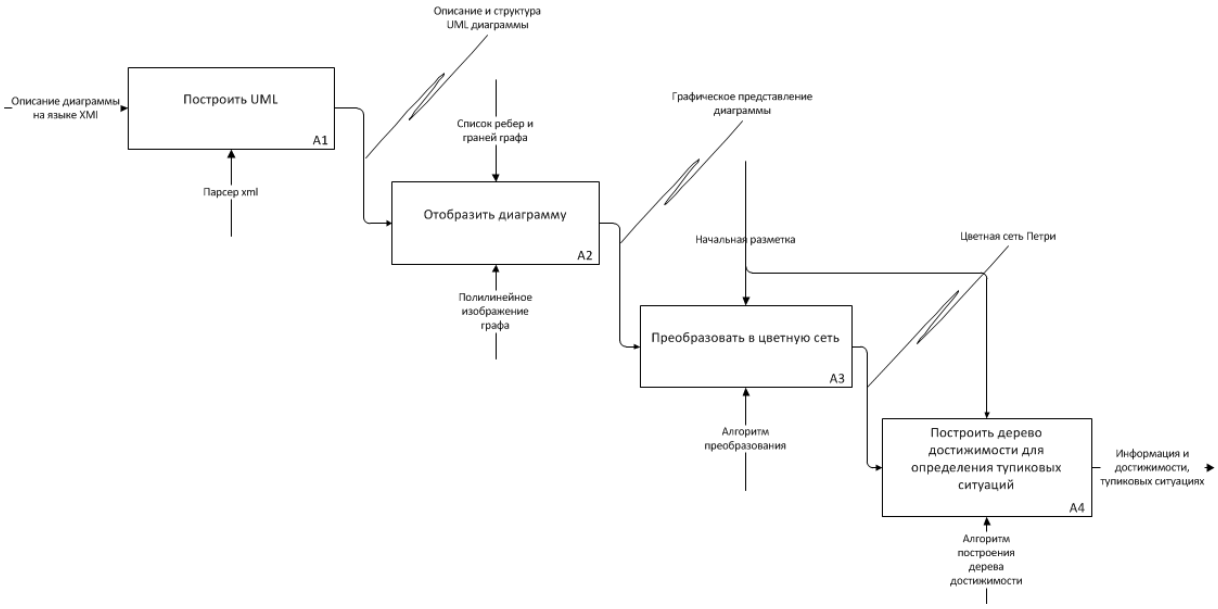


Рисунок 2.1 — Общий процесс функционирования.

Диаграмма деятельности представляет собой граф, вершины которого обозначают действия, а дуги — переходы от одного действия к другому. Способы преобразования UML-диаграмм в простую сеть Петри широко известны, но для преобразования в раскрашенную сеть одних данных диаграммы не хватает: для раскрашенной сети необходимо сформировать множество типов и определить раскраски вершин (т.е. определить тип входных переменных). Процесс преобразования в раскрашенную сеть Петри представлен на рис. 2.2.

2.2 Рисование бесконтурных графов

Любую диаграмму деятельности можно свести к планарному графу, а значит и сеть Петри, построенную по этой диаграмме можно отобразить как планарный граф. Будем предполагать, что исходный бесконтурный граф является так называемым *st-графом*, т.е. в нем имеется только одна начальная вершина, обозначаемая через s, и одна конечная, обозначаемая через t.

Аналитический алгоритм планарного представления графа описывает последовательность различных преобразований, приводящую к построению укладки. [9] Краткий план работы алгоритма таков:

1. Построение ассоциированного орграфа G*.
2. Топологическая сортировка G и G*.
3. Мозаичное представление графа G.
4. Полилинейное изображение графа G, основанное на мозаичном представлении и информации о типах входных и выходных позиций.

Рассмотрим каждый этап работы алгоритма подробнее.

2.2.1 Построение ассоциированного орграфа G*

Пусть G* — планарный *st-граф*, и пусть V, EF - множества вершин, ребер и граней графа G, где внешние грани представлены в F двумя элементами $s*$ и $t*$, называемыми левой и правой внешней гранью G. Определим орграф G*, ассоциированный с G, следующим образом:

— вершинами G* являются элементы из F;

— для любой дуги $e \neq (s,t)$ в G, граф G* имеет дугу $e* = (f,g)$, где f — левая по отношению к e грань, а g — правая.

Для построения G* необходимо получить список всех граней графа G. Для данной задачи условимся понимать под гранью простой цикл графа, не содержащий в себе других циклов. Для поиска циклов используем модификацию алгоритма поиска в глубину:

```
function findBaseFaces(matrix, index, used[], track[]):
    used[index] = true;
```

```
foreach outgoing vertex : v from vertex[index]:
    if not used[v]:
        track << v
        // Запуск поиска в глубину из текущей вершины.
        findBaseFaces(matrix, v, used, track)
        // Если вершина еще содержится в списке, то удаляем ее отт
        // т.к. она не содержится в цикле.
        if (track.last = v)
            track remove last
    else if (track contains v) and (track.last <> v):
        // Берем вершины из списка пути следования
        // пока не встретится текущая вершина.
        previous = v;
        while v <> k:
            k << track
            faces << (previous, k)
            used[k] = not used[k]
            previous = k

        // Удаляем первую дугу цикла из графа, тем самым нарушая ц
        // и запускаем поиск из текущей вершины заново.
        (u, v) << faces.last
        matrix[u, v] = matrix[v, u] = 0
        continue cycle from begining
```

Для поиска циклов необходимо преобразовать граф G в неориентированный граф простым симметричным отображением относительно главной диагонали. Алгоритм использует массив used для определения уже просмотренных вершин и список track, содержащий в себе вершины текущей итерации обхода. Если текущая вершина еще не была просмотрена, то запускаем поиск из нее, иначе, если эта вершина уже встречалась и это вершина не является предшественником текущей, мы нашли цикл.

Для нахождения всех вершин, содержащихся в цикле, мы забираем все вершины из списка track на пути до текущей, помечаем их как не

пройденные, т.к. они могут содержаться в другом цикле, и добавляем дуги в список faces. После этого удаляем из графа первую дугу цикла. Так как поиск в глубину идет с лева на право, то если эта дуга содержится в другом цикле, то этот цикл уже был найден на предыдущих шагах.

На основе множества граней графа G необходимо для каждого ребра определить левую и правую грани (рис. 2.4) и по ним построить граф G*. Для определения ориентации ребер воспользуемся тем, что номера вершин и переходов упорядочены, т.е. дуга $1 \to 2$ будет раньше, чем дуга $1 \to 3$. Отсюда следует, что для определения положения граней относительно ребра необходимо определить на правом или левом пути из истока лежит эта грань. Для определения истока необходимо найти на грани вершину, из которой есть только исходящие дуги, но воспользовавшись тем, что вершины упорядочены, можно утверждать, что истоком будет вершина с наименьшим номером. Получаем, что если ребро лежит на левом пути из истока в сток грани, то грань относительно этого ребра правая, иначе — левая.

Таким образом, получив для каждого ребра графа G левую и правую грань, выстраиваем матрицу смежности графа G* как переход из левой грани в правую. Матрица смежности G* понадобится в дальнейшем для топологической сортировки уровней G*, таким образом, нет нужды учитывать количество одинаковых дуг из одной грани в другую.

2.2.2 Топологическая сортировка сетей

Топологическая сортировка представляет собой алгоритм упорядочивания вершин бесконтурного ориентированного графа согласно частичному порядку, заданному ребрами орграфа на множестве его вершин. На первом шаге алгоритма мы выбираем все вершины с нулевой полустепенью захода, заносим их в нулевой уровень и исключаем их из графа. В результате, (больше не учитываются строки матрицы, соответствующие обработанным вершинам) полустепени захода остальных вершин уменьшатся. Шаги повторяются пока все вершины не будут просмотрены и распределены по уровням.

На рисунке 2.5 приведена итеративная работа алгоритма.

Если граф содержит цикл, то алгоритм на очередном шаге не сможет найти такую вершину, у которой полустепень захода равна нулю, другими словами, нельзя топологически представить цикл, т.к. для него нарушается правило $X(p) > X(q) \Leftrightarrow (q,p)\epsilon E$.

2.2.3 Мозаичное представление графа

Мозаичное представление Θ графа G — это такое отображение каждого объекта O из $V \bigcup E \bigcup F$ в плитку $\Theta(O)$, что справедливы следующие свойства:

— если $O_1 \neq O_2$, то нет общих внутренних вершин у $\Theta(O_1)$ и $\Theta(O_2)$;

— объединение всех плиток $\Theta(O)$, $O\epsilon V \bigcup E \bigcup F$ является прямоугольником;

— $\Theta(O_1)$ и $\Theta(O_2)$ горизонтально инцидентны тогда и только тогда, когда $O_1 = left(O_2), O_1 = right(O_2)$ или $O_2 = left(O_1), O_2 = right(O_1)$;

— $\Theta(O_1)$ и $\Theta(O_2)$ вертикально инцидентны тогда и только тогда, когда $O_1 = orig(O_2), O_1 = dest(O_2)$ или $O_2 = orig(O_1), O_2 = dest(O_1)$;

Вершины $p\epsilon V$ графа G помечаются числами Y(p) таким образом, что Y(s) = 0 и $Y(p) > Y(q) \leftrightarrow (q,p)\epsilon E$ — дуга G.

Вершины $p*\epsilon V*$ графа G* помечаются числами X(p*) таким образом, что X(s*) = 0 и $X(p) > X(q) \leftrightarrow (q,p)\epsilon E*$ — дуга G*.

Для каждой вершины вычисляются координаты горизонтального сегмента 2.1:

$$\begin{cases} y = Y(v) \\ x_1 = X(left(v)) \\ x_2 = X(right(v)) \end{cases} \tag{2.1}$$

Для каждого ребра графа G вычисляются координаты вертикального сегмента 2.2:

$$\begin{cases} y = X(left(e)) \\ x_1 = Y(orig(e)) \\ x_2 = Y(dest(e)) \end{cases} \tag{2.2}$$

Обзорным представлением Γ заданного *st-графа* G называется такое его изображение, в котором каждая его вершина p представлена горизонтальным отрезком Γ(p), называемым вершинным отрезком, а каждая дуга (p, q) — вертикальным отрезком Γ(p, q), называемым реберным отрезком, таким образом, что справедливы следующие свойства:

— вершинные отрезки не накладываются друг на друга;

— реберные отрезки не накладываются друг на друга;

— реберный отрезок Γ(p, q) имеет нижнюю границу, лежащую на Γ(p), и верхнюю границу, лежащую на Γ(q), и не пересекают ни один другой вершинный отрезок.

2.2.4 Полилинейное изображение графа G

Легко конструируется полилинейное изображение планарного *st-графа* G на основе его обзорного представления. Для этого каждая вершина G может быть представлена некоторой точкой соответствующего вершинного отрезка, а каждая дуга (p, q) графа G ломанной, среднее звено которого образовано частью реберного отрезка, изображающего жту дугу (p, q).

Для каждой вершины заменить вершинный отрезок Γ(v) на произвольную (среднюю) точку P(v) = (x(v), y(v)) отрезка Γ(v).

Для каждого ребра (u, v), если это короткое ребро, т.е. расстояние y(v) - y(u) = 1, заменить реберный отрезок Γ(u, v) на отрезок с конечными вершинами P(u), P(v). Иначе, если это ребро длинное, то заменить реберный отрезок Γ(u, v) на ломаную линию, соединяющую точки P(u) и P(v) через точки $(x(U(u,v)), y(u)+1)$ и $(x((u,v)), y(u)-1)$.

Наиболее оптимальное размещение точки P(v) — это середина вершинного отрезка Γ(v). При этом будет получаться плоское восходящее изображение, имеющее не более 6n - 12 сгибов, а каждое ребро имеет не более двух сгибов. При выборе позиции вершины P(v) в случае стратегии "выбора длинного ребра"можно размещать ее на внешней стороне вершинного отрезка и алгоритм будет строить полилинейное изображе-

ние графа в области размера $O(n^2)$ с общим числом вершин $(10*n-31)/3$ и не более, чем с двумя сгибами на одном ребре.

2.3 Описание UML диаграмм с помощью XMI

Для описания диаграммы деятельности используется стандарт XMI. Каждая вершина имеет имя, тип и уникальный идентификатор, описанные как атрибуты, и список входных и выходных вершин. Идентификатор используется для связи между вершинами и переходами. Если вершина имеет тип *action*, то она может содержать описание действий, по которым в дальнейшем будет выстраиваться множество переменных.

Каждый переход имеет строго одну исходящую и входящую вершину и может содержать в себе правила защиты перехода (например, для условного перехода *condition*).

Основываясь на описанных правилах, получим структуру описания диаграммы:

```
<activity_diagram>
    <states>
        <state id, name, type>
            <incoming transitions>
            <outgoing transition>
            <action>
        </state>
    </states>
    <transitions>
        <transition id>
            <source state>
            <target state>
            <guard>
        </transition>
    </transitions>
<activity_diagram>
```

2.4 Преобразование диаграммы деятельности в раскрашенную сеть Петри

Наибольшей сложностью в процессе преобразования диаграммы в раскрашенную сеть Петри является формирование самой раскраски сети. Не существует четких правил, по которым можно определить достаточность раскраски, а так же возникает проблема избыточности, при которой сформированная сеть не будет точно моделировать работу диаграммы или не будет работать вовсе из-за невозможности совершения перехода.

Сам алгоритм построения раскрашенной сети Петри можно разделить на четыре этапа:

1. Выделение списка переменных для каждой вершины (рис. 2.9).
2. Преобразование диаграммы деятельности в простую сеть Петри.
3. Введение начальной разметки и формирование типов для соответствующих переменных. Предварительное определение множества раскрасок (рис. 2.10).
4. Определение максимальной области видимости переменных и формирование результирующей раскраски (рис. 2.11).

Исходя из общих правил формирования множества раскрасок, необходимо каждое составное действие разделить на объект, субъект и само действие (2.12).

Сначала выделяются субъект и объект действия. Если на объект действия накладываются условные ограничения (не всегда может быть доступен, например, канал передачи данных, свободный блок оперативной памяти и т.п.), его необходимо отделить от субъекта действия, так как в сети Петри он будет обеспечивать второе условие для срабатывания перехода действия. Субъект и объект действия в общем случае связаны правилом, в соответствии с которым это действие происходит. [10]

2.4.1 Получение списка переменных

На первом этапе построение раскрашенной сети необходимо рассмотреть множество переменных, описывающих работу диаграммы.

Условимся, что если переменные (в любой части описания переменной) имеют одинаковые имена, то они имеют и одинаковые типы.

На основе этого мы можем построить предварительное описание типов. Предположим, что на диаграмме имеются три блока действия:

```
player.cell = field.cell;
player.resource = player.resoure + field.resource;
player.resource = player.resoure - resource.
```

По этому описанию можно заключить, что в системе фигурируют три переменные: player, field и resource, причем первые две имеют составной тип. Так же получаем то, что типы player и resource имет одинаковое описание типов: $cell, resource$, причем тип всех переменных resource одинаковый. Таким образом, после первого этапа анализа получаем:

```
player : struct { cell, resource }
field : struct { cell, resource }
resource : simple type
```

2.4.2 Введение начальной разметки

На втором этапе пользователю предлагается задать входные данные для диаграммы, основываясь переменных, выделенных на пером этапе. Основываясь на этих значениях, можно построить заключение о типах переменных. Остановим выбор на трех простейших типах: *integer*, *boolean*, *string*, из которых можно образовывать кортежи и сложные структуры.

Для примера, приведенного на первом этапе, построим множество типов:

```
player : [ ((1, 1), 0) ]
field : [ ((1, 2), 2); ((1, 3), 1); ((2, 1), 1); ... ]
resource : 1
```

Учитывая предварительное описание типов, получаем

```
player : struct { cell : (int * int), resource : int }
field : struct { cell : (int * int), resource : int }
resource : int
```

Раскраска характеризуется кортежом типов. Так как каждая позиция содержит список переменных, на этом этапе можно сформировать предварительное множество раскрасок.

При введении начальной разметки значения переменных присваиваются первому вхождению этой переменной на пути из начальной вершины, а для остальных вершин, содержащих эту переменную, значения будут передаваться при функционировании системы с помощью фишек.

2.4.3 Преобразование в простую сеть Петри

Диаграмма деятельности во многом подобна сетям Петри. Но в сетях Петри действия моделируются переходами, а на диаграмме деятельности узлами. Кроме того, если в диаграмме деятельности движение фишки необходимо приостановить, то это делается между узлами на дугах, а не внутри блока. Таким образом, правильный перевод в сеть Петри заменяет узлы на переходы, а дуги на позиции. Узлы диаграммы представляются по-разному, в зависимости от типа (2.13):

Условия для принятия решения в сетях Петри задаются как свойства соответствующих переходов. Позиция, предшествующая ветвлению, имеет два выхода, и маркер позиции перейдет через тот переход, условию которого удовлетворяет его значение. Стоит отметить потенциальную возможность ошибок, предупреждение которых целесообразно рассматривать как одно из формальных правил.

— Если для переходов ветвления не заданы условия их срабатывания, возникает неопределенность, в какую ветку перейдет маркер.

— В случае если ни одно из условий не является истинным, переход становится недоступным — маркер остается в своей позиции.

2.4.4 Наложение раскраски

На последнем этапе необходимо получить результирующую раскраску сети. На втором этапе была введена предварительная раскраска сети, основанная только на множестве переменных, принадлежащих вершине.

Для каждой переменной необходимо найти максимальную область ее видимости — т.е. все возможные пути из вершины, где она встретилась в первый раз, до вершины, после которой она больше не содержится во множестве используемых переменных.

Основываясь на этом, мы расширяем список используемых переменных вершины и меняем ее раскраску.

2.5 Обратная польская запись

При переходе от диаграммы деятельности к раскрашенной сети Петри мы выделяем список переменных, на основе которых строится раскраска. При функционировании моделируемой системы переход сети может менять раскраску используемых фишек, а значит, и производить над ними вычисления. В связи с этим встает задача разбора и вычисления математических выражений. Как правило, арифметические выражения удобно преобразовывать в обратную польскую запись, чтобы избавиться от скобок, содержащихся в выражении. Выражения, преобразованные в польскую запись, можно вычислять последовательно, слева направо.

Отличительной особенностью обратной польской нотации является то, что все аргументы (или операнды) расположены перед знаком операции. В общем виде запись выглядит следующим образом:

— Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.

— Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её

операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.

— Результатом вычисления выражения становится результат последней вычисленной операции.

Автоматизация вычисления выражений в обратной польской нотации основана на использовании стека. Алгоритм вычисления для стековой машины элементарен:

1. Обработка входного символа.
   1.1. Если на вход подан операнд, он помещается на вершину стека.
   1.2. Если на вход подан знак операции, то соответствующая операция выполняется над требуемым количеством значений, извлечённых из стека, взятых в порядке добавления. Результат выполненной операции кладётся на вершину стека.
2. Если входной набор символов обработан не полностью, перейти к шагу 1.
3. После полной обработки входного набора символов результат вычисления выражения лежит на вершине стека.

Реализация стековой машины, как программная, так и аппаратная, чрезвычайно проста и может быть очень эффективной. Обратная польская запись совершенно унифицирована — она принципиально одинаково записывает унарные, бинарные, тернарные и любые другие операции, а также обращения к функциям, что позволяет не усложнять конструкцию вычислительных устройств при расширении набора поддерживаемых операций.

Существует несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. Наиболее распространен переработанный алгоритм, идея которого предложена Э.В. Дейкстрой.

Для хранения переменных используется стек типа char.

Рассматриваем поочередно каждый символ:

1. Если этот символ - число (или переменная), то просто помещаем его в выходную строку.

2. Если символ — знак бинарной операции (+, -, *, /), то проверяем приоритет данной операции. Получив один из этих символов, мы должны проверить стек:

2.1. Если стек все еще пуст, или находящиеся в нем символы имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

2.2. Если символ, находящийся на вершине стека имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие; затем переходим к пункту 2.1.

3. Если текущий символ - открывающая скобка, то помещаем ее в стек.

4. Если текущий символ - закрывающая скобка, то извлекаем символы из стека в выходную строку до тех пор, пока не встретим в стеке открывающую скобку, которую следует просто уничтожить. Закрывающая скобка также уничтожается.

Если вся входная строка разобрана, а в стеке еще остаются знаки операций, извлекаем их из стека в выходную строку.

2.6 Алгоритм построения дерева достижимости

Алгоритм начинает свою работу с определения начальной разметки. До тех пор, пока имеются граничные вершины, они обрабатываются алгоритмом.

Пусть x – граничная вершина, которую необходимо обработать, и с которой связана разметка $\mu(x)$.

а) Если в дереве имеется другая вершина y, не являющаяся граничной, и с ней связана разметка $\mu(y) = \mu(x)$, то вершина x дублируется.

б) Если для разметки $\mu(x)$ ни один из переходов неразрешим, т.е. $\mu(x)$ тупиковая разметка, то x терминальная вершина.

в) Для любого перехода t_j, из множества T разрешенного в разметке $\mu(x)$, создать новую вершину z дерева достижимости. Разметка $\mu(z)$,

связанная с этой вершиной, определяется для каждой позиции p_i следующим образом:

1) если $\mu(x)_i = \omega, \mu(z)_i = \omega$;
2) если на пути от корневой вершины к x существует вершина y такая, что $\mu(y) \rightarrow^{t_j} \mu(x), \mu(y) < \mu(x)\mu(y)_i < \mu(x)_i$, то $\mu(z)_i = \omega$;
3) в противном случае $\mu(z)_i = \mu(x)_i$.

Дуга, помеченная t_j, направлена от вершины x к вершине z. Вершина x переопределяется как внутренняя, вершина z становится граничной. Когда все вершины дерева становятся терминальными, дублирующими или внутренними, алгоритм останавливается. Из алгоритма построения дерева достижимости следуют следующие выводы:

— сеть ограничена тогда и только тогда, когда символ ω отсутствует в дереве;

— сеть безопасна, если число фишек в каждой позиции не превышает 1;

— сеть живая, если в дереве отсутствуют циклы и тупики.

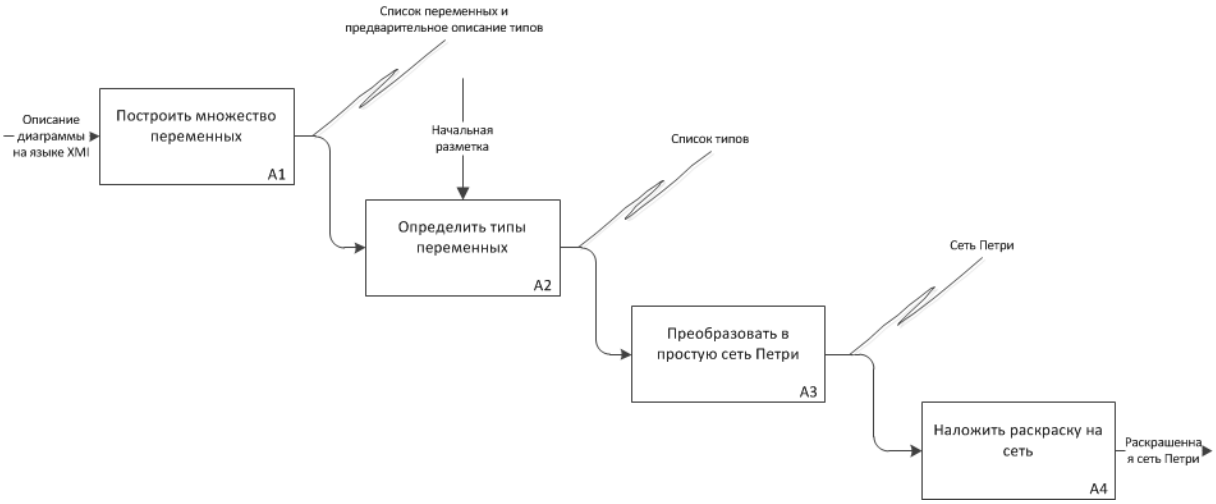
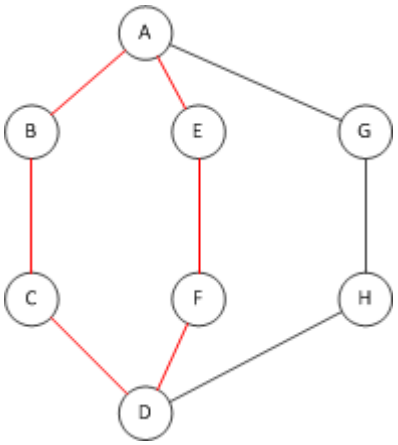
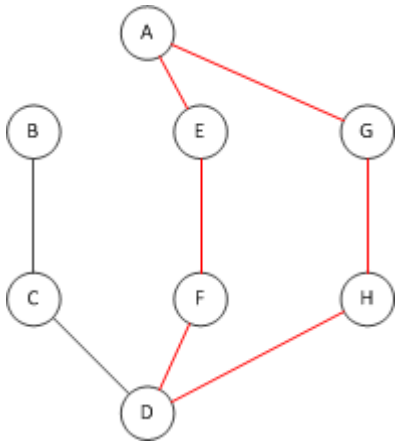


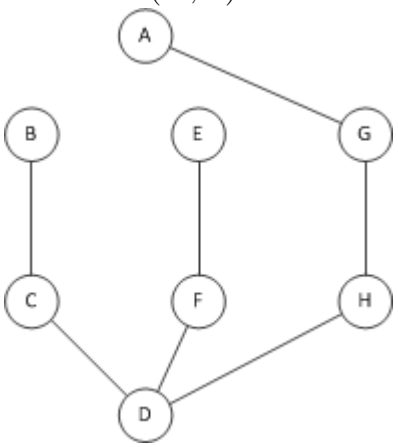
Рисунок 2.2 — Процесс преобразования UML диаграммы деятельности в раскрашенную сеть Петри.



Найден цикл (A,B) $\to$ (B,C) $\to$ (C,D) $\to$ (D,F) $\to$ (F,E) $\to$ (E,A). Удаляется первое ребро цикла (A,B).

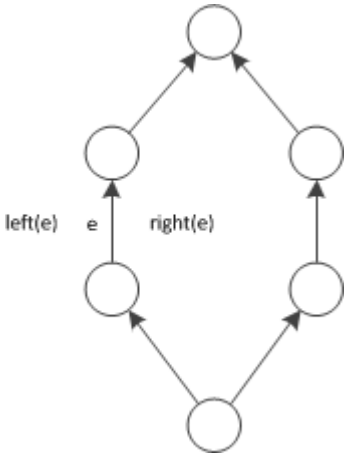


Найден цикл (A,E) $\to$ (E,F) $\to$ (F,D) $\to$ (D,H) $\to$ (H,G) $\to$ (G,A). Удаляется первое ребро цикла (A,E).



Циклов больше нет.

Рисунок 2.3 — Демонстрация работы алгоритма поиска реберных граней.



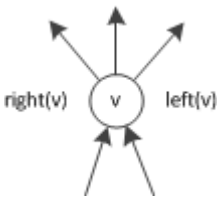
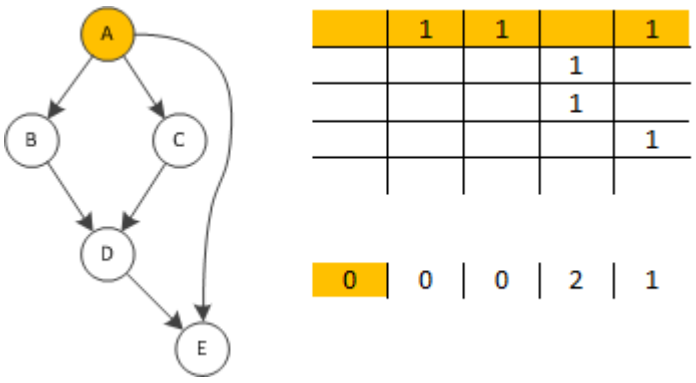
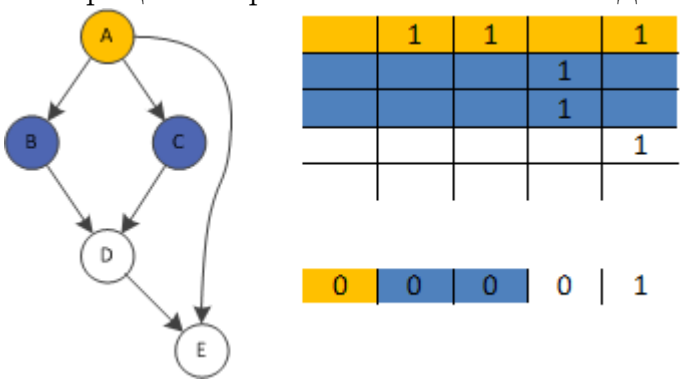


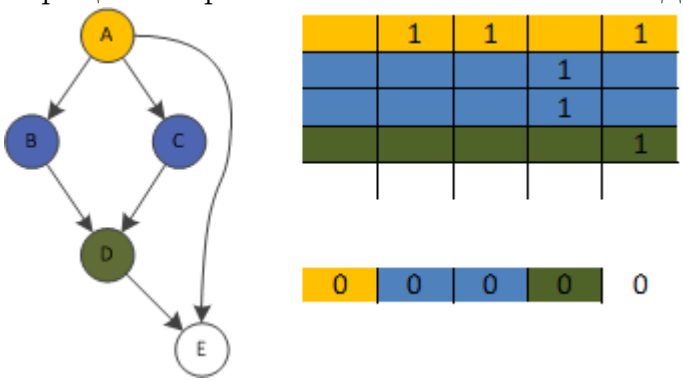
Рисунок 2.4 — Функции *left* и *right* для ориентированного графа.



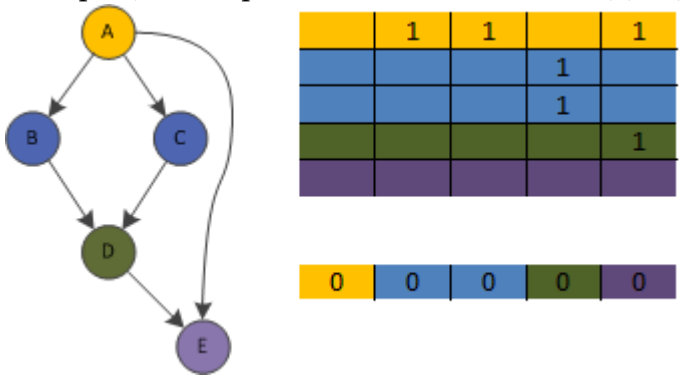
Первая итерация. Вершина A не имеет входящих дуг.



Вторая итерация. Вершины B и C не имеют входящих дуг.



Третья итерация. Вершина D не имеет входящих дуг.



Четвертая итерация. Вершина E не имеет входящих дуг.

Все вершины просмотрены, окончание работы алгоритма.

Рисунок 2.5 — Итеративная работа алгоритма топологической сортировки сети.

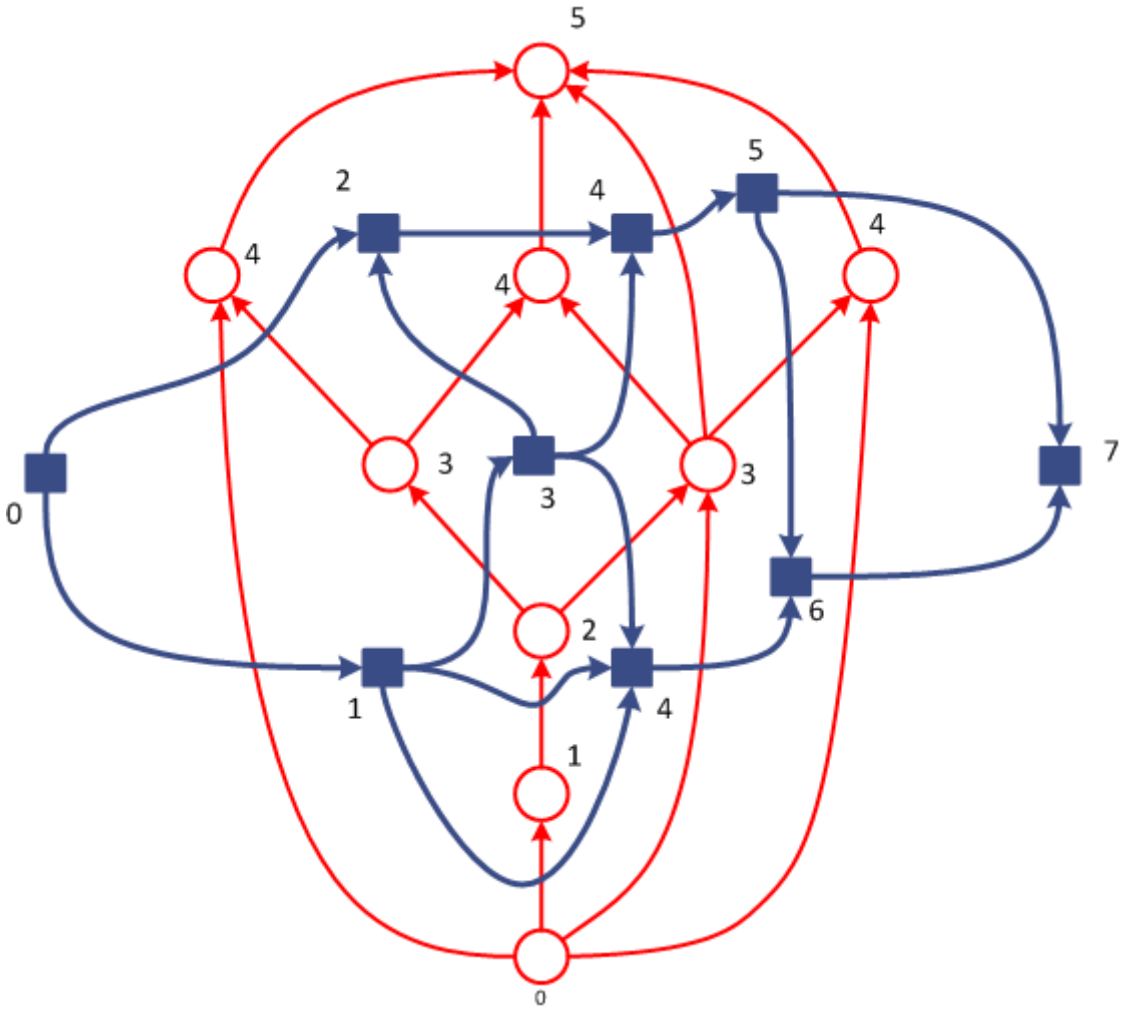


Рисунок 2.6 — Планарный *st*-граф G и ассоциированный граф G*.

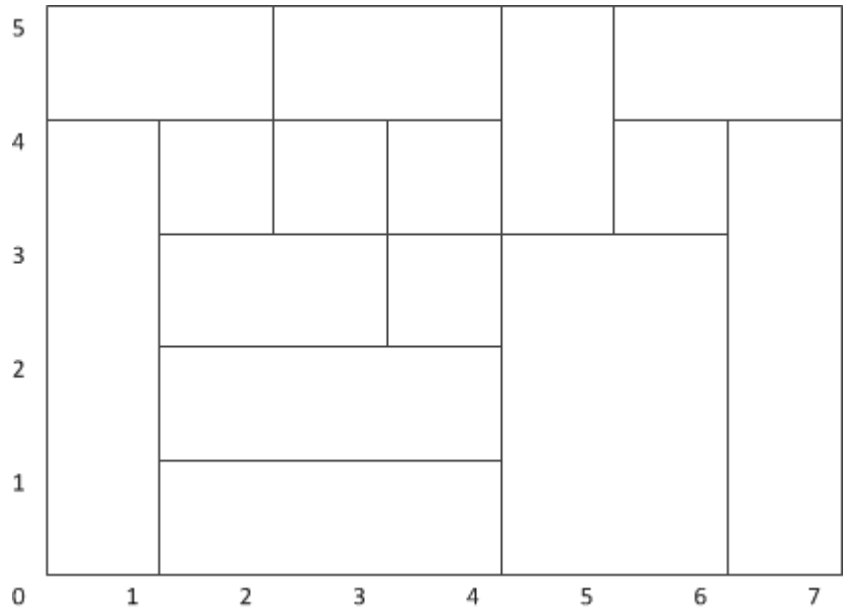


Рисунок 2.7 — Мозаичное представление графа G.

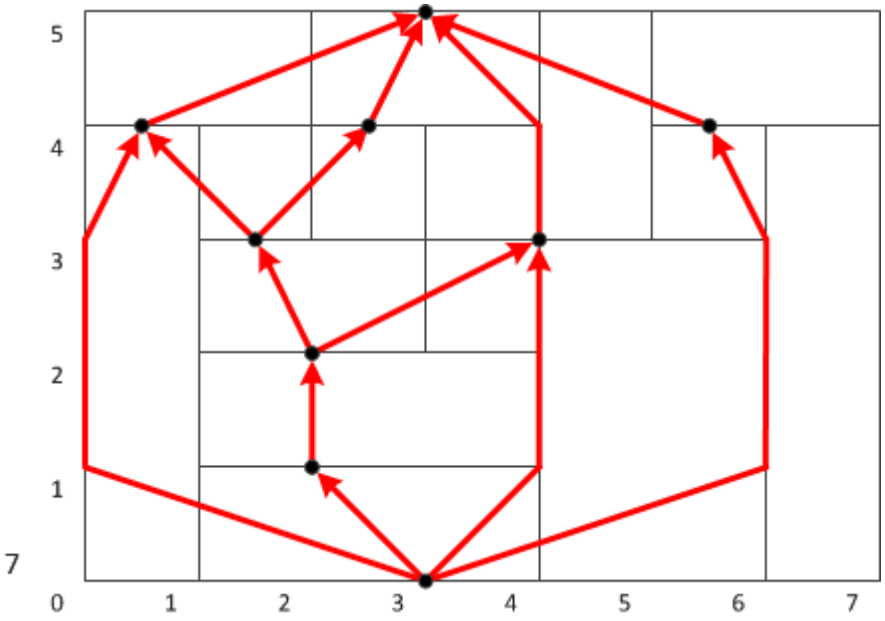


Рисунок 2.8 — Полилинейное изображение графа G.

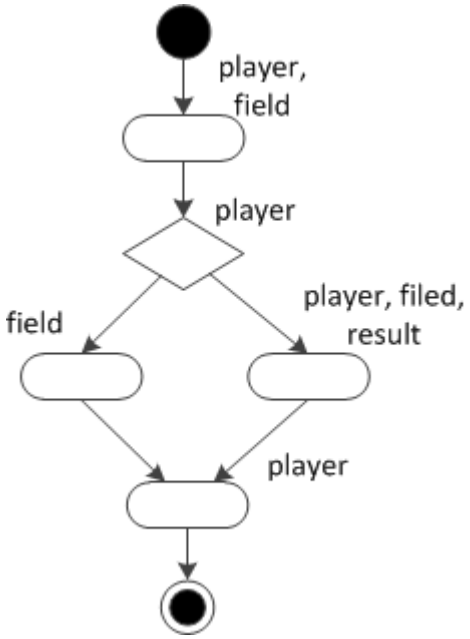


Рисунок 2.9 — Выделение списка переменных.

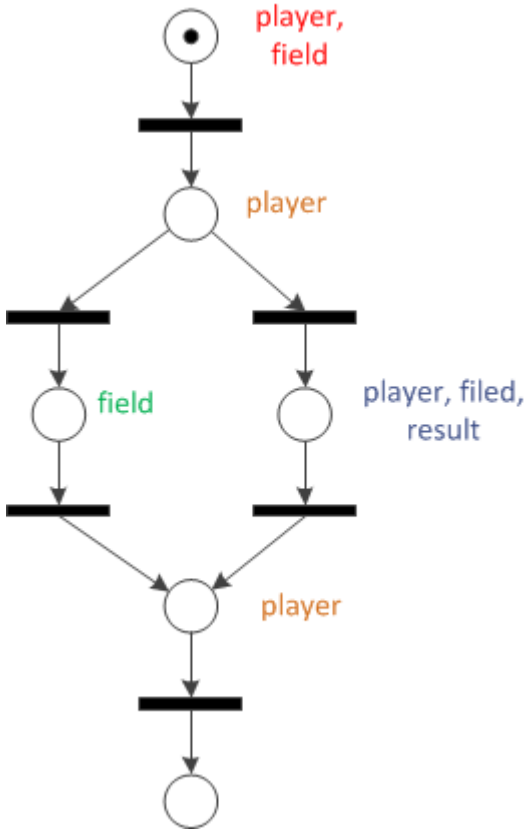


Рисунок 2.10 — Преобразование в простую сеть Петри и предварительное формирование множества раскрасок.

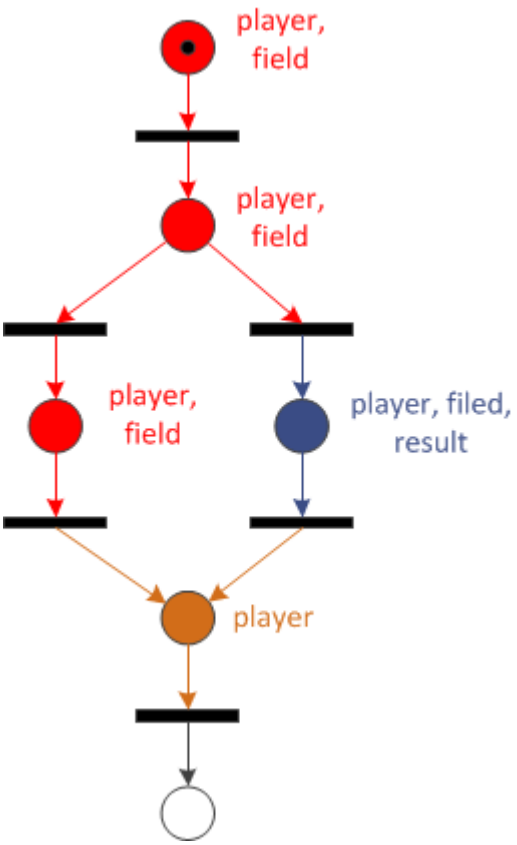


Рисунок 2.11 — Формирование результирующей раскраски.

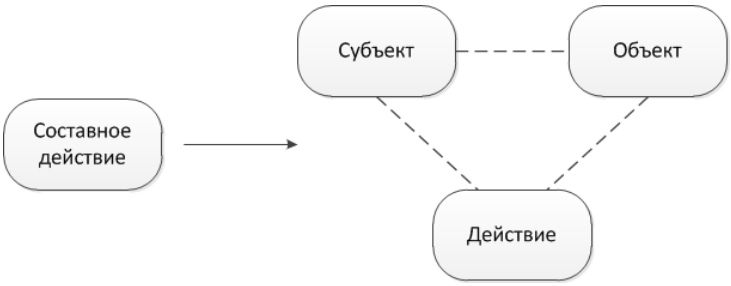


Рисунок 2.12 — Треугольник сложного действия.

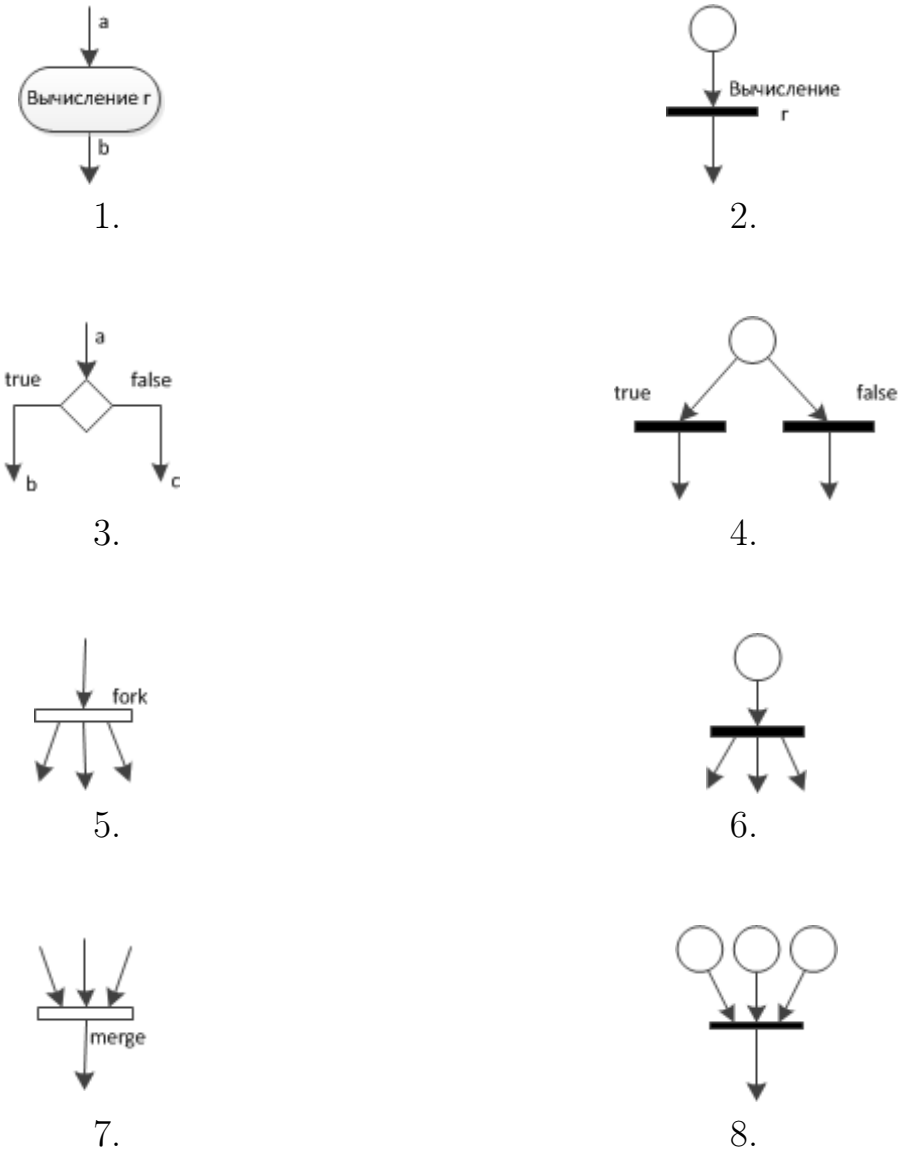


Рисунок 2.13 — Преобразование узлов диаграммы деятельности в сеть Петри. деятельность; 3,4 условие; 5,6 ветвление; 7,8 синхронизация.

3 Технологический раздел

3.1 Выбор и обоснование языка программирования

Для разработки программного комплекса был выбран язык C++ и среда разработки Qt 4.8. Данная среда является кроссплатформеной и обладает огромной библиотекой классов, что существенно упрощает разработку. Для перенесения программы на другую операционную систему требуется лишь ее перекомпиляция. Язык C++ является мощнейшим инструментом разработки: он имеет строгую типизацию и объектно-ориентирован, обеспечивает возможность быстрой разработки приложений, но при этом сохраняет выразительность и элегантность.

3.2 Структура программного комплекса

Внутреннее строение системы имеет модульную структуру, и каждый модуль отвечает за определенные функции. Все модули и классы, реализованные в нем, можно разделить на два типа:

— классы, реализующие интерфейс программы;
— классы, отвечающие за логику работы.

Было принято решение не выделять отдельной сущностью менеджера, т.к. система имеет практически линейный сценарий и к ней не предъявляются требования по сохранению собственного состояния. Вся нагрузка по взаимодействию элементов ложится на класс главной формы. Стоит отметить, что реакция системы на неверные входные данные реализована в виде негативного результата функции, т.к. механизм исключений *try ... catch* не имеет должной поддержки в Qt из-за особенностей реализации фреймворка.

На вход системе поступает XMI-файл, содержащий описание диаграммы. Разбор и построение списка состояний и переходов диаграммы производится в методе *parse* класса *XMLEngine* . Исходя из особенностей структуры XMI разбор файла производится вручную. Для этого, перед основной функцией разбора вызывается метод *validate* класса *XMLValidator*, который проверяет соответствие структуры файла xml-

схеме (см. 4.3). В ходе разбора сначала заполняется список состояний диаграммы *states*, а потом, основываясь на этом списке строиться список переходов *transitions*. Каждое состояние имеет свой уникальный id, по которому происходит связывание текущего перехода transition с исходной и целевой вершиной.

В случае удачного разбора файла происходит отображение диаграммы. Для большей гибкости приложения, само отображение диаграммы на плоскости вынесено в отдельный класс и для визуализации используется паттерн посетитель. Парадигма ядра графического движка Qt основана на представлении отдельных примитивов как объектов и дальнейшим их позиционировании на плоскости.

Каждый потомок абстрактного класса *State* имеет переопределенный метод *diagramItem()*, который создает соответствующий класс, наследуемый от абстрактного *QGraphicsItem*, и отвечающий за отображение этого типа элемента диаграммы.

Диаграмма классов представлена на рисунке 3.1.

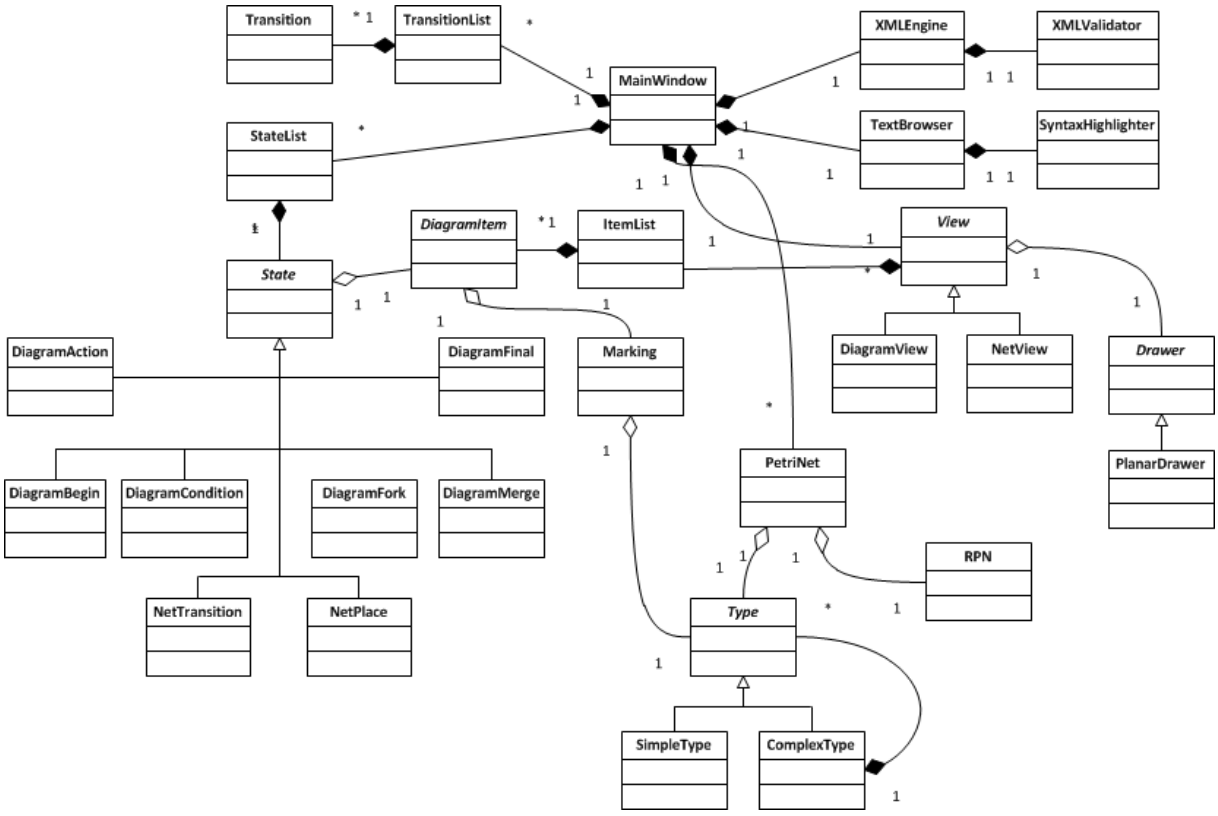


Рисунок 3.1 — Диаграмма классов системы

3.3 Реализация алгоритмов работы системы

3.3.1 Построение списка переменных

Для преобразования в раскрашенную сеть Петри необходимо иметь информацию о переменных, используемых диаграммой. Переменные могут фигурировать в диаграмме в блоках действий или в спусковых функциях переходов. Для разбора переменных будем использовать алгоритм поиска в глубину. Структура данных для описания типов приведена на рисунке 3.2.

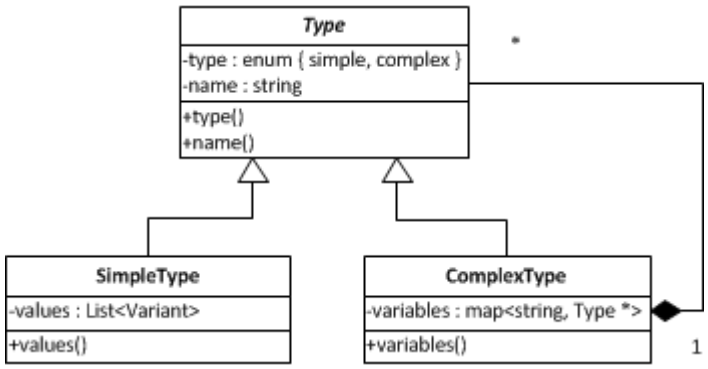


Рисунок 3.2 — Структура классов для описания типов переменных.

Все переменные можно разделить условно на два типа: простые и составные (структурные). Для простых переменных в классе SimpleType используется список сериализованных данных *list<variant>*. Структурный тип, реализованный классом ComplexType, имеет именованный список переменных *map<string, type *>*.

3.3.2 Преобразование в простую сеть Петри

Каждый элемент диаграммы преобразуется в позиции и переходы таким образом, что позиции соединены с переходами, но они не имеют входных дуг, а переходы выходных. Исключения составляют начальное и конечное состояние. Начальное состояние имеет только одну выходную дугу, а значит мы можем его вообще не учитывать, а выходной переход преобразуется только в соответствующую позицию. Так как ограничения переходов могут накладываться только на блок условного перехода, при преобразовании условие преобразуется в спусковую функцию внутрен-

ней дуги между позицией и переходом. Алгоритм преобразования состоит из двух частей:

— преобразование каждого элемента диаграммы в соответствующий набор позиций и переходов сети;

— связывание преобразовыванных элементов друг с другом.

Позиции и переходы являются наследниками абстрактного класса *State*, а значит для их идентификации используется поле *id*. Для каждой позиции идентификатор формируется как *id* преобразовываемого элемента, метка *place* и порядковый номер, если позиций больше одной. Аналогично для переходов. Из-за этого нарушается четкая взаимосвязь между состояниями и переходами сети, т.е. после первого этапа преобразования нельзя четко сказать какая дуга из перехода в позицию будет соответствовать дуге в исходном описании. Необходимо реализовать связь элементов сети, производя поиск незанятых позиций и переходов.

Листинг 3.1 — Алгоритм преобразования в простую сеть Петри

```
queue << states.find(begin)
while !queue.empty():
    State * state = queue.takeFirst()
    if !netStates->find(state->id()):
        convertState(state)
    foreach transition in state.outgoing:
        if !netStates->find(target->id()):
            queue.append(target)
            convertState(target)
        if state->type() != begin_state:
            State * transition, * place;
            List<State *> transitions =
               netStates->find(state->id() + "transition")
            if state->type() = condition_state:
                transition = findFreeTransition(transitions)
            else
                transition = transitions.first()
            List<State *> places = netStates->find(target->id()
              + "place");
            if target->type() == merge_state:
```

```
        place = findFreePlace(places)
    else
        place = places.first()
    Transition * tr = new Transition(state->id() + "-"
      + target->id())
```

3.3.3 Формирование множества активных переходов

В процессе моделирования на каждом шаге возникает задача поиска множества активных переходов. Для решения этой задачи можно последовать двумя путями:

— поиск всех переходов и выделение списка активных переходов;

— поиск всех позиций, содержащих фишки, и на основании этого построение списка возможных переходов.

Так как каждая позиция может иметь несколько исходящих переходов, а каждый переход может иметь несколько входных позиций, задача определения множества активных переходов в общем случае сводится к полному перебору. С учетом специфики преобразования диаграммы, получаем, что из одной позиции может быть несколько переходов только в случае условного перехода и все исходящие переходы для этой позиции имеют лишь одну входящую дугу. Переход может иметь несколько входных позиций только в случае слияния, а значит, все позиции имеют лишь одну исходящую дугу. Для всех остальных случаев каждый переход имеет лишь одну входную дугу, а каждая позиция лишь одну исходящую.

Основываясь на этом, опишем алгоритм формирования множества активных переходов:

Листинг 3.2 — Алгоритм формирования множества активных переходов

```
List<Place *> list, places = states->find(place)
MultiMap<Place *, Transition *> activeTransitions
foreach place in places:
    if place->marking > 0:
        list.append(place)
foreach transition in place->outgoing:
    bool flag = true
```

```
foreach place in transition→incoming:
    if place→marking = 0 and calculate(transition→guard,
       place→marking):
        flag = false
if flag:
    activeTransitions.insert(place, transition)
```

3.4 Формирование раскраски сети

Задача формирования раскраски сети состоит из двух этапов:

— выделение множества цветов на основе используемых переменных;

— определение максимальной области видимости переменных.

Алгоритм опередедения максимальной области видимости переменных реализуется поискомв глубину. Список *track* хранит путь из корня до текущей вершины. Основаная идея алгоритма заключается в том, что если мы для некоторой позции список переменных содержит искомую переменную, то добавляем эту переменную для всех просмотренных вершин из текущей до предыдущей вершины, содержащей эту переменную.

Листинг 3.3 — Алгоритм опередедения максимальной области видимости переменных

```
def path(state, variable, track):
    foreach transition in state→outgoing:
        Place * place = transition→target()
        if variable in place→variables():
            foreach state in track:
                if state→type = place and variable in
                   state→varibles():
                    foreach place from current state to
                       track.last():
                        place→variables << variable
        if target not in track:
            track << target
            path(target, variable, track);
            track remove last
```

3.5 Анализ работоспособности программного комплекса

Рассмотрим базовые элементы UML диаграмм деятельности в аспекте их отражения на элементарную базу сетей Петри.

Пусть имеется квадратное поле 3x3, каждая клетка которого условно содержит некоторое количество ресурсов. На одной из клеток изначально находится игрок, способный перемещаться за один ход на одну клетку в любую сторону (рис. 3.3).

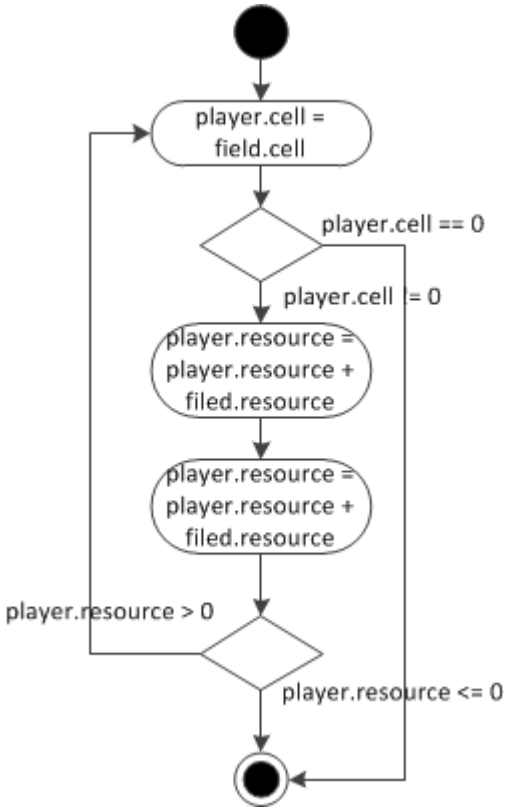


Рисунок 3.3 — Диаграмма деятельности, представляющая алгоритм решения поставленной задачи.

Цикл работы данной симуляции состоит из следующих шагов:

1. выбор смежной клетки, которая еще не была посещена. Если таковых не осталось, игра заканчивается, иначе переход к пункту 2;

2. перемещение в выбранную клетку;

3. ресурсы, сопоставленные с этой клеткой, становящиеся достоянием игрока, т. е. прибавляемые к некоторому счетчику, изначально нулевому. Однако на перемещение затрачивается некоторое фиксированное

количество ресурсов. Если суммарное накопление ресурсов стало отрицательным, игрок не может продолжать движение и игра останавливается.

Для анализа рабоспособности метода проведем сравнение результатов моделирования задачи с помощью разработанного метода и моделирующей программы CPN Tools.

Раскрашенная сеть Петри, построенная в среде CPN Tools, приведена на рисунке 3.4.

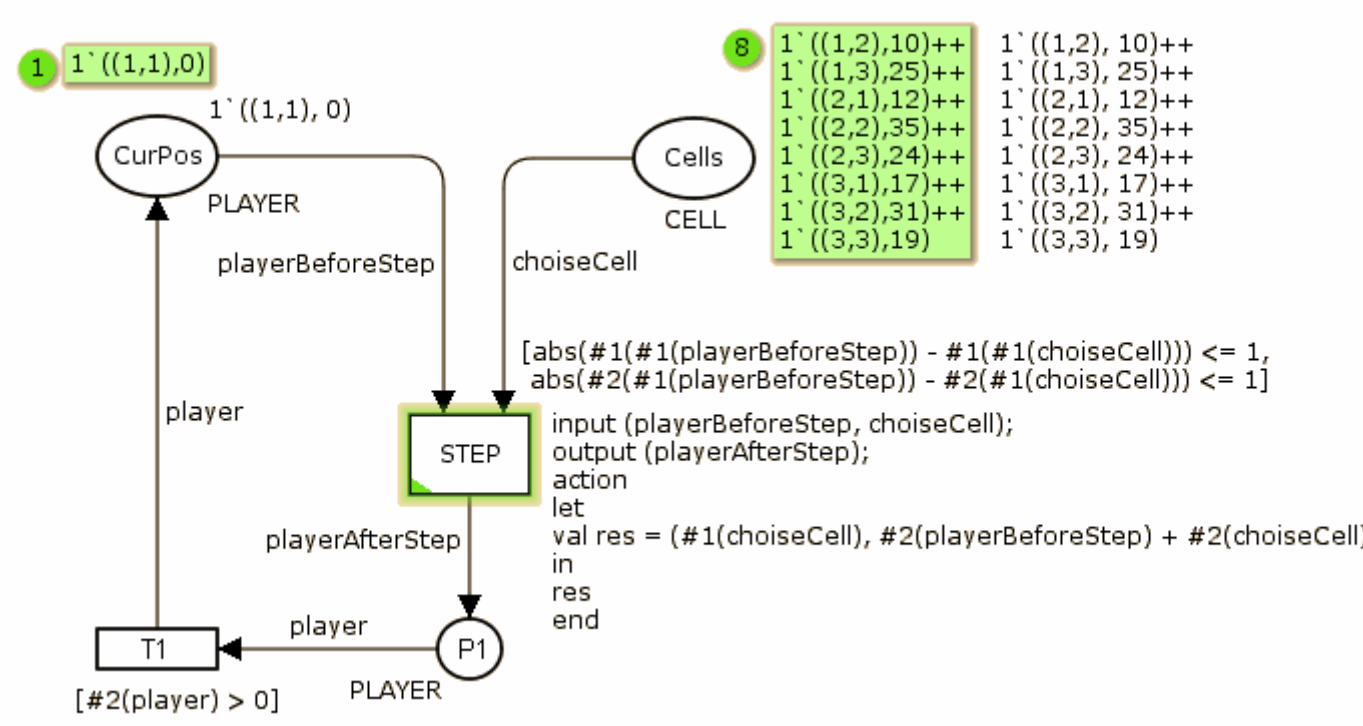


Рисунок 3.4 — Раскрашенная сеть Петри в CPN Tools.

Для обработки модели в CPN Tools вводятся следующие типы данных:

```
colset POSITION = product INT * INT
colset PRICE = INT
colset CELL = PRODUCT POSITION * PRICE
colset PLAYER = product POSITION * PRICE
```

Тип POSITION обозначает пару целых чисел, предназначенных для обозначения номера строки и столбца клетки, тип PRICE — целое

значение количества ресурсов клетки, тип CELL обозначает клетку и включает в себя позицию и цену, тип PLAYER введен для обозначения игрока и хранит в себе его текущие позицию и количество собранных ресурсов. На следующем этапе объявляются дуговые переменные:

```
playerBeforeStep, playerAfterStep, player : PLAYER
choiseCell : CELL
```

В curPos заданы начальный маркер, обозначающий игрока в позиции (1;1), и обнуленный счетчик накопленных ресурсов. В позиции Cells определено восемь маркеров, соответствующих прочим клеткам игрового квадрата, каждая клетка представлена своими координатами и предопределенным уровнем ресурсов. Переход STEP обозначает действие: выбор смежной клетки и получения ее ресурсов. Для данного перехода задано следующее условие срабатывания:

```
[abs(#1(#1(playerBeforeStep)) - #1(#1(choiceCell))) <= 1
 abs(#2(#1(playerBeforeStep)) - #2(#1(choiceCell))) <= 1]
```

Конструкция 1(x) означает «взять первый элемент в x». Таким образом, условие осуществляет проверку: следующая клетка, выбранная для перемещения, смежная с текущей позицией игрока. Благодаря этому условию игрок перемещается в случайном порядке, за один ход преодолевая один квадрат. После каждого хода количество маркеров в позиции Cells уменьшается на единицу, поскольку переход STEP их изымает. Это соответствует тому, что уже посещенные клетки недоступны для перемещения. Переход STEP содержит следующую подпрограмму:

```
input(playerBeforeStep, choiseCell);
output(playerAfterStep);
action
    let
    val res = (#1(choiseCell), #2(playerBeforeStep) +
    + #2(choiseCell) - 1)
    in
```

```
    res
end
```

Здесь описаны входные (игрок и доступная клетка) и выходные параметры (игрок, совершивший ход). Алгоритм подпрограммы прост: в маркер игрока записывается позиция выбранной клетки, а к его счетчику ресурсов добавляется количество ресурсов этой клетки и вычитается 15 (число, взятое за цену перемещений).

Стоит отметить, что не каждая из возможных последовательность выбора следующей клетки при оговоренных ограничениях приведет к посещению всех клеток. Симуляция сети Петри в CPN Tools позволяет построить пространство возможных состояний и увидеть, какие ходы оптимальны, какая стратегия выбора более выигрышная.

При преобразовании диаграммы с помощью разработанного метода автоматически выделяется список переменных для каждого состояния, и у пользователя запрашивается ввод их исходных значений:

```
player : struct { cell, resource } = [ ((1, 1), 0) ]
field : struct { cell, resource } =
[ ((1, 2), 10); ((1, 3), 25); ((2, 1), 12); ((2, 2), 35);
  ((2, 3), 24); ((3, 1), 17); ((3, 2), 31); ((3, 3), 19) ]
resource : simple type = 15
```

На основе алгоритма опеределения максимальной области видимости переменных строится результирующая раскраска графа (рис. 3.5)

В процессе моделирования мы получаем, что фишка приходит в конечное состояние и все фишки сети были использованы. Это означает отсутствие блокировок и недостижимых состояний сети, а значит корректность исходных данных для поставленной задачи.

В отличии от CPN Tools, где построением сети занимается пользователь, разработнная система выстраивает сеть автоматически, для этого лишь требуется полное описание диаграммы деятельности и начальные значения переменных.

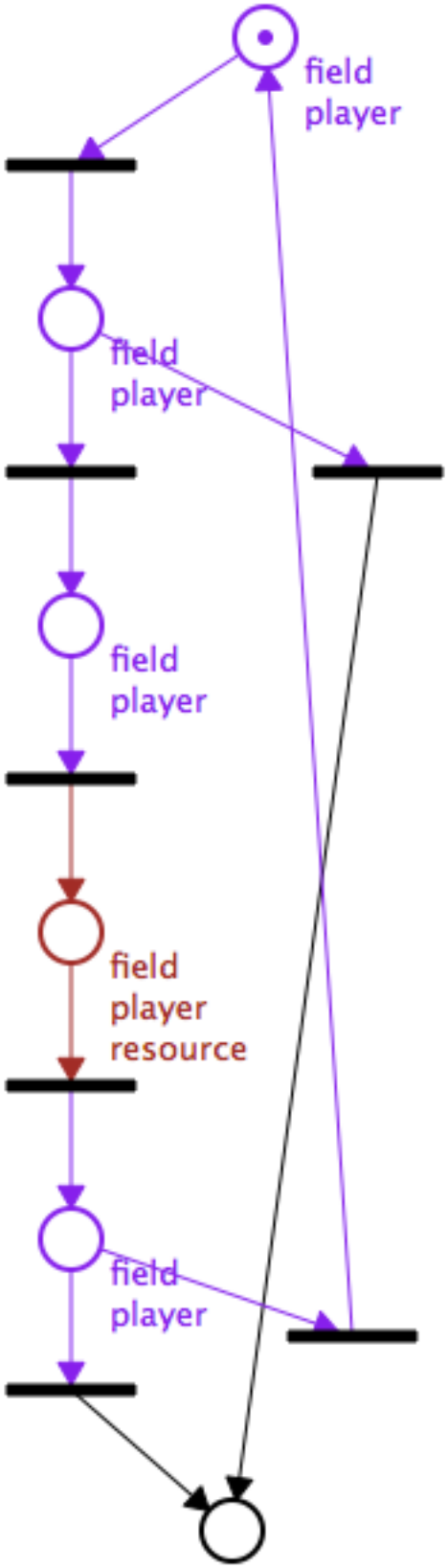


Рисунок 3.5 — Раскрашенная сеть Петри, полученная с помощью разработнного метода.

4 Экспериментальный раздел

4.1 Исследование появления блокировок

Основной задачей исследования сетей Петри является выявление недоступных состояний и блокировок. Выявим причины, по которым они могут возникнуть в диаграммах деятельности. В процессе исследования представим диаграмму деятельности в виде простой и раскрашенной сети Петри и сравним результаты.

В диаграммах деятельности блокировки могут возникать по причине:

— неверной структуры диаграммы;

— невозможности перехода из-за невыполнения логического условия спусковой функции.

4.1.1 Простая сеть Петри

Так как простые сети Петри могут моделировать лишь поток выполнения, для них возможно выявления лишь структурных несоответствий диаграммы.

Блокировки в сетях Петри могут быть вызваны лишь ситуацией, когда переходу принадлежит два и более места, но не все места имеют фишки. Применительно к диаграмме деятельности, только блок ожидания завершения параллельного процесса может иметь больше одной входной дуги во внутреннем переходе. Таким образом, для получения блокировки внутри блоков многопоточной обработки должен находиться условный переход (рис. 4.1, 4.2).

Ситуация блокировки, изображенная на рисунке 4.1 искусственная, т.к. по логике выход за пределы параллельных процессов невозможен. Так же, при моделировании такой ситуации получается, что метки C и D являются небезопасными, и в них будет происходить бесконечное накопление фишек.

Ситуация, изображенная на рисунке 4.2 возможна и в результате в системе возникает блокировка, т.к. переход никогда не сработает. Для избежания такой ситуации используется введение дополнительной вер-

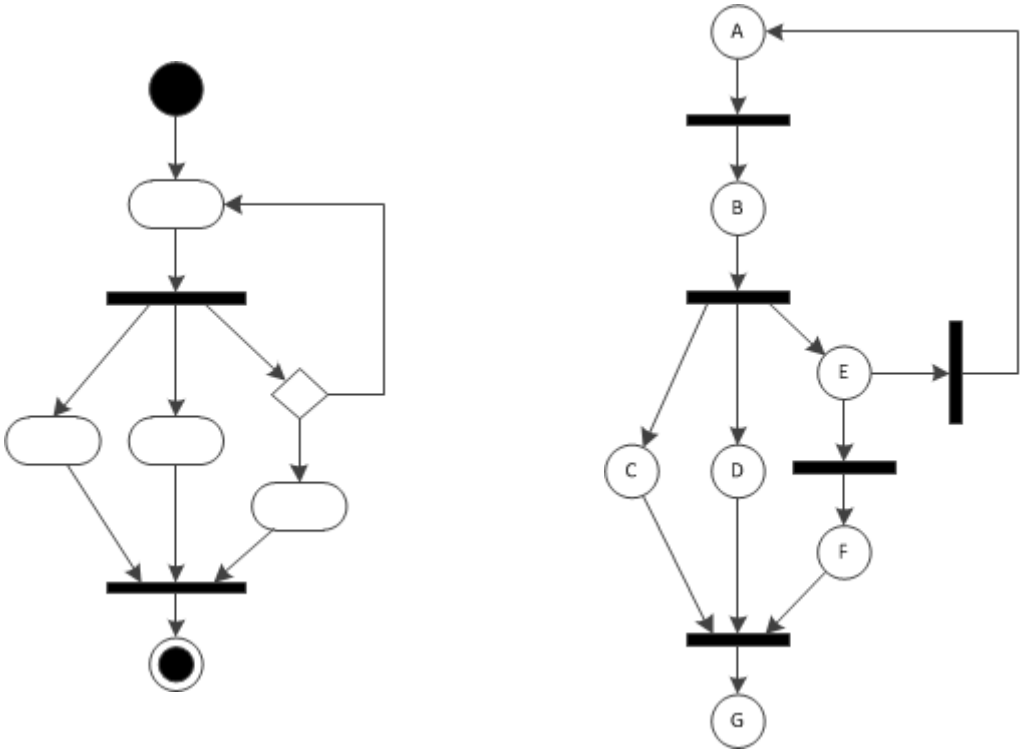


Рисунок 4.1 — Блокировка из-за выхода за пределы блоков.

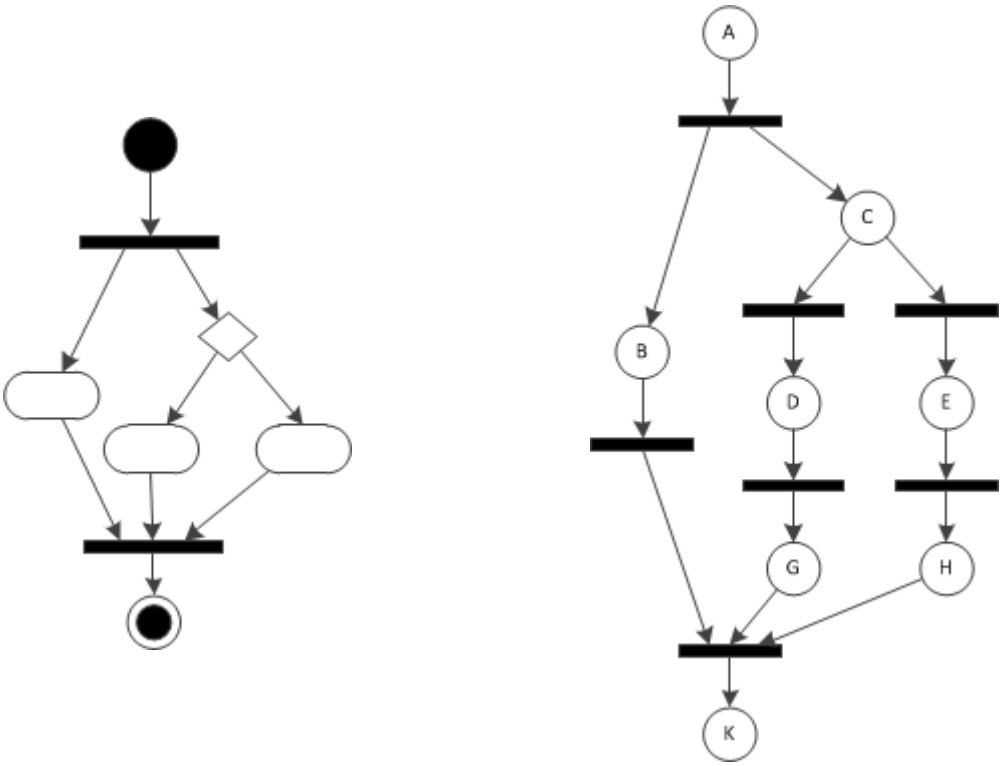


Рисунок 4.2 — Блокировка из-за неверной структуры внутреннего условия.

шины, из которой будет единственная дуга в блок синхронизации (рис. 4.3).

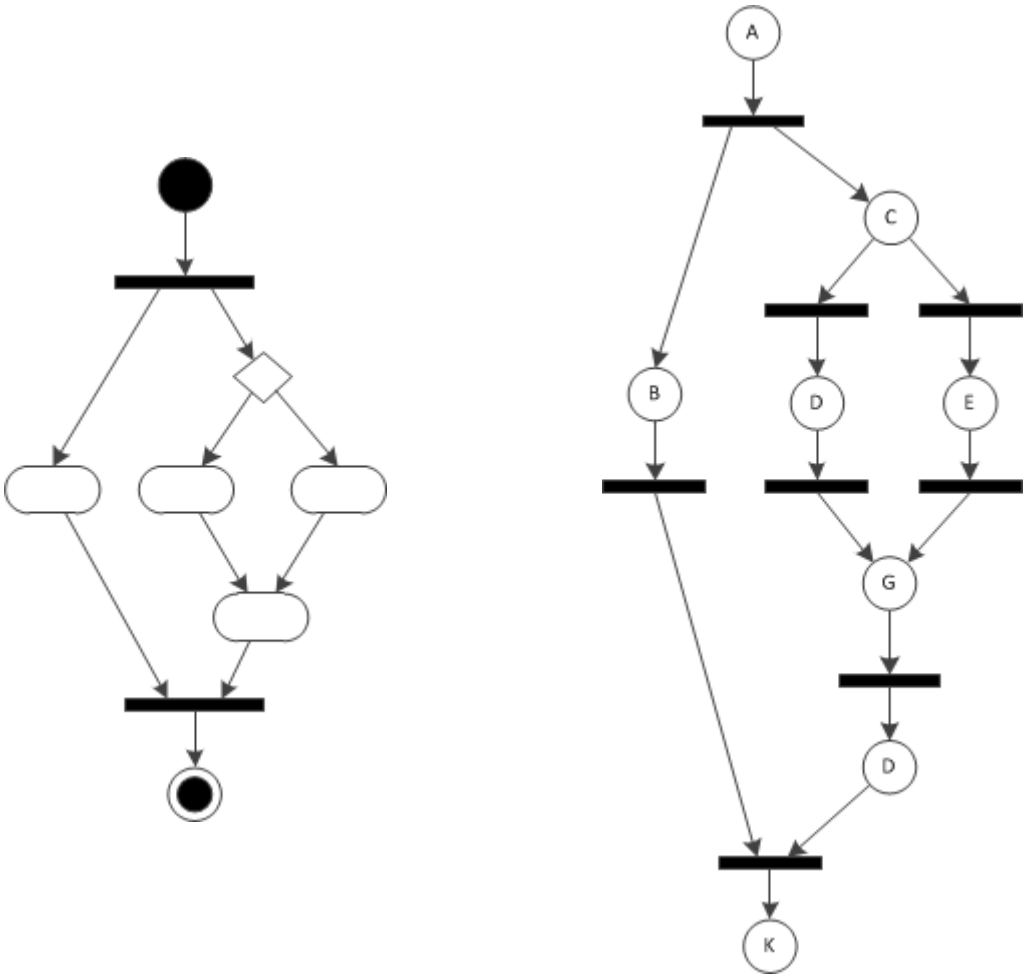


Рисунок 4.3 — Корректная структура внутреннего условия.

4.1.2 Раскрашенная сеть Петри

Диаграмма деятельности имеет практически линейную структуру, за исключением вершины условного перехода, которая при преобразовании в простую сеть будет давать два возможных пути выполнения, т.к. переходы имеют равный приоритет и в простой сети срабатывание перехода определяется лишь наличием фишки в позиции. При наличии некоторого цикла в диаграмме, анализ с помощью простой сети Петри выдаст лишь его наличие.

Моделирование работы диаграммы деятельности с помощью раскрашенных сетей Петри приводит к тому, что пути процесса выполнения будут иметь вид линейную структуру, а не древовидную. Блокировка в

сети может быть вызвана не только отсутствием фишки в позиции, но и не срабатыванием спусковой функции из-за невыполнения логического условия.

По этой причине, раскрашенная сеть Петри не всегда способна выявить некорректную структуру диаграммы, т.к. при конкретной начальной разметке, для которой построена сеть, не все переходы будут активны.

4.2 Исследование корректности построения раскраски сети

Предложенный метод формирования раскраски сети на основе определения максимальной видимости переменных в ходе исследования работоспособности метода дал хорошие результаты, но при этом были выявлены ограничения, связанные со структурой преобразуемой диаграммы.

Если в диаграмме имеется хотя бы один цикл, то при наложении раскраски возникает вопрос об определении области видимости переменных. В связи с этим получается два подхода:

1. определять раскраску всех элементов цикла как кортеж переменных наибольшей мощности (рис. 4.4);
2. определять раскраску не учитывая обратную дугу цикла (рис. 4.5).

Так как фишка хранит в себе набор переменных, сопоставленных ее цвету, то при реализации первого подхода на переходах внутри цикла будут меняться лишь значение переменных. Такой подход будет вести к некорректному моделированию работы, т.к. если какая либо позиция в цикле содержит несколько фишек, то в процессе моделирования будет использована лишь одна из них, и весь процесс работы будет привязан лишь к значениям переменных.

При реализации второго подхода в сети может случиться блокировка, т.к. фишка из раскраски меньшей мощности будет переходить к более мощной раскраске в позиции не содержащей фишек новой раскрас-

ки. Другими словами, при моделировании работы диаграммы на рисунке 4.5 не сможет сработать переход t_6 даже если будет выполнено логическое условие, ограничивающее этот переход, т.к. мощность раскраски позиции p_6, содержащей фишку, меньше мощности позиции p_2.

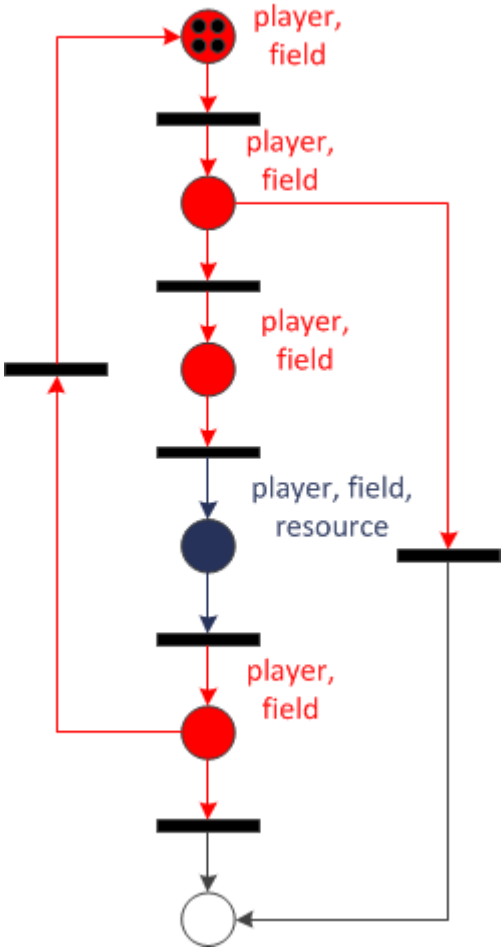


Рисунок 4.4 — Сеть с раскраской, наложенной с учетом циклов.

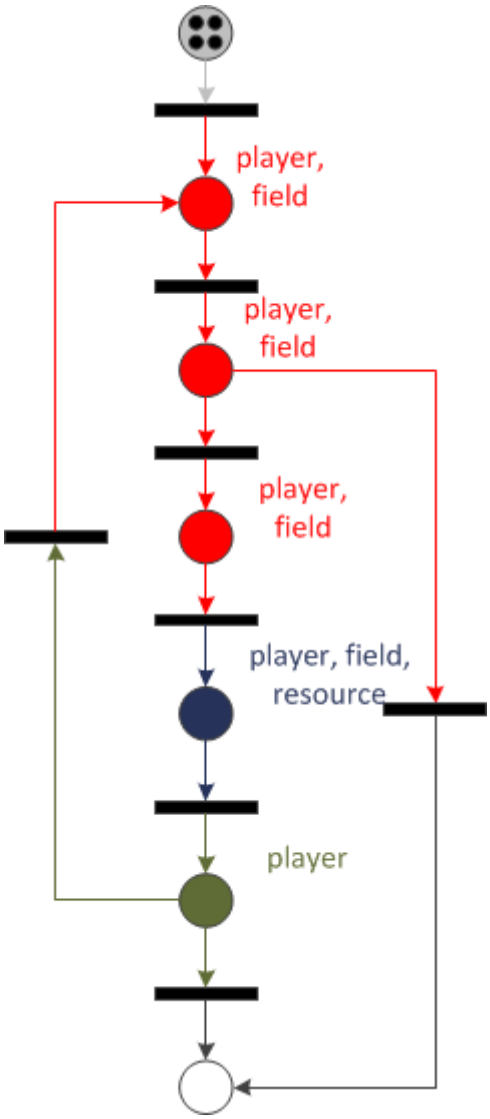


Рисунок 4.5 — Сеть с наложением раскраски без учета циклов.

С учетом вышесказанного было принято решение использовать первый подход, но при нахождении в позиции с ненулевой разметкой заменять фишку, содержащуюся в маркере на новую. Это решение позволяет избежать некорректности работы модели, т.к. в процессе будут использоваться все фишки сети.

4.3 Сравнение скорости работы простой и раскрашенной сети Петри

Проведем сравнение времени работы и затраченных ресурсов для моделирование работы диаграммы деятельности, представленной простой и раскрашенной сетью Петри.

В сравнении времени моделирования учитывалось только время преобразования диаграммы деятельности в сеть и проведение анализа. Результаты сравнения показаны в 4.1.

Таблица 4.1 — Сравнение времени моделирования.

	8 состояний	15 состояний	25 состояний
Простая сеть Петри	0.012 сек.	0.015 сек.	0.020 сек.
Раскрашенная сеть Петри	0.42 сек.	0.67 сек.	0.114 сек.

В результате исследования мы получили, что время, затраченное для моделирования раскрашенной сеть Петри значительно больше. Это объясняется тем, что на этапе преобразования диаграммы помимо самого преобразования в сеть Петри происходит так же анализ множества используемых переменных и построение раскраски. Так же во время моделирования для раскрашенной сети на каждом переходе происходит вычисление новых значений переменных, а для каждой дуги из позиции в переход вычисляется спусковая функция.

Для подсчета затраченных ресурсов использовался профилировщик кода VTune. Профилировщик запускал моделирующую программу и по окончанию моделирования выводился список затраченных ресурсов для каждого объекта программы.

Таблица 4.2 — Сравнение затраченных ресурсов.

	8 состояний	15 состояний	25 состояний
Простая сеть Петри	250KB	340KB	585KB
Раскрашенная сеть Петри	345KB	420KB	730KB

В результате исследования было выявлено, что моделирование диаграммы деятельности в виде простой и раскрашенной сети Пери тре-

бует приблизительно одинаковое количество ресурсов. Небольшая разница объясняется тем, что в раскрашенной сети для каждой позиции хранится список используемых переменных и цвет, а в переходах и на внутренних дугах содержится выражения.

Заключение

В рамках данно проекта был разработан и реализован метод преобразования диаграммы деятельности в раскрашенную сеть Петри, позволяющий выявить блокировки и недостижимые состояния. В результате работы над проектом были проделаны следующие этапы:

1. проведена классификация существующих методов анализа диаграмм деятельности;
2. реализован алгоритм отображения диаграмм деятельности;
3. разработан метод представления диаграммы деятельности в виде раскрашенной сети Петри;

Было проведено исследование причин появления блокировок в диаграммах деятельности и применимость простой и раскрашенной сетей для задачи их выявления. Так же было проведено исследование подходов к построению раскраски сети и на основе этих результатов было принято решение о реализации метода.

Список использованных источников

1. *Д., Паронджанов В.* Теоретические основы языка ДРАКОН / Паронджанов В. Д. — М.: Новая книга, 1995.

2. *Ермаков И. Е., Жигуненко Н. А.* Двумерное структурное программирование; класс устремлённых графов. (Теоретические изыскания из опыта языка «ДРАКОН» / Жигуненко Н. А. Ермаков И. Е. // Сборник трудов V Международной конференции «Инновационные информационно-педагогические технологии в системе ИТ-образования». — М.: МГУ, 2010. — Pp. 452–461.

3. *Тюгашев, А. А.* Графические языки программирования и их применение в системах управления реального времени / А. А. Тюгашев. — Самара: Научный центр РАН, 2009.

4. *В.Е., Котов.* Сети Петри / Котов В.Е. — М.: Наука, 1984.

5. *Дж., Питерсон.* Теория сетей Петри и моделирование систем / Питерсон Дж. — М.: Мир, 1984.

6. *Jensen K., Kristensen L.M.* Coloured Petri nets. Modelling and validation of concurrent systems / Kristensen L.M. Jensen K. — N.Y.: Springer Dordrecht Heidelberg, 1998.

7. *В., Шахов.* Моделирование программно-аппаратных «реактивных» систем раскрашенными сетями Петри / Шахов В. // *RSDN Magazine.* — 2006. — no. 3.

8. *Kristensen L.M. Christensen S., Jensen K.* The practitioner's guide to Coloured Petri Nets / Jensen K. Kristensen L.M., Christensen S. — N.Y.: Springer, 1998.

9. *Battista G. Garg A., Liotta G. Tamassia R. Tassinari E. Vargiu F.* An experimental comparison of four graph drawing algorithms / Liotta G. Tamassia R. Tassinari E. Vargiu F. Battista G., Garg A. — Computational Geometry, 1997. — Pp. 303–325.

10. *С.В., Коротиков.* Применение сетей Петри в разработке программного обеспечения центров дистанционного контроля и управления: Ph.D. thesis / НГТУ. — 2007.

XML-схема

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified">
    <xs:element name="activity_diagram">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="states"/>
                <xs:element ref="transitions"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="states">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" ref="state"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="transitions">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" ref="transition"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="state">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="expression"/>
                <xs:element ref="incoming"/>
                <xs:element ref="outgoing"/>
            </xs:choice>
```

```
        <xs:attribute name="id" type="xs:string"/>
        <xs:attribute name="idref" type="xs:string"/>
        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="type" type="xs:string"/>
    </xs:complexType>
</xs:element>
<xs:element name="expression" type="xs:string"/>
<xs:element name="incoming">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded" ref="transition"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="outgoing">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded" ref="transition"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="transition">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="guard"/>
            <xs:element ref="source"/>
            <xs:element ref="target"/>
        </xs:choice>
        <xs:attribute name="id" type="xs:string"/>
        <xs:attribute name="idref" type="xs:string"/>
    </xs:complexType>
</xs:element>
<xs:element name="guard" type="xs:string"/>
```

```
    <xs:element name="source">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element ref="state"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="target">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element ref="state"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```