Министерство образования и науки Российской Федерации

Государственное учебно-научное учреждение

Факультет вычислительной математики и кибернетики

Московского государственного университета имени М.В. Ломоносова

(Факультет ВМК МГУ имени М.В. Ломоносова)

УДК	УТВЕРЖДАЮ
№ госрегистрации	декан
Инв. №	академик РАН
	____________ Е.И. Моисеев
	«___» ____________ 2012 г.

Государственный контракт от «20» сентября 2010 г. № 14.740.11.0399

Шифр заявки «2010-1.1-215-138-007»

ОТЧЕТ
О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

в рамках федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009-2013 годы

по теме:

«СОЗДАНИЕ ПРОТОТИПА ИНТЕГРИРОВАННОЙ СРЕДЫ И МЕТОДОВ КОМПЛЕКСНОГО АНАЛИЗА ФУНКЦИОНИРОВАНИЯ РАСПРЕДЕЛЁННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ (РВС РВ)»

(промежуточный, этап № 4)

Наименование этапа: «Разработка второй очереди методов и инструментальных средств, интеграция средств»

Руководитель работ, д.ф.-м.н. _______________ Смелянский Р.Л.

профессор подпись, дата

Москва 2012

Обозначения и сокращения

АГЭА	Адаптивный гибридный эволюционный алгоритм
БПМ	Библиотека поддержки моделирования
ВС	Вычислительная система
МОО	Механизм обеспечения отказоустойчивости
НИР	Научно-исследовательская работа
ПНМ	Полунатурное моделирование
РВ	Реальное время
РВС РВ	Распределённая вычислительная система реального времени
ТЗ	Техническое задание
ЭА	Эволюционный алгоритм

API	Интерфейс программирования приложений (Application Programming Interface)
CRC	Центральный компонент RTI (Central RTI Component)
DOM	Программный интерфейс для доступа к документу (Document Object Model)
FOM	Федеративная объектная модель (Federation Object Model)
HLA	Высокоуровневая архитектура (High Level Architecture)
HTA	Иерархический временной автомат (Hierarchical Timed Automata)
LRC	Локальный компонент RTI (Local RTI Component)
OMT	Шаблон объектной модели федерации (Object Model Template)
RTI	Среда имитационного моделирования (RunTime Interface)
SCXML	Расширяемый язык разметки для диаграмм состояний (State Chart eXtensible Markup Language)
SOM	Имитационная объектная модель (Simulation Object Model)
TCP	Протокол управления передачей (Transmission Control Protocol)
UDP	Протокол пользовательских дейтаграмм (User Datagram Protocol)
UML	Универсальный язык разметки (Universal Markup Language)
WAN	Глобальная компьютерная сеть (Wide Area Network)
WCET	Наихудшее время выполнения программы (Worst-case Execution Time)

XMI	Расширяемый язык разметки для обмена метаданными (eXtensible Markup Language for Metadata Interchange)
XML	Расширяемый язык разметки (eXtensible Markup Language)

Реферат

Основной целью данной НИР является разработка прототипа интегрированной программной среды с открытыми исходными кодами для поддержки разработки и интеграции РВС РВ через моделирование, а также методов количественного и качественного анализа функционирования РВС РВ. Выполнение НИР должно обеспечивать достижение научных результатов мирового уровня, подготовку и закрепление в сфере науки и образования научных и научно-педагогических кадров, формирование эффективных и жизнеспособных научных коллективов.

Основной целью четвёртого этапа НИР была разработка второй очереди методов и инструментальных средств, интеграция средств. Основное содержание работ по четвёртому этапу следующее: разработка второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ; экспериментальное исследование второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ; интеграция разработанных методов и инструментальных средств; подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 36 академических часов.

Результатом работы по четвёртому этапу является: промежуточный отчёт о НИР за четвёртый этап. Промежуточный отчёт о НИР за четвёртый этап включает в себя: описание разработанных второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ; описание результатов их экспериментального исследования; описание интеграции разработанных методов и инструментальных средств; разработанные учебные материалы.

Все задачи, поставленные в рамках четвёртого этапа НИР, выполнены.

Содержание

Введение ... 7

1 Разработка второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ ... 8

1.1 Модификация среды выполнения моделей, совместимых со стандартом HLA ... 9

1.2 Модификации транслятора UML в модели, совместимые с HLA ... 24

1.3 Обзор схем трассировки моделей и разработка средства трассировки моделей ... 29

1.4 Обоснование корректности алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов ... 37

1.5 Минимизация временных автоматов ... 48

1.6 Модификация транслятора UML в UPPAAL ... 54

1.7 Алгоритм восстановления параметров модели по контрпримеру в UPPAAL ... 58

1.8 Обзор методов оценки наихудшего времени выполнения и реализация метода оценки наихудшего времени выполнения ... 65

1.9 Обзор моделей процессоров для оценки наихудшего времени выполнения ... 75

1.10 Метод решения задачи выбора механизмов обеспечения отказоустойчивости для РВС РВ. ... 82

1.11 Разработка средства внесения неисправностей ... 90

2 Интеграция разработанных методов и инструментальных средств ... 98

2.1 Интеграция среды моделирования со средствами оценки WCET ... 98

2.2 Интеграция средств моделирования со средствами построения расписаний и синтеза архитектур ... 100

2.3 Интегрированная среда разработки и анализа моделей ... 109

3 Экспериментальное исследование второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ ... 119

3.1 Модель поведения бортовых компьютеров автомобилей ... 119

3.2 Эксперименты по сравнению сред моделирования РВС РВ ... 135

3.3 Эксперименты с модифицированным транслятором UML в исполняемые модели, совместимые со стандартом HLA ... 146

3.4 Эксперименты по восстановлению параметров модели по контрпримеру в UPPAAL. 154

3.5 Эксперименты со средствами оценки наихудшего времени выполнения ... 162

3.6 Эксперименты со средствами оптимизации надежности РВС РВ 165
3.7 Эксперименты со средствами трассировки моделей и внесения неисправностей.. 169

4 Подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 36 академических часов 174

Заключение 176

Список использованных источников 178

Введение

Настоящий документ представляет собой научно-технический отчет по четвёртому этапу НИР «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)». Документ содержит отчет по пунктам 4.1-4.4 календарного плана, в соответствии с техническим заданием (ТЗ) по государственному контракту № 14.740.11.0399 от 20 сентября 2010 г. между Государственным учебно-научным учреждением Факультет вычислительной математики и кибернетики Московского государственного университета имени М.В. Ломоносова и Министерством образования и науки Российской Федерации.

В первой главе в соответствии с пунктом 4.1 календарного плана ТЗ приводится описание разработанных второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ.

Во второй главе в соответствии с пунктом 4.3 календарного плана ТЗ приводится описание интеграции разработанных методов и инструментальных средств.

В третьей главе в соответствии с пунктом 4.2 календарного плана ТЗ описываются результаты экспериментального исследования второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ.

В четвёртой главе в соответствии с пунктом 4.4 календарного плана ТЗ приводятся описание разработанных учебных материалов.

В заключении изложены основные результаты четвёртого этапа НИР.

1 Разработка второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ

В данном разделе описываются разработанные на четвёртом этапе средства, входящие в состав второй очереди методов и инструментальных средств поддержки анализа и разработки распределённых вычислительных систем реального времени (РВС РВ), либо описываются доработки средств, разработанных на предыдущих этапах.

На предыдущих этапах данной работы были выбраны и доработаны следующие средства [1],[2],[3]:

- редактор UML-диаграмм;
- среда выполнения моделей, совместимых со стандартом HLA;
- средство визуализации трасс;
- средство верификации.

Для того, чтобы было возможно использовать эти средства совместно были разработаны:

- средство трансляции из UML в исполняемые модели, совместимые со стандартом HLA;
- средство трансляции из UML во временные автоматы.

Как было показано на предыдущих этапах работы, разработанные средства необходимо было модифицировать, а также необходимо было внести изменения в среду выполнения моделей, совместимых со стандартом HLA, для того чтобы ускорить скорость работы этих средств.

Стандарт HLA предполагает, что отдельные имитационные модели (или несколько имитационных моделей), предназначенные для использования в одном приложении, могут быть легко использованы в другом приложении, если их разработчики придерживаются концепции федератов. Федерат – это одна или набор взаимодействующих систем имитационных моделей. Природа федератов может быть весьма разнообразной (программные системы, тренажёры). Федераты объединены в федерации.

Исходя из стандарта HLA, необходимо для каждой имитационной модели отдельно описать несколько десятков интерфейсов для взаимодействия с Runtime Infrastructure (RTI). Под RTI понимают среду передачи данных между компонентами, входящими в модель.

Следовательно, помимо затрат на построение самой модели, разработчику необходимо дополнительно заниматься описанием интерфейсов для взаимодействия с RTI.

Кроме уже разработанных средств необходимо было на этом этапе разработать и включить в состав среды моделирования следующие средства:

- средство восстановления параметров модели по контрпримеру в UPPAAL;
- средство оценки наихудшего времени выполнения программы;
- средство синтеза архитектуры РВС РВ с учётом требований на надёжность;
- средство внесения неисправностей.

В разделе 1.1 приводится описание модификации среды выполнения моделей, совместимых со стандартом HLA. В разделе 1.2 описываются модификации средства трансляции из UML-диаграмм в исполняемые модели, совместимые со стандартом HLA. Раздел 1.3 содержит обзор схем трассировки моделей и описание выбранных вариантов для реализации. В разделе 1.4 приведено обоснование корректности алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов. Раздел 1.5 содержит описание модификаций средства трансляции из UML-диаграмм во временные автоматы. В разделе 1.6 приведён алгоритм восстановления параметров модели по контрпримеру в UPPAAL. Раздел 1.7 содержит обзор методов оценки наихудшего времени выполнения и описание реализации метода оценки наихудшего времени выполнения программы. В разделе 1.8 приводится обзор моделей процессоров для оценки наихудшего времени выполнения. Раздел 1.9 содержит описание метода синтеза архитектуры РВС РВ с учётом требований на надёжность. В разделе приводится описание средства внесения неисправностей.

1.1 Модификация среды выполнения моделей, совместимых со стандартом HLA

На предыдущих этапах настоящей работы [1],[2],[3] в качестве базы для построения среды выполнения на основе стандарта распределённого моделирования HLA [4] была выбрана система CERTI [5]. Основными преимуществами CERTI по сравнению с существующими аналогами являются открытый исходный код, активная поддержка проекта, большое количество целевых платформ, а так же реализация на языке C++, характеристики которого позволяют решать задачи в реальном времени [6].

Одной из главных проблем, стоящих на пути создания системы моделирования реального времени на базе системы CERTI является её низкая производительность: по результатам проведённых на предыдущем этапе тестов, CERTI значительно уступает специализированной среде полунатурного моделирования Стенд ПНМ [3]. Анализ CERTI [7] позволил выявить несколько серьёзных архитектурных недостатков и сформулировать предложения для их преодоления. В результате сопоставления предполагаемого прироста производительности с объёмом трудозатрат, необходимых для реализации каждой идеи, было выбрано изменение модели потоков управления CERTI как наиболее приоритетное направление доработки CERTI.

На предыдущем этапе работы было проведено исследование существующих моделей потоков управления, используемых различными системами моделирования, основанными на стандарте HLA. По его результатам наиболее актуальной была признана асинхронная модель, реализация прототипа (MT-CERTI) которой была начата на предыдущем и завершена на текущем этапе работы. В данном разделе рассматриваются теоретические, алгоритмические и технические вопросы, связанные с реализацией данного прототипа.

Кроме того, настоящий раздел содержит подробное описание идеи построения системы моделирования с каскадной архитектурой, совмещающей преимущества простой и понятной централизованной модели управления с более производительной моделью peer-to-peer. Дополнительно приводится анализ сильных и слабых сторон предложенной архитектуры. Раздел содержит приблизительные оценки трудозатрат на построение системы с подобной архитектурой на основе CERTI и разработку необходимых вспомогательных средств поддержки моделирования.

1.1.1 Описание архитектуры CERTI

Любая реализация RTI HLA является распределённой программной системой и обеспечивает множество подключённых к ней участников моделирования стандартным интерфейсом HLA, скрывая при этом детали их взаимодействия, включая сетевое. Эта цель достигается благодаря построению RTI, как совокупности удалённых компонентов, соответствующих федератам. Таким компоненты работают локально на той же машине, что и федерат, и обычно называются *локальными компонентами RTI* (Local RTI Component, LRC) [8].

Инфраструктура RTI постоянно должна поддерживать согласованность отдельных компонентов имитационной модели с помощью встроенных механизмов синхронизации.

Поэтому эффективность синхронизации оказывает значительное влияние на общие показатели производительности системы. Полностью распределённая архитектура RTI предполагает равнозначность и самодостаточность её локальных компонентов LRC, а её реализация требует использования сложных распределённых алгоритмов согласования. Поэтому разработчики обычно отказываются от полностью распределённой архитектуры и вводят понятие *центрального компонента RTI* (Central RTI Component, CRC), который хранит разделяемые данные выполняемой модели и производит синхронизацию участников моделирования локально. Обе модели, как централизованная, так и децентрализованная, имеют свои сильные и слабые стороны, и их взвешенное и обоснованное совмещение является краеугольным камнем любой эффективной реализации RTI [9].

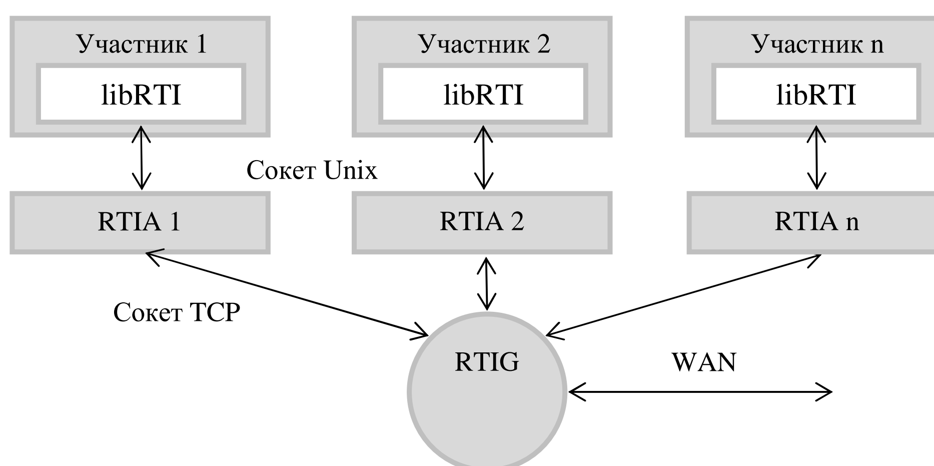

Рисунок 1- Архитектура CERTI RTI

В общем виде архитектура CERTI RTI может быть представлена в виде совокупности компонентов трёх типов [5]: RTI Gate (RTIG), RTI Ambassador (RTIA) и libRTI (Рисунок 1). Процесс RTIG может выполняться на отдельном вычислительном узле, к которому не подключён ни один участник моделирования, и является центральным компонентом CRC системы CERTI. Процесс RTIA, напротив, выполняется на том же узле, что и участник моделирования. Таким образом, число запущенных процессов RTIA во время выполнения модели равно числу участников моделирования. Взаимодействие между процессами RTIG и RTIA происходит через сокет TCP.

В то время как процессы RTIG и RTIA формируют «внутренности» CERTI RTI, библиотека времени выполнения libRTI непосредственно реализует интерфейс стандарта HLA. Библиотека libRTI линкуется с процессом участника моделирования и отвечает за его

соединение с соответствующим процессом RTIA через канал передачи данных операционной системы (Unix Socket). Таким образом, локальный компонент LRC системы CERTI состоит из процесса RTIA и библиотеки libRTI.

Процессы RTIA никогда не взаимодействуют друг с другом напрямую – весь обмен данными идёт через процесс RTIG. Таким образом, система CERTI имеет полностью централизованную архитектуру, и её центральный компонент CRC единолично управляет выполнением имитационной модели. Процесс RTIG реализует большую часть сервисов и служб RTI, в то время как единственная цель процесса RTIA и библиотеки libRTI – создание удобной коммуникационной инфраструктуры между центральным процессом RTIG и процессами федератов. Другими словами, LRC системы CERTI служит, главным образом, каналом передачи данных между её CRC и участниками моделирования.

1.1.2 Модификация LRC

Исходная схема LRC CERTI

Итак, компонент LRC [8] системы CERTI состоит из библиотеки libRTI и процесса RTIA, соединённых сокетом Unix. Хотя библиотека libRTI и линкуется с процессом федерата, обеспечивая его интерфейсом к службам и сервисам HLA, эта библиотека не реализует логику инфраструктуры RTI. Вместо этого библиотека просто перенаправляет получаемые от федерата запросы присоединённому к ней процессу RTIA. Более точно, при каждом вызове одного из сервисов RTI библиотека пересылает процессу RTIA сообщение с идентификатором вызванного метода и набором поступивших при этом аргументов. Процесс RTIA обрабатывает поступившее сообщение и отвечает федерату.

Таким образом, библиотека libRTI является интерфейсной частью LRC системы CERTI, а процесс RTIA реализует внутреннюю логику приложения. Благодаря такому разделению компонента LRC на независимые модули, возможные изменения спецификаций стандарта HLA не способны повлиять на внутреннюю логику компонента LRC, и соответствующие изменения инфраструктуры моделирования потребуют минимальных трудозатрат. На данный момент система CERTI использует описанную возможность для одновременной поддержки сразу двух версий стандарта HLA: более старого DMSO 1.3 и более нового IEEE 1516 2000 [5].

Другим преимуществом двухкомпонентного LRC перед монолитным является его большая надежность и защищённость. При двухкомпонентной организации библиотека libRTI и процесс RTIA выполняются независимо друг от друга и проверяют каждое

поступающее к ним сообщение. Поэтому федерат никак не может прочитать или изменить внутренние данные инфраструктуры моделирования RTI, кроме как через стандартные интерфейсы HLA. По этой же причине отказ внутри федерата сам по себе не может стать причиной отказа всей системы моделирования.

К сожалению, гибкость модульного компонента LRC снижает общие показатели производительности системы. Рассмотрим передачу сообщений между федератами с точки зрения процессов CERTI. Федерат-отправитель вызывает соответствующий метод библиотеки libRTI. Библиотека отправляет их процессу RTIA в виде сообщения через сокет Unix. Процесс RTIA анализирует сообщение и переправляет полученные данные процессу RTIG сообщением через сокет TCP. Затем RTIG передаёт данные федерату-получателю, используя аналогичную цепочку процессов в обратном порядке. При этом важно отметить, что передача каждого сообщения через сокет требует предварительной сериализации передаваемых данных и их последующей распаковки. Таким образом, передача одного сообщения между федератами приводит к передаче четырёх сообщений и восьми конвертаций формата данных. Первый федерат отправляет сообщение своему RTIA, RTIA передаёт его процессу RTIG, далее цепочка повторяется в обратном порядке. Конвертация данных требуется при каждой передаче.

Схема построения LRC с использованием потоков

На предыдущем этапе работы было выдвинуто предложение об объединении процесса RTIA и процесса федерата в один единственный процесс – процесс RTIA будет выполняться в контексте процесса федерата как самостоятельный поток управления [3]. При такой организации процессов RTIA и libRTI будут выполняться в общем контексте, поэтому информация между ними может передаваться как обычный указатель. Тем самым исключается необходимость механизма сообщений, приводящего к дополнительной конвертации и излишнему копированию данных, что позволяет снизить расходы на передачу данных между федератами почти вдвое.

Аналогичных результатов можно достичь и с помощью разделяемой памяти, однако организация взаимодействия на уровне потоков позволяет использовать более эффективные механизмы синхронизации, чем существующие на уровне взаимодействия полновесных процессов. В частности, потоки способны более эффективно использовать доступные вычислительные ресурсы.

До недавнего времени все процессоры были одноядерными, и использование обычных процессов с единственным потоком управления не приводило к существенному

недоиспользованию производительности процессора. Однако с появлением многоядерных архитектур, способных эффективно выполнять несколько потоков одновременно, потери относительно пиковой производительности стали гораздо более ощутимыми [10]. В следующем разделе рассматривается альтернативная реализация среды выполнения MT-CERTI (Multi-Threaded CERTI), локальные компоненты которой основаны на механизме потоков.

Прямые и обратные вызовы

Один из разделов стандарта HLA IEEE 1516-2000 описывает спецификации интерфейсов и правила взаимодействия среды выполнения RTI с подключёнными к ней федератами [4]. В нём вводятся понятия *прямого* и *обратного вызова*. Прямой вызов происходит при обращении федерата к RTI, то есть во время использования этим федератом одним из сервисов инфраструктуры RTI. Обратный вызов, наоборот, происходит при обращении инфраструктуры RTI к одному из методов подключённого к ней федерата. Например, один из методов федерата вызывается при получении им сообщения или при его уведомлении об успешности предшествующего прямого вызова.

Разработчики стандарта HLA стремились создать такую среду выполнения, которая бы накладывала как можно меньше ограничений на подключаемых к ней федератам. Согласно этой логике федерат может быть написан с использованием единственного потока управления, и тем более его разработчик не должен заботиться о возможных состояниях гонки, которые могут возникнуть в случае одновременного доступа к одним и тем же данным во время обратного вызова. Поэтому инфраструктура RTI никогда не обращается к федерату, пока он сам это обращение не разрешит.

В итоге разработчики HLA предусмотрели два принципиально различных способа получения обратных вызовов: обратиться к методу RTI, который совершит однократный обратный вызов внутри себя, или же разрешить асинхронные вызовы. В последнем случае разработчик федерата берёт ответственность за корректное выполнение программы на себя. При этом инфраструктура RTI может вызывать один из методов федерата в одном из собственных потоков управления в произвольный момент времени.

Асинхронная модель управления

Прямые и обратные вызовы на практике могут быть реализованы несколькими различными способами в зависимости от модели потоков управления, лежащей в основе конкретной RTI. Текущая версия RTI CERTI использует в своей реализации только полновесные процессы, поэтому вопросы выбора подходящей модели потоков управления

для неё не возникают в настоящем отчёте. Однако данные вопросы приобретают актуальность в свете того, что предложенная модификация CERTI предполагает замену полновесного процесса RTIA отдельным потоком управления в контексте процесса федерата.

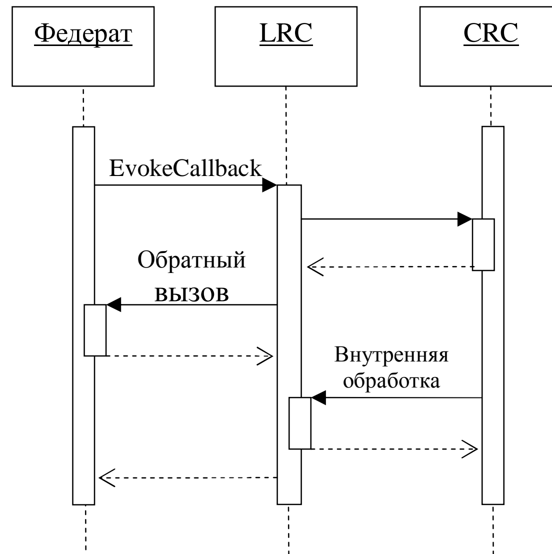

Рисунок 2 - . Схема работы асинхронной модели управления

На предыдущемм этапе настоящей работы было рассмотрено три используемых в модели потоков управления: однопоточная, асинхронная и многопоточная [8]. Наиболее подходящей из них для целей данного проекта с точки зрения эффективности модели и требуемых для её реализации затрат оказалась асинхронная модель потоков, успешно применяющаяся, в частности, в RTI от компании Pitch – pRTI [11].

В асинхронной модели (рисунок 2) инфраструктуры RTI использует один или несколько внутренних потоков управления, которые самостоятельно обеспечивают взаимодействие с удалёнными компонентами инфраструктуры. Поэтому разработчику федерата не нужно заботиться о *постоянном* и *своевременном* выделении процессорного времени на нужды RTI, что было бы критично в случае выполнения RTI и федерата в одном единственном потоке. В результате, асинхронная модель не требует дополнительных трудозатрат при изменении условий решаемой задачи.

Кроме того, асинхронная модель потоков управления наиболее похожа на модель процессов, используемую в системе CERTI. Процесс федерата в ней отвечает за прямые и обратные вызовы, а процесс RTIA – за корректную работу инфраструктуры RTI. В результате, асинхронная модель потребует минимального количества изменений в коде, поэтому именно она была выбрана для реализации в рамках прототипа.

1.1.3 Реализация MT-CERTI

Структура библиотеки

В рамках данного этапа настоящей работы был создан прототип многопоточной версии системы моделирования CERTI – Multi-Threaded CERTI (MT-CERTI). Внесённые при этом изменения в исходный код были оформлены в виде соответствующего патча, и представлены разработчикам системы. В момент написания настоящего отчёта идёт согласование произведённых модификаций и рассматривается возможность их внесения в основную ветку разработки.

Изменение поточной модели CERTI потребовало изменений не только в исходном коде системы моделирования, но и в её структуре. Результатом проведённых модификаций стала разработанная на данном этапе библиотека libRTI (название библиотеки оставлено прежним, так как оно определено стандартом HLA [4].). С одной стороны, новая библиотека включает код одноимённой библиотеки CERTI и может по-прежнему линковаться с федератом. С другой стороны, она запускает процесс RTIA как дополнительный поток управления в процессе связанного с ней федерата. При этом любое взаимодействие оригинальной библиотеки libRTI и RTIA реализуется внутри библиотеки libRTIA на уровне её потоков управления.

Многокомпонентная процессная архитектура оригинальной системы CERTI обладает некоторыми преимуществами по сравнению с разработанной на данном этапе версией с многопоточной архитектурой. Например, при совместном моделировании устройств конкурирующих организаций многопоточная инфраструктура RTI, к которой подключены точные модели разрабатываемых устройств, может стать источником получения закрытых данных, и привести к нечестному конкурентному преимуществу.

В связи с этим, внесение изменений в код системы CERTI было сделано опциональным. Для реализации опциональности к параметрам системы сборки был добавлен специальный конфигурационный флаг RTIA_USE_THREAD. Если флаг задан со значением false или не указан, то собирается традиционная версия системы моделирования, в случае значения true – разработанная многопоточная версия библиотеки.

Выбранная асинхронная модель потоков управления во многом схожа с существующей процессной моделью CERTI, поэтому большую часть кода процесса RTIA удалось использовать повторно. С нуля потребовалось разработать лишь несколько

вспомогательных модулей. Внесённые же в существующий исходный код модификации проявляются лишь при объявлении макроопределения, которое отсутствует по умолчанию и автоматически создаётся системой сборки при поднятии флага RTIA_USE_THREAD.

Для работы с потоками управления была использована сторонняя библиотека boost_thread с открытым исходным кодом и свободной лицензией [12]. Данная библиотека написана на языке C++ и, фактически, является стандартным решением для кроссплатформенной организации работы с потоками. На момент написания отчёта boost_thread поддерживает более широкий диапазон систем, чем CERTI (Windows, Linux, MacOS, MinGW, и так далее). Поэтому код, написанный с её помощью, не сужает первоначальной области применения данной системы моделирования. Таким образом, все дополнительные затраты, связанные с использованием boost_thread, сводятся к необходимости её линковки к оригинальным библиотекам системы CERTI.

Организация обменов данными между процессом RTIA и федератом

В традиционной версии системы CERTI обмен данными между процессом RTIA и соответствующим ему федератом происходит через локальный сокет Unix и выполняется в синхронном режиме: процесс RTIA записывает в канал своё сообщение только в ответ на сообщение процесса федерата. На уровне модели потоков управления аналогичная функциональность может быть организована с использованием одной разделяемой переменной. Действительно, каждый из потоков может записывать в неё указатель на своё сообщение и ждать, пока там не появится указатель на сообщение от другого процесса. Однако описанная организация взаимодействия не позволяет реализовать полностью асинхронную модель, в которой процесс федерата и соответствующий ему процесс RTIA могли бы послать друг другу сообщения одновременно. Кроме того, при данной организации ни один из участников взаимодействия не может послать другому сразу несколько сообщений, не дожидаясь получения его ответов.

Чтобы не ограничивать потенциал разработанной библиотеки, взаимодействие её потоков управления было организовано с использованием двух независимых очередей, каждая из которых служит промежуточным буфером для указателей на сообщения одного из потоков (рисунок 3). В перспективе такой подход позволит использовать не только асинхронную, но и многопоточную модель, которая может быть более эффективной в случае должной поддержки со стороны федерата.

Таким образом, каждый поток записывает указатели на свои сообщения в одну очередь и проверяет другую очередь, чтобы получить указатели на сообщения от другого

потока. Целостность каждой из очередей защищается отдельным семафором, поэтому в каждый момент времени доступ к очереди может получить лишь один поток. Если во время обращения к очереди одного из потоков она уже используется, то поток блокируется и ожидает освобождения очереди. Стоит заметить, что при чтении данных такой подход может быть менее эффективным, чем неблокирующая проверка. Данный вопрос должен быть исследован дополнительно.

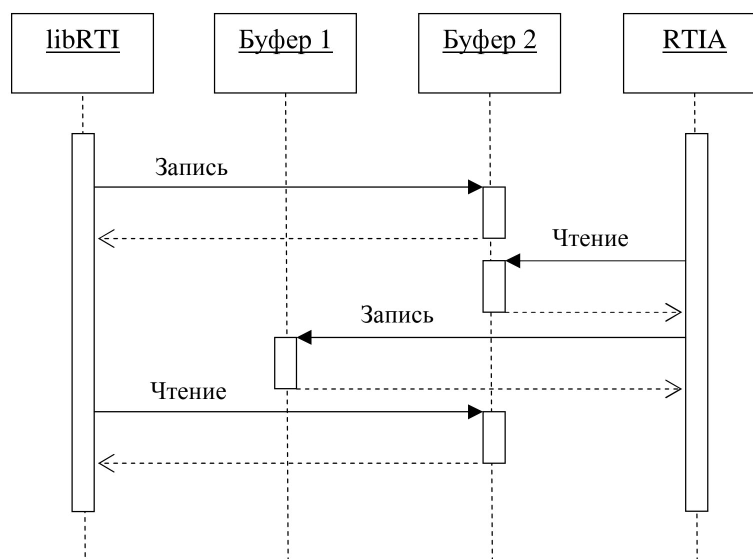

Рисунок 3 - Обмен данными через две очереди

Ещё одной областью для дальнейшей доработки может служить неэффективный протокол и логика обмена данными между потоками федерата и RTIA. На данный момент все взаимодействия между ними происходят синхронно: логика каждого из компонентов системы CERTI организована так, что он не может передать новое сообщение, пока не получит ответ на предыдущее сообщение. Особенно заметным недостаток становится при выполнении обратных вызовов, когда процесс RTIA пытается передать федерату все накопленные сообщения. Однако подобная модификация логики системы моделирования потребует значительных временных и трудозатрат, и поэтому данные изменения малоцелесообразны.

Организация ожидания

Также важным вопросом на данном этапе разработки является вопрос об эффективности работы потока RTIA, в обязанности которого входит ожидание сообщений от удалённых компонентов RTI и от других потоков федерата одновременно. Данное поведение может быть организовано с помощью одной из двух известных парадигм: *активного* или

пассивного ожидания. При активном ожидании поток находится в вечном цикле, поочерёдно проверяя все возможные источники входящих сообщений, что требует дополнительных расходов процессорного времени. При пассивном ожидании поток просит операционную систему оповестить его о приходе сообщения и засыпает до тех пор, пока не получит соответствующее уведомление. Данный подход требует специальной поддержки со стороны операционной системы, имеет более сложную организацию, и его эффективность обратно пропорциональна частоте входящих сообщений.

Так как процесс RTIA получает сообщения достаточно редко, то базовая версия CERTI использует пассивное ожидание. Однако поток RTIA получает сообщения от удалённых компонентов RTI через сеть (уровень процессов), а сообщения от других потоков федерата через отдельную очередь (уровень потоков). При этом операционные системы обычно не предоставляют средств организации пассивного ожидания сразу на двух уровнях взаимодействия. В результате выборка сообщений при текущей логике системы моделирования может быть реализована лишь в виде активного ожидания. Использование более эффективного пассивного ожидания требует создания ещё одного потока управления, единственной целью которого станет прослушивание сетевого соединения и отображение получаемых сообщений на уровень потоков.

Созданный на данном этапе прототип многопоточной системы моделирования использует более простой механизм активного ожидания. Однако практическое внедрение обычно более эффективного пассивного ожидания может дать существенный прирост производительности, и соответствующее предложение должно быть дополнительно рассмотрено в дальнейшем.

Работа с памятью

Другим важным аспектом реализации многопоточной версии системы CERTI является эффективность работы с памятью. При использовании независимых процессов все данные необходимо было передавать в виде массива, что неизбежно приводило к копированию памяти. Использование потоков позволило передавать сообщения простой передачей указателя. Такой подход требует использования динамической памяти, выделение и освобождение которой приводит к дополнительным накладным расходам. Однако, эти затраты можно уменьшить с помощью *кольцевого буфера* [13].

Кольцевой буфер создаётся в процессе инициализации инфраструктуры системы моделирования. При этом под него выделяется заданный объём динамической памяти. Во время создания нового сообщения под него отдаётся часть памяти буфера, реального

выделения памяти при этом не происходит. Аналогично, во время удаления сообщения часть памяти буфера помечается как свободная, реальное же её освобождение происходит только при удалении буфера.

Неэффективность использования памяти проявляется в использовании разных форматов сообщений между библиотекой libRTI и процессом RTIA, и между процессом RTIA и процессом RTIG. Если бы эти сообщения имели единый общий формат, то процесс RTIA мог бы передавать полученные от федерата сообщения напрямую процессу RTIG. При этом не требовалась бы лишнее преобразование формата данных.

Описанные предложения по оптимизации использования памяти также могут привести к существенному приросту производительности системы моделирования, и должны быть изучены более детально.

1.1.4 Каскадная архитектура CERTI

Описание идеи

Выраженная централизованная архитектура CERTI приводит к чрезмерной загрузке её компонента CRC во время выполнения сложных имитационных моделей, что является серьёзным архитектурным недостатком. Существует множество реализаций RTI, которые совмещают централизованную и децентрализованную архитектуру более сбалансированно, что позволяет им достичь лучших показателей производительности [14]. Однако смешение этих подходов в случае CERTI потребует коренной модификации системы, и приведёт к потере многих преимуществ, которые даёт её простая модульная архитектура сегодня. Настоящий раздел рассматривает альтернативный подход к уменьшению нагрузки на CRC – каскадную архитектуру инфраструктуры RTI.

Несмотря на то, что каждый федерат соответствует конкретному компоненту моделируемой системы, уровень их абстракции может существенно различаться и не отражать её логической структуры. Например, модель бортовой системы может состоять из одного федерата, соответствующего множеству второстепенных систем, и множества федератов, моделирующих поведение наиболее важной подсистемы. При этом последние имеют меньший уровень абстракции, так как соответствуют более мелким компонентам системы. Лишь совокупность этих федератов формирует логическую подсистему того же уровня абстракции, что и остальные федераты. Поэтому агрегированные федераты зависят друг от друга, они логически связаны между собой.

Во время выполнения имитационной модели, группа агрегированных федератов выделяется на фоне остальных участников моделирования высокой интенсивностью взаимодействий. Практика проведения экспериментов показывает, что агрегированные федераты взаимодействуют друг с другом гораздо чаще, чем с внешними по отношению к этой группе федератами. Таким образом, происходит естественная кластеризация всех федератов модели на множество *агрегированных групп*, внутри которых происходит большая часть обменов данными.

Ввиду того, что система CERTI имеет ярко выраженную централизованную архитектуру, любые взаимодействия федератов происходят с помощью её компонента CRC, вне зависимости от логической структуры модели. В результате, если число обменов данными между участниками моделирования *достаточно* велико, центральный компонент CRC перегружается и становится узким местом системы, замедляя скорость моделирования.

В то же время, выполнение конкретного федерата на самом деле не зависит от обмена данными внутри агрегированной группы, к которой он не принадлежит. Поэтому трафик может быть эффективно разделён в соответствии с логической структурой исследуемой системы с помощью набора вспомогательных компонентов CRC, каждый из которых соответствует одной из агрегированных групп. С одной стороны, эти CRC будут самостоятельно контролировать сопоставленную им группу федератов, выполняя при этом часть функций основного CRC и, тем самым, снимая с него часть нагрузки. С другой стороны, с точки зрения основного CRC, вспомогательные CRC будут выглядеть как обычные федераты, которые выполняются согласно правилам HLA, но генерируют поток трафика, соответствующий целой группе федератов. Описанные вспомогательные CRC могут быть реализованы как новый интерфейс к локальному компоненту LRC, что не повлечёт за собой существенного увеличения сложности CERTI.

Рассмотрим несколько естественных расширений описанной идеи. Во-первых, на практике федераты могут объединяться в группы сразу по нескольким признакам, не связанным с логической структурой моделируемой системы: хорошим критерием может служить, например, неравномерное распределение интенсивности обменов данными. Во-вторых, один и тот же приём можно использовать несколько раз. Агрегированная группа может быть, в свою очередь, разбита на несколько подгрупп, тем самым сформировав новый *каскад* управления моделью. Именно поэтому описанная архитектура описывается в настоящей работе как *каскадная* (рисунок 4).

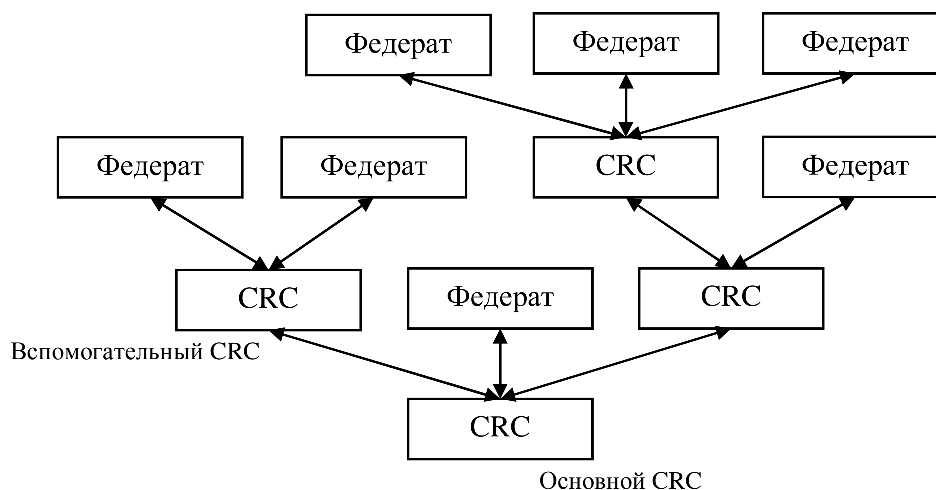

Рисунок 4 - Каскадная архитектура

Анализ каскадной архитектуры

Построение системы моделирования на основе каскадной архитектуры на практике даёт сразу несколько преимуществ.

Прежде всего, данный подход позволяет решить проблему чрезмерной загрузки компонента CRC и повышает масштабируемость системы. В самом деле, вспомогательные CRC самостоятельно обрабатывают внутренние обмены данных между федератами соответствующей им группы и, тем самым, снимают часть обязанностей с основного CRC. Каждый вспомогательный CRC может выполняться на отдельном вычислительном узле, при этом загрузка автоматически перераспределяется по множеству доступных ресурсов и не требует дублирования данных или использования сложных алгоритмов согласования состояния распределённой системы.

Во-вторых, агрегация позволяет уменьшить потери на синхронизацию федератов. Внутренние взаимодействия агрегированной группы происходят через вспомогательный компонент CRC, который не учитывает множество зависимостей от других участников моделирования, и поэтому может выполняться более эффективно, чем основной CRC. В то же время, внешние взаимодействия агрегированных федератов реализуются описанной схемой с использованием и вспомогательного, и основного компонента CRC, и поэтому менее эффективны. Однако, если структура взаимодействия в группе федератов выбрана *удачно*, синхронизационные потери уменьшаются соответствующим образом.

В третьих, агрегация позволяет провести точную настройку RTI под конкретную имитационную модель. Например, федераты могут быть кластеризованы в соответствии с

набором сервисов RTI, которые они используют. При этом неиспользуемые сервисы RTI могут быть безболезненно удалены из вспомогательных CRC, тем самым уменьшая их сложность и увеличивая эффективность. Далее, оставшиеся сервисы CRC, в свою очередь, могут быть подстроены под требования подключённых к ним федератов. Например, каждый из вспомогательных компонентов CRC может использовать свой собственный алгоритм продвижения модельного времени. В случае полунатурного моделирования основной CRC всегда должен использовать только консервативную схему продвижения времени, однако вспомогательный CRC, к которому не подключено оборудование, может использовать внутри себя и обычно более эффективную оптимистическую схему.

Наконец, агрегация даёт возможность увеличения эффективности взаимодействия федератов, выполняющихся на единственном узле. Централизованная архитектура CERTI не учитывает относительное расположение федератов. Даже в случае, если они запущены на одном вычислительном узле, каждый обмен данными происходит через процесс RTIG. При этом RTI использует два сетевых сообщения, чтобы передать данные между двумя процессами, выполняющимися на одном узле, что приводит к значительному падению производительности системы. В то же время агрегация всех федератов, выполняющихся на одном узле, позволяет реализовать внутренний обмен данными между ними без использования сетевых взаимодействий. Таким образом, концепция теоретически позволяет достичь эффективности децентрализованных peer-to-peer систем без внесения каких-либо изменений в логику работы CERTI.

Слабой стороной рассмотренной каскадной архитектуры является недетерменизм и зависимость структуры RTI от конкретной имитационной модели. Построение каскадной инфраструктуры, соответствующей эффективному разделению федератов на группы, требует разработки автоматических статических и динамических анализаторов выполняемой модели. Тем не менее, полученный прирост производительности системы моделирования, по всей видимости, стоит затраченных усилий, и каскадная архитектура выглядит достаточно перспективно.

1.1.5 Выводы

На данном этапе настоящей работы был реализован прототип среды выполнения MT-CERTI с многопоточным локальным компонентом LRC. В ходе описания различных аспектов реализации данного прототипа было рассмотрено сразу несколько идей по его дальнейшему развитию: поддержка многопоточной модели, изменение протокола обмена

сообщениями между потоком RTIA и другими потоками федерата, реализация неблокирующего обмена данными, использование пассивного ожидания во время обработки сообщений внутри потока RTIA и, наконец, организация кольцевого буфера для более эффективной работы с памятью.

Использование каждой из рассмотренных идей может принести существенный прирост показателей производительности системы моделирования, и поэтому каждая из них должна быть рассмотрена подробнее. Затраты на реализацию некоторых из них, по-видимому, будут приемлемы для целей настоящей научно-исследовательской работы, и их практической внедрение в разработанный прототип может стать целесообразным.

Кроме того, в разделе была рассмотрена идея построения системы моделирования с каскадной архитектурой. Затраты на её построения превосходят затраты на доработку уже созданного прототипа многопоточной CERTI. Тем не менее, использование каскадной архитектуры имеет ряд преимуществ, и её практическое внедрение представляется довольно перспективным.

1.2 Модификации транслятора UML в модели, совместимые с HLA

В данном разделе описывается основной процесс генерации исходного кода моделей, заданных при помощи диаграмм состояний UML. Рассмотрены основные примитивы UML необходимые для создания моделей, и представлены основными этапы генерации исходного кода моделей. В данном разделе описывается средство, реализующее подход к созданию федерации и федератов при помощи конечных автоматов и развивающее возможности средства, описанного в отчете за третий этап [3]. Для создания и редактирования конечного автомата предполагается использование любого редактора диаграмм состояний UML с возможностью сохранения диаграммы в формате XMI. Однако, авторами был на предыдущих этапах проведен обзор UML редакторов, который показал, что ArgoUML лучше других кандидатов подходит для создания диаграмм состояний моделей. Основными критериями обзора были:

- некоммерческая лицензия на распространение продукта;
- удобство интерфейса;
- возможность работы с XMI;

- поддержка всех необходимых примитивов для работы с диаграммами состояний.

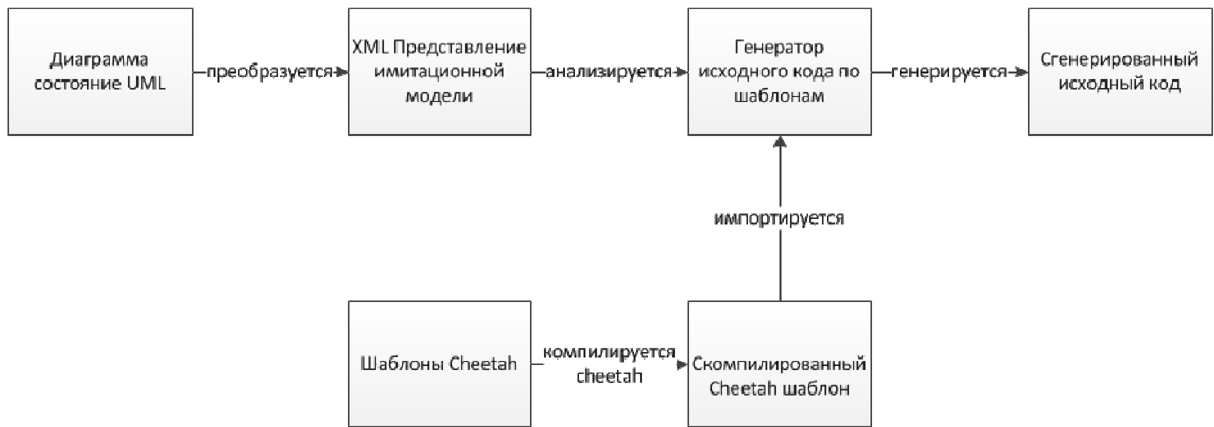

Рисунок 5 - Общая схема генерации исходного кода федерата

1.2.1 Конечный автомат

Конечный автомат (state machine) представляет собой граф состояний и переходов, который описывает, каким образом федерат реагирует на получение событий в HLA федерации [15]. Конечный автомат специфицирует последовательности состояний, через которые проходит в течение своего функционирования федерат, или взаимодействие федератов. Конечный автомат внутренней логики прикреплен к каждому федерату и служит для определения поведения данного федерата.

Состояние (state). Условие или ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоему условию, выполняет определенную деятельность, или ожидает какого-либо события.

За время своей жизни объект может находиться в различных состояниях. В каждом из них объект пребывает в течение какого-либо конечного времени. Для удобства при моделировании можно использовать фиктивные состояния, в которых осуществляются тривиальные действия и немедленный выход.

Каждый конечный автомат описывает поведение одного федерата. Реакция федерата, находящегося в определенном состоянии, на некое событие описывается переходом: федерат выполняет действие, прикрепленное к переходу, и меняет состояние. Каждому состоянию соответствует свое множество возможных переходов.

Текущее поведение федерата связано с его состоянием. Текущую деятельность можно представить в виде пары действии; действие при входе, которое осуществляется по входу в состояние, и действие при выходе, которое осуществляется при выходе из этого состояния.

Несколько состояний можно сгруппировать в одно композитное состояние. Любой переход в композитное состояние затрагивает одно состояние, находящееся внутри него. Данное состояние является начальным для данного композитного состояния. Композитное состояние может быть как последовательным, так и параллельным. В конкретный момент времени активно только одно подсостояние последовательного композитного состояния. В параллельном композитном состоянии подсостояния активны одновременно в разных параллельных регионах.

Вложенный автомат (submachine). Конечный автомат, который может быть вызван как часть другого конечного автомата. Вложенный автомат является своего рода подпрограммой конечного автомата. Его семантика соответствует ситуации композитного состояния, когда его содержимое копируется и вставляется в то состояние, которое на него ссылается.

Переход (transition). В конечном автомате — отношение между двумя состояниями, показывающее, что федерат, находящийся в первом состоянии, будет выполнять некоторые действия и перейдет в другое состояние, как только наступит некоторое событие и при этом будут удовлетворены необходимые сторожевые условия. У перехода есть одно начальное и одно целевое состояние. У внутреннего перехода есть начальное, но нет целевого состояния. Он представляет собой реакцию на событие, при которой не происходит изменения состояния. Состояния и переходы представляют собой вершины и дуги конечного автомата.

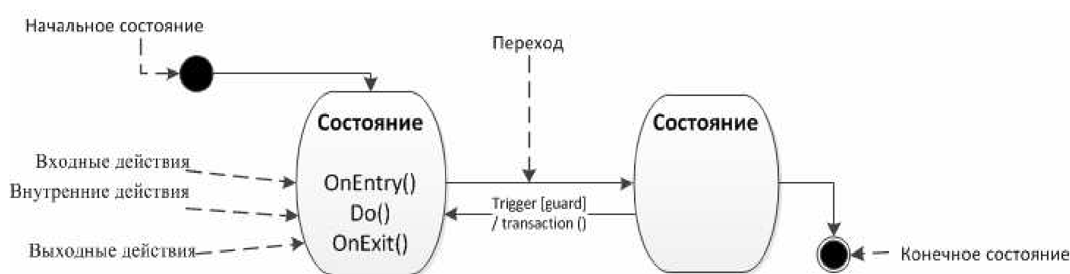

Рисунок 6 - Схема автомата внутренней логики федерата

1.2.2 Общая схема генерации кода федерата

На вход средству генерации подается диаграмма состояний языка UML и шаблон для генерации кода. Далее диаграмма переводится во внутреннее представление языка Питон.

После создания XML файла [16], он подаётся на вход непосредственно генератору кода по шаблону, который генерирует исходный код модели. Для генерации кода федератов (с интерфейсами для подключения к RTI) были созданы особые шаблоны [17]: отдельно для .h, .cpp и .fed файлов. На рисунке 6 представлена схема работы генератора кода моделей совместимых со стандартом HLA. Примеры работы представлены в статье [18].

Для генерации кода внутренней логики федерата были созданы шаблоны для трансляции кода автомата, описывающую данную логику. При реализации автомата пользователь системы моделирования использует следующие примитивы:

- *Заметка (UML — Note).* Заметки могут использоваться как для пояснения, так и для задания дополнительных атрибутов объектам автомата внутренней логики.
- *Обобщение (UML — Generalization).* Соединяет объекты и заметки "UML — Note", содержащие расширенную информацию об объекте.
- *Начальное/конечное состояние (UML — State Term).* Обозначают начальные и конечные состояния. Начальное/конечное состояние не имеет названия, для начального состояния оно не требуется, для конечного необходимо добавить заметку "UML — Note" с именем состояния и связать с конечным состоянием при помощи "UML — Generalization".
- *Состояние (UML — State).* Обозначает состояние автомата. Атрибуты State:
    - Входное действие (entry action) — действие, выполняемое при входе в состояние. Не выполняется при внутренних переходах.
    - Внутренние действие (do action) — действие, выполняемое после внутреннего перехода.
    - Выходное действие (exit action) — действие, выполняемое при выходе из автомата. Не выполняется при внутренних переходах.
- *Переход (UML — Transition).* Задает переход из одного состояния в другое. Атрибуты Transition:
    - *Триггер (trigger)* — условие (событие) перехода.
    - *Действие (action)* — действие, выполняемое при переходе.
    - *Хранитель (guard)* — дополнительное условие перехода.

Каждое состояние автомата преобразуется в отдельный класс на языке C++. Для этого были созданы шаблоны для трансляции исходных файлов и файлов заголовков для состояний автомата.

Для управления переходами из одного состояния в другое используется класс *Controller*. Экземпляр данного класса создается для каждого автомата внутренней логики отдельно. Перед данным классом ставятся следующие задачи:

- Предоставлять для каждого состояния метод *InitializeSM*(). При вызове данного метода происходить инициализация состояния и выполняется внутренний код состояния.
- Предоставлять метод *triggerEvent*(). Метод позволяет менять состояния исходы из вновь поступивших входящих событий.

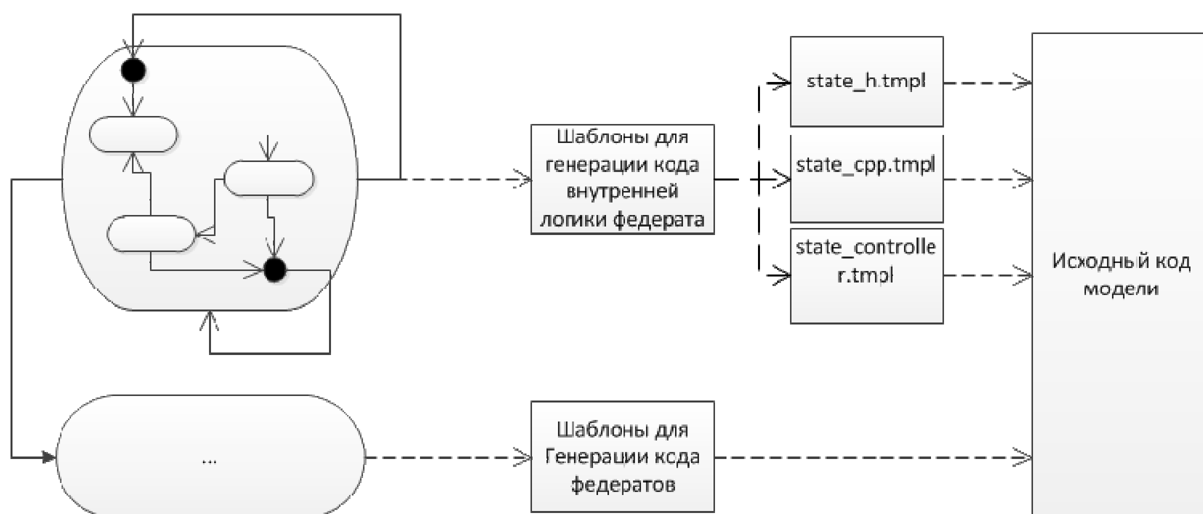

Рисунок 7 - Схема работы транслятора кода федерата

При трансляции класса можно задать заголовочные файлы, которые будут подключены перед определением класса. Для этого с классом автомата связывается объект "UML — Note", первой строкой которого является строка расширенных атрибутов, в остальных — имена подключаемых файлов в угловых скобках или кавычках. Можно также добавлять код методов прямо на диаграмму. Для этого с классом связывается "UML — Note" с кодом метода.

1.3 *Обзор схем трассировки моделей и разработка средства трассировки моделей*

Средства регистрации и трассировки событий моделирования предназначены для фиксации изменений состояния и параметров моделей компонентов РВС РВ и обменов сообщениями между ними в трассе имитационного эксперимента. На первом этапе данной НИР [1] в качестве основной технологии для распределенного имитационного моделирования в реальном времени была выбрана технология HLA RTI, в качестве реализации HLA RTI, принятой за основу разрабатываемой среды моделирования РВС РВ – среда распределенного моделирования CERTI.

Целью данного обзора является выбор схемы трассировки событий моделирования для реализации в среде распределенного моделирования РВС РВ на основе CERTI. Для достижения поставленной цели необходимо:

- Рассмотреть особенности архитектуры CERTI.
- Рассмотреть существующие общие подходы к трассировке.
- Сформулировать требования к схеме трассировки.
- Рассмотреть возможные реализации схем трассировки применительно к средам моделирования на основе HLA RTI.
- Выбрать схему трассировки.

1.3.1 Особенности архитектуры CERTI RTI

Архитектура CERTI RTI представлена на рисунке 8.

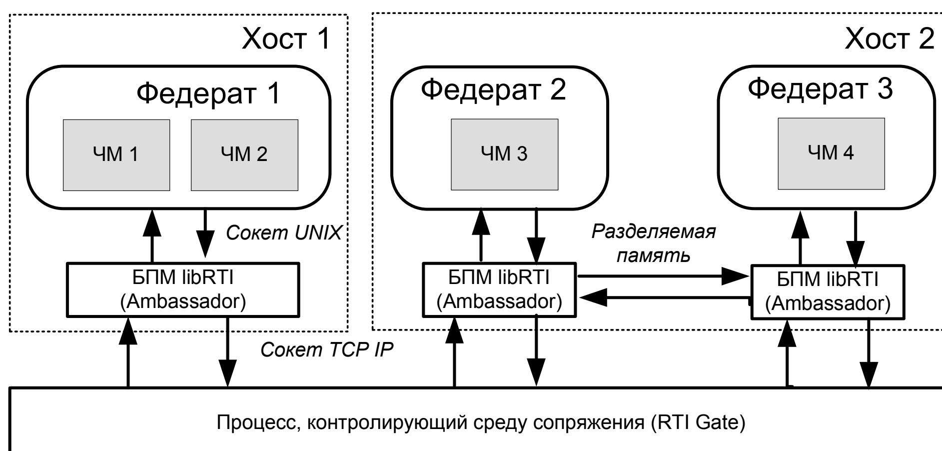

Рисунок 8 - Архитектура среды моделирования на основе CERTI RTI

RTI обеспечивает набор общих и независимых сервисов, успешно отделяет реализацию модели, управление выполнением модели и управление взаимодействием моделей. Шесть типов сервисов управления, как показано на рисунке 9, обеспечивают связь и координацию сервисов в федератах. В CERTI основная часть сервисов RTI реализована внутри глобального процесса RTIG, хранящего всю информацию, необходимую для управления выполняемой федерацией. Остальные компоненты RTI служат для обмена информацией между процессом RTIG и подключенными федератами. Таким образом, в CERTI используется централизованная архитектура, и часто синхронизация распределённой имитационной модели сводится к последовательному выполнению запросов федератов на единственном вычислительном узле.

Рисунок 9 - Функциональное представление моделирования на основе HLA RTI

В терминах HLA отдельная модель компонента рассматривается как «федерат». Группа федератов, которые намереваются взаимодействовать друг с другом, образуют систему моделирования, называемую «федерацией». В общем HLA определяется тремя компонентами: спецификацией интерфейса, набором правил и шаблоном объектной модели (OMT).

HLA интерфейс определяет правила взаимодействия федератов через набор общих интерфейсов. Спецификация интерфейса устанавливает интерфейс между федератами и RTI, который имеет имя, данное в программной реализации спецификации интерфейса.

HLA правила определяют набор соответствующих правил для обеспечения надлежащего взаимодействия федератов во время выполнения федерации, которая управляет поведением всей федерации и федератов.

HLA OMT обеспечивает стандартный фреймворк для записи информации, включенной в требуемую HLA модель объекта для каждой федерации и федератов. Основные компоненты OMT - объектная модель федерации (FOM) и имитационная объектная модель (SOM). FOM – это спецификация на уровне федераций, описывающая всю информацию (объекты, атрибуты, связи и взаимодействия), которая может быть разделяемой федератами, принимающими участие в отдельных имитационных экспериментах. SOM – это спецификация на уровне федератов, описывающая объекты, атрибуты и взаимодействия в отдельных федератах, которые доступны для других федератов.

Нужно отметить, что одним из преимуществ CERTI является более эффективное взаимодействие между моделями посредством разделяемой памяти в том случае, если федераты, которым принадлежат модели, располагаются на одном хосте. К достоинствам архитектуры CERTI можно отнести её простоту.

Централизованная архитектура CERTI обладает и рядом недостатков, главным из которых является большая загруженность процесса RTIG. При активном обмене информацией между федератами глобальный процесс RTIG становится узким местом. Кроме того, централизация управления федерацией порождает необходимость пересылки большого числа сетевых сообщений, что пагубно сказывается на производительности системы в целом.

1.3.2 Общие подходы к сбору трасс

Существующие механизмы сбора трассы данных о выполнении имитационных моделей компонентов РВС РВ можно разделить на две категории: централизованный сбор и распределенный сбор.

Централизованный сбор данных - это централизованный подход, который предполагает единую точку сбора данных имитационного эксперимента. Этот подход требует, чтобы сборщик захватывал все необходимые данные в одной точке сети. Главными достоинствами централизованного подхода являются присущая простота и отсутствие необходимости сортировки данных по временной метке. Также достоинством является гибкость и простота управления процессом трассировки из единой точки сбора данных. Главным недостатком является скопление большого объема трафика в одном узле сети, что может быть послужить причиной заторов трафика в сети и, таким образом, сбор данных может быть узким местом всей системы моделирования, особенно в масштабах HLA-систем

моделирования на основе WAN. В результате использования централизованного подхода по окончании процесса моделирования получается одна трасса.

Распределенный сбор данных предполагает наличие нескольких точек сбора данных. Каждый сборщик отвечает за сбор части данных имитационного эксперимента. Главным достоинством распределенного сбора является предотвращение в одной точке сети чрезмерного трафика, связанного с трассировкой имитационного эксперимента. Основной недостаток подхода заключается в том, что необходима дополнительная обработка данных с целью их сортировки при анализе всего эксперимента. Если требуется формирование единой базы данных или трассы всего эксперимента, то все распределенные данные должны быть перемещены по сети в одно место и упорядочены по временной метке, поэтому особое значение уделяется времени выполнения этой операции, чтобы избежать перегрузки сети и ее компонентов. В результате использования распределенного подхода по окончании процесса моделирования получается множество трасс, находящихся в разных точках сети.

Описанные общие подходы к процессу трассировки могут иметь различную реализацию. Ниже будут рассмотрены следующие схемы трассировки:

- Централизованные схемы:
    - Схема на основе прослушивания сетевого трафика.
    - Схема «Федерат-сборщик».
- Распределенные схемы:
    - Сбор трасс для отдельных федератов
        - Схема на основе встраивания функций трассировки в федерат.
        - Схема «RTI-интерфейс сборщик».
    - Сбор трасс для отдельных хостов
        - Схема на основе общей памяти в рамках хоста.
    - Сбор трасс федератов с разных хостов
        - Схема трассировки на основе федератов-сборщиков
        - Схема трассировки на основе многоагентной системы.

1.3.3 Требования к схеме трассировки

Критерии выбора схемы трассировки:

- Минимальное влияние на производительность разрабатываемой среды моделирования.

- Гибкость управления процессом трассировки. До начала имитационного эксперимента пользователю должны быть доступны возможности настройки перечня трассируемых компонентов РВС РВ, перечня регистрируемых событий и их параметров, фильтрации данных.
- Возможность реализации подхода в CERTI.

1.3.4 Централизованные схемы трассировки

1.3.4.1 Схема трассировки на основе прослушивания сетевого трафика

Схема трассировки на основе прослушивания сетевого трафика предложена в [19] и основана на использовании библиотеки поддержки моделирования, удовлетворяющей требованиям HLA RTI. Она заключается в том, что дополнительный процесс прослушивает сетевой интерфейс, по которому осуществляется передача пакетов HLA RTI. Пакеты разбираются, на их основании формируется трасса (рисунок 10). Хотя такой способ и представляется универсальным, он требует настройки на конкретную библиотеку, так как реализация механизма сбора трассы в рамках HLA не определена. Другим недостатком такого подхода оказывается неспособность перехватывать события в том случае, когда объекты работают в рамках одной федерации и библиотека использует общую память для ускорения обмена.

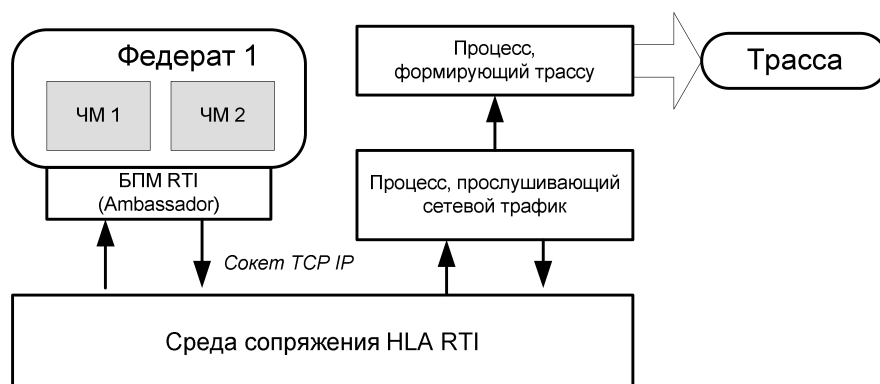

Рисунок 10 - Схема формирования трассы на основе прослушивания сетевого канала

1.3.4.2 Схема трассировки «Федерат-сборщик»

Схема трассировки «Федерат-сборщик» заключается в создании отдельного федерата (федерат 2 на рисунке 11), занимающегося исключительно сбором информации о событиях в

среде моделирования и их записью в трассу. Данная схема является централизованной, поэтому ей присущи достоинства и недостатки подобного типа схем.

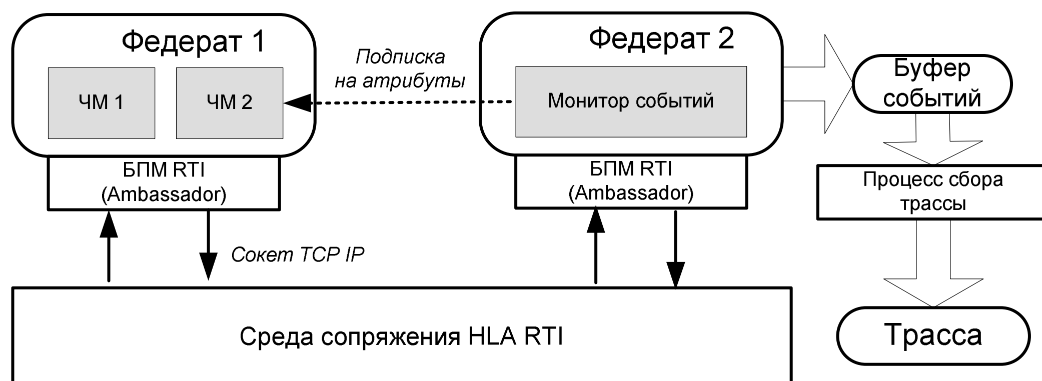

Рисунок 11 - Схема сбора трассы с помощью федерата-сборщика

Если в федерации только один федерата-сборщика, то сетевой трафик может резко возрасти, потому что ему придется подписаться на все данные из других федератов. Другой недостаток федерата-сборщика связан с неполнотой временных меток, поступающих в него данных: нет показаний времени, когда данные были получены федератом, подписавшимся на эти данные. Таким образом, становится трудным вычисление задержек. Для того, чтобы временные метки были адекватными, сборщик-федерат должен располагаться в каждом удаленном узле и время прибытия данных в такой узел должно записываться локальным сборщиком-федератом. Подобная схема реализована в MAK HLA.

1.3.5 Распределенные схемы трассировки

1.3.5.1 Встраивание функций сбора трассы в ПО федерата

В данной распределенной схеме трассировки отдельная трасса собирается для каждого федерата. Для реализации данной схемы в каждый федерат необходимо добавить дополнительный программный код, осуществляющий сбор и формирование трассы по мере поступления событий. Основной недостаток подхода заключается в том, что код, который должен быть добавлен в федерат, будет специфичным для федерата или федерации и не может повторно использоваться в других федератах или федерациях.

1.3.5.2 RTI-Interface logger

Взаимодействие федерата со средой сопряжения HLA RTI в CERTI осуществляется посредством библиотеки поддержки моделирования (БПМ) libRTI (Ambassador). Таким образом, этот RTI-интерфейс собирает все данные, которые пересылаются от федерата к RTI и от RTI к федерату [20]. Следовательно, функции трассировки можно внедрить в БПМ libRTI. В данной схеме трасса собирается для каждого отдельного федерата, процесс трассировки осуществляется в точках взаимодействия федератов и RTI.

1.3.5.3 Схема трассировки на основе общей памяти

В Стенде ПНМ [21] реализуется данная распределенная схема трассировки на основе общей памяти, и на каждом хосте собирается своя отдельная трасса. На каждом хосте события заносятся основным процессом моделирования в область общей памяти. Специальный вспомогательный процесс считывает из общей памяти события, передаёт их средству оперативного управления и периодически сохраняет события в трассу. Общая схема трассировки на основе общей памяти изображена на рисунке 12.

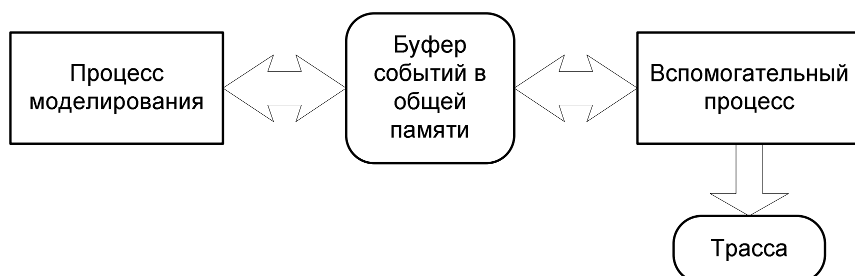

Рисунок 12 - Схема трассировки на основе общей памяти, реализованная в Стенде ПНМ

Поскольку CERTI поддерживает эффективное взаимодействие федератов, расположенных на одном хосте, через разделяемую память, то такая схема может использоваться на каждом хосте среды моделирования.

1.3.5.4 Схема трассировки на основе федератов-сборщиков

Данная схема является развитием централизованной схемы «федерат-сборщик», описанной выше и заключается в том, что для решения проблемы временных меток «федераты-сборщики» устанавливаются в удаленных узлах, осуществляющих моделирование, которые могут включать в себя несколько хостов.

1.3.5.5 Схема трассировки на основе многоагентной системы

С целью повышения надежности и производительности среды моделирования на основе HLA RTI в работе [22] предлагается распределенная схема трассировки на основе многоагентной системы сбора данных.

Многоагентная система состоит из множества взаимодействующих агентов-сборщиков. Каждый сборщик включает в себя:

- Коммуникационный модуль, отвечающий за связь с моделями и другими агентами-сборщиками.
- Совместный модуль принятия решений.
- Модуль обработки данных, получаемых от моделей.
- Модуль записи данных в трассу

Каждый сборщик загружает данные в соответствующий сервер баз данных.

Сборщики каким-то образом располагаются на хостах сети. Каждый сборщик может трассировать одну или несколько моделей, расположенных на разных хостах. Причем модели для трассировки могут определяться по некоторому эвристическому алгоритму в зависимости от количества событий, выдаваемых каждой моделью. Набор трассируемых моделей для каждого сборщика в процессе работы может динамически изменяться в зависимости от загрузки сети и количества выдаваемых моделями событий. К недостаткам данной схемы можно отнести сложность её реализации.

1.3.6 Выводы

В ходе анализа различных схем трассировки были рассмотрены их особенности реализации, достоинства и недостатки. Нужно отметить, что централизованные схемы больше подходят для ситуаций, когда модели не генерируют большое количество событий, и как следствие не создают большого сетевого трафика, связанного с трассировкой, и когда моделирование ведется в локальной сети. При этом собирается единая трасса с упорядоченными по временным меткам событиями.

Наиболее предпочтительными для реализации в распределенной среде моделирования РВС РВ являются распределенные схемы трассировки.

Для дальнейшей реализации и проведения экспериментального исследования были выбраны следующие схемы трассировки:

1. Централизованная схема «Федерат-сборщик» (сбор трассы в одной точке).

2. Распределенная схема «RTI-интерфейс сборщик» (сбор трасс на уровне федератов).

Нужно отметить, что перспективными является подходы на основе многоагентной системы трассировки, на основе общей памяти и множества федератов-сборщиков, однако данные схемы несколько сложнее с точки зрения реализации. Для проведения их экспериментального исследования потребуются более сложные примеры моделей РВС, чем имеются на данный момент, поэтому они могут быть реализованы и исследованы на последующих этапах НИР в случае необходимости.

1.4 Обоснование корректности алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов

На предыдущих этапах проекта[2],[3] был разработан и реализован алгоритм трансляции иерархических временных автоматов, описанных посредством UML-диаграмм, в сети простых (плоских) временных автоматов, описанных в формате системы верификации распределенных программ UPPAAL. Предложенный алгоритм трансляции позволяет совместить два удобных для применения и хорошо зарекомендовавших себя на практике средства проектирования сложных информационных систем, работающих в реальном времени – графическое средство описания иерархических систем взаимодействующих процессов UML и программно-инструментальное средство автоматической проверки правильности поведения систем взаимодействующих процессов, работающих в реальном времени, UPPAAL. Но для того чтобы быть уверенными в том, что результаты верификации системы UPPAAL, примененной к сети плоских временных автоматов, адекватно отражают свойства вычислений исходного иерархического временного автомата, представленного в виде UML-диаграмм, необходимо обосновать корректность предложенного алгоритма трансляции.

В данном разделе приводится строгое математическое обоснование корректности разрабатываемого алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов (далее – просто сеть). Корректность алгоритма состоит в том, что при трансляции диаграммы сохраняется выполнимость достаточно широкого класса свойств.

Алгоритм работает в два этапа:

1. трансляция UML-диаграммы в иерархический временной автомат (далее – просто автомат) и
2. трансляция автомата в сеть.

Корректность первого этапа работы алгоритма не нуждается в обосновании, так как семантика UML-диаграммы совпадает с таковой для автомата. В данном разделе приводятся описания моделей автомата и сети и краткое описание алгоритма трансляции, после чего обосновывается корректность алгоритма. Подробнее описание моделей автомата и сети можно найти, например, в [23].

1.4.1 Модель иерархических временных автоматов

Автомат — это система

$$(S, S_0, \eta, Type, Var, Clocks, Chan, Inv, T, Chantype),$$

где

- S — множество состояний,
- $S_0 \subseteq S$ — множество инициальных состояний,
- $\eta : S \rightarrow 2^S$ — функция вложенности состояний,
- Type : S → {AND, XOR, BASIC, ENTRY, EXIT} – функция типов состояний,
- Var – множество переменных ограниченного целочисленного типа,
- Clocks – множество таймеров,
- Chan – множество каналов,
- Inv : S → Invariant – функция инвариантов состояний,
- $T \subseteq S \times (Guard \times Sync \times Reset) \times S$ — множество переходов,

Каждый инвариант (Invariant) представляет собой константу true, константу false или выражение вида «x op n», «(x-y) op n», где x, y — таймеры множества Clocks, n – целочисленная константа, op — одно из отношений <, ≤, =.

Каждое предусловие (Guard) представляет собой константу true, константу false или булеву формулу над выражениями вида «E' op E''», «x op n», где E', E'' – арифметические выражения над переменными множества Var, x – таймер множества Clocks, n — целочисленная константа, op – одно из отношений <, =, >, ≤, ≥.

Каждая синхронизация (Sync) имеет вид «c!» (посылка сообщения по каналу c), «c?» (прием сообщения по каналу c) или «none» (отсутствие посылки/приема сообщения).

Действие (Reset) есть множество присваиваний вида «x ← 0» и «v ← E», где x – таймер множества Clocks, v – переменная множества Var, E – арифметическое выражение над переменными множества Var.

Состояния типов AND и XOR будем называть метасостояниями, состояния типа ENTRY — входами, состояния типа EXIT – выходами, состояния типа BASIC – простыми состояниями.

Автомат без функции вложенности может рассматриваться как помеченный ориентированный мультиграф с множеством вершин S. В связи с этим к автомату применима терминология, используемая в теории графов.

Далее будут рассматриваться только корректные автоматы. Корректным считается автомат, удовлетворяющий следующим ограничениям:

- ограничения на структуру состояний:
  - функция η задает ориентированное корневое дерево с множеством вершин S и дугами, идущими от корня к листьям дерева (в дальнейшем будем называть его деревом состояний);
  - множество всех простых состояний, входов и выходов образуют листья дерева состояний;
  - состояние типа AND не соединено дугой ни с одним простым состоянием;
- ограничения на множество инициальных состояний:
  - множество S_0 содержит корень дерева состояний;
  - множество S_0 не содержит входов и выходов;
  - вместе с каждым состоянием множество S_0 содержит и его предка;
  - среди потомков инициального состояния типа XOR ровно один является инициальным;
  - все потомки инициального состояния типа AND являются инициальными;
- ограничения на переходы:
  - из входа, вложенного в состояние типа XOR, исходит ровно одна дуга;
  - из входа, вложенного в состояние типа AND, исходит ровно по одной дуге в каждое вложенное состояние;
  - все ограничения на соотношение инцидентных переходам состояний приведены на рисунке [13] (кругами обозначены простые состояния, тонкими вертикальными линиями — входы, толстыми вертикальными линиями — выходы, квадратами — метасостояния, стрелками —

переходы, вложенность состояний соответствует геометрической вложенности на рисунке);

- если дуга автомата исходит из входа, то она помечена предусловием true и синхронизацией none;
- если дуга автомата ведет в выход, то она помечена пустым действием и синхронизацией none;
- если дуга автомата соединяет два выхода, то она помечена предусловием true и пустым действием.

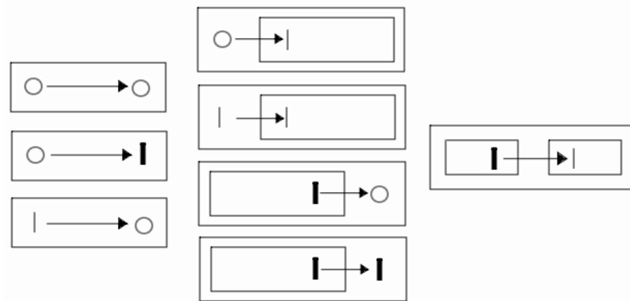

Рисунок 13 - Ограничения на соотношение инцидентных переходам состояний

Конфигурацией автомата является подмножество множества его состояний вместе со значениями переменных и таймеров. Семантика автомата описывается вычислением – последовательностью конфигураций, в которой каждая следующая получается из предыдущей применением одного перехода или множества переходов, связанных синхронизацией. При этом переход может сработать в конфигурации, если его предохранитель истинен в данной конфигурации, и изменения значений переменных и таймеров определяются действиями сработавших переходов. Кроме того, вместо срабатывания переходов возможно увеличение значений таймеров на одно и то же действительное число.

1.4.2 Модель сети временных автоматов

Сеть — это система

(A, Vars, Clocks, Chan, Chantype, Chanurg),

где

- A – множество процессов,
- Vars – множество переменных ограниченного целочисленного типа,

- Clocks – множество таймеров,
- Chan – множество каналов синхронизации,
- Chantype : Chan → {h, b} – функция типов каналов,
- Chanurg : Chan → {o, u} – функция срочности каналов.

Каждый канал имеет тип «точка-точка» или «широковещательный» и может быть помечен как срочный. Тип и срочность определяются функциями Chantype и Chanurg соответственно.

Каждый процесс p_i в векторе процессов является системой

$$(L_i, T_i, Type_i, Inv_i, l_i^0),$$

где

- L_i — множество состояний,
- $T_i \subseteq L_i \times (Guard \times Sync \times Reset) \times L_i$ — множество переходов,
- $Type_i : L_i \rightarrow \{o, u, c\}$ — функция типов состояний,
- $Inv_i : L_i \rightarrow Invariant$ — функция инвариантов,
- l_i^0 — инициальное состояние.

Множества Guard, Sync, Reset, Invariant полностью совпадают с одноименными множествами в модели автоматов.

Посредством функции $Type_i$ каждое состояние процесса может быть помечено как обычное, срочное и сверхсрочное.

Конфигурацией сети является набор состояний процессов этой сети вместе со значениями переменных и таймеров. Семантика сети описывается вычислением – последовательностью конфигураций, в которой каждая следующая получается из предыдущей применением одного перехода или множества переходов, связанных синхронизацией. При этом переход может сработать в конфигурации, если его предохранитель истинен в данной конфигурации, и изменения значений переменных и таймеров определяются действиями сработавших переходов. Если в конфигурации не присутствуют срочные и сверхсрочные вершины, то вместо срабатывания переходов может происходить продвижение времени. В этом случае показания всех таймеров увеличиваются на одну и ту же величину δ. На эту величину δ налагается единственное ограничение: в каждом из автоматов для обновленных значений таймеров должны выполняться инварианты всех текущих состояний. Срабатывание переходов, начинающихся в срочных вершинах, имеет более высокий приоритет по сравнению с обычными вершинами; начинающихся в сверхсрочных – по сравнению со срочными.

1.4.3 Описание алгоритма трансляции

Алгоритм принимает на вход корректный автомат

$$HTA = (S, S_0, \eta, Type_{hta}, Var_{hta}, clocks_{hta}, chan_{hta}, Inv, T, Chantype_{hta}).$$

В процессе работы алгоритмом формируется модель плоских временных автоматов

$$M = (A, Vars, Clocks, Chan, Chantype, Chanurg).$$

По окончании работы алгоритма сформированная модель выдается в качестве результата.

В начале работы алгоритма создается система плоских временных автоматов, не содержащая ни одного процесса и содержащая все каналы и таймеры, соответствующие каналам и таймерам HTA.

В процессе работы алгоритм последовательно обрабатывает все метасостояния HTA от корня к листьям, транслируя каждое метасостояние в отдельный процесс в системе M. Каждый получаемый при трансляции метасостояния процесс содержит состояние «idle», соответствующее неактивности исходного метасостояния в HTA. После добавления состояния «idle» метасостояние обрабатывается в зависимости от его типа — AND или XOR.

Обработка метасостояния типа AND.

При трансляции состояния типа AND в получаемый процесс добавляется состояние «active», соответствующее его активности в HTA. Каждый вход, вложенный в обрабатываемое метасостояние, порождает дополнительные сверхсрочные вершины, число которых равно числу вложенных в метасостояние компонент, также являющихся метасостояниями. Дополнительные вершины упорядочиваются, после чего добавляется дуга из состояния «idle» в первую вершину и дуга из последней вершины в состояние «active». Кроме того, каждая дополнительная вершина соединяется со следующей по порядку.

Добавленные дуги несут синхронизацию, соответствующую активации исходного метасостояния и всех его вложенных компонент, также являющихся метасостояниями.

Для обеспечения корректной деактивации метасостояния в получаемый процесс добавляется дуга, несущая особую синхронизацию.

Обработка метасостояния типа XOR.

Для каждого метасостояния и каждого простого состояния в получаемом процессе заводится состояние, соответствующее активности этого вложенного состояния. Кроме того, для каждого входа, вложенного в каждую вложенную компоненту, в процессе заводится отдельное сверхсрочное метасостояние. Из каждого состояния процесса, соответствующего входу, входящему в состав вложенной компоненты, ведется дуга в соответствующую

вложенную компоненту, отвечающая активации данной компоненты с использованием указанного входа входа.

Переходы в НТА, соединяющие два простых состояния, простое состояние со входом, вход с простым состоянием, два входа, простое состояние с выходом или два выхода, при трансляции порождают одну дугу с соответствующими метками. Остальные переходы при трансляции порождают дополнительные сверхсрочные вершины, последовательно соединенные дугами и обеспечивающие деактивацию не только вложенной компоненты, но и всех ее потомков в дереве состояний вплоть до листьев. Более подробно это рассказано в пункте «Обеспечение деактивации метасостояний».

Обеспечение корректного начала работы.

Предполагается, что в начале работы каждый получаемый процесс находится в состоянии «idle». Для перехода системы плоских временных автоматов в состояние, соответствующее начальному состоянию НТА, делается следующее.

До начала основной обработки в систему M добавляется процесс Global_kickoff, представляющий собой вершины, последовательно соединенные дугами, несущими особую синхронизацию, обеспечивающую корректное начало работы системы. Все вершины, кроме последней, являются сверхсрочными, что гарантирует немедленное выполнение всех переходов процесса, т.е. немедленное приведение системы в нужное состояние.

В каждом обрабатываемом инициальном метасостоянии, помимо прочего, добавляется дуга, корректно активизирующая данное метасостояние и несущая синхронизацию, парную к одной из синхронизаций дуг процесса Global_kickoff.

Трансляция срочных переходов НТА.

Из-за того что в модели автоматов могут быть срочные переходы, тогда как в модели системы плоских временных автоматов их нет, приходится моделировать срочные переходы средствами результирующей модели.

Для обработки срочных переходов, не несущих синхронизации, в систему M перед началом основной обработки добавляется процесс, состоящий из одного состояния и одной дуги, несущей синхронизацию «Hurry?» по срочному каналу типа «точка-точка». При обработке срочного перехода, не несущего синхронизации, соответствующей дуге процесса приписывается синхронизация «Hurry!».

Каждый канал «c» в множестве каналов НТА порождает в результирующей модели четыре канала для моделирования взаимодействия двух несрочных, несрочного со срочным, срочного с несрочным и двух срочных переходов с синхронизацией по каналу «c» в НТА.

При обработке перехода, несущего синхронизацию, вместо соответствующей дуги процесса может порождаться несколько дуг с синхронизацией по различным каналам.

Обеспечение деактивации метасостояний.

При обработке дуги, ведущей из выхода вложенной в метасостояние типа XOR компоненты во вход другой вложенной компоненты или во вложенное простое состояние, необходимо добавить в получаемый процесс дополнительные сверхсрочные вершины и соединить дугами состояния, соответствующие активности вложенных компонент, через эти вершины.

Добавление вершин связано с тем, что помимо деактивации вложенной компоненты провести также деактивацию всех ее потомков вплоть до листьев, если они были активны. Для обеспечения такой деактивации делается следующее.

Для каждого выхода вложенной в метасостояние типа XOR компоненты вычисляются все возможные наборы простых состояний ее потомков в дереве состояний вплоть до листьев, из которых потенциально возможен (хотя может и не быть осуществлен, например, из-за несовместности наборов предусловий) одновременный переход с деактивацией всех активных потомков. Для этого строится ориентированное дерево, дуги которого направлены к корню — рассматриваемому выходу, у которого листьями являются простыми состояния, а остальные вершины образуют все выходы, из которых переходом по дугам через другие выходы достижим рассматриваемый.

После построения дерева достаточно перебрать все его поддеревья, удовлетворяющие следующим условиям: если предок выхода, являющегося вершиной дерева, имеет тип XOR, то у выхода ровно один потомок, если же предок имеет тип AND, то у выхода столько потомков, сколько в исходном дереве.

После получения всех поддеревьев у них отбрасываются листья (простые вершины). При этом, возможно, некоторые деревья станут одинаковыми — тогда можно отбросить все повторения. В получаемый процесс добавляются сверхсрочные вершины, соответствующие выходам в поддереве, и они последовательно соединяются дугами, обеспечивающими корректную деинициализацию. Каждое дерево порождает свою последовательность последовательно соединенных вершин.

Кроме того, в процессе трансляции создаются переменные, по значению которых можно утверждать, может или не может сработать каждая последовательность деинициализаций метасостояний, и предусловия, соответствующие возможности

деинициализации, добавляются дуге, ведущей в первую вершину добавленной для деинициализации последовательности.

1.4.4 Обоснование корректности алгоритма трансляции

Для обоснования корректности будет введено понятие структуры Крипке, позволяющее единообразно описать семантику автоматов и сети. Затем будет введено понятие выполнимости формул на структурах Крипке и выделен класс формул, равновыполнимых на структурах, описывающих исходный автомат и сеть, получаемую в результате трансляции.

Считаем заданным в дальнейших построениях множество A атомарных высказываний.

Структурой Крипке будем называть систему (S, s_0, L, R), где

- S – конечное множество состояний,
- s_0 – начальное состояние, выделенное в множестве S,
- $L : S \rightarrow 2^A$ – функция разметки,
- $R \subseteq S \times S$ – множество переходов.

Путем в структуре Крипке $K = (S, s_0, L, R)$ будем называть максимальную последовательность $tr = s_0, s_1, s_2, \ldots, s_n, \ldots$ состояний структуры K таких, что s_i R s_{i+1} для любых двух соседних элементов последовательности.

Допустимая формула Φ задается следующей грамматикой:

$$\Phi ::= \mathbf{AG}\ \varphi \mid \mathbf{AF}\ \varphi \mid \mathbf{EG}\ \varphi \mid \mathbf{EF}\ \varphi \mid \varphi\ \mathbf{imply}\ \varphi, \backslash\backslash$$

где φ – подформула, задаваемая следующей грамматикой:

$$\varphi ::= p \mid \mathbf{not}\ \varphi \mid \varphi\ \mathbf{or}\ \varphi,$$

где p – высказывание множества A.

Семейство допустимых формул Φ для множества A всевозможных предохранителей совпадает с множеством формул, проверяемых средством верификации UPPAAL, и является подмножеством формул логики LTL_{-X} [24] с учетом следующих замечаний:

- допустимая формула φ_1 **imply** φ_2 соответствует формуле **AG**(**not** φ_1 **or** F φ_2) логики LTL и
- допустимая формула **E**η выполнима тогда и только тогда, когда невыполнима LTL-формула **A**(**not** η).

Отношение выполнимости $\models$ допустимых формул на структурах Крипке определяется, с учетом сделанных замечаний, стандартным для формул логики LTL способом [24].

Пусть $tr = s_0, s_1, \ldots$; $tr' = s_0', s_1', \ldots$ – бесконечные пути в структурах Крипке (S, s_0, L, R) и (S', s_0', L', R') соответственно. Пути tr, tr' будем называть эквивалентными по прореживанию (stuttering equivalent), если найдутся такие бесконечные последовательности $0 = i_0 < i_1 < \ldots$, $0 = j_0 < j_1 < \ldots$, что для любого $k \geq 0$ выполняются равенства

$$L(s_{ik}\}) = L(s_{ik+1}) = \ldots = L(s_{i(k+1)-1}) = L'(s_{jk}') = L'(s_{jk+1}') = \ldots = L'(s_{j(k+1)-1}').$$

Иначе говоря, два бесконечных пути эквивалентны по прореживанию, если их можно разбить на конечные блоки так, чтобы состояния k-го блока одного пути были помечены одинаково и так же, как и состояния k-го блока второго пути. Аналогично вводится понятие прореживания для конечных путей (последовательности i_k и j_k в этом случае конечны и оканчиваются числами, на единицу меньшими длин трас tr и tr' соответственно).

Свойство прореживания легко переносится с путей на сами структуры Крипке. Структуры Крипке K, K' будем называть эквивалентными по прореживанию, если

- для любого пути в K существует эквивалентный ему по прореживанию путь в K' и
- для любого пути в K' существует эквивалентный ему по прореживанию путь в K.

Доказано [24], что для любых структур Крипке K, K', эквивалентных по прореживанию, и любой формулы Φ логики LTL_{-X} верно соотношение

$$K \models \Phi \Leftrightarrow K' \models \Phi.$$

Прямым следствием данного факта является корректность алгоритма трансляции.

Теорема. *Для любых структур Крипке* K, K', *эквивалентных по прореживанию, и любой допустимой формулы* Φ *верно соотношение*

$$K \models \Phi \Leftrightarrow K' \models \Phi.$$

Средство верификации UPPAAL позволяет проверять также свойства системы, содержащие особый символ **deadlock**.

Синтаксис допустимых формул с символом **deadlock** остается тем же, что и для допустимых формул без него, если добавить символ **deadlock** к множеству A.

Семантика допустимых формул с символом **deadlock** остается той же, что и для формул логики LTL, если к существующим правилам выполнимости добавить следующее: формула ``**deadlock**'' выполнима в состоянии s структуры Крипке $K = (S, s_0, L, R)$ тогда и

только тогда, когда ни для какого состояния s' множества S не верно s R s'. Так как символ **deadlock** обозначает конец пути в структуре Крипке, справедлива следующая теорема.

Теорема. *Для любых структур Крипке* K, K', *эквивалентных по прореживанию, и любых допустимых формул с символом* **deadlock** *вида* Φ = **AF deadlock**, Φ = **AG**(**not deadlock**), Φ = **EF deadlock**, Φ = **EG**(**not deadlock**), Φ = φ **imply deadlock** *верно соотношение*

$$K \models \Phi \Leftrightarrow K' \models \Phi.$$

Для обоснования корректности алгоритма трансляции теперь достаточно показать, что структуры, описывающие поведения автомата и сети, эквивалентны по прореживанию. Это можно показать следующим образом.

Сначала определим структуры Крипке, описывающие поведение автомата и сети. Затем обозначим соответствие состояний описанных структур. Затем обозначим соответствие их переходов. Затем обозначим соотношение их начальных состояний, позволяющее заключить, что упомянутые структуры Крипке находятся в состоянии прореживания.

Состояниями структур Крипке K_A, K_N, описывающих поведение автомата и сети соответственно, являются их конфигурации. Множества переходов структур Крипке K_A, K_N соответствуют всевозможным изменениям конфигураций.

Множество таймеров сети, получаемой в результате применения алгоритма трансляции, совпадает с множеством таймеров автомата. Множество переменных сети включает в себя множество переменных автомата.

Множество A, над которым строятся структуры Крипке K_A, K_N и допустимые формулы, проверяющиеся на них, есть множество всевозможных предусловий над общими переменными и таймерам автомата и сети (то есть над переменными и таймерами автомата).

Значение функций разметки в каждом состоянии структур Крипке K_A, K_N есть множество истинных в этом состоянии предусловий. Согласно такому заданию функции разметки, ее значения в состояниях структур Крипке K_A, K_N с совпадающими значениями общих переменных и таймеров автомата также совпадают.

Значения переменных сети, отсутствующих в автомате, однозначно определяются вектором состояний процессов сети. Кроме того, алгоритм трансляции обеспечивает взаимно однозначное соответствие семейства всех допустимых множеств состояний в конфигурации автомата и некоторого множества векторов состояний процессов сети. Такое соответствие можно естественным образом расширить на состояния структур Крипке K_A, K_N, добавив к соответствующим состояниям одинаковые значения общих переменных и таймеров и

восстановив значения добавочных переменных. Таким образом, значения функций разметки на соответствующих состояниях совпадают.

Алгоритм трансляции ставит в соответствие переходу в структуре Крипке K_A последовательность переходов в структуре Крипке K_N, начинающуюся и оканчивающуюся в состояниях, соответствующих началу и концу перехода в K_A. В процессе выполнения этой последовательности значение общих переменных и таймеров (а значит, и значение функции разметки) может измениться лишь однажды за всю последовательность. Кроме того, между начальным состоянием структуры Крипке K_N состоянием, соответствующим начальному состоянию структуры Крипке K_A, находится лишь конечное число состояний с одинаковым значением функции разметки.

Проведенные рассуждения позволяют считать, что структуры Крипке K_A и K_N, описывающие поведение автомата и сети, являются эквивалентными по прореживанию, что обосновывает корректность алгоритма трансляции.

1.5 Минимизация временных автоматов

Система верификация, разработанная в рамках настоящего проекта, позволяет сводить проверку свойств вычислений распределенных систем, описание которых представлено UML-диаграммами, к задаче анализа поведения сетей временных автоматов, для решения которой применяется программно-инструментальное средство верификации UPPAAL.

Состояние вычисления временного автомата характеризуется двумя компонентами – состоянием управления автомата и конечным набором вещественных чисел, представляющих собой значения (показания) таймеров. Таким образом, в отличие от дискретного конечного автомата, временной автомат допускает бесконечное (вообще говоря, континуальное) множество состояний вычисления. Для того чтобы эффективно проверить свойства вычислений временных автоматов, множество всех состояний вычисления этих автоматов разбивается на конечное семейство регионов; каждый регион описывается двумя составляющими – состоянием управления автомата и конечной системой неравенств вида $t \leq c$, $c \leq t$ и $t' \leq t'' + c$, задающих выпуклый многогранник (полиэдр) в многомерном вещественном пространстве показаний таймеров. На множестве регионов определяется отношение переходов, соответствующее отношению переходов анализируемого временного

автомата, и в результате этого образуется конечная система размеченных переходов, которая полностью отражает всевозможные вычисления этого автомата. Именно для этой системы переходов проводится проверка темпоральных свойств временного автомата.

Этот метод верификации вычислительных систем реального времени был предложен в серии работ [25, 26]. В этих работах было показано, что число регионов в системе переходов, соответствующей временному автомату, может оказаться экспоненциально зависящим от количества таймеров, используемых автоматом. В связи с этим возникает задача минимизации временных автоматов: сократить размер описания временного автомата и/или соответствующей ему системы переходов при сохранении множества вычислений, порождаемых исходным временным автоматом. Существует два основных подхода к решению этой задачи. Первый из них оставляет без изменения описание исходного автомата, но осуществляет минимизацию пространства регионов в системе переходов по ходу ее построения. Второй подход проводит оптимизацию самого временного автомата подобно тому, как это осуществляется для дискретных конечных автоматов. В данном разделе отчета рассмотрен первый из указанных подходов и описан один из возможных алгоритмов минимизации системы переходов (графа регионов), соответствующих временному автомату.

Вначале приведем определение временного автомата, адаптированное для решения задачи минимизации пространства регионов в системе переходов.

Рассмотрим конечное множество вещественных переменных $X = \{x_1, x_2, \ldots, x_n\}$, которые будем называть *таймерами*. *Элементарным линейным ограничением* называется всякое неравенство одного из следующих видов $x \le c,\ x < c,\ c \le x,\ c < x,\ x \le x' + c,\ x < x' + c,$ где x, x' - таймеры, а c - целое число. Решением элементарного линейного ограничения является множество наборов вещественных чисел $< r_1, r_2, \ldots, r_n >$, удовлетворяющих соответствующему неравенству. Очевидно, что решением всякого линейного ограничения является некоторое полупространство n-мерного вещественного пространства. Любое конечное множество элементарных линейных ограничений называется *линейным ограничением*. Решением линейного ограничения является пересечение решений неравенств, составляющих это ограничение. Это множество, являющееся выпуклым многогранником (возможно, пустым), будем называть *временной зоной*, или просто *зоной*. Семейство всех возможных зон обозначим записью $Z(n)$.

Временной автомат описывается четверкой $A = (S, X, s_{init}, T)$, где

S - конечное множество *состояний управления*,

X - конечное множество таймеров,

s_{init} - начальное состояние управления,

$T \subseteq S \times 2^X \times Z(n) \times S$ - конечное отношение переходов.

Каждая четверка $< s, Y, z, s' >$ из множества T означает, что из состояния управления s в состояние управления s' возможен переход в том случае, когда показания таймеров образуют набор, принадлежащий зоне z. При этом по окончании перехода значения всех таймеров из множества Y полагаются равными нулю (сброс таймеров). Переходы вида $< s, Y, z, s' >$ будем обозначать записью $s \xrightarrow{Y,z} s'$.

Вычислением автомата $A = (S, X, s_{init}, T)$ называется последовательность пар

$(s_0, \vec{x}_0) \Rightarrow (s_1, \vec{x}_1) \Rightarrow (s_2, \vec{x}_2) \Rightarrow \cdots \Rightarrow (s_k, \vec{x}_k) \Rightarrow (s_{k+1}, \vec{x}_{k+1}) \Rightarrow \cdots$,

удовлетворяющая следующим двум условиям:

- $s_0 = s_{init}$, $\vec{x}_0 =< 0,0,\ldots,0 >$, для любого k, $k \geq 0$, верно одно из двух
- $s_k = s_{k+1}$ и $\vec{x}_{k+1} = \vec{x}_k + < \delta, \delta, \ldots, \delta >$ для некоторого вещественного числа δ, $\delta > 0$ (продвижение времени), существует такой переход $s_k \xrightarrow{Y,z} s_{k+1}$ из множества T, что $\vec{x}_k \in z$ и набор $\vec{x}_{k+1}$ отличается от набора $\vec{x}_k$ только тем, что все показания всех таймеров из множества Y в наборе $\vec{x}_{k+1}$ равны 0 (срабатывание перехода).

Для эффективной верификации свойств вычислений конечного автомата в статье [26] было введено понятие региона и на основании этого понятия введена система размеченных переходов (граф регионов).

Регионом называется всякое множество F, $F \subseteq S \times R^n$. Регион $\{< s, \vec{x} >: \vec{x} \in Z\}$, где Z - некоторая зона, обозначается записью (s, Z). На множестве регионов вводится следующее *отношение регионального продвижения* . Пусть F, F' и $< s, \vec{x} > \in F$. Тогда отношение продвижения $< s, \vec{x} > \rightarrow_F F'$ имеет место в том случае, когда выполняется одно из следующих двух условий:

- существует такое вещественное число δ, $\delta > 0$, для которого выполняются два включения $< s, \vec{x} + \delta > \in F$ и $\{< s, \vec{x} + \delta' >: 0 \leq \delta' \leq \delta\} \subseteq F \cup F'$ (продвижение времени),

- существует такое вещественное число $\delta, \delta \geq 0$, и такой набор $< s', \vec{x} + \delta > \in F'$, для которых выполняются два включения $\{< s, \vec{x} + \delta' >: 0 \leq \delta' \leq \delta\} \subseteq F$ и $< s, \vec{x} > \Rightarrow < s', \vec{x} + \delta >$ (продвижение управления).

Разбиение пространства $S \times R^n$ на регионы называется *стабильным* в том случае, когда для любой пары регионов F, F' и для любого набора $< s, \vec{x} > \in F$, если выполняется отношение регионального продвижения $< s, \vec{x} > \to_F F'$, то отношение регионального продвижения того же типа $< s', \vec{x}' > \to_F F'$ выполняется и для любого другого набора $< s', \vec{x}' > \in F$.

Графом регионов, соответствующим временному автомату A и начальному разбиению $Q_0 = \{< s, R^n >: s \in S\}$ пространства $S \times R^n$ называется граф $GR(A, Q_0)$ вершинами которого являются регионы, образующие некоторое стабильное разбиение Q пространства $S \times R^n$, удовлетворяющее следующему требованию: для любого региона $F, F \in Q$, существует регион $F', F' \in Q_0$, для которого верно включение $F \subseteq F'$. В графе $GR(A, Q_0)$ из региона F_1 в регион F_2 ведет дуга в том и только том случае, когда для некоторого набора $< s, \vec{x} > \in F$ выполняется отношение регионального продвижения $< s, \vec{x} > \to_{F_1} F_2$. В статье [26] было установлено, что для любого временного автомата A и любого начального разбиения Q_0 пространства $S \times R^n$ существует конечный граф регионов $GR(A, Q_0)$. Задача минимизации системы переходов, соответствующих временному автомату A состоит в построении графа регионов как можно меньшего размера.

Рассмотрим следующую схему алгоритма, который применяется для решения задачи минимизации числа состояний в системе переходов (недетерминированном конечном автомате).

Пусть задана некоторая система переходов $B = (S, s_0, \to)$ с множеством состояний S, начальным состоянием s_0 и отношением переходов $\to \subseteq S \times S$. Для состояния s и множества состояний X запись $s \to X$ будет обозначать выполнимость отношения переходов $s \to s'$ для некоторого состояния $s', s' \in X$. Для произвольного разбиения ρ множества состояний S на классы состояний класс $X, X \in \rho$, называется стабильным, если для любого состояния s из класса X выполнимость отношения $s \to Y$ влечет выполнимость отношения $s' \to Y$ для любого состояния s' из класса X. Разбиение ρ называется бисимуляцией, если каждый класс этого разбиения стабилен.

Известно, что выполнимость формул темпоральных логик ветвящегося времени (и в том числе логики TCTL) инвариатна относительно отношения бисимуляции. Это означает, что при решении задачи минимизации систем переходов достаточно для заданной системы переходов вычислить максимальное отношение бисимуляции этой системы и затем построить минимальную систему переходов состояниями которой являются классы эквивалентности пространства состояний по максимальному отношению бисимуляции. Таким образом, задача минимизации системы переходов сводится к задаче вычисления максимального отношения бисимуляции в пространстве состояний этой системы. Последняя задача имеет следующую простую схему решения.

Рассмотрим систему переходов $B = (S, s_0, \to)$, для которой введем следующие три функции:

- функция $split(X, \rho)$ вычисляет для класса X разбиения ρ минимальное расщепление класса X на стабильные относительно разбиения ρ подклассы;
- функция $succ(X, \rho)$ вычисляет множество $\{Y : \exists x (x \in X \wedge x \to Y)\}$ всех классов, хотя бы одно из состояний которых достижимо из некоторого состояния класса X;
- функция $pred(X, \rho)$ вычисляет множество $\{Y : \exists y (y \in Y \wedge y \to X)\}$ всех классов, хотя бы из одного состояния которых достижимо некоторое состояния класса X.

Тогда алгоритм, вычисляющий максимальную бисимуляцию ρ, являющуюся сужением заданного начального разбиения ρ_0, имеет следующий вид (здесь α - множество классов, достижимых из класса, содержащего начальное состояние системы переходов, а σ - множество классов текущего разбиения ρ, которые стабильны относительно ρ) :

$\rho := \rho_0$; $\alpha := \{[s_0]_\rho\}$; $\sigma := \varnothing$;

while $\alpha \neq \sigma$ do

let $X \in \alpha \setminus \sigma$;

if $split(X, \rho) = X$

then { $\sigma := \sigma \cup \{X\}$; $\alpha := \alpha \cup succ(X, \rho)$ }

else

{

$\alpha := \alpha \cup succ(X, \rho)$;

if $\exists Y\ (Y \in \alpha' \wedge s_0 \in Y)$ then $\alpha := \alpha \cup Y$;

$\sigma := \sigma \setminus pred(X, \rho);\ \ \rho := (\rho \setminus \cup\{X\}) \cup \alpha'$;

}

fi

od

Применение данного алгоритма к решению задачи минимизации графа регионов требует уточнения функций $split(X,\rho)$, $succ(X,\rho)$ и $pred(X,\rho)$, поскольку в отличие от конечных систем переходов граф регионов допускает бесконечное множество регионов. Для того чтобы преодолеет эту трудность, предлагается ввести ряд специальных операций на множестве регионов.

Для произвольной пары зон Z_1, Z_2 обозначим записью

$Z_1 \setminus Z_2$ такое множество попарно непересекающихся зон, что объединение $\{Z_2\} \cup Z_1 \setminus Z_2$ образует разбиение зоны Z_1. Тогда положим $Z_1 \bigcup Z_2 = \{Z_1 \cap Z_2\} \cup Z_1 \setminus Z_2 \cup Z_2 \setminus Z_1$;

$Z_1 \Uparrow Z_2$ множество Z векторов (наборов показаний таймеров) $\vec{x}$, которые для некоторого положительного вещественного числа δ удовлетворяют двум условиям:

$\vec{x} + <\delta, \delta, \ldots, \delta> \in Z_2$,

для любого $\delta', 0 \le \delta' \le \delta,$ верно включение $\vec{x} + <\delta', \delta', \ldots, \delta'> \in Z_1$.

Тогда в терминах этих операций функции $split(X,\rho)$, $succ(X,\rho)$ и $pred(X,\rho)$ могут быть определены следующим образом.

Вначале введем функцию расщепления для пары регионов:

$split((s,Z),(s',Z')) = (s,Z) \bigcup \bigcup_{s \xrightarrow{z,a} s'} (s, Z \Uparrow (Z \cap z \cap a^{-1}(Z')))$, для всякого состояния управления $s \neq s'$;

$split((s,Z),(s,Z')) = (s,Z) \bigcup (s, Z \Uparrow Z') \bigcup \bigcup_{s \xrightarrow{z,a} s'} (s, Z \Uparrow (Z \cap z \cap a^{-1}(Z')))$;

На основании этой функции можно определить функцию расщепления региона относительно заданного разбиения ρ

$split((s,Z),\rho) = \bigcup_{(s',Z') \in \rho} split((s,Z),(s',Z'))$.

Функции вычисления предшествующих и последующих регионов определяются так:

$$pred((s,Z),\rho)=\{(s,Z'):Z'\Uparrow Z\neq\varnothing\}\cup\bigcup_{s\xrightarrow{z,a}z'}\{(s',Z')\in\rho:\ a(Z'\cap z)\cap Z\neq\varnothing\};$$

$$succ((s,Z),\rho)=\{(s,Z'):Z\Uparrow Z'\neq\varnothing\}\cup\bigcup_{s\xrightarrow{z,a}z'}\{(s',Z')\in\rho:\ a(Z\cap z)\cap Z'\neq\varnothing\}.$$

Таким образом, из описанной выше общей схемы вычисления максимальной бисимуляции в конечной системе переходов получаем алгоритм минимизации системы переходов (графа регионов), соответствующей временному автомату. Эффективность этого алгоритма существенно зависит от эффективности реализации процедур, вычисляющих функции $split(X,\rho)$, $succ(X,\rho)$ и $pred(X,\rho)$, которые, в свою очередь, зависят от методов вычисления операции $Z_1 \setminus Z_2$ и $Z_1 \Uparrow Z_2$. Исследование этого вопроса планируется провести на следующем этапе выполнения проекта.

В данном разделе отчета описан алгоритм минимизации системы переходов для простейшего временного автомата. Однако в системе UPPAAL в качестве моделей систем реального времени используются сети взаимодействующих временных автоматов, в которых используются операции синхронизации. Описанный выше метод минимизации может быть применен и для сетей временных автоматов. В этом случае регионами будут являться наборы вида $< s_1, s_2, \dots, s_n, v_1, v_2, \dots, v_k, Z >$, где $s_1, s_2, \dots, s_n$ - состояния управления отдельных временных автоматов, образующих рассматриваемую сеть, $v_1, v_2, \dots, v_k$ - значения предметных переменных, которыми оперируют временные автоматы сети, а Z - зона. Для сетей временных автоматов соответствующим образом модифицируется система переходов (граф регионов). Более подробное изучение вопроса об адаптации описанного в настоящем разделе алгоритма минимизации для упрощения графов регионов сетей временных автоматов планируется провести на последующем этапе выполнения проекта.

1.6 Модификация транслятора UML в UPPAAL

1.6.1 Введение

На предыдущих этапах проекта были разработаны алгоритм и средство преобразования диаграмм состояний UML во временные автоматы, использующиеся в среде верификации UPPAAL. Такое средство позволяет верифицировать свойства системы на этапе проектирования, до написания кода и, тем самым, сократить сроки разработки системы и повысить качество разработки.

Язык UML предназначен для моделирования широкого класса систем, поэтому в его стандарте намеренно не фиксирована семантика. Однако для формальной проверки свойств модели необходимо строго задать синтаксис и семантику всех используемых элементов диаграмм состояний. В данной работе заданы дополнительные ограничения на структуру диаграмм состояний и синтаксис выражений и, таким образом, устраняется неоднозначность интерпретации элементов диаграмм.

Транслятор диаграмм UML в автоматы UPPAAL получает на вход диаграмму состояний в формате XMI, экспортируемую из средства ArgoUML. После преобразования диаграммы во внутреннее представление непосредственная трансляция осуществляется в два этапа. Сначала диаграмма переводится в промежуточное представление – иерархический временной автомат (HTA) [23], затем этот автомат преобразуется в сеть временных автоматов в соответствии с разработанным алгоритмом.

Для формальной проверки свойств модели строго задается синтаксис и семантику всех используемых элементов диаграмм состояний. Чтобы избежать неоднозначности интерпретации элементов диаграмм, наложены дополнительные ограничения на структуру диаграмм состояний и синтаксис выражений.

Синтаксис и семантика простых состояний определяется в рамках стандартной метамодели UML. Также разрешается использование композитных состояний двух типов: последовательных и содержащих параллельно выполняющиеся регионы. При выборе синтаксиса предусловий, обусловливающих срабатывание переходов, инвариантов, обусловливающих пребывание в состояниях, и действий переходов (то есть операторов, выполняющихся при осуществлении переходов) мы руководствовались стремлением достичь совместимости со стандартным синтаксисом языков, подобных языку Си. В этих выражениях допускается использование целочисленных и булевых переменных, арифметических отношений и операций, а также таймеров, принимающих вещественные значения. Семантика этих выражений совпадает с семантикой операторов в языке Си.

1.6.2 Расширенный синтаксис предусловий

Имевшийся синтаксис предусловий [2] позволял теоретически записывать любые выражения, приводя их предварительно к конъюнктивной нормальной форме. Однако использовать КНФ не всегда удобно, так как выражение может получиться более громоздким, чем могло бы быть. Поэтому синтаксис предусловий был расширен, и стали

допустимы любые логические выражения с операциями конъюнкции, дизъюнкции и отрицания.

Например, теперь выражение

!task2_time_ex && !port1 || !task2_time_ex && !port2 || !hard_rt && !port2 || !hard_rt && !port1,

встречающееся в модели расписания для процессора, можно записать в более простой и естественной форме:

(!port1 || !port2) && (!task2_time_ex || !hard_rt)

Новый синтаксис предусловий:

```
Guard ::= OrExpr
OrExpr  ::= AndExpr | AndExpr ‘||’ AndExpr
AndExpr  ::= Disj | Disj ‘||’ Disj
Disj  ::= (OrExpr) | NegExpr | in(S) | Expr
NegExpr  ::= !(OrExpr) | ! BoolExpr
Expr  ::= BoolExpr | CompAtom
CompAtom  ::= Sum (==|!=|>|<|>=|<=) Sum
Sum ::= IntExpr | IntExpr ‘+’ IntExpr
IntExpr ::= IntVar | IntConst | ClockVar
ClockVar ::= <идентификатор>
IntVar   ::= <идентификатор>
IntConst ::= <целочисленная константа>
BoolExpr ::= BoolVar | false | true
BoolVar  ::= <идентификатор>
```

1.6.3 Новый алгоритм преобразования UML в HTA

По определению HTA в этих автоматах запрещены предусловия и действия на переходах между служебными состояниями (входами и выходами). Однако в сложных моделях с вложенными метасостояниями из-за этого необходимо добавлять длинные цепочки входов и выходов, которые после оптимизации автомата удаляются. При этом требование не ставить предусловий на такие переходы иногда противоречит естественным стремлениям разработчиков систем, и, как показала практика, многие ошибки возникают именно из-за нарушений в соблюдении этого ограничения. Кроме того, при трансляции

наличие предусловий и действий на выходных переходах обрабатывается транслятором без ошибок, но сами предусловия удаляются, в результате иногда становится непросто понять, почему модель работает неверно.

Исходя из этого, было решено изменить схему трансляции UML в HTA, чтобы транслятор сам находил описанные выше ситуации и переносил предусловия и действия на нужные переходы, не теряя их.

При удалении составных состояний входы и выходы в них автоматически не удаляются, в результате остаются цепочки вида «простое состояние» – «выход» – «простое состояние». В такой цепочке есть два перехода. Если предусловия и действия есть не более чем на одном из переходов, то выход удаляется, между простыми состояниями остается один переход с предусловием и действием (рисунок 14). Если предусловия и действия есть на обоих переходах, выход заменяется на простое состояние (рисунок 15).

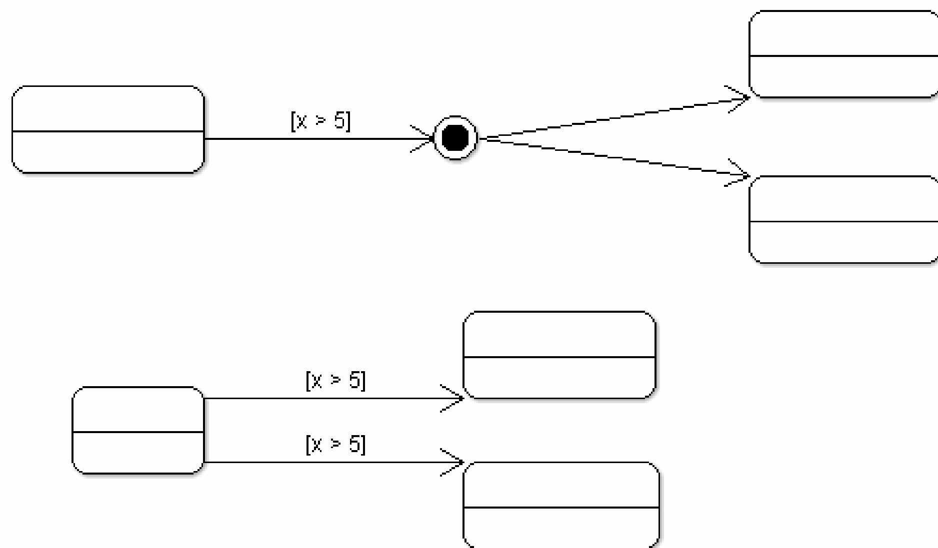

Рисунок 14 - Удаление выходного состояния

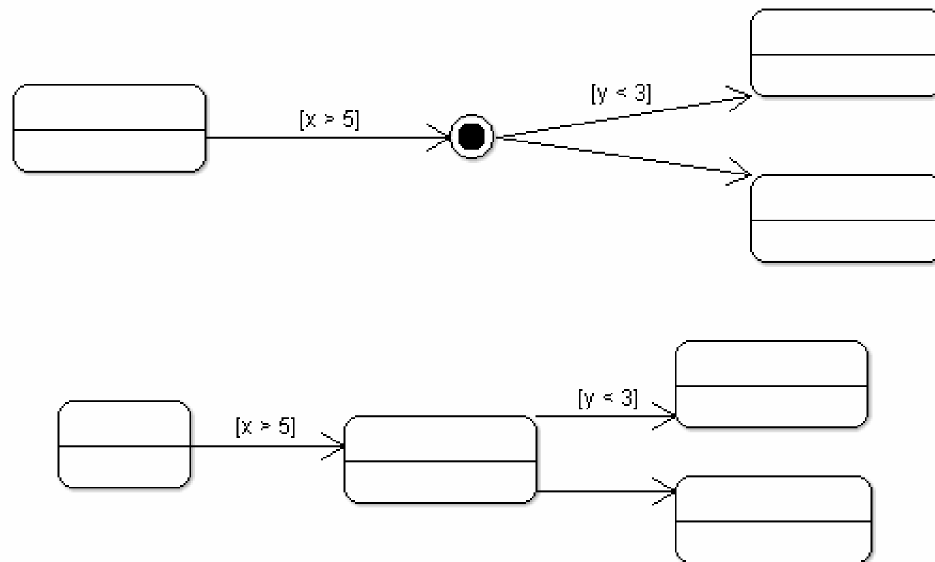

Рисунок 15 - Замена выходного состояния на простое.

1.7 Алгоритм восстановления параметров модели по контрпримеру в UPPAAL

1.7.1 Введение

Задача построения и анализа контрпримеров неизбежно возникает при верификации моделей вычислительных систем. В том случае, если некоторое проверяемое свойство поведения системы не выполняется, необходимо установить причину этого явления. Для этого необходимо отыскать хотя бы одно из тех вычислений системы, которые не удовлетворяют проверяемому свойству. Большинство средств верификации, включая UPPAAL, снабжено процедурами построения контрпримеров. Однако описания этих контпримеров проводятся на языке входной модели средства верификации. В случае использования UPPAAL таким языком является язык сетей временных автоматов. Поскольку в разрабатываемой нами системе проектирования РСРВ модели представляются в виде UML диаграмм, которые транслируются в сети временных автоматов, возникла необходимость в отображении трасс вычислений в сетях временных автоматов в трассы вычислений в UML диаграммах, т.е. необходимо по трассе, полученной в средстве верификации, восстановить последовательность шагов исходной программы. Поскольку трансляция из UML в UPPAAL делается автоматически, преобразование трасс также следует сделать автоматическим. Для этого необходимо сделать следующее:

1. Проанализировать файл трассы, сохраняемого из UPPAAL
2. Установить соответствие между состояниями временного автомата и состояниями UML-диаграммы
3. Восстановить значения переменных и таймеров на каждом шаге трассы

1.7.2 Формат трасс UPPAAL

Файлы трасс UPPAAL предназначены прежде всего для визуализации трассы в пошаговом режиме в графическом интерфейсе UPPAAL, и изначально не предусматривается их читаемость человеком. Тем не менее, формат трасс текстовый, и его синтаксис возможно установить. Ниже приведено общее описание формата трассы, полученное в результате обратной инженерии.

Файл трассы UPPAAL представляет собой текстовый файл, в котором каждый элемент, описанный ниже, расположен в отдельной строке. Разделителем служат строки со знаком точки.

Описание формата:

- Все состояния каждого автомата UPPAAL нумеруются с нуля в порядке их появления в xml-файле.
- Все переходы нумеруются с нуля в порядке их появления в xml.
- Все процессы (автоматы) нумеруются с нуля в порядке их появления в xml.
- Все переменные нумеруются с нуля в порядке их появления в xml.
- Все таймеры нумеруются с **единицы** в порядке их появления в xml. Значение 0 зарезервировано под глобальный таймер.
- Трасса содержит состояния, значения таймеров и переменных и переходы (именно в таком порядке).
- После каждого состояния стоит одна точка в отдельной строке.
- После блока со значениями таймеров стоят две точки в отдельных строках (длина блока таймеров может меняться, поэтому необходим отдельный признак конца).
- После блока со значениями переменных стоит одна точка в отдельной строке.
- После каждого перехода стоит одна точка в отдельной строке.
- В конце трассы стоит точка в отдельной строке.
- В начале трассы записано начальное состояние и значения переменных и таймеров.
- Дальше записаны блоки, сначала очередное состояние, затем значения таймеров и переменных, затем сделанный переход.

- Состояние записывается как набор номеров активных состояний во всех процессах, каждый номер в отдельной строке.
- Переход записывается как пара чисел: номер процесса и номер перехода, в одной строке, эти два числа разделены пробелом.
- Если при переходе задействуются сразу несколько ребер (при посылке и приеме сигналов), то каждый обозначается описанной выше парой чисел в отдельной строке.
- Значения переменных записываются подряд, в каждой строчке – значение соответствующей переменной, каждый раз для всех переменных.
- Значения таймеров записываются блоками по три числа, разделенными строкой, содержащей точку (блок таймеров заканчивается двумя точками). Три числа в блоке имеют следующие значения: номер первого таймера, номер второго таймера, верхняя граница их разницы, умноженная на 2.

Общая схема файла трассы:

<Начальное состояние>
.
<Второе состояние>
.
<Переход из начального во второе состояние>
.
<Третье состояние>
<…>

Схема описания состояния:

<Номер состояния 1-го автомата>
<Номер состояния 2-го автомата>
<…>
<Номер состояния последнего автомата>
.
<Выражение над таймерами>
.
<Выражение над таймерами>
.
<…>
.
<Значение 1-й переменной>
<…>
<Значение последней переменной>

Поясним перечисленные правила на примере простой трассы.

Пусть есть два процесса (рисунки 16-17):

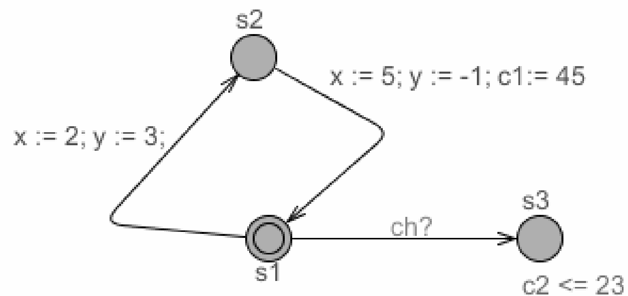

Рисунок 16 – Процесс 1

Рисунок 17 – Процесс 2

- В первом состояния s1, s2, s3, во втором - s1, s2
- Инвариант s3: c2 <= 23
- Переменные x, y
- Таймеры c1, c2
- Канал ch
- Переходы и действия в первом процессе: s1 -> s2 (x := 2; y := 3), s2 -> s1 (x := 5; y := -1; c1 := 45), s1 -> s3 (ch?); во втором процессе: s1 -> s2 (ch!)

```
0
0 // Начальные состояния - s1 и s1
. // Начинается блок таймеров
0
1
0 // c1 >= 0
.
1
2
0 // c1 - c2 <= 0
.
2
1
0 // c2 - c1 <= 0
.
. // Конец блока таймеров, начало блока переменных
0 // x = 0
```

```
0 // y = 0
. // Конец описания начального состояния, начало описания второго состояния
1 // Первый процесс перешел в s2
0 // Второй остался в s1
. // Таймеры на втором шаге
0
1
0 // c1 >= 0
.
1
2
0 // c1 - c2 <= 0
.
2
1
0 // c2 - c1 <= 0
.
. // Переменные на втором шаге
2 // x = 2
3 // y = 3
.
0 3 // Переход 3 в первом (нулевом) процессе
. // Начало описания третьего состояния
0
0 // Вернулись в s1
.
0
1
-90 // c1 >= 45
.
1
2
90 // c1 - c2 <= 45
.
.
5 // x = 5
-1 // y = -1
.
0 2 // Переход 2
. // Начало описания четвертого состояния
2 // Перешли в s3
1 // Перешли в s2
.
0
1
-90 // c1 >= 45
.
1
2
```

```
90 // c1 - c2 <= 45
.
2
0
46 // c2 <= 23
.
.
5
-1 // x и y не поменялись
.
1 1
0 1 // Задействованы сразу два перехода
.
. // Конец трассы
```

1.7.3 Преобразование имен состояний

Так как перед экспортом иерархического временного автомата в UPPAAL создается внутреннее представление для временных автоматов, получить по номерам имена состояний, переменных и таймеров несложно.

В автоматах UPPAAL каждому простому состоянию UML соответствует одно несрочное состояние. Помимо этого в UPPAAL есть множество служебных промежуточных состояний, которые не имеют прямых соответствий в UML. Однако в UML нас интересуют только простые состояния, в которых модель может находиться некоторое время. Составные состояния интереса не представляют, поскольку модель всегда находится в одном из вложенных в них простых состояний, либо в процессе входа или выхода, который считается мгновенным и с точки зрения поведения программы нас не интересует.

Таким образом, необходимо для каждого простого состояния UML указать соответствующее ему состояние UPPAAL. Согласно алгоритму трансляции, простое состояние s преобразуется в состояние с именем вида <s>_active_in_<P>. В процессе трансляции в UPPAAL можно при каждом создании такого состояния можно записывать этот факт в специальный лог.

Однако здесь возникает проблема с переименованием: к моменту трансляции в UPPAAL название простого состояния s вовсе не обязательно будет таким же, как в изначальном UML. Это может случиться из-за переименования состояний при преобразовании в НТА, которое делается во избежание совпадения имен состояний из разных вложенных автоматов. В результате требуется при каждом переименовании состояний записывать этот факт в лог.

Применяя записанные в лог переименования в обратном порядке, можно получить исходное название состояния. Если отбросить служебные состояния UPPAAL, в итоге для каждого состояния в трассе UPPAAL получается состояние диаграммы UML. Это по сути и есть трасса для UML-диаграммы, но в ней не хватает значений переменных и таймеров, которые необходимо знать для анализа трассы.

1.7.4 Вычисление значений таймеров

Как было сказано выше, значения таймеров в трассе UPPAAL не хранятся в явном виде, вместо этого там записаны неравенства, связывающие различные таймеры. Это связано с тем, что в самой системе UPPAAL значения таймеров показываются в таком виде. Тем не менее, для анализа трассы было бы полезно знать абсолютные значения таймеров. В особенности это было бы важно для проверки, соблюдаются ли в модели директивные сроки, заданные абсолютными константами.

В общем случае задача поиска решения системы линейных неравенств достаточно сложна, однако в данном случае рассматривается упрощенная задача. Все неравенства, как следует из описания формата трассы, имеют вид x_i-x_j<=c_k. Кроме того, система обладает следующим свойством: если разбить все таймеры на две непересекающиеся группы, то для любого такого разбиения обязательно найдется неравенство, в котором есть таймер из первой и второй группы. Такую систему можно решить алгоритмом описанным ниже.

Часть неравенств содержит «нулевой» таймер, обозначающий глобальное время. Фактически неравенство вида x_i-x_o<=c означает, что абсолютное значение таймера x_i не превышает c.

В результате изначально имеется ряд неравенств, задающих нижние и верхние границы некоторых таймеров. Далее эти неравенства складываются с остальными, в результате чего получаются неравенства для других таймеров, и так пока не будут найдены все значения. Можно представить алгоритм следующим псевдокодом.

```
function SolveTimers(inequalities):
    values = {} // Список нижних и верхних границ таймеров
    for (xi-xj<=ck) in inequalities:
        if i == 0:
            values = values ? (“xj >= -ck / 2”)
        else if j == 0:
            values = values ? (“xi <= ck / 2”)
    while True:
        leng = |values|
        for (xi-xj<=ck) in inequalities:
            for (xl <= d) in values:
                if l == j:
                    values = values ? (“xi <= d + ck/2”)
```

```
        for (x_l >= d) in values:
            if l == i:
                values = values ∪ ("x_j >= d - c_k/2")
    if |values| == leng: // Если не добавилось новых неравенств - выходим
        break
return values
```

Следует обратить внимание на то, что знак объединения в данном алгоритме должен работать так, чтобы не допускать добавления во множество неравенств values не только повторяющихся неравенств, но и более широких неравенств. Например, если уже есть неравенство $x < 5$, то $x < 10$ добавлять не нужно.

1.7.5 Заключение

В данном разделе был описан алгоритм построения трассы переходов UML по трассе UPPAAL. Алгоритм позволяет разобрать файл в формате трасс UPPAAL и, используя специально собранную при трансляции информацию, восстановить соответствующие события в терминах диаграммы UML. Также рассчитываются значения переменных и таймеров.

Описанный метод построения трассы был реализован в рамках средства трансляции UML в UPPAAL. Программа берет на вход трассу UPPAAL и файл со вспомогательной информацией. Примеры работы транслятора приведены в разделе 3.4.

1.8 Обзор методов оценки наихудшего времени выполнения и реализация метода оценки наихудшего времени выполнения

1.8.1 Введение

Во многих ситуациях вычислительным системам предъявляются строгие требования ко времени их реакции на внешние события. В случаях, когда запаздывание реакции даже на доли секунды может привести к существенным потерям, применяются системы реального времени. При этом возникает задача проверки, отвечает ли система предъявляемым ко времени реакции требованиям.

Задача оценки наихудшего времени выполнения (WCET) решается для определения, удовлетворяет ли система требованиям ограничений работы по времени. В равной степени оценка WCET важна и на этапе разработки системы, как метод априорного анализа кода и выявления несоответствий до начала тестирования системы.

При моделировании РВС РВ разработчику модели некоторые действия компонента моделируемой системы удобней описывать на алгоритмическом языке программирования, чем при помощи UML диаграмм. Поэтому некоторые простые состояния UML диаграмм, помечаются комментариями, содержащими код на языке C++. Знание этого кода позволяет, оценить время его выполнения и, таким образом, вычислить максимальное время пребывания системы в том или ином состоянии.

Согласно [27], невозможно найти оценку наихудшего времени выполнения произвольной программы. Как известно, в общем случае невозможно даже выявить завершимость задачи. Однако программы, функционирующие в РВС РВ, используют ограниченный набор конструкций языков программирования, гарантирующих, что программа закончит выполнение. Во-первых, эти программы в основном последовательные. Во-вторых, эти программы обрабатывают целочисленные данные. В-третьих, для этих программ явно задаются ограничения на число итераций циклов. В-четвёртых, запрещается рекурсивный вызов процедур для программ, функционирующих в РВС РВ.

На рисунке 18 изображено типичное распределение времен выполнения программы в зависимости от различных наборов входных данных. Внутренний интервал показывает распределение, которое можно получить при профилировке. Реальное распределение времен выполнения ограничено действительными наилучшим (BCET) и наихудшим (WCET) временами выполнения программы. Для программ с нетривиальным потоком управления, данные значения практически невозможно получить с помощью измерений. Оценки наихудшего времени выполнения, получаемые современными методами, превышают реальное WCET. Задачей оценки наихудшего времени выполнения является нахождение наиболее точного значения, которое при этом не ниже реального WCET.

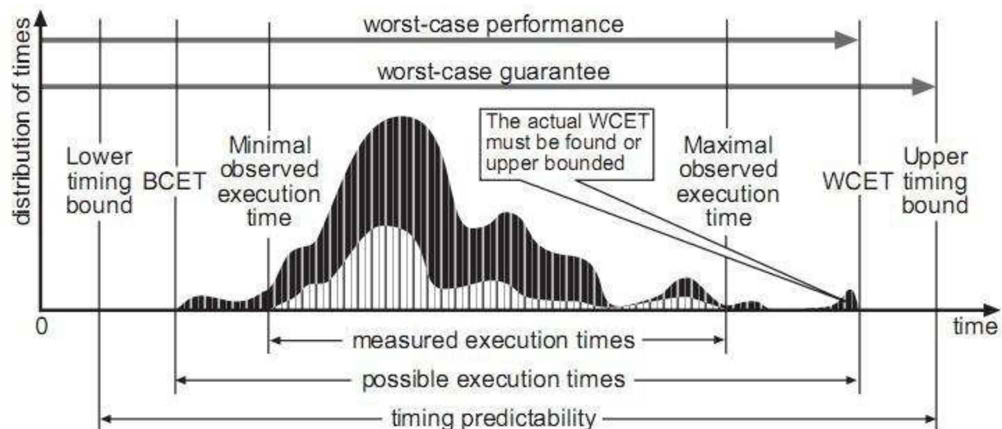

Рисунок 18 - Распределение времени выполнения задачи в зависимости от различных входных данных

1.8.2 Существующие методы оценки наихудшего времени выполнения

Все методы оценки наихудшего времени выполнения можно разделить на следующие классы:

- Статические методы.
- Измерительные методы.
- Гибридные методы.

Статические методы

Все статические методы (такие как методы, описанные в работах [27], [28], [29], [30]) основаны на аналитическом исследовании программы без её выполнения на целевом оборудовании.

Типичная схема анализа программы по статическому методу включает в себя следующую последовательность шагов (рисунок 19):

- дизассемблирование программы (в случае наличия исполняемого файла и недоступности исходного кода);
- построение графа потока управления (CFG);
- анализ потока управления (Control-Flow Analysis);
- анализ поведения процессора (Processor-Behavior Analysis);
- вычисление оценки наихудшего времени выполнения (estimate calculation).

Рисунок 19 - Типичная схема анализа программы статическим методом

Общая схема статических методов

В процессе анализа программы выполняются следующие операции:

1. Дизассемблирование программы.

Данный этап используется для получения исходного или ассемблерного кода программы в случае его отсутствия.

2. Построение графа потока управления.

На данном этапе производится анализ кода и построение графа потока управления программы (граф, у которого вершины – линейные участки, ребра – переходы между ними). В процессе анализа код программы разбивается на линейные участки, которые являются вершинами графа. Ребра формируются из переходов между линейными участками.

3. Анализ потока управления.

Производится анализ кода и графа потока управления программы и выявление таких свойств, как количество итераций циклов, границы рекурсий, наличие недостижимых линейных участков (участки кода, выполнение которых никогда не происходит).

После обхода графа к узлам и ребрам графа потока управления сопоставляется дополнительная информация, полученная в результате анализа. Например, к ребрам, соответствующим циклическим переходам, может быть сопоставлена информация о количестве проходов по ним.

4. Анализ поведения процессора.

На данном этапе происходит вычисление времени выполнения каждого линейного участка. При этом учитывается влияние архитектурных компонент: кэшей, конвейера, предсказателя переходов. При наличии любой из перечисленных компонент время выполнения каждой инструкции зависит от истории выполнения программы, а именно от последовательности ранее выполненных инструкций, переходов, обращений к памяти.

Для вычисления времени выполнения линейных участков используются такие подходы, как эмуляция выполнения и абстрактная интерпретация. Описание упомянутых подходов можно найти в работе [27].

5. Вычисление оценки наихудшего времени выполнения.

Производится проход по графу и вычисление наихудшего времени работы программы с учетом данных о времени выполнения и о выполнимости линейных участков. Существуют следующие подходы оценки наихудшего времени выполнения:

- метод, основанный на представлении программы в виде дерева (structure-based/tree-based)
- метод полного перебора путей (path-based)
- метод неявного перебора путей (implicit path enumeration techniques — IPET)
- метод, основанный на верификации программ на моделях(model checking)

Метод полного перебора

Перебираются возможные варианты прохода программы, для каждого из них вычисляется время выполнения и выбирается наихудшее.

Пример анализа наихудшего времени выполнения методом полного перебора изображен на рисунке 20. Среди всех путей выполнения на графе потока управления выбирается путь с наибольшим временем выполнения.

Данный метод имеет вычислительную сложность, экспоненциальную по числу ветвлений в программе. Однако он предоставляет возможность анализировать программы с недетерминированным выполнением (каждый вариант выполнения является отдельным путем выполнения программы и перебирается данным методом).

Structure-base метод

Граф потока управления представляется в виде дерева, у которого листья соответствуют линейным участкам программы, узлы соответствуют операторам условия, цикла или последовательного перехода. Листья помечаются временем выполнения. Существуют правила, согласно которым определённые группы листьев и узлов дерева можно объединить в один лист с вычислением наихудшего времени выполнения этого листа.

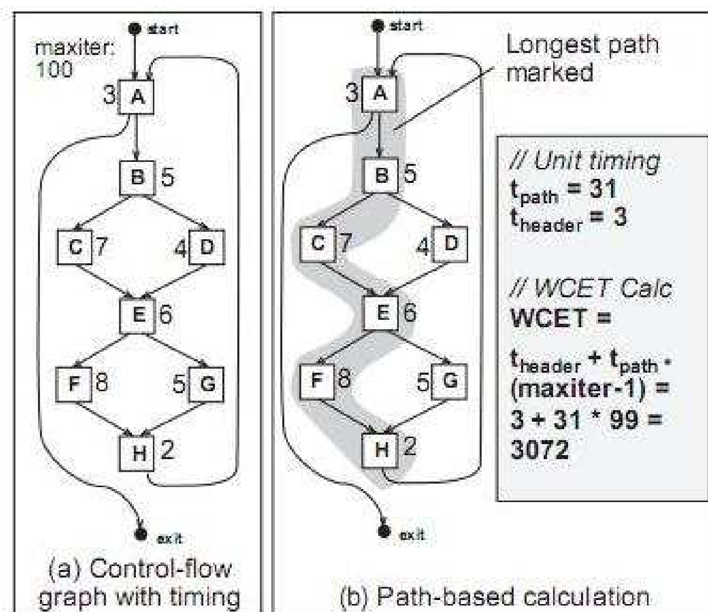

Рисунок 20 - Анализ наихудшего времени выполнения методом полного перебора

Дерево проходится снизу вверх, в нем находятся группы вершин, для которых выполнимо одно из известных правил объединения, и формируется один лист согласно этому правилу. Полученному листу присваивается наихудшее время выполнения соответствующей группы вершин. Проход по дереву продолжается до тех пор, пока не останется одна вершина. Наихудшим временем выполнения программы считается время выполнения этой вершины.

Пример работы данного метода изображен на рисунке 21**Ошибка! Источник ссылки не найден.**. В примере изображена последовательность действий по объединению вершин синтаксического дерева программы с вычислением времени выполнения.

Данный метод характеризуется относительно низкой вычислительной сложностью. Метод решает задачу за полиномиальное время по числу вершин построенного дерева.

Недостатком метода является узкий класс анализируемых программ. В исследуемых программах не должно присутствовать недетерминированного выполнения, иначе объединение вершин дерева будет давать неверную оценку наихудшего времени выполнения.

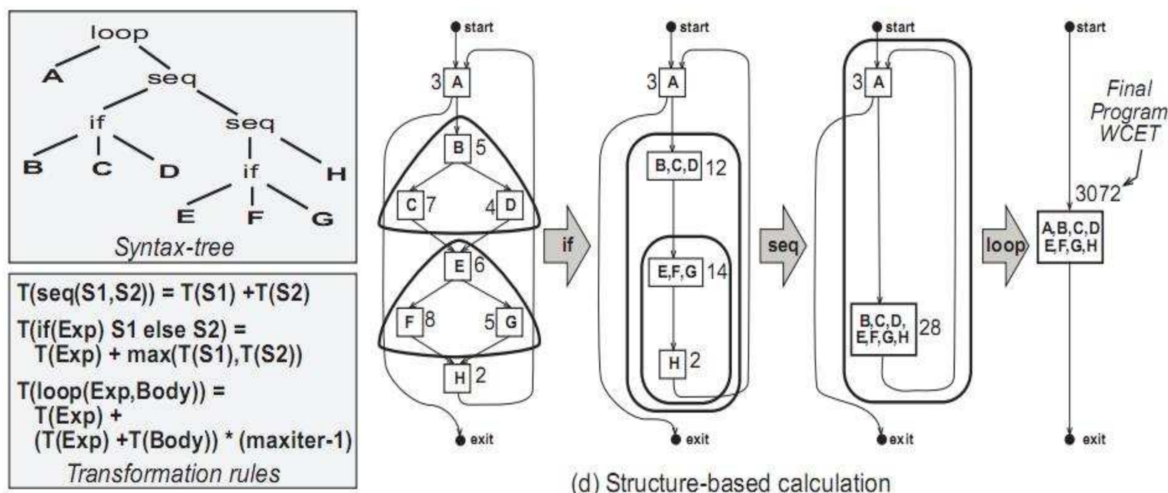

Рисунок 21 - Анализ наихудшего времени выполнения structure-based методом

Метод неявного перебора

Данный метод основан на приведении исходной задачи к задаче целочисленного линейного программирования.

Ко всем рёбрам и вершинам графа потока управления добавляется коэффициент, обозначающий количество проходов по ребру или через вершину. Далее составляется система линейных уравнений. При этом каждой вершине сопоставляется одно уравнение, в котором коэффициенту вершины приравнивается сумма коэффициентов входящих в неё рёбер, и ещё одно уравнение, в котором коэффициенту вершины приравнивается сумма коэффициентов исходящих рёбер. Составляется целевая функция, равная сумме произведений коэффициентов на время выполнения каждого линейного участка. Находится целочисленное решение системы уравнений, максимизирующее значение целевой функции. Наихудшим временем выполнения задачи считается значение целевой функции, которое она принимает на вычисленном решении системы.

Пример работы данного метода изображен на рисунке 22. По заданному графу, в котором ребра помечены количеством проходов, а вершины помечены временем выполнения, строится задача оптимизации в виде набора ограничений на число проходов по вершинам. После решения задачи линейного программирования находится наихудшее время выполнения.

Преимуществом данного метода является его вычислительная сложность. Приведение задачи к задаче целочисленного линейного программирования происходит за

полиномиальное по числу линейных участков время. Существуют методы решения задачи целочисленного линейного программирования, более эффективные, чем полный перебор.

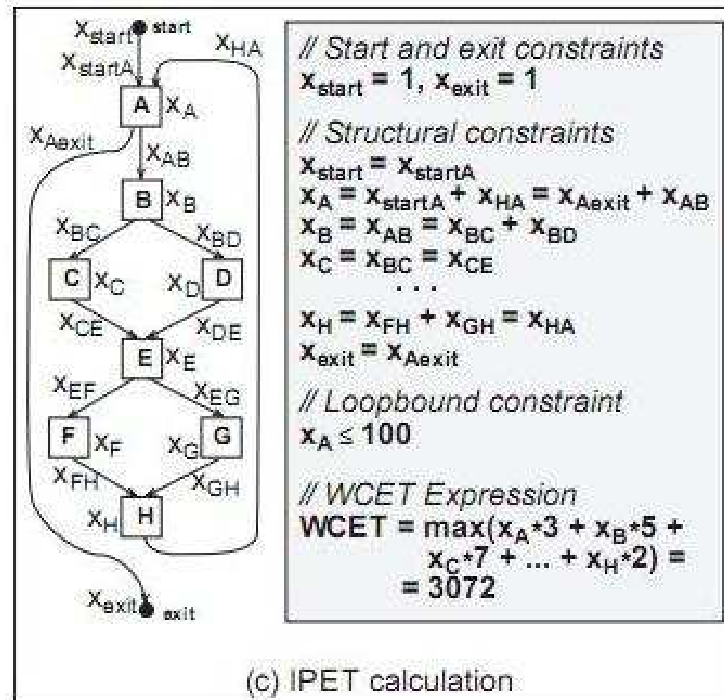

Рисунок 22 - Анализ наихудшего времени выполнения методом неявного перебора

Недостатком данного метода является то, что нет возможности учесть недетерминизм выполнения программ, который может привести к недостижимости линейных участков в зависимости от той или иной истории выполнения программы. В результате оценка может получиться завышенной.

Метод, основанный на верификации программ на моделях

Данный метод заключается в описании модели поведения программы и проверки выполнимости программы за определенный временной промежуток с помощью верификации. Модель программы может быть представлена, например, в виде диаграмм переходов, как, например, в [31], или в виде конъюнктивной нормальной формы, как описано в [32].

В данном методе производится построение модели программы с добавлением описания временных свойств её поведения. Далее формируется свойство проверки выполнимости программы за время, большее некоторой оценки. Производится итеративная проверка выполнения данного свойства (с помощью запуска верификатора) с уточнением

оценки до того момента, когда свойство перестает выполняться. Полученная оценка считается наихудшим временем выполнения программы.

Данная схема подробно описана в [33].

Преимуществом данного метода является то, что имеется возможность анализировать недетерминированное выполнение программы. Недостатком является вычислительная сложность, так как на каждой итерации производится запуск верификатора, который должен произвести поиск в полном пространстве состояний программы для проверки выполнимости поставленного свойства.

Измерительные методы

Суть измерительных методов описана в работах [27], [34] и состоит в том, что программа многократно выполняется на целевом оборудовании при различных значениях входов программы. На каждом запуске программы измеряются времена выполнения и затем запоминаются наихудшее и наилучшее из полученных значений. При этом точность получаемого результата существенно зависит от наборов входных данных, на которых осуществляется серия запусков. При неправильном построении наборов входных данных возможна ситуация, в которой ни один из запусков программы не привёл к выполнению самого длинного пути программы.

Гибридные методы

Гибридные методы описаны в [27], [34], [35] и соединяют особенности измерительных и статических методов.

Общая схема анализа включает в себя следующую последовательность шагов:

1. Построение графа потока управления.
2. Вычисление времени выполнения каждого линейного участка измерительным методом.
3. Вычисление оценки наихудшего времени выполнения по графу статическим методом.

1.8.3 Реализация метода оценки наихудшего времени выполнения

Для реализации анализатора оценки наихудшего времени выполнения выбран статический метод с использованием технологии верификации программ на моделях.

Обоснование выбранного метода

Данный метод был выбран в связи со следующими причинами:

- при оценке статическими методами не требуется запуск программы на целевом вычислителе, в отличие от других методов
- при использовании верификации программ на моделях имеется возможность анализировать программы с недетерминированным поведением, в то же время не требуется полностью перебирать все пространство путей выполнения (как, например, в методе полного перебора)

Описание реализации

Реализация основана на инструменте Metamoc, который в процессе анализа оценки WCET использует верификатор UPPAAL. Описание работы инструмента можно найти в [36].

Анализатор поддерживает программы, написанные на подмножестве языке программирования Си, и поддерживает следующие структуры языка:

- целочисленные переменные
- массивы целочисленных данных
- указатели
- арифметика над целочисленными данными и указателями
- условные операторы

Общая схема реализации

Общая схема работы анализатора изображена на рисунке 23 и заключается в следующей последовательности шагов:

1. Построение объектного файла по исходному коду программы.

На данном этапе используется кросс-компилятор GCC для целевой архитектуры для получения исполнимого кода программы. Текущей целевой архитектурой является процессора ARMT9.

2. Получение целевого кода вычислителя по объектному файлу.

Производится дизассемблирование объектного файла с помощью утилиты objdump для целевой архитектуры для получения ассемблерного кода программы.

3. Преобразование кода программы в модель для верификации.

На данном этапе используется специальный преобразователь Arm-to-uppaal целевого кода в модель для верификации с помощью инструмента UPPAAL.

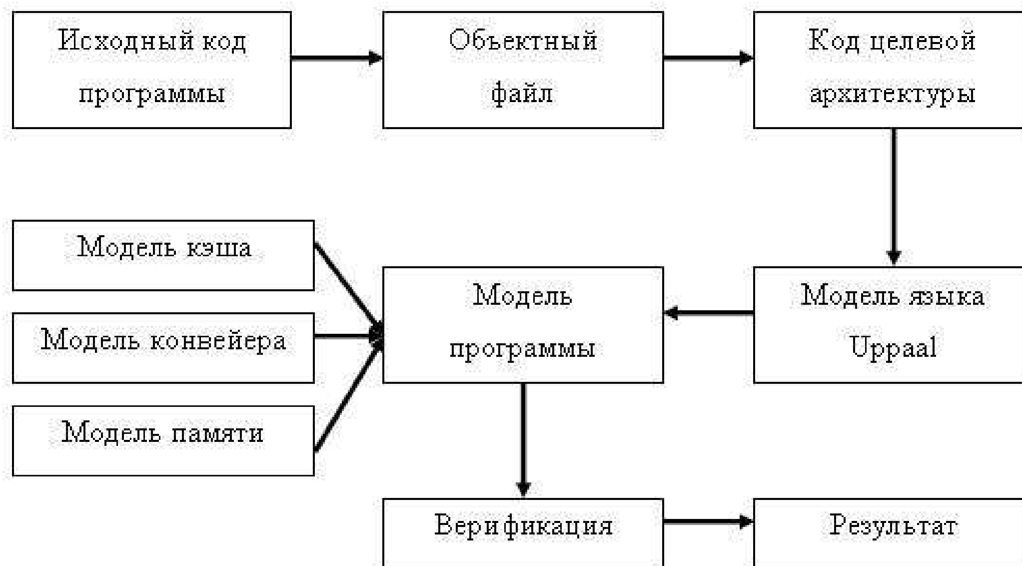

Рисунок 23 - Схема работы анализатора

4. Присоединение модели вычислителя к модели программы.

Описание модели компонентов целевого вычислителя, такие как модель кэша, конвейера и памяти, соединяются с построенной моделью программы. Обзор существующих моделей вычислителей приводится в следующем разделе.

5. Запуск верификации с поиском наихудшего времени выполнения программы.

На данном этапе используется специальная возможность верификатора UPPAAL – возможность поиска наибольшего значения переменной, при котором выполняется заданное свойство (выполняется с помощью операции SUP инструмента UPPAAL). Производится поиск наибольшего значения переменной, задающей время выполнения программы.

Найденное значение переменной является наихудшим временем выполнения программы.

1.9 Обзор моделей процессоров для оценки наихудшего времени выполнения

1.9.1 Введение

В первых работах, посвященных оценке наихудшего времени выполнения программ (таких как [37]), предполагалось, что время выполнения отдельных участков кода не зависит

от контекста выполнения. При таком предположении время выполнения двух участков кода можно вычислить, просуммировав времена выполнения этих участков.

В современных вычислителях, содержащих в себе такие компоненты, как кэш и конвейер, независимость от контекста уже не имеет места. Время выполнения отдельной инструкции может существенно варьироваться в зависимости от состояния процессора. В общем случае для определения времени выполнения текущей инструкции необходимо знать историю выполнения программы непосредственно до выполнения данной инструкции. При этом необходимо учитывать влияние таких компонент вычислителя, как память, кэш, конвейер и предсказатель переходов. Например, время считывания значения переменной, находящейся в кэше, на порядок меньше, чем время считывания переменной из оперативной памяти. Анализ попадания/промаха при поиске переменной в кэше может существенно улучшить точность оценки наихудшего времени выполнения. Влияние компонент на время выполнения описано в работе [27].

Анализ времени выполнения инструкций и линейных участков в процессе оценки наихудшего времени выполнения статическими методами проводится на этапе анализа поведения процессора. Время выполнения вычисляется исходя из модели вычислителя и истории выполнения программы. При этом модель вычислителя состоит из набора моделей компонент, влияющих на время выполнения. Точность полученных результатов времени выполнения инструкций и линейных участков зависят от точности модели вычислителя.

1.9.2 Описание существующих моделей вычислителей

Network Timed Automata(NTA)

Один из подходов заключается в представлении модели вычислителя в виде Network Timed Automata – сетей автоматов с описанием временных свойств. Данная модель используется для верификации с помощью инструмента UPPAAL. В работе [38] описан пример использования данного подхода при моделировании архитектуры вычислителя ARM9T. Каждая из компонент модели, таких как кэш, конвейер и память, моделируются в виде сети автоматов. В процессе анализа строится модель программы в виде NTA, и к ней подключаются модели компонент вычислителя. Полученная модель подается на вход верификатору UPPAAL и производится верификация с проверкой временных свойств программы с учетом влияния поведения вычислителя.

Данный подход используется для оценки наихудшего времени выполнения в инструменте Metamoc. В работе [36] описаны принципы работы данного инструмента. В работе также описаны модели конвейера и кэша в виде NTA для архитектуры ARM9T.

Конвейер представляет собой набор последовательно работающих функциональных устройств, каждый из которых выполняет некоторый этап выполнения инструкции программы. Каждый из функциональных устройств моделируется в виде диаграммы, которая состоит из нескольких состояний, таких как ожидание, начало выполнения соответствующего этапа, конец выполнения этапа, передача операции следующей части конвейера или выдача результата. В процессе моделирования программы каждая инструкция подается на вход конвейеру и ожидает завершения его работы с вычислением времени выполнения данной инструкции. На некоторых этапах работы конвейера (таких как чтение или запись значения переменной) происходит обращение к другим моделям компонент вычислителя, таким как кэш и память. Схема модели пятиступенчатого конвейера изображена на рисунке 24.

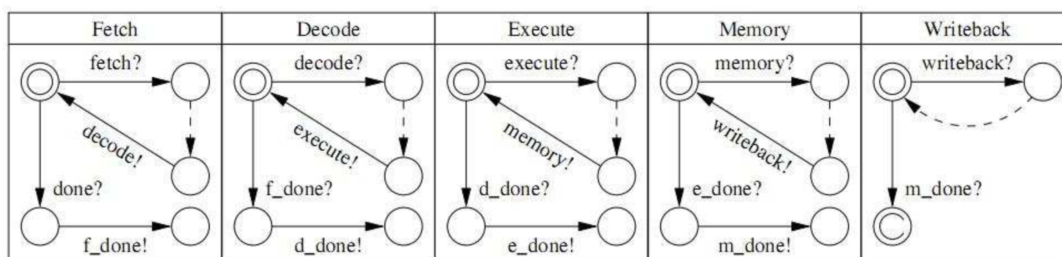

Рисунок 24 - NTA-модель пятиступенчатого конвейера

В инструменте Metamoc моделируются как кэш данных, так и кэш инструкций. Модель каждого кэша представляется в виде множества ячеек, в которых хранятся переменные. На этапе чтения/записи значения переменной конвейер обращается к кэшу для выявления наличия в нем переменной. При попадании моделируется чтение/запись значения и вычисляется время доступа (например, один такт работы процессора). При промахе производится обращение к модели памяти, и время доступа к переменной существенно увеличивается (например, до 30 тактов). Значение после промаха записывается в кэш.

В работе [36] показаны результаты проведения экспериментов, показывающие существенное повышение точности оценки наихудшего времени выполнения при использовании описанной модели вычислителя.

SimpleScalar processor model

В некоторых инструментах, таких как Chronos (данный инструмент описан в работе [39]), для оценки времени выполнения участков кода используется эмуляция выполнения (данная технология описана в [27]). Одной из основных систем для эмуляции, используемых в системах оценки наихудшего времени выполнения, является система SimpleScalar. Данная система подробно описана в работе [40].

SimpleScalar предоставляет возможность настраивать компоненты архитектуры целевого вычислителя, такие как центральный процессор, кэши, конвейеры, память, элементы ввода-вывода и др., или использовать настройки по-умолчанию. Существуют настройки для таких архитектур, как MIPS и ARM. Описания временных свойств компонентов можно производить с помощью специального графического интерфейса или с помощью описания на Си-подобном языке. Пример описания поведения кэша представлен ниже:

```
time_t cache_access(addr_t addr)
{
        word_t index = cache_hash(addr)
        if (tag[index] == addr)
        { /* hit */
                cache_update_lru(index);
                return 1;
        }
        else
        { /* miss */
                cache_handle_miss(addr);
        }  return 9;
}
```

В примере описана функция, которая возвращает количество тактов работы кэша при промахе и при попадании. Набор данных функций для описания временных свойств работы каждой компоненты вычислителя может быть использован в процессе эмуляции работы программы для поиска оценки наихудшего времени выполнения.

Для получения времен выполнения линейных участков с помощью модели вычислителя и эмулятора SimpleScalar в инструменте Chronos выполняется следующая последовательность действий:

- Компиляция программы с помощью кросс-компилятора GCC для целевой архитектуры.
- Построение кода программы целевого вычислителя с помощью дизассемблирования объектного файла программы.
- Разметка программы, выделение линейных участков кода.
- Получение времен выполнения линейных участков кода с использованием системы SimpleScalar и описанных моделей компонент вычислителя.

VHDL

Для многих вычислителей существуют спецификации на низкоуровневом языке описания аппаратуры VHDL [41]. Данный подход используется во многих системах оценки наихудшего времени выполнения, таких как ait и SWEET[42]. Описания включают в себя как компоненты вычислительной системы, такие как кэши, конвейеры и память, так и взаимодействия между компонентами. При этом описывается функциональность элементов системы с потактовой точностью, что может быть использовано для получения точных оценок времени выполнения.

Однако, как упомянуто в статье [43], полные модели архитектур являются слишком громоздкими, состоящими из тысяч строк описаний низкоуровневых деталей и эмуляция работы вычислителей является слишком долгим процессом. Для менее громоздких моделей вычислителей производятся специальные преобразования, строящие небольшие модели, сохраняющие при этом временные свойства работы вычислителей. Основные фазы преобразования изображены на рисунке 25 и состоят из следующих этапов:

- Удаление неиспользуемого кода

  Производится удаление кода модели, который не влияет на временные свойства.
- Уточнение конфигурации.

  Многие элементы модели являются конфигурируемыми, т.е. позволяют настраивать параметры целевой аппаратуры. Выявление и настройка конфигурации помогает существенно уменьшить модель.
- Построение абстракций для элементов модели.

Производится обобщение многих элементов системы с сохранением временных свойств. На данном этапе может использоваться документация аппаратуры.

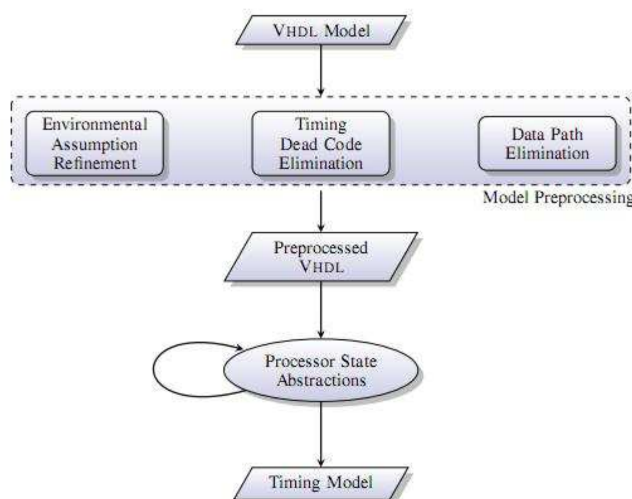

Рисунок 25 - Схема получения модели с сохранением свойств времен выполнения

Получаемые модели могут быть использованы для эмуляции и измерения времени выполнения линейных участков на этапе анализа поведения процессора. Модели данного типа отличаются высокой точностью и большим классом исследуемых архитектур, так как написание спецификаций является одним из этапов разработки вычислителей.

Таблицы времен выполнения и временных эффектов

Еще одним методом является описание времени выполнения каждой инструкции как суммы базового времени выполнения и времени временных эффектов. Данный метод описан в работе [44]. Базовое время выполнения – это время выполнения инструкции без влияния архитектурных особенностей вычислителя. Временные эффекты являются обычно отрицательной величиной (вычитаются из базового времени выполнения) и связаны с влиянием таких компонентов, как кэш, конвейер и предсказатель ветвлений. Для каждого вида команд целевого процессора известны значения как базового времени выполнения, так и временных эффектов под влиянием каждого из компонентов. Эти значения могут быть получены путем измерений, либо из документации вычислителей. При этом в процессе анализа используются простые модели кэша, конвейера и предсказателя ветвлений. Для каждой инструкции выявляется, будет ли учитываться тот или иной временной эффект. Для

получения времени выполнения линейного участка для всех входящих в него команд суммируются базовые времена выполнения и временные эффекты.

Данный метод используется, например, в инструменте Bound-T [45]. Характеристической особенностью является большая скорость обработки программ, так как не приходится производить эмуляцию выполнения. Недостатком является небольшая точность оценки по сравнению с эмуляцией.

Также стоит отметить метод, предложенный в статье [46]. В этом методе строится виртуальная модель процессора, которая управляет выполнением инструкций только при помощи разделяемых или блокируемых ресурсов. Состоянием модели процессора будет набор времен освобождения всех ее ресурсов. Статическая оценка для инструкции (или всей программы) будет абстрактной моделью инструкции (программы), изменяющей состояние модели процессора. В данном контексте, статический этап оценки можно рассматривать как "компиляцию" модели программы, а динамический — как ее абстрактную интерпретацию на модели процессора. К сожалению, данная методика применима для ограниченног класса процессоров, но для тех процессоров, где она применима (например, векторно-скалярный процессор NM6403 NeuroMatrix), была достигнута потактовая (абсолютная) точность оценки.

1.9.3 Описание выбранной модели процессора

Для анализа наихудшего времени выполнения выбран метод представления вычислителя в виде сетей автоматов с временными свойствами.

Основными причинами выбора данного метода являются следующие особенности:

- Существуют готовые описания компонент процессора ARM9T в виде сетей автоматов. Модели входят в состав инструмента Metamoc [36].
- Модели написаны на языке верификатора UPPAAL.

Описание модели

Модель состоит из следующих компонент:

- кэш
- конвейер
- основная память

При этом существует специальный преобразователь arm-to-uppaal, преобразующий код целевой архитектуры в модель языка UPPAAL, которая в совокупности с моделями архитектурных компонент описывает модель временного поведения программ.

Общая схема анализа времени выполнения линейных участков производится в следующей последовательности шагов:

- На вход подается объектный файл программы.
- Производится дизассемблирование объектного файла с помощью утилиты objdump для целевого процессора.
- Производится разметка кода, выделение линейных участков.
- Производится построение модели на языке UPPAAL с помощью преобразователя arm-to-uppaal
- Модель программы на языке UPPAAL объединяется с моделями компонент вычислителя.
- Полученная модель подается на вход верификатору UPPAAL, который вычисляет время выполнения линейных участков

1.10 Метод решения задачи выбора механизмов обеспечения отказоустойчивости для РВС РВ.

Все РВС РВ имеют четыре фундаментальные характеристики: функциональность, производительность, надежность и стоимость. В ходе разработки РВС РВ возникает задача построения системы, имеющей заданную функциональность (описываемую в спецификациях) и максимальную надежность при ограничениях на производительность и стоимость. Надежность РВС РВ может быть повышена посредством использования механизмов обеспечения отказоустойчивости (МОО), однако это снижает производительность и увеличивает стоимость системы.

В процессе построения конфигурации РВС РВ необходимо оценивать ее производительность, то есть время выполнения программ на заданных устройствах. Произвести такую оценку аналитически невозможно, поэтому необходимо обращаться к процедуре имитационного моделирования.

В данном разделе описан метод решения задачи выбора МОО РВС РВ. Описание схемы интеграции программного средства, решающего эту задачу, со средой имитационного моделирования приведено в разделе 2.2, описание апробации этой схемы – в разделе 3.6.

1.10.1 Неформальная постановка задачи

Структура РВС РВ задана в виде набора модулей и связей между ними. Каждый модуль состоит из аппаратного компонента, программного компонента и, возможно, МОО. Известны наборы вариантов устройств и программ, которые могут присутствовать в модуле в качестве соответственно аппаратного и программного компонентов. Для каждого варианта устройства или программы известны надежность и цена. Для каждой пары (устройство, программа) известно время выполнения программы на данном устройстве. Также известен объем выходных данных программного компонента каждого модуля. Для каждого модуля известны модули, от которых необходимо получать входные данные и которым нужно передавать выходные данные.

Необходимо для каждого модуля системы выбрать такой МОО и варианты аппаратных и программных компонентов, при которых надёжность максимальна при ограничениях на стоимость и время выполнения программ.

Под надежностью системы будем понимать вероятность ее безотказной работы в течение некоторого заданного промежутка времени.

Приведем описание рассматриваемых механизмов обеспечения отказоустойчивости (МОО) [47].

1) N-версионное программирование (NVP). Данный МОО включает модуль принятия решения (голосователь) и N независимо разработанных версий программы (N – нечетное). Все N версий работают параллельно, результат их работы обрабатывает голосователь. Тот результат, который выдает большая часть версий, принимается за финальный. В данном случае рассматривается два варианта NVP, в каждом из которых N=3. В NVP/0/1 все версии программы работают на одном аппаратном компоненте, в то время как в NVP/1/1 каждая версия программы работает на отдельном аппаратном компоненте. Таким образом NVP/0/1 допускает одну неисправность в программных версиях, а NVP/1/1 – одну неисправность либо в программных версиях, либо в аппаратных.

2) Восстановление блоками (RB/1/1). Данный МОО включает в себя модуль принятия решений (контрольный тест) и минимум две программных версии. Когда первая из версий завершает свою работу, то результат её работы тестируется контрольным тестом. Если результат теста оказался неудачным, то процесс откатывается на начало работы и запускается на выполнение следующая версия. Процесс продолжается пока результат одной из версий не будет принят, либо результат работы всех версий не будет отклонён. В данной

постановке задачи используются два аппаратных компонента, на каждом из которых существует две разные версии программы.

1.10.2 Формальная постановка задачи

Для формального описания структуры РВС РВ введём следующие обозначения:

- n – количество модулей РВС РВ;
- U_i – i-ый модуль системы;
- p_i , q_i – количество доступных версий соответственно аппаратного и программного компонентов в модуле U_i;
- H_{ij} – j-ая версия аппаратного компонента модуля U_i, $j \in [1, p_i]$;
- S_{ij} – j-ая версия программного компонента модуля U_i, $j \in [1, q_i]$;
- FT_i – множество доступных МОО для i-ого модуля;
- $FT_i \subseteq \{None, NVP/0/1, NVP/1/1, RB/1/1\}$, *«None»* обозначает отсутствие МОО;
- $F_i \in FT_i$ – МОО, используемый в модуле U_i;
- $H_i^{F_i}$ – мультимножество версий H_{ij}, выбранных для аппаратного компонента модуля U_i;
- $S_i^{F_i}$ – множество версий S_{ij}, выбранных для программного компонента модуля U_i;

Конфигурацию *System* РВС РВ можно представить в виде ориентированного графа без циклов, в котором каждой вершине соответствует модуль U_i, а множество ребер содержит ребро (U_i, U_j) тогда и только тогда, когда выходные данные программного компонента модуля U_i являются входными данными для программного компонента модуля U_j. В таком случае будем считать, что модуль U_i непосредственно зависит по данным от модуля U_j.

Конфигурация i-ого модуля однозначно определяется тройкой $\{H_i^{F_i}, S_i^{F_i}, F_i\}$.

- R_{ij}^{hw}, C_{ij}^{hw} – надежность и стоимость j-ой версии аппаратного компонента модуля U_i;
- R_{ij}^{sw}, C_{ij}^{sw} – надежность и стоимость j-ой версии программного компонента модуля U_i;

- x_{ij}, y_{ij} – количество экземпляров H_{ij} и S_{ij} в модуле U_i;

Стоимость системы можно вычислить по формуле:

$$C_{System} = \sum_{i=1}^{n}(\sum_{j=1}^{p_i} x_{ij} \cdot C_{ij}^{hw} + \sum_{j=1}^{q_i} y_{ij} \cdot C_{ij}^{sw}) = \sum_{i=1}^{n}\sum_{j=1}^{p_i} x_{ij} \cdot C_{ij}^{hw} + \sum_{i=1}^{n}\sum_{j=1}^{q_i} y_{ij} \cdot C_{ij}^{sw}.$$

- P_{rv} – вероятность отказа между двумя версиями программного компонента;
- P_{all} – вероятность одновременного отказа всех версий программного компонента;
- P_d – вероятность отказа схемы голосования;

Надежность i-ого модуля можно вычислить, используя R_{ij}^{hw}, R_{ij}^{sw}, P_{rv}, P_{all}, P_d. Формулы для этого приведены в [47]. Например, если в модуле U_i используется МОО RB/1/1, выбраны версии k_1, k_2 аппаратного компонента и версии l_1, l_2 программного компонента, то надежность этого модуля вычисляется по формуле:

$$\begin{aligned} R_i = 1 - (P_{rv} + (1-P_{rv}) \cdot P_d + (1-P_{rv}) \cdot (1-P_d) \cdot P_{all} + (1-P_{rv}) \cdot (1-P_d) \cdot (1-P_{all}) \cdot \\ \cdot (1-R_{ik_1}^{hw}) \cdot (1-R_{ik_2}^{hw}) + (1-P_{rv}) \cdot (1-P_d) \cdot (1-P_{all}) \cdot (1-(1-R_{ik_1}^{hw}) \cdot (1-R_{ik_2}^{hw})) \cdot \\ \cdot (1-R_{il_1}^{sw}) \cdot (1-R_{il_2}^{sw})) \end{aligned}.$$

Надежность всей системы равна:

$$R_{System} = \prod_{i=1}^{n} R_i.$$

- D_i – директивное время выполнения программного компонента S_i модуля U_i;
- T_i – время выполнения программного компонента S_i модуля U_i; оценивается с помощью имитационного моделирования; будем считать, что работа программного компонента модуля завершена, если его выходные данные полностью переданы всем зависящим от него модулям.

В данной постановке задачи для каждого модуля U_i время выполнения программного компонента для каждой пары версий (H_{ij_1}, S_{ij_2}), $j_1 \in [1, p_i]$, $j_2 \in [1, q_i]$ считается известным. Также известны времена работы голосователей, контрольных тестов и времена откатов к начальным состояниям в схеме RB/1/1. Кроме того, известен объем данных, передаваемых между программными компонентами, и скорость передачи данных.

- $C_{System}^{\max}$ – максимальная допустимая стоимость конфигурации РВС РВ;
- *Systems* – множество всевозможных конфигураций РВС РВ;

В рамках введённых обозначений задачу сбалансированного выбора МОО РВС РВ можно сформулировать следующим образом:

Дано:

1. n;
2. $p_i, q_i, \forall i \in [1,n]$;
3. R_{ij}^{hw}, C_{ij}^{hw}, $\forall i \in [1,n], \forall j \in [1, p_i]$;
4. R_{ij}^{sw}, C_{ij}^{sw}, $\forall i \in [1,n], \forall j \in [1, q_i]$;
5. FT_i, $\forall i \in [1,n]$;
6. P_{rv}, P_{all}, P_d;
7. D_i, $\forall i \in [1,n]$;
8. $C_{system}^{\max}$.

Требуется определить:

Конфигурацию $System_{best} \in Systems$, такую что:

$$R_{System_{best}} = \max_{Systems} R_{system}$$

при этом должны выполняться условия:

$$C_{System_{best}} \le C_{system}^{\max},$$

$$T_i \le D_i, \ \forall i \in [1,n]$$

то есть стоимость системы не должна превышать максимально допустимой, а время работы каждого модуля не должно превышать индивидуального директивного срока для этого модуля.

Данную задачу можно записать как задачу дискретной оптимизации:

$$\begin{cases} System_{best} \in Systems \\ R_{System_{best}} = \max\limits_{Systems} R_{system} \\ C_{System_{best}} \le C_{system}^{\max} \\ \mathrm{T_i} \le \mathrm{D_i}, \forall \mathrm{i} \in [1, \mathrm{n}] \end{cases}.$$

1.10.3 Алгоритм решения задачи

В [48] был предложен метод решения поставленной задачи, состоящий из следующих шагов: построение модели РВС РВ в общем виде; поиск оптимальной конфигурации РВС РВ с помощью адаптивного гибридного эволюционного алгоритма (АГЭА).

Модель РВС РВ строится на основе графа, представляющего внутреннюю структуру системы.

Далее подробно описана схема работы АГЭА.

Каждое возможное решение кодируется в виде строки, состоящей из блоков, которые соответствуют модулям РВС РВ. Каждый блок представляет собой тройку вида $<H, S, F>$. H - номер конфигурации аппаратной составляющей модуля, S – номер конфигурации программной составляющей, а F – номер МОО, используемого в данном модуле. По строке такого вида может быть вычислена целевая функция – надёжность системы, которая характеризует качество решения, и однозначно восстановлена соответствующая конфигурация РВС РВ.

АГЭА, предлагаемый в [48] для решения поставленной задачи, представляет собой эволюционный алгоритм с использованием блока нечеткой логики.

Описание схемы работы алгоритма:

1. Генерация случайным образом популяции решений.
2. Оценка популяции: вычисление целевой функции и проверка ограничений стоимости и времени; фиксация лучшего на текущий момент решения, вычисление среднего значения целевой функции для всей популяции. Проверка ограничений на время происходит на модели РВС РВ, которая модифицируется для учёта МОО. Модифицированная модель запускается для вычисления задержек времени выполнения программ РВС РВ с учётом МОО.
3. Отбор особей для скрещивания в отдельную промежуточную популяцию: популяция сортируется, и отбираются лучшие N_{sel} % особей по значению целевой функции.
4. Операция скрещивания (одноточечное скрещивание).
5. Формирование новой популяции: в новую популяцию берется некоторый процент лучших особей от текущей популяции, остальная часть популяции формируется из лучших особей промежуточной популяции, полученной после операции скрещивания.
6. Операция мутации (модификация одноточечной мутации).
7. Повторение п.2.

8. Проверка критерия останова: если он выполнен, то переход к п.10, если нет, то переход к п.9. Критерием останова в данном случае является выполнение алгоритмом априорно заданного числа итераций без улучшения целевой функции.

9. Блок нечеткой логики осуществляет автоматическую подстройку алгоритма и переход к п.3.

10. Завершение алгоритма, вывод наилучшей конфигурации.

Опишем подробно блок нечеткой логики. Он нужен для того, чтобы в автоматическом режиме корректировать настройки ЭА, управлять степенью влияния операций селекции, скрещивания и мутации на эволюционный процесс согласно некоторым правилам в зависимости от результатов работы алгоритма, которые получаются в каждом поколении.

Ключевыми параметрами для оценки популяции являются среднее значение целевой функции текущей популяции и лучшее значение целевой функции текущей популяции.

Введём следующие обозначения:

- R_1^{av} и R_0^{av} – средние значения целевой функции в текущей и предыдущей популяциях;
- $R_1^{\max}$ и $R_0^{\max}$ – лучшие значения целевой функции в текущей и предыдущей популяциях.

Изменяемые параметры АГЭА:

- N_{sel} – процент от популяции лучших особей, которые затем будут скрещиваться;
- P_{cross}– вероятность скрещивания;
- N_{nmut} – процент лучших особей текущей популяции, которые не мутируют;
- P_{mut} – вероятность мутации;

Параметры N_{nmut} и P_{mut} имеют три значения (большое, среднее и малое). Параметры N_{sel} и P_{cross} имеют два значения (большое и малое).

В зависимости от изменений параметров R^{av} и $R^{\max}$ в процессе работы АГЭА происходит переключение значений параметров алгоритма в соответствии с таблицей 1.

Таблица 1- Правила работы блока нечеткой логики

	$R_0^{av} < R_1^{av}$	$R_0^{av} \approx R_1^{av}$ (~3%)	$R_0^{av} > R_1^{av}$
$R_0^{\max} < R_1^{\max}$	Малый P_{mut} Большой N_{nmut} Большой N_{sel} Большой P_{cross}	Средний P_{mut} Средний N_{nmut} Большой N_{sel} Большой P_{cross}	Большой P_{mut} Малый N_{nmut} Большой N_{sel} Большой P_{cross}
$R_0^{\max} \approx R_1^{\max}$ (~3%)	Малый P_{mut} Большой N_{nmut} Малый N_{sel} Малый P_{cross}	Средний P_{mut} Средний N_{nmut} Малый N_{sel} Малый P_{cross}	Большой P_{mut} Малый N_{nmut} Малый N_{sel} Малый P_{cross}

Такой блок нечеткой логики имеет малую сложность по сравнению с вычислением целевых функций и проверкой ограничений для всех особей популяции, но при этом позволяет производить автоматическую перенастройку эволюционного алгоритма в процессе его работы.

Также авторами были предложены, реализованы и исследованы следующие модификации описанного алгоритма:

- случайная с ограничениями генерация начальной популяции решений;
- использование штрафных функций.

Экспериментальное исследование показало, что наилучшие решения получаются при использовании штрафных функций. Поэтому далее рассматривается именно эта модификация АГЭА.

Идея использования штрафных функций заключается в том, чтобы искусственно понизить значение целевой функции (надежности) для тех решений, которые не удовлетворяют заданным ограничениям. Таким образом, решения, не удовлетворяющие ограничениям, будут взяты в следующую популяцию с меньшей вероятностью.

В [48] штрафная функция представляет собой коэффициент, на которой умножается значение целевой функции. Этот коэффициент должен быть равен 1, если решение удовлетворяет ограничениям, и должен принадлежать (0;1] иначе. Кроме того, логично выбирать его таким образом, чтобы большему значению стоимости (времени) соответствовало меньшее значение штрафного коэффициента. В соответствии с такими

условиями был выбран следующий вид штрафных функций: функция равна 1, если решение удовлетворяет ограничениям, иначе обратно пропорциональна времени (стоимости):

- для времени:

$$E_{time}^{i} = \begin{cases} 1, T_i < D_i \\ \dfrac{D_i}{T_i}, иначе \end{cases} \qquad R_i^{*} = R_i \cdot E_{time}^{i} \qquad R_{system}^{*} = \prod_{i=1}^{n} R_i^{*}$$

- для стоимости:

$$E_{\cos t} = \begin{cases} 1, C_{system} < C_{system}^{\max} \\ \dfrac{C_{system}^{\max}}{C_{system}}, иначе \end{cases} \qquad R_{system}^{**} = R_{system}^{*} \cdot E_{\cos t}$$

В качестве значения целевой функции в эволюционных операторах берется измененное значение надежность R_{system}^{**} решения.

Когда в ходе работы алгоритма возникает необходимость оценить время выполнения программных компонентов, происходит обращение к процедуре имитационного моделирования. Общая схема интеграции средств синтеза архитектур со средой имитационного моделирования описана в разделе 2.2. В разделе 3.6 приведено описание апробации этой схемы для вышеописанного средства.

1.11 Разработка средства внесения неисправностей

1.11.1 Методы внесения неисправностей

Одним из методов тестирования РВС РВ является внесение неисправностей. Главной идеей метода внесения неисправностей является внесение неисправностей в компоненты системы с целью анализа, каким образом влияет та или иная неисправность на систему, приводит ли она к ошибке или нет, а также способна ли система обработать эту ошибку. Таким образом, оценивается способность системы обнаруживать и устранять те или иные ошибки[49]. Главной задачей метода внесения неисправностей для оценки надежности системы является внесение неисправностей в компоненты системы с целью анализа того, каким образом влияет та или иная неисправность на систему, приводит ли она к ошибке или нет, а также способности системы восстанавливаться после тех или иных ошибок.

Аппаратное внесение неисправностей

Аппаратное внесение неисправностей – внесение неисправностей в аппаратные компоненты системы.

На этапе реализации аппаратуры используются языки ее описания, такие как VHDL и Verilog. Одним из вариантов применения методов внесения неисправностей является модификация кода на этих языках[50]. Для этого используются два различных подхода – мутация и саботаж. Мутант – копия одного из оригинальных компонентов, чье поведение отличается от поведения оригинального компонента, саботер - дополнительный компонент, который располагается между двумя оригинальными компонентами и изменяет значения и время прохождения сигнала между ними.

На этапе производства аппаратуры возможны два варианта применения методов внесения неисправностей – контактное[51], когда неисправность физически вносится непосредственно в прототип аппаратуры, и бесконтактное, когда этот прототип подвергается действия извне, изменяющий его свойства на определенный промежуток времени или навсегда. Примером бесконтактного внесения неисправностей может быть ионно-радиационный метод, когда аппаратура облучается при помощи специальных инструментов.

Программное внесение неисправностей

Программное внесение неисправностей – это внесение неисправностей в программные компоненты системы. Существует два различных подхода к программному внесению неисправностей:

- Внесение неисправностей до компиляции;
- Внесение неисправностей при выполнении.

В первом подходе неисправность вносится в исходный код программных компонентов системы. С помощью этого подхода можно эмулировать аппаратные неисправности[52], в том числе неисправности при передаче данных. Этот подход не нуждается в каком-либо дополнительном программном обеспечении, не наносит никаких повреждений системе и очень прост. Однако у него есть ряд недостатков, среди которых основными являются невозможность внесения неисправностей во время работы системы и необходимость временных затрат для обеспечения возможности внесения неисправностей новых классов.

1.11.2 Выбор метода внесения неисправностей

Рассмотрим возможность адаптации метода внесения неисправностей до компиляции и при выполнении:

До компиляции

При применении метода внесения неисправностей до компиляции для оценки надёжности необходимо будет вводить изменения в модель на этапе описания модели. Эти изменения зависят от наборов неисправностей, на которых необходимо оценивать надежность распределенных вычислительных систем реального времени, соответствующих моделям. Данный метод достаточно трудоёмок с точки зрения написания кода и последующих компиляций и прогонов модели. Однако, при написании удобного средства автоматического внесения неисправностей в код модели, эти проблемы не столь значительны.

При выполнении

При применении метода внесения неисправностей при выполнении для оценки надёжности необходимо будет вносить неисправности на этапе прогона модели. Внесение неисправностей на этапе прогона возможно двумя способами.

Первый способ предполагает внесение неисправностей на уровне среды выполнения. Данный метод предполагает внесение неисправность в сообщения, передаваемые между RTIG и RTIA (Рисунок 26). Такой подход хорош тем, что исходный код исполняемой модели остается неизменным. Однако при данном подходе необходимо существенно менять среду выполнения. Главным недостатком данного подхода является необходимость изменения системы прогона моделей в случае добавления новых классов неисправностей.

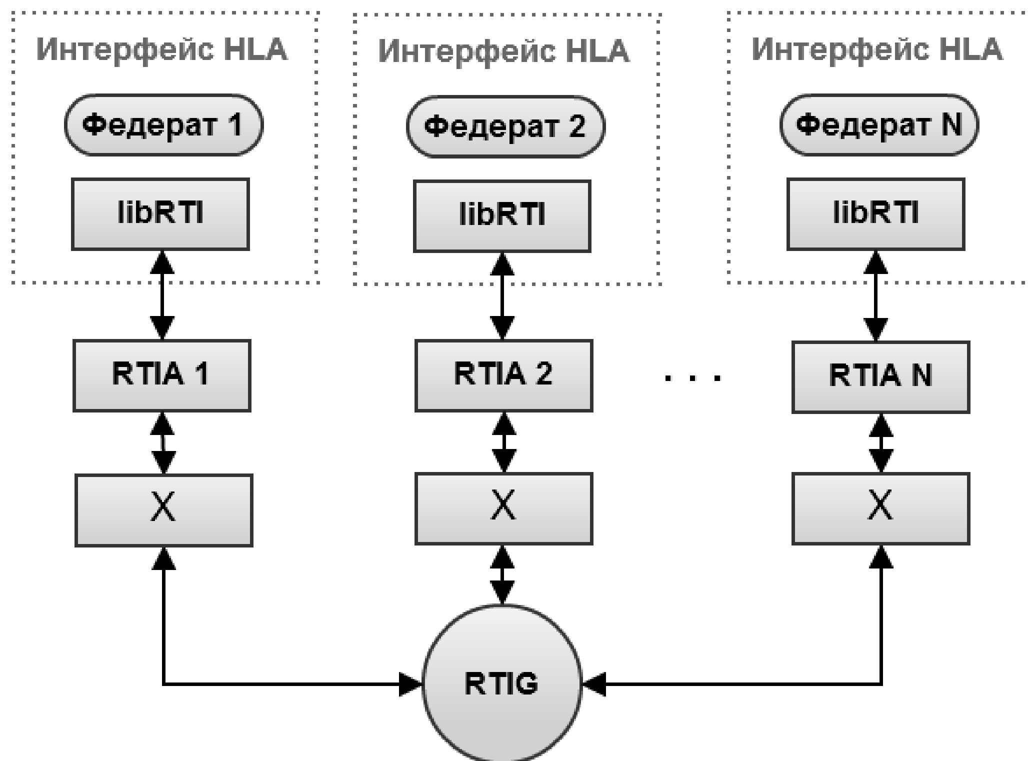

Рисунок 26 - Внесение неисправностей на уровне среды выполнения

Второй подход предполагает добавление нескольких федератов, выступающих в роли перехватчиков сообщений. Добавление нескольких перехватчиков служит для того, чтобы распределять нагрузку между перехватчиками. Главным преимуществом этого метода является отсутствие необходимости изменения среды выполнения. Также придется вносить незначительные изменения в исходных участниках моделирования. Федераты-перехватчики выступают в роли посредников, через которых проходят все сообщения, которые могут либо пересылаться получателю, либо предварительно редактироваться (Рисунок 27).

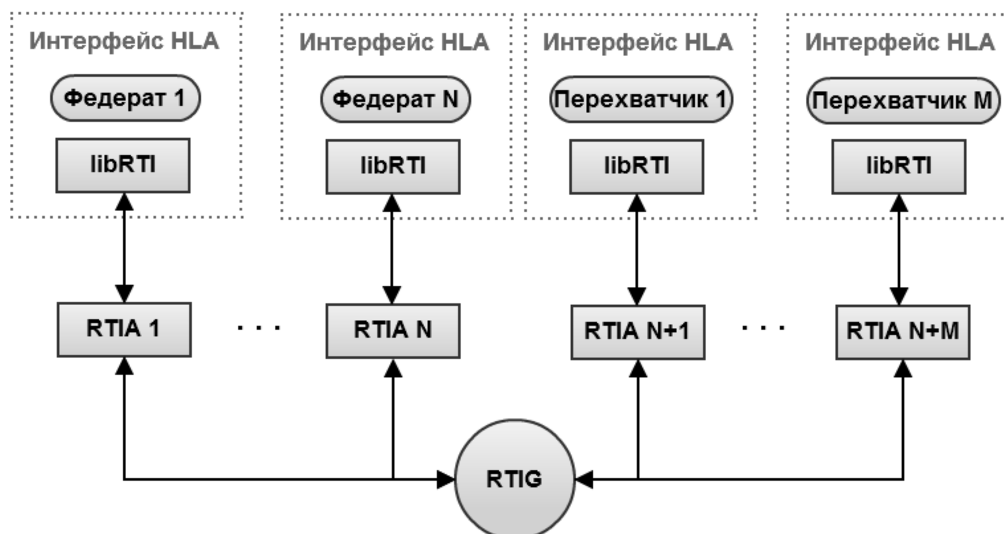

Рисунок 27 - Внесение неисправностей на уровне федератов

Используя данный метод, необходимо вносить незначительные изменения в каждый федерат и создавать группу федерат-перехватчиков.

Сравним возможные реализации средств автоматического внесения неисправностей при данных подходах. При выборе метода внесения неисправностей необходимо учитывать не только итоговые возможности средства автоматического внесения неисправностей, но и сложность технической реализации.

При внесении неисправностей до компиляции отсутствует возможность контролировать внесение неисправностей во время прогона модели, что является существенным недостатком данного подхода. Добавление новых неисправностей также является довольно сложным процессом.

Методы внесения неисправностей при выполнении лишены таких недостатков. Оба подхода к внесению неисправностей во время прогона имитационных моделей позволяют контролировать процесс внесения неисправностей. С точки зрения сложности технической реализации эти методы равнозначны. Добавление федерата-перехватчика меняет логику выполнения модели, однако не меняет саму среду выполнения, что является большим преимуществом. Именно поэтому метод внесения неисправностей на уровне федератов был выбран для дальнейшего исследования и последующей реализации.

1.11.3 Схема работы метода

Главной задачей данного метода является создание федерата-перехватчика, играющего роль посредника в передачи сообщений. Для начала необходимо создать шаблон федерата, который в зависимости от тестируемой модели будет заполняться необходимой информацией о других федератах.

Было принято решение о реализации упрощенной схемы ВН, т.е. при М = 1(1 федерат-перехватчик). В этом случае сообщения от всех участников моделирования поступают сначала к федерату-перехватчику, а затем отправляются по назначению.

Чтобы федерат-перехватчик имел возможность редактировать все сообщения, необходимо чтобы все сообщения изначально направлялись ему. Рассмотрим пример: пусть Федерат 1 хочет передать сообщение типа Т1. Федерат 2 может принимать сообщения типа Т1, значит исходное сообщение предназначается для Федерата 2 (Рисунок 28).

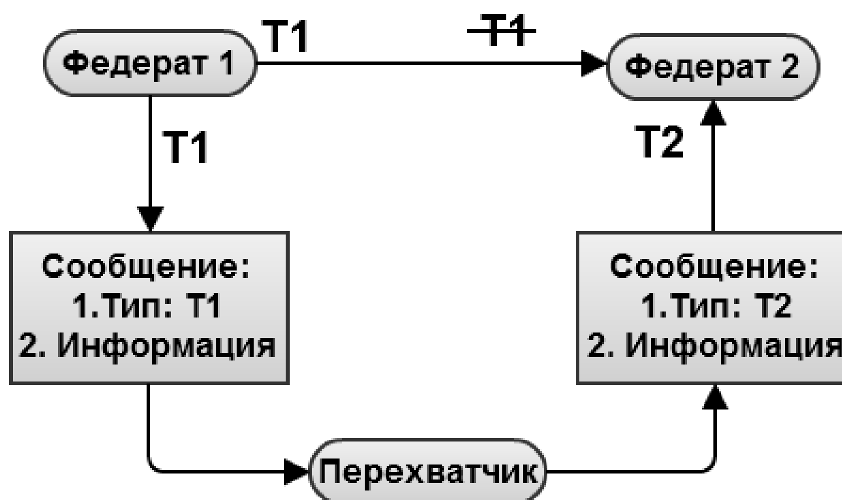

Рисунок 28 - Пересылка сообщений

Для того, чтобы Перехватчик мог иметь возможность принимать сообщение Т1, он подписывается на этот тип сообщений. Чтобы исходное сообщение приходило к Федерату 2 только после обработки его в Перехватчике, необходимо поменять в Федерате 2 тип принимаемых сообщений на Т2. Получается что исходное сообщение имеет тип Т1 и предназначается для Перехватчика. Далее перехватчик обрабатывает сообщения(вносит необходимые изменения), меняет его тип на Т2 и отправляет Федерату 2.

Этот алгоритм применим к моделям любого объема, однако, после тестирования, было принято решение о его модификации. Предположим, что идет тестирование взаимодействия двух федератов, состоящих в федерации с большим числом федератов.

Выполнив предложенный алгоритм, федерат-перехватчик подпишется на сообщения всех типов, а значит, любое сообщение сначала придет к перехватчику и только затем к получателю. Таким образом, число передаваемых сообщений удвоится, что сильно повлияет на скорость работы CERTI, использующей централизованную архитектуру. Поэтому было принято решение о том, что пользователь заранее указывает типы сообщений, на которые должен подписаться федерат.

Таким образом, для реализации схемы с федератом-перехватчиком необходимо не только сгенерировать сам Перехватчик, но и произвести замену типов сообщений таким образом, чтобы все сообщения сначала шли Перехватчику и затем шли получателям.

1.11.4 Архитектура средства

Как было сказано выше, все сообщения, передаваемые в исполняемой модели, сначала приходят федерату-перехватчику. Для удобства пользователей был сделан графический интерфейс перехватчика с использованием кроссплатформенной библиотеки wxWidgets[52]. Процесс внесения неисправностей может проходить как в автоматическом режиме, так и с возможностью перехода в ручной режим.

В ручном режиме внесения неисправностей пользователю показывается каждое пришедшее сообщение и предлагается внести свои корректировки. В этом режиме предполагается что пользователь будет обрабатывать каждое сообщение, что не всегда удобно и нужно. В этом случае невозможно тестирование моделей в режиме реального времени. Для таких случаев был создан автоматический режим внесения неисправностей.

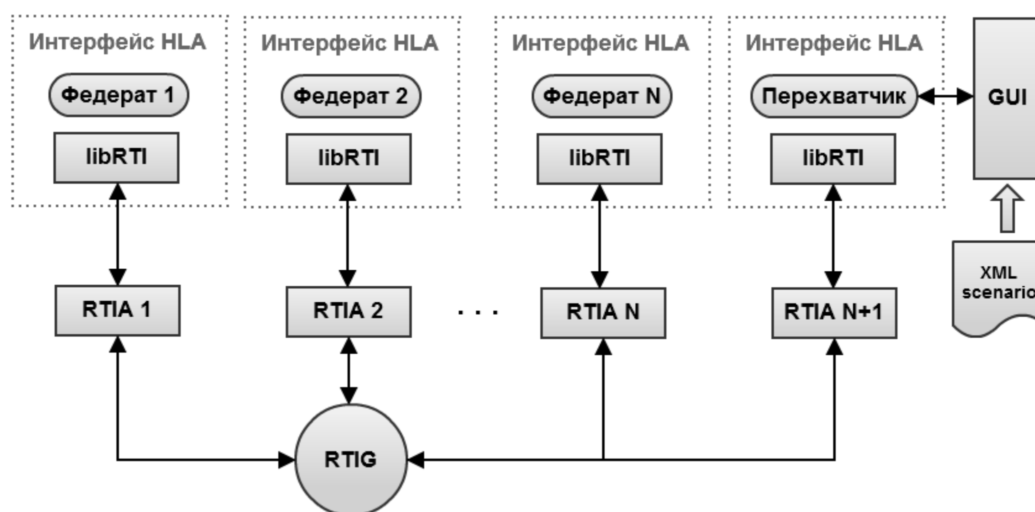

Рисунок 29 - Архитектура средства

В автоматическом режиме внесения неисправностей пользователю предлагается загрузить XML файл со сценарием неисправностей. XML-сценарий содержит набор типов сообщений и описания действий для их модификации. Средство ВН позволяет менять режим внесения неисправностей во время выполнения (Рисунок 29).

После обработки сценария средство готово к использованию и запускается тестируемая модель. Вся информация от федерата-перехватчика к пользовательскому интерфейсу и обратно передается с помощью сокетов.

После запуска тестируемой модели, средство получает, обрабатывает согласно сценарию и передает измененные данные федерату-перехватчику (Рисунок 12).

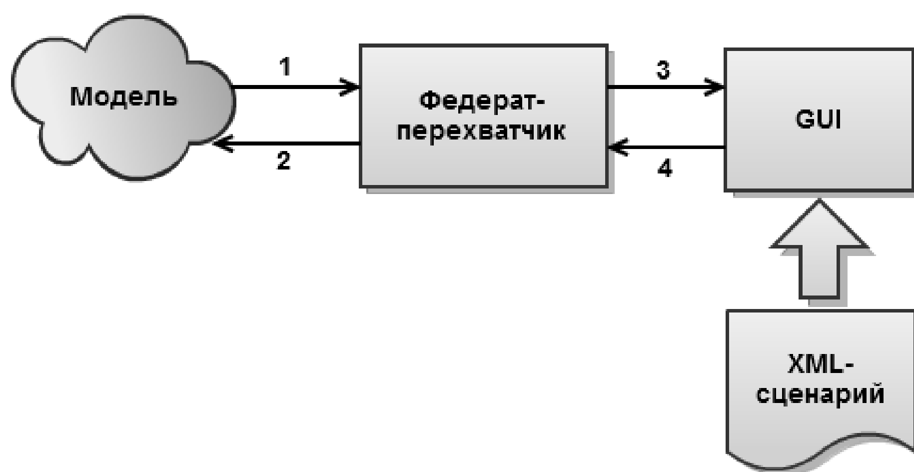

Рисунок 30 – Схема работы средства

1.11.5 Выводы

В данном разделе описан метод внесения неисправностей и предлагается схема средства, реализующего данный метод. Такое средство позволит вносить неисправности в компоненты системы с целью анализа, каким образом влияет та или иная неисправность на систему.

2 Интеграция разработанных методов и инструментальных средств

Разработанные в данной работе средства необходимо было интегрировать в единую среду. В данном разделе описываются результаты этой интеграции. В разделе 2.1 приведено описание интеграции с WCET. Раздел 2.2 содержит описание интеграции с средстваи синтеза архитектур и планирования расписаний. В 2.3 приведено описание интеграции средств, разработанных на предыдущих этапах работы.

2.1 Интеграция среды моделирования со средствами оценки WCET

2.1.1 Встраивание средств оценки WCET в транслятор UML в UPPAAL

Синтаксис UML допускает использование комментариев для пометки некоторых состояний диаграмм. Эти комментарии могут содержать важную информацию о поведении проектируемой системы, но обработка комментариев должна осуществляться процедурами, выходящими за пределы обычных средств моделирования.

В частности, простые состояния, в которых выполняются содержательные операции, могут быть помечены комментариями, содержащими код на языке C++. Знание этого кода позволяет, например, оценить время его выполнения и, таким образом, указать ограничение на время пребывания системы в том или ином состоянии.

Для оценки времени выполнения программного кода может быть использовано программно-инструментальное средство, использующее один из алгоритмов WCET, которое позволяет по заданному программному коду и модели аппаратуры, выполняющей этот код, получить верхнюю оценку времени выполнения кода. Оценка времени выполнения программного кода, которым помечены простые состояния UML диаграмм, проводится на этапе преобразования UML в HTA. Для каждого состояния вводится дополнительный таймер t. Показания этого таймера сбрасываются на каждом переходе, ведущем в любое простое состояние UML диаграммы. При этом верхняя оценка времени выполнения кода преобразуются в ограничения вида $t \le c$, которые добавляются к инвариантам, приписанным простым состояниям. После этого комментарий состояния удаляется, а вместо него добавляются таймер, инвариант и таймаут, как показано на рисунке 31. Здесь E – это оценка времени выполнения, полученная из средства WCET.

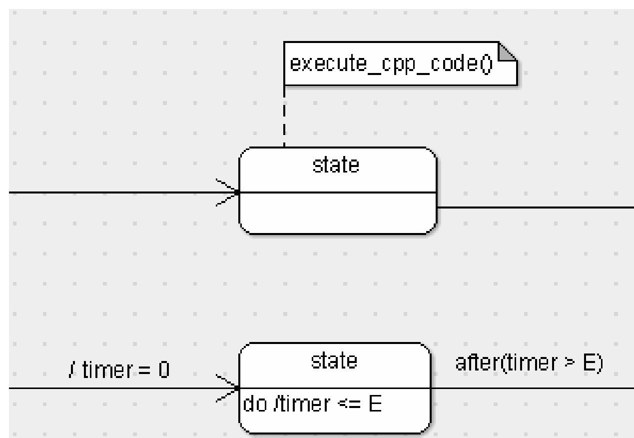

Рисунок 31 – Добавление таймера, инварианта и таймаута в модель по результатам оценки наихудшего времени выполнения программы

2.1.2 Запуск анализатора WCET

Для запуска анализатора оценки максимального времени выполнения линейного участка используется специальный скрипт count_wcet_basic_block, которому на вход подается файл с описанием кода линейного участка на языке C++ и который возвращает наихудшее время выполнения линейного участка в тактах работы процессора.

При вызове анализатора из транслятора UPPAAL по умолчанию происходит оценка времени выполнения с использованием модели ARM9TDMI. Можно настраивать параметры модели вычислителя, указывая следующие параметры:

- модель конвейера; в настоящее время доступны модели для архитектур arm7, avr, arm9
- модель кэша; в этой модели можно настраивать такие параметры, как размер блока кэша, максимальное количества блоков, поведение кэша при промахе и попадании
- модель памяти; эта модель позволяет настраивать такие параметры, как время доступа к памяти, размер блока памяти, количество блоков памяти.

Перечисленные параметры указываются с помощью задания соответствующих ключей при вызове анализатора оценки максимального времени выполнения.

2.2 *Интеграция средств моделирования со средствами построения расписаний и синтеза архитектур*

Средства построения расписаний и синтеза архитектур решают задачи построения архитектур и расписаний, оптимальных по некоторому критерию и удовлетворяющих определенным ограничениям. Зачастую в ходе работы средств построения и синтеза требуется проверить, удовлетворяет ли определенная архитектура или расписание некоторым требованиям, однако сделать это аналитически невозможно. В таких случаях обращаются к процедуре проведения имитационных экспериментов. Типичным примером ограничения, проверка выполнимости которого часто проводится с помощью имитационного моделирования, является ограничение на время выполнения программ. Примером задачи синтеза архитектуры является задача выбора механизмов обеспечения отказоустойчивости для РВС РВ, описанная в разделе 1.10.

Таким образом, возникает задача интеграции средств построения и синтеза архитектур со средой имитационного моделирования. Так как большинство алгоритмов построения расписаний и синтеза архитектур требуют многократных проведений имитационных экспериментов, то взаимодействие средств построения со средой моделирования должно быть полностью автоматизированным. Кроме того, разрабатываемая схема интеграции не должна требовать от авторов средств построения и синтеза знания принципов работы и внутренней структуры среды моделирования.

2.2.1 Формальное определение расписания

Формальное определение расписания было построено на основе определений, рассматриваемых в работах [54] и [55].

Аппаратная часть РВС РВ, на которой выполняется расписание, представляет собой множество процессоров, соединенных средой передачи данных. Время передачи единицы данных от одного процессора к другому одинаково для всех процессоров и считается заданным.

Программа, для которой строится расписание, состоит из конечного множества заданий (подпрограмм). Для каждого из заданий известен объем результата, получаемого на выходе. Некоторые из заданий получают на вход выходные данные других заданий. Таким образом, программу можно представить в виде графа потока данных, представляющего собой ориентированный граф без циклов. Каждой вершине графа соответствует задание программы; из i-ой вершины идет ребро в j-ую вершину тогда и только тогда, когда

выходные данные i-ого задания являются входными данными для j-ого задания. i-ое задание зависит по данным от j-ого задания тогда и только тогда, когда в графе программы существует путь из j-ой вершины в i-ую, непосредственно зависит – из j-ой вершины в i-ую идет ребро.

Будем считать, что для программы задано расписание, если у каждого из заданий есть привязка – однозначно определено, на каком процессоре оно выполняется, и задан порядок – для каждого процессора известно, в какой очередности выполняются задания.

Определим понятие расписания формально. Пусть:

V – множество всех заданий программы;

M – множество процессоров в ВС.

Тогда под расписанием будем понимать множество S троек (v,m,n), $v \in V$, $m \in M$, $n \in \mathbb{N}$, таких, что:

$$\forall v \in V \quad \exists!\, s_j = (v_j, m_j, n_j) \in S : v = v_j,$$

$$\forall s_i = (v_i, m_i, n_i) \in S, \forall s_j = (v_j, m_j, n_j) \in S : (s_i \neq s_j, m_i = m_j) \Rightarrow n_i \neq n_j.$$

Элемент расписания $s_i = (v_i, m_i, n_i) \in S$ непосредственно зависит от элемента $s_j = (v_j, m_j, n_j) \in S$, если либо v_i непосредственно зависит по данным от v_j, либо $m_i = m_j$ и $n_i = n_j + 1$. Расписание может быть представлено в виде ориентированного графа такого, что вершинам графа соответствуют элементы расписания, и из i-ой вершины идет дуга j-ую вершину тогда и только тогда, когда элемент s_j непосредственно зависит от элемента s_i. Элемент s_i зависит от элемента s_j, если из j-ой вершины существует путь в i-ую.

Будем говорить, что расписание S является корректным, если оно удовлетворяет свойству ацикличности, то есть не существует набора элементов расписания $s_1, ..., s_n$, такого что s_i зависит от s_{i-1} для всех $i \in [2, n]$ и s_n зависит от s_1 [55].

Для каждой пары (процессор, задание) известно время выполнения данного задания на данном процессоре без учета времени получения и отправки данных. Для каждого задания задан директивный срок выполнения. Задание считается выполненным, если его выходные данные переданы всем непосредственно зависящим от него заданиям. Каждое задание должно быть выполнено не позже своего директивного срока.

Для каждого элемента расписания время начала работы равно $\max_{s_i}\{0, T_i\}$, где s_i – элементы расписания, от которых данный элемент непосредственно зависит, а T_i – их времена завершения.

2.2.2 Формат представления расписаний

Для того чтобы не обязывать авторов средств построения и синтеза самостоятельно реализовывать построение модели, выполнимой в среде CERTI, необходимо ввести единый формат представления расписаний и разработать отдельное программное средство, строящее модель системы по представленному в этом формате расписанию. Данный формат не должен зависеть от особенностей реализации средств построения и синтеза архитектур и среды выполнения моделей.

В качестве формата представления расписаний был выбран формат, основанный на XML. При описании расписания используются следующие теги:

<system> – корневой элемент в описании системы. Имеет атрибут rt, принимающий значение 1, если рассматриваемая система является системой жесткого реального времени, и 0 – иначе. Содержит внутри себя теги <processor>.

<processor> – описание процессора. Имеет атрибут id – уникальное имя процессора. Содержит внутри себя теги <task>.

<task> – описание задания. Имеет атрибуты num – порядковый номер задания в расписании; id – уникальное имя задания; time – время выполнения задания на процессоре, к которому привязано данное задание; dirtime – директивный срок выполнения задания; datavol – объем выходных данных; link – ссылка на исходный код задания на C++ (в текущей реализации не используется). Внутри тега <task> могут содержаться теги <prev> и <next>.

<prev> – задание (атрибут id – имя), от которого текущее задание непосредственно зависит по данным.

<next> – задание (атрибут id – имя), которое зависит по данным от текущего задания.

Средства построения должны по внутреннему представлению расписания генерировать файл описанного формата. Расписание, формально определенное в предыдущем разделе, можно представить в указанном формате.

Ниже приведен пример файла, описывающего расписание:

```
<system rt="0">
    <processor id="Processor_1">
        <task id="task_1" num="1" time="5" dirtime="15" datavol="1" link="nolink">
            <next id="task_4"></next>
        </task>
```

```
            <task id="task_4" num="2" time="10" dirtime="50" datavol="2" link="nolink">
                <prev id="task_1"></prev>
                <prev id="task_2"></prev>
                <prev id="task_3"></prev>
            </task>
        </processor>
        <processor id="Processor_2">
            <task id="task_2" num="1" time="6" dirtime="25" datavol="3" link="nolink">
                <next id="task_4"></next>
            </task>
            <task id="task_3" num="2" time="8" dirtime="35" datavol="4" link="nolink">
                <next id="task_4"></next>
            </task>
        </processor>
    </system>
```

2.2.3 Модель системы ВС

По описанному в формате XML расписанию строится модель РВС РВ, представляющая собой диаграмму состояний в формате SCXML [56]. Далее по ней будет сгенерирован код федератов на C++. Данная модель выполнение расписания задач на процессорах с учётом задержек на выполнение задач и обмен данными.

Опишем общую логику построения модели в виде диаграммы состояний.

Всей вычислительной системе соответствует AND-состояние *system*, включающее в себя составные состояния, соответствующие процессорам.

Каждое состояние, соответствующее процессору, представляет собой последовательный автомат, моделирующий выполнение заданий, привязанных к заданному процессору. i-ому заданию соответствует последовательность состояний, начинающаяся состоянием *task_<i>_entry* и завершающаяся состоянием *task_<i>_exit*. $\forall i \in [1, n-1]$ существует переход из состояния *task_<i>_exit* в состояние *task_<i+1>_entry*. *task_1_entry* – начальное состояние автомата.

Рассмотрим последовательность состояний, соответствующих i-ому заданию (префикс *task_<i>_* будет опущен).

entry – начальное состояние.

Переход в **waiting.**

waiting – ожидание входных данных.

Переход в **working**:

- условие: для всех j *task_j_ready* == true, j – номера всех заданий других процессоров, от которых необходимо получать данные;
- действия: *current_time* = max{ *task_j_task_i_sending_end* } (значение счетчика времени становится равным времени получения последней порции входных данных).

working – выполнение задания.

Переход в **time_exceeded:**

- условие: (*current_time + time > dir_time) && (hard_rt)* (превышение директивного времени для систем жесткого реального времени);
- действия: *task_i_ time_exceeded = true;*

Переход в **sending:**

- условие: (*current_time + time <= dir_time) || (!hard_rt)*
- действия: *task_i_time_exceeded=current_time+time>dir_time; current_time=time+current_time; task_i_task_j_sending_ready=false;* (*task_j* – непосредственно зависящие от *task_i* задания на других процессорах)

time_exceeded – нарушен директивный срок для системы жесткого реального времени.

Конечное состояние, переходов нет.

sending – состояние перед передачей данных.

Переход в **task_i_task_j_sending:**

- условие: *task_i_task_j_sending_ready*==false (*task_j* – непосредственно зависящие от *task_i* задания на других процессорах)

Переход в **end**:

- условие: для всех i *task_i_task_j_sending_ready*==true;
- действия: *task_i_ready*=true;

task_i_task_j_sending – передача данных от i-ой задачи к j-ой (передачи данных между заданиями одного процессора не моделируются).

Переход в **sending:**

- действие: task*_i_task_j_sending_ready*=true;
  current_time=data_vol+current_time; (время передачи пропорционально объему данных)

end – завершение работы задания.

Переход в **time_exceeded:**

- условие: (*current_time > dir_time) && (hard_rt)* (превышение директивного времени для систем жесткого реального времени);

Переход в **exit :**

- условие: (*current_time <= dir_time) || (!hard_rt)*

exit – выход.

Переход в **task_<i+1>_entry**, либо переходов нет.

Взаимодействие между автоматами, соответствующими процессорам, осуществляется с помощью глобальных переменных. В терминах среды моделирования CERTI при изменении значения глобальной переменой изменивший ее федерат должен выполнить вызов RTI send interaction с параметром-значением переменной, а все федераты, использующие эту переменную, – receive interaction. В SCXML-модели такое взаимодействие отображается как переход между состояниями параллельных регионов, соответствующих процессорам. Поле «event» такого перехода содержит имя глобальной переменной. При генерации кода федерата переходы такого вида рассматриваются как указания на то, что при нахождении автомата, задающего логику работы процессора в некотором состоянии, необходимо выполнить вызов RTI send (receive) interaction.

На рисунке 32 приведен фрагмент файла в формате SCXML, соответствующий одному заданию, а на рисунке 33 - его визуализация в редакторе SCXML-gui:

```
<state id="task_2_entry">
  <transition cond="1" target="task_2_waiting" />
  </state>
<state id="task_2_waiting">
  <transition cond="1" target="task_2_working" />
  </state>
<state id="task_2_task_4_sending">
  <transition cond="1" event="c2=task_2_trans+task_2_task_4_sending_start;
  IRT_task_2_task_4_sending_end=c2;task_2_task_4_sending_ready=true;"
target="task_2_sending" />
  </state>
<state id="task_2_sending">
  <transition cond="!task_2_task_4_sending_ready" event="task_2_task_4_sending_start=c2;"
  target="task_2_task_4_sending" />
  <transition cond="task_2_task_4_sending_ready" event="IRT_task_2_ready=true;
  task_2_time_ex=task_2_dirTime≤c2;" target="task_2_end" />
  </state>
<state id="task_2_working">
  <transition cond="(c2>task_2_dirTime)&&hard_rt" target="task_2_time_exceeded" />
  <transition cond="!hard_rt||task_2_dirTime>=task_2_time+c2"
event="c2=task_2_time+c2;task_2_time_ex=(c2>task_2_time);task_2_task_4_sending_ready=false;"
target="task_2_sending" />
  </state>
<state id="task_2_end">
  <transition cond="task_2_time_ex&&hard_rt" target="task_2_time_exceeded" />
  <transition cond="!task_2_time_ex||!hard_rt" target="task_2_exit" />
<transition cond="1" event="IRT_task_2_task_4_sending_end" target="task_4_waiting">
  <parametr name="IRT_task_2_task_4_sending_end" />
  </transition>
<transition cond="1" event="IRT_task_2_ready" target="task_4_waiting">
  <parametr name="IRT_task_2_ready" />
  </transition>
  </state>
  <state id="task_2_time_exceeded" />
<state id="task_2_exit">
  <transition cond="1" target="task_3_entry" />
  </state>
```

Рисунок 32 - Фрагмент SCXML-файла, описывающий выполнение одного задания

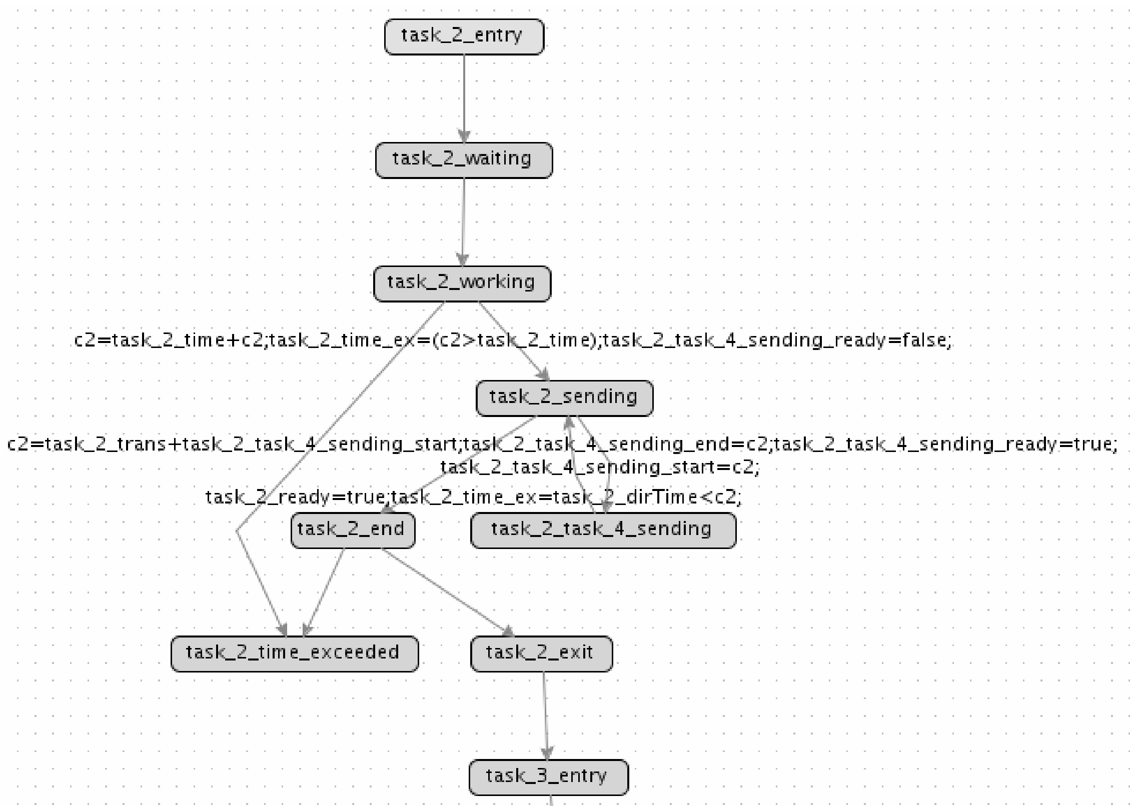

Рисунок 33 - Визуализация фрагмента SCXML-файла, описывающего выполнение одного задания

2.2.4 Программная реализация

Общая схема интеграции средства построения (синтеза) и среды выполнения моделей представлена на рисунке 34.

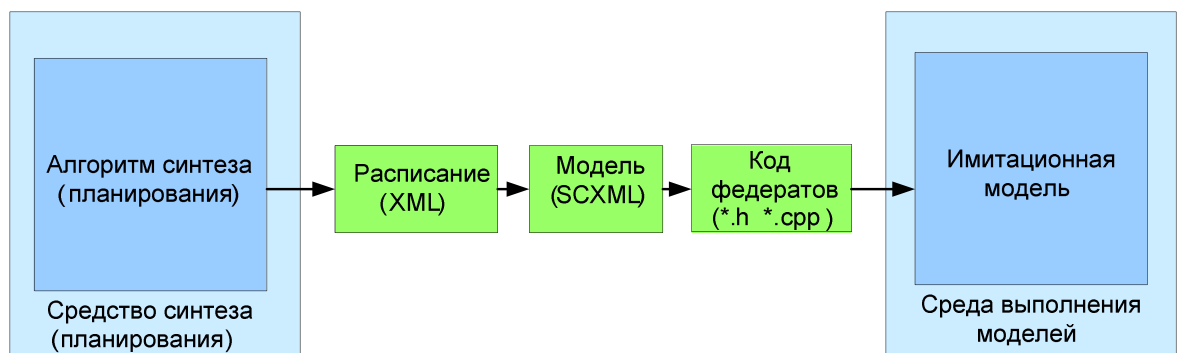

Рисунок 34 - Схема интеграции средства построения (синтеза архитектур) со средой выполнения моделей

Средство автоматической генерации модели РВС по заданному в формате XML файлу представляет собой программу на языке Python. В ходе выполнения программы содержимое входного файла представляется в виде модели DOM (Document Object Model) и по нему строится модель DOM для целевого SCXML-файла. Считается, что заданное во входном файле расписание корректно. Основной функцией данной программы является функция createTask, строящая последовательность состояний, соответствующую одному заданию. Функция createTask вызывается циклически для всех заданий определенного процессора, и сгенерированные ею последовательности состояний объединяются в один последовательный автомат.

Рассмотрим саму схему интеграции. Когда в ходе работы средства построения требуется точное вычисление времен выполнения заданий расписания, происходит вызов функции, строящей XML-файл с расписанием в формате, описанном в разделе 2.2.2. Затем вызывается shell-скрипт, выполняющий последовательно следующие действия:

1. Запуск программы, строящей по XML-файлу с расписанием SCXML-файл с моделью ВС.

2. Запуск программы, генерирующей по SCXML-файлу с моделью РВС исходный код федератов на C++, удовлетворяющий стандарту HLA. Для осуществления полностью автоматизированного запуска имитационных экспериментов данная программа была модифицирована так, что она также генерирует файлы CMakeLists.txt и launcher.py, использующиеся на следующих этапах работы скрипта. Содержимое данных файлов зависит от структуры модели РВС, поэтому их нельзя задать заранее.

3. Запуск утилиты cmake с файлом CMakeLists.txt в качестве входного параметра. При этом происходит генерация Makefile-а для сборки исполняемых файлов федератов.

4. Сборка исполняемых файлов каждого федерата с помощью утилиты make.

5. Запуск скрипта launcher.py, осуществляющего автоматический запуск имитационного эксперимента.

6. Запуск скрипта, осуществляющего поиск в выведенном федератами тексте значений переменных, соответствующих искомым временам работы программных компонентов. Найденные значения записываются в определенный файл.

Далее продолжает работу средство построения (синтеза). Из сгенерированного на 6 этапе работы скрипта файла считываются искомые значения времен.

Результаты аппробации данного средства приведены в разделе 3.6.

2.3 Интегрированная среда разработки и анализа моделей

2.3.1 Введение

На предшествующих этапах данной работы [1],[2],[3] была создана среда выполнения моделей и разработаны средства трансляции моделей из UML в язык среды моделирования (C++), средства верификации и средства визуализации результатов моделирования. Возникает задача объединить все разработанные средства в единый программный комплекс, который поддерживал бы весь цикл работы с моделями. Далее описан интерфейс программы, интегрирующей все разработанные средства.

2.3.2 Описание интерфейса

Главное окно программы (рисунок 35), помимо меню и панели инструментов, содержит три основных области: слева находится список файлов, с которыми идет работа, справа сверху – информация об открытых в данный момент файлах, справа снизу – область, куда выводятся диагностические сообщения и логи работы системы.

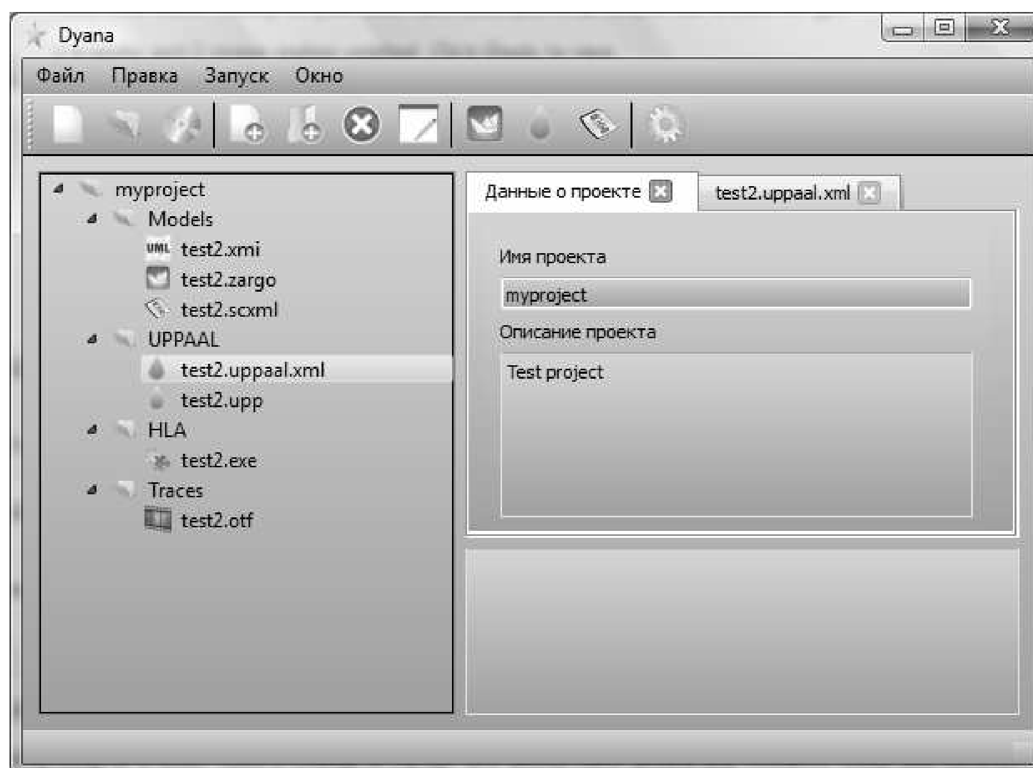

Рисунок 35 - Главное окно программы

Основная сущность, с которой работает программа – проекты. В файловой системе проект представляет собой древовидную структуру файлов и директорий, вложенную в одну общую директорию, в которой также находится файл с расширением .dyana, описывающий структуру проекта. В левой части рабочего окна в виде дерева визуализируется структура проекта, полностью совпадающая со структурой каталогов в файловой системе. У каждого типа объектов, с которыми работает программа, есть своя иконка в дереве. Типы объектов следующие:

Папки – для объединения файлов в группы.

Модели на UML – файлы с расширениями .uml и .zargo, предназначенные для обработки в ArgoUML.

Модели на XMI – файлы с расширением .xmi, экспортируемые из ArgoUML.

Модели на SCXML – файлы с расширением .SCXML, создаваемые в ручную или генерируемые на основе XMI.

Файлы UPPAAL – системы автоматов в файлах с расширением .uppaal.xml, служебная информация для анализа трасс UPPAAL в файлах с расширением .upp, свойства верификатора в файлах с расширением .q

Модели HLA – файлы исходных файлов с расширением .cpp или .h, и файлы .exe, готовые для прогона в CERTI

Файлы трасс – трассы в формате OTF.

Для каждого типа файлов существует свой диалог с настройками, который появляется в правой части окна в виде вкладки при двойном клике на файл в дереве.

В главном меню доступны следующие действия:

- File – New Project (также кнопка на панели инструментов). Создает новый проект в указанной директории. В этой директории создаются пустые папки Models, HLA, UPPAAL, Traces, а также конфигурационный файл проекта с расширением .dyana и названием таким же, как у выбранной директории.
- File – Open Project (также кнопка на панели инструментов). Загружает проект из заданного файла с расширением .dyana.
- File – Save Project (также кнопка на панели инструментов). Сохраняет состояние проекта в конфигурационный файл.
- File – Exit. Завершает работу программы.

- Edit – Add File (также кнопка на панели инструментов и пункт в контекстном меню списка файлов). Запускает стандартный диалог выбора файла, после чего файл добавляется в выделенную директорию. Физически файл копируется в директорию проекта. Тип файла определяется автоматически по расширению.
- Edit – Add Folder (также кнопка на панели инструментов и пункт в контекстном меню списка файлов). Добавляет папку с выбранным названием
- Edit – Delete (также кнопка на панели инструментов и пункт в контекстном меню списка файлов). Удаляет выбранный элемент в дереве и все элементы, вложенные в него. Файл также удаляется из файловой системы, поэтому данная операция необратима.
- Edit – Change Type (также кнопка на панели инструментов и пункт в контекстном меню списка файлов). Меняет тип, приписанный к файлу. Следует использовать, если у файла нестандартное расширение.
- Launch – ArgoUML (также кнопка на панели инструментов). Запускает ArgoUML без входного файла.
- Launch – UPPAAL (также кнопка на панели инструментов). Запускает UPPAAL без входного файла.
- Launch – SCXMLGui (также кнопка на панели инструментов). Запускает SCXMLGui без входного файла.
- Window – Settings (также кнопка на панели инструментов). Открывает окно настроек. В этом окне можно указать пути к используемым внешним программам.

Далее подробно рассмотрим окна для каждого типа файлов.

Окно для файлов UML

На рисунке приведено окно для работы с файлами UML. Основную часть окна занимает виджет, в котором можно просмотреть содержимое текущего файла. В текущей версии редактирование файла вручную не допускается.

Кнопка внизу запускает ArgoUML и сразу открывает в нем файл, соответствующий данной вкладке.

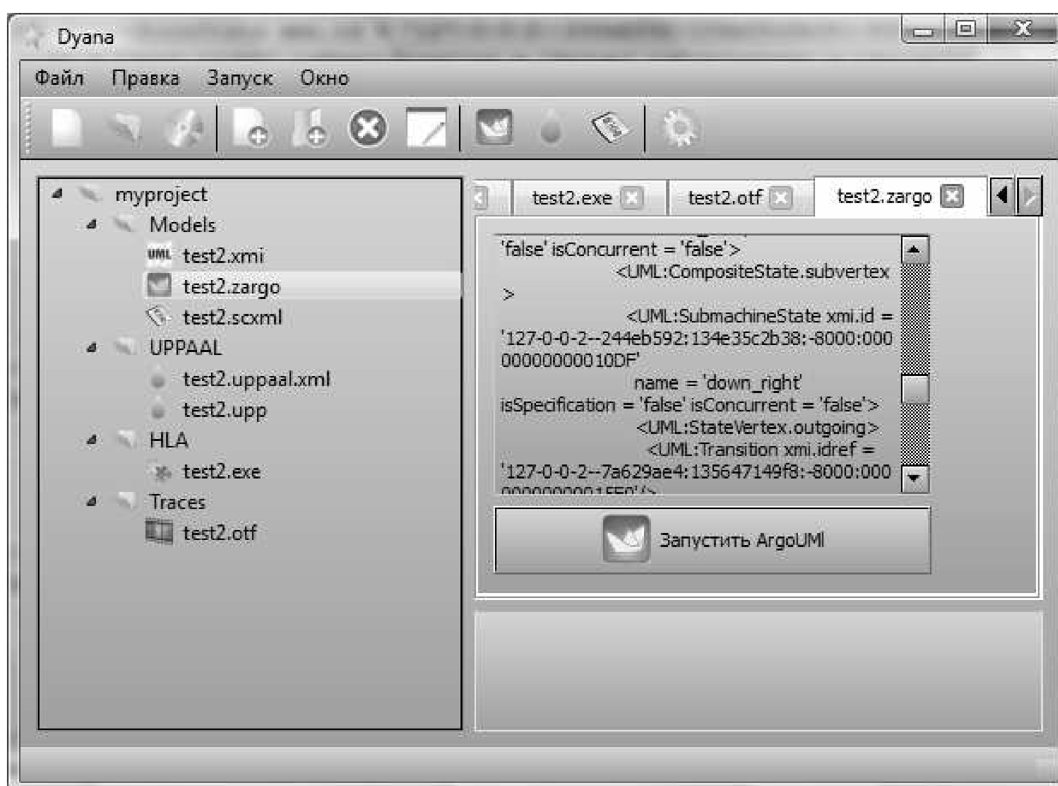

Рисунок 36 - Окно для файлов UML

Окно для файлов XMI

На рисунке приведено окно для работы с файлами XMI. При открытии XMI-файла он предварительно анализируется, чтобы определить, какие автоматы в нем заданы.

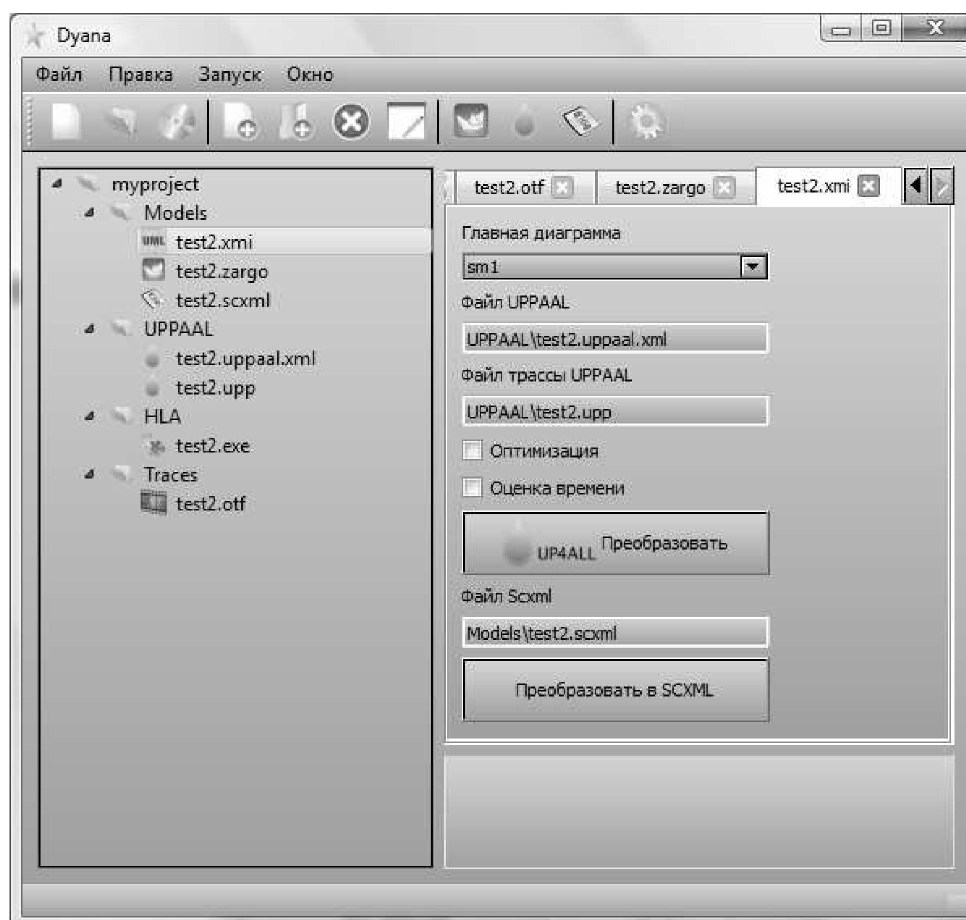

Рисунок 37 - Окно для файлов XMI

Список позволяет выбрать, какой из автоматов следует транслировать в UPPAAL. Две редактируемые строки задают путь, по которому следует записать результаты трансляции в UPPAAL. Путь задается относительно корня дерева файлов проекта. По умолчанию файлы имеют то же имя, что и XMI-файл и расширения .uppaal.xml и .upp для файла UPPAAL и вспомогательных данных для анализа трассы соответственно. По умолчанию файлы помещаются в директорию «UPPAAL». Галочки задают, необходимо ли оптимизировать автомат и оценивать время выполнения соответственно.

Окно для файлов SCXML

На рисунке приведено окно для работы с моделями в формате SCXML. Возможны три основных действия. Во-первых, можно просто открыть файл для просмотра и редактирования в редакторе SCXML GUI. Во-вторых, можно сгенерировать код модели HLA, задав путь, по которому будут сохранены файлы модели. В-третьих, можно конвертировать модель в систему временных автоматов UPPAAL, аналогично XMI-файлу.

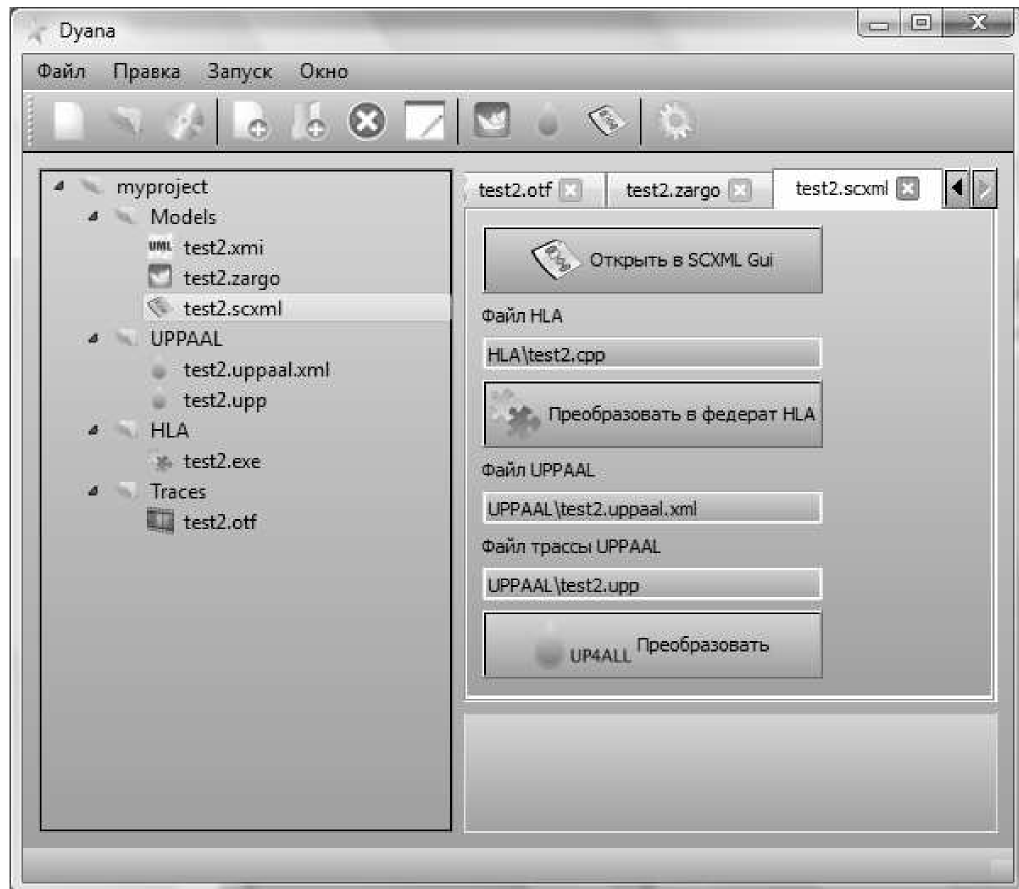

Рисунок 38 Окно для файлов моделей SCXML

Окно для файлов UPPAAL

На рисунке приведено окно для работы с файлами UPPAAL. Запускать можно только файлы с системами временных автоматов (не служебный файлы .upp и .q). Кнопка внизу вкладки запускает GUI средства UPPAAL и открывает в нем выбранный файл. Можно также сразу загрузить файл со свойствами для верификации

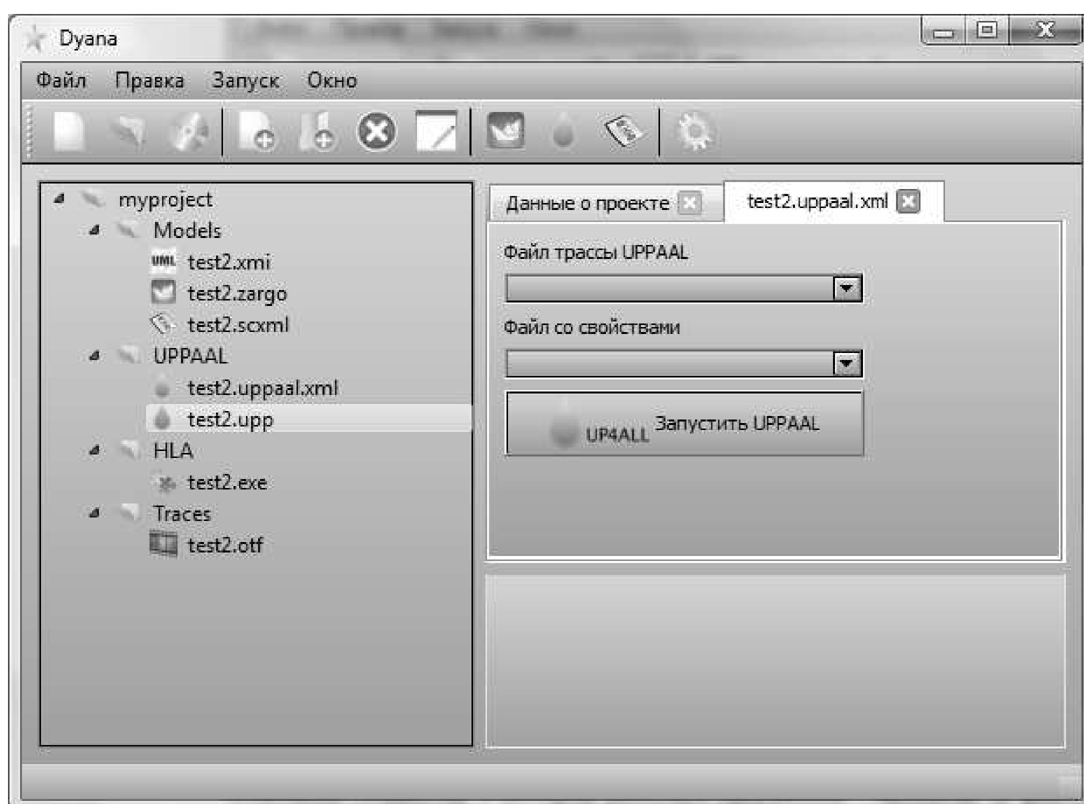

Рисунок 39 - Окно для файлов UPPAAL

Окно для файлов HLA

На рисунке приведено окно для запуска экспериментов в CERTI. Большую часть окна занимает виджет, в котором можно просмотреть содержимое файла федерата в текстовом виде. В нем отображаются исходные коды федератов на C++. Для уже скомпилированных моделей это окно пустое.

В строке задается путь, по которому будет создан файл с трассой в формате OTF.

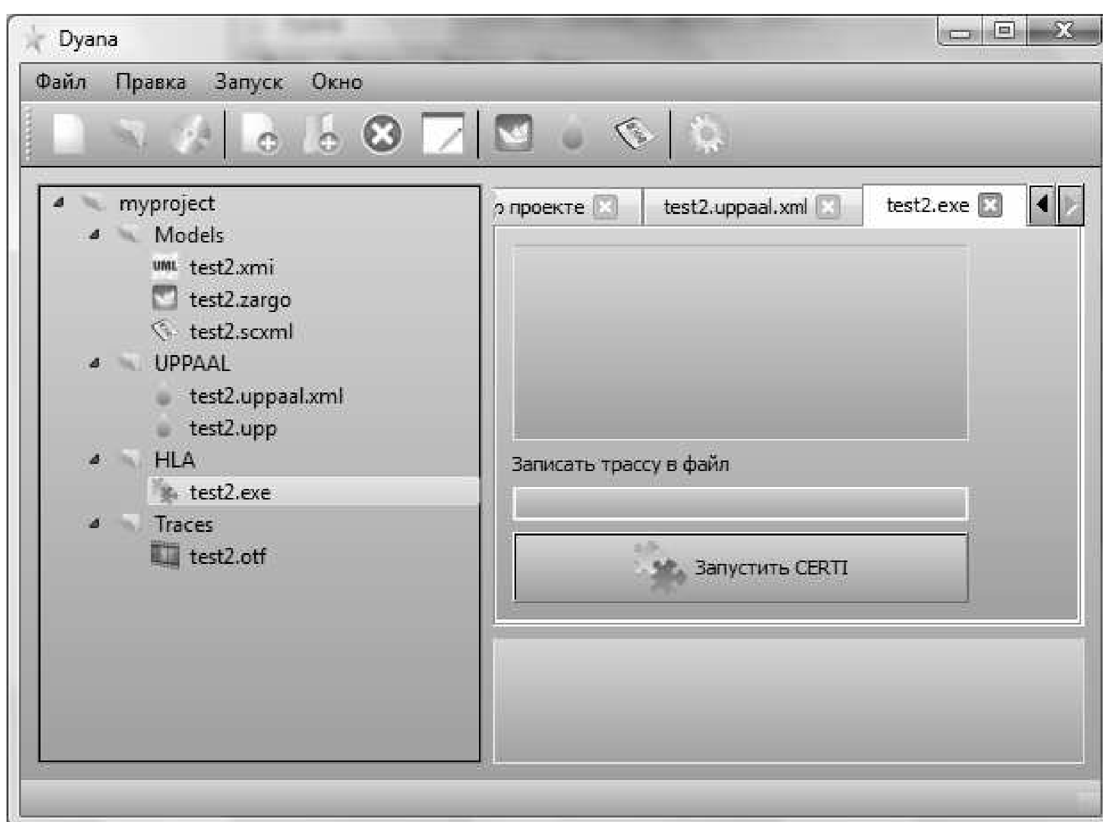

Рисунок 40 - Окно для файлов CERTI

Окно для файлов трасс

На рисунке приведено окно для работы с трассами в формате OTF. Большую часть окна занимает виджет, в котором можно просмотреть содержимое трассы в текстовом виде. Ручное редактирование трасс не допускается из соображений безопасности.

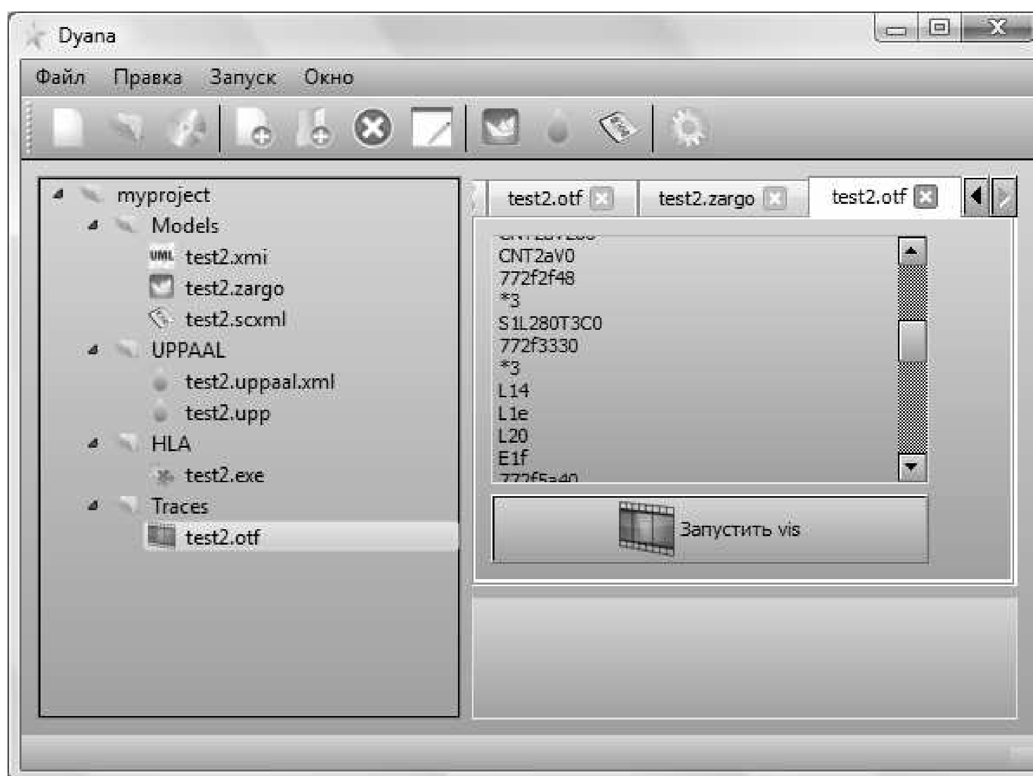

Рисунок 41 - Окно для файлов трасс

Кнопка внизу запускает vis и сразу открывает в нем соответствующую трассу.

2.3.3 Структура кода программы

Графический интерфейс для системы моделирования написан на языке Python 2.7 с использованием библиотеки Qt4 и ее привязок к языку Python, PyQt4. Для создания графических элементов использовалось средство Qt Designer.

Код главного окна содержит обработчики глобальных действий (создать / открыть / сохранить проект) и два главных виджета: ProjectView (список файлов) и TabWidget (вкладки).

Класс ProjectView унаследован от QTreeWidget. Он содержит контекстное меню и все обработчики действий для него.

Для каждого из типов вкладок существует собственный объект QWidget, все описания их собраны в файле TabWidgets.py. Вид виджетов задается одноименными формами, созданными при помощи QtDesigner. Также в этом файле задано соответствие типа объекта и соответствующего класса виджета.

3 Экспериментальное исследование второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ

В данном разделе описывается экспериментальное исследование второй очереди методов и инструментальных средств поддержки анализа и разработки РВС РВ. В разделе 3.1 приводится описание модели поведения бортовых компьютеров автомобилей. Раздел 3.2 содержит описание экспериментов по сравнению сред моделирования РВС РВ. В разделе 3.3 приводится описание экспериментов с модифицированным транслятором UML->HLA Раздел 3.4 содержит описание экспериментов по восстановлению параметров модели по контрпримеру в UPPAAL. В разделе 3.5 приведено описание экспериментов со средствами оценки наихудшего времени выполнения. В разделе 3.6 приведено описание оптимизации средства трансляции UML во временные автоматы. Раздел 3.6 содержит описание экспериментального исследования средств оптимизации надежности РВС РВ. В разделе 3.7 содержатся результаты экспериментов со средствами трассировки моделей и внесения неисправностей.

3.1 Модель поведения бортовых компьютеров автомобилей

Для апробации разрабатываемы в данной работе средств моделирования необходимо при помощи этих средств разработать исследовать модель РВС РВ. В данном разделе приведена модель РВС РВ, описывающая поведение машин на нерегулируемом перекрестке. За основу взята модель нерегулируемых перекрестков, предложенная в [57]. Данная модель переработана и разработана в средстве ArgoUML в виде UML диаграмм состояний. При этом проведены некоторые упрощения, например, не учтены в полной мере арифметические операции над действительными числами, проводимые бортовым компьютером, такие как вычисление длины тормозного пути. Далее приведены содержательное описание модели и все содержащиеся в ней диаграммы.

3.1.1 Содержательное описание модели

Два основных объекта модели – это перекресток и машина.

Перекресток представляет собой пересечение двух однополосных дорог с прилегающей к пересечению достаточно длинной проезжей частью. Дорожное полотно разбито на отдельные секции, как обозначено на рисунке 42 красными прямоугольниками.

Размеры дорожных секций в модели явно не учтены и моделируются минимальным временем, которое машина должна потратить, чтобы проехать от начала до конца секции. В каждый момент времени в каждой секции пересечения дорог может находиться не более одной машины. На прилегающей проезжей части при этом может находиться сколь угодно много машин.

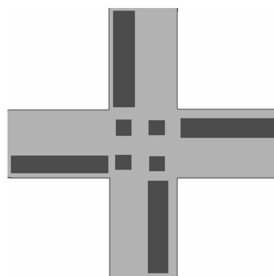

Рисунок 42 – Модель машин, подъежающих к перекрёстку.

Каждая машина моделируется двумя наборами параметров.

Первый набор описывает физические характеристики машины: занимаемая секция, положение в секции и скорость. Положение в секции в сочетании со скоростью моделируется определением минимального количества секций, которые необходимо проехать до полной остановки. Точное значение скорости машины заменено в модели тремя возможными скоростными характеристиками: машина стоит, машина едет и машина едет на максимальной скорости.

Второй набор описывает управляющую часть бортового компьютера машины. Для моделирования задержки обработки поступающей информации управляющая часть разбита на две компоненты: компонента принятия решений (водитель) и реагирующая компонента (обработчик). Водитель в каждый момент времени имеет доступ ко всей информации о ситуации на перекрестке и способен мгновенно принимать решения, однако не влияет напрямую на изменение физических характеристик машины. Обработчик способен считывать решение, принятое водителем, и изменять соответствующим образом скорость машины, однако делает это с задержкой.

При появлении машины на перекрестке и до проезда перекрестка считается фиксированным направление движения машины: поворот направо, проезд прямо, поворот налево.

Считается, что на перекрестке в каждый момент времени могут находиться машины в количестве, не превышающем заранее определенное число (в дальнейшем описании – не более двух). Каждая машина, находящаяся на перекрестке, моделируется отдельной StateChart-диаграммой. Появившись на перекрестке, машина начинает движение с соблюдением набора правил.

3.1.2 Учтенные правила дорожного движения

В модели учтены следующие правила, являющиеся упрощенной записью правил дорожного движения и правил движения «по договоренности».

1. Водитель обязан уступить дорогу машинам, приближающимся справа.
2. При повороте налево водитель обязан уступить дорогу машинам, движущимся со встречного направления и едущим прямо или направо.
3. Если машина, подъезжающая слева, не имеет возможности остановиться, чтобы пропустить водителя согласно первому правилу, водитель должен пропустить эту машину во избежание аварии.
4. Аналогичное действие совершается для второго пункта.

Кроме того, в модели учтены и правила «здравого смысла», например, торможение, если попутная секция перекрестка занята.

3.1.3 Синтаксис диаграмм

В данном подразделе кратко описывается синтаксис UML диаграмм состояний, используемый при описании модели.

Для описания состояний модели используются композитные и простые состояния. Композитное состояние может использовать параллельные регионы для обозначения параллельно работающих вложенных компонент. Диаграмма, содержащаяся в каждом параллельном регионе, описывает последовательно работающую компоненту.

С каждым композитным состоянием, кроме объемлющего, ассоциированы одно входное состояние и не более одного выходного. Из входного состояния ведет дуга в ассоциированное композитное состояние. Из композитного состояния, в свою очередь, ведет дуга в выходное.

Разрешается использование ссылок на диаграммы. При трансляции вместо ссылок подставляются диаграммы, на которые они ссылаются, пока не будут устранены все ссылки.

Например, на рисунке 44 изображено объемлющее композитное состояние с четырьмя параллельными регионами, и каждый из регионов описан ссылкой на диаграмму.

Состояния внутри последовательно работающей компоненты могут быть соединены дугами. Каждая дуга может быть помечена

1. временным событием (на диаграммах – «after(...)»),
2. событием приема сообщения,
3. предохранителем (на диаграммах – «[...]») и
4. действием (на диаграммах – выражение после символа «/»).

Событие приема сообщения представляет собой имя канала. Действие посылки сообщения имеет вид «!!c», где c – имя канала. Примеры выражений представлены, например, на рисунке 49.

Каждому простому состоянию может быть приписан инвариант (на диаграммах – «do / assume(...)»). Примеры записи инвариантов представлены, например, на рисунке 48.

При построении выражений может использоваться запись «self.c» для обозначения времени нахождения в состоянии, которому приписан инвариант или из которого исходит дуга.

Все используемые выражения имеют C-подобную структуру. Переменные, локальные для композитного состояния и использующиеся при построении выражений, определяются в комментарии, ассоциированном с данным композитным состоянием. Также разрешено использование макроподстановок, имеющих тот же смысл, что и в языке C.

3.1.4 Описание модели

Для краткости записи далее группы переменных будут объединены в массивы с использованием C-подобного синтаксиса. Кроме того, все формулы переписаны в синтаксически неверном, но более понятном виде. Для восстановления исходных формул по предложенным далее записям достаточно записать все массивы как группы переменных, заменить в диаграммах каждую дугу, помеченную массивом, на дуги, помеченные переменными, и привести все формулы к виду дизъюнктивной нормальной формы.

Характеристики системы пронумерованы следующим образом (Таблица 2):

Таблица 2 – Нумерация характеристик системы

Характеристика	Значение	Номер
Секция пересечения дорог	Правая верхняя	0
	Правая нижняя	1
	Левая нижняя	2
	Левая верхняя	3
Направления подъезда к перекрестку	Справа	0
	Снизу	1
	Слева	2
	Сверху	3
Направление движения машины	Направо	0
	Прямо	1
	Налево	2
Относительное положение машины	Не появилась	0
	Подъезжает к пересечению дорог	1
	На первой секции пересечения	2
	На второй секции пересечения	3
	На третьей секции пересечения	4
	Проехала	5
Действия водителя	Проезжать	0
	Остановиться до перекрестка	1
	Остановиться не позже ближайшей секции пересечения	2
	Не позже второй секции	3
	Проезжать с максимальной скоростью	4
Скорость машины	Ненулевая	0
	Нулевая	1
	Максимальная	2

В модели заведены следующие каналы (Таблица 3). При числе каналов, начиная от двух, подразумевается массив каналов.

Таблица 3 - Каналы

Имя	Число каналов	Значение
free	16	free[i + 4*j] обозначает освобождение секции i машиной, прибывшей к перекрестку с направления j
occupy	16	occupy[i + 4*j] обозначает занятие секции i машиной, прибывшей к перекрестку с направления j
arrive	4	arrive[i] обозначает появление машины на перекрестке со стороны i
cant_stop_1	4	канал для подсчета числа машин, участвующих в правиле движения 3
urgent_chan	1	приписывается переходам, которые должны сработать как можно быстрее

В объемлющем состоянии crossroads заведены следующие переменные, таймеры и макроопределения (Таблица 4). В данной таблице bool означает булев тип, int[i,j] – целочисленный от i до j, суффикс [k] – «массив из k элементов».

Таблица 4 – Переменные, таймеры и макроопределения

Имя	Тип	Значение
place	bool [4]	place[i] обозначает возможность занять секцию i
cars_b	int[0..2] [4]	cars_b[i] обозначает количество машин на дороге, прилегающей к перекрестку со стороны i
cars_n2	int[0..2] [4]	cars_n2[i] обозначает количество машин, подъезжающих к перекрестку со стороны i и участвующих в правиле движения 3
in_broad	таймер	необходим для рассылки сигнала urgent_chan
TM_B	макроопределение	«4»; минимальное время прохождения машиной проезжей части, прилегающей к перекрестку
TM_CC	макроопределение	«1»; минимальное время прохождения машиной секции на пересечении дорог
TM_CS	макроопределение	«1»; минимальное время между изменениями значений счетчика stop_counter в диаграмме машины
TM_P	макроопределение	«1»; задержка между действиями обработчика
TM_SU	макроопределение	«2»; минимальное время перехода от нолевой скорости к максимальной

В таблице опущены начальные значения переменных. По умолчанию подразумевается, что начальные значения – 0 для типа int и true для типа bool.

Объемлющее состояние crossroads изображено на Рисунок 43 и имеет три вложенных компонента следующего назначения:

1. controller – автомат, собирающий информацию о наличии на различных частях перекрестка машин и о занятости секций на пересечении дорог и рассылающий сигнал по каналу urgent_chan;
2. car1 – первая машина; после проезда перекрестка может опять появиться с любой из сторон и ехать в любом допустимом направлении;

3. car2 – вторая машина; после проезда перекрестка также может появиться с любой из сторон и ехать в любом допустимом направлении.

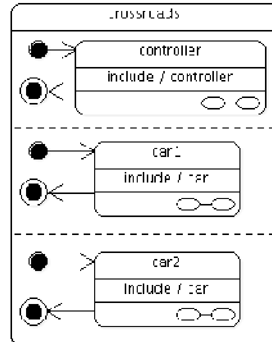

Рисунок 43 – Объемлющее состояние crossroads

Компонент controller изборажен на рисунке 44 и имеет, в свою очередь, три вложенных компоненты:

1. counters – компонент, считающий число машин, участвующих в учтенных правилах движения;
2. places – компонент, контролирующий занятость секций перекрестка и наличие на нем аварий;
3. urgent_sender – компонент, рассылающий каждый такт времени сигнал по каналу urgent_chan.

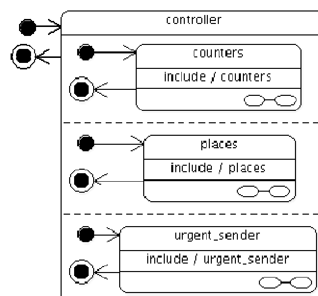

Рисунок 44 - Компонент controller

Компоненты counters, places и urgent_sender изображены, соответственно, на рисунках 45, 46 и 47.

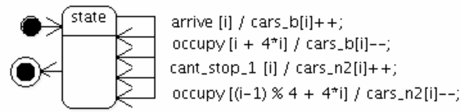

Рисунок 45 - Компонент counters

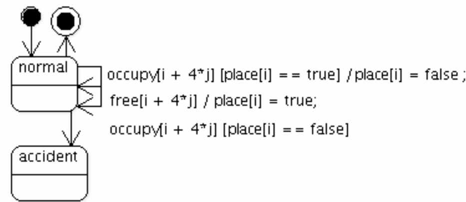

Рисунок 46 - Компонент places

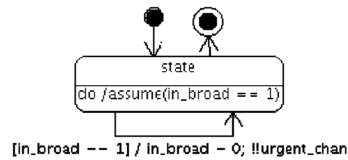

Рисунок 47 - Компонент urgent_sender

Диаграмма car, экземплярами которой являются компоненты car1 и car2 объемлющего состояния, изображена на рисунке 48 и содержит два параллельно работающих вложенных состояния: отвечающее физическим характеристикам машины (characteristics) и отвечающее управляющей компоненте бортового компьютера (intentions).

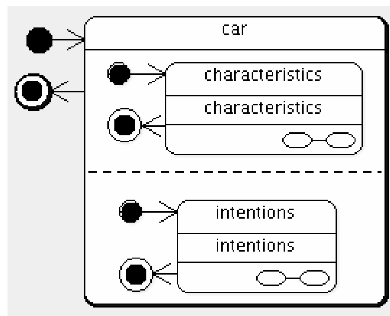

Рисунок 48 - Диаграмма car

Кроме того, в диаграмме car определены следующие локальные переменные, таймеры и макроопределения (Таблица 5):

Таблица 5 - Локальные переменные, таймеры и макроопределения диаграммы car

Имя	Тип	Значение
in_pos	таймер	время пребывания в текущей секции перекрестка
pos_counter	int[0..5]	относительное положение машины в текущий момент
driver_counter	int[0..4]	действие водителя, отвечающее текущей ситуации на перекрестке
intention_counter	int[0..4]	в данную переменную обработчик с задержкой записывает текущее действие водителя
speed_counter	int[0..2]	скорость машины в текущий момент
stop_counter	int[0..5]	минимальное относительное положение машины, на котором она сможет остановиться при текущей скорости и текущем положении
from	int[0..3]	направление, с которого машина подъезжает к перекрестку
to	int[0..2]	направление движения машины

Компонента characteristics изображена на рисунке 49 и содержит в себе три вложенные компоненты: относительное положение машины (position), миинимальное относительное положение машины, на котором она может остановиться (stop_comp) и скорость машины (speed).

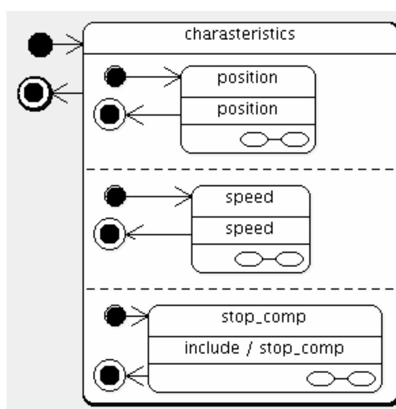

Рисунок 49 - Диаграмма characteristics

Компоненты position, speed и stop_comp представлены на рисунках 50, 51 и 52, соответственно.

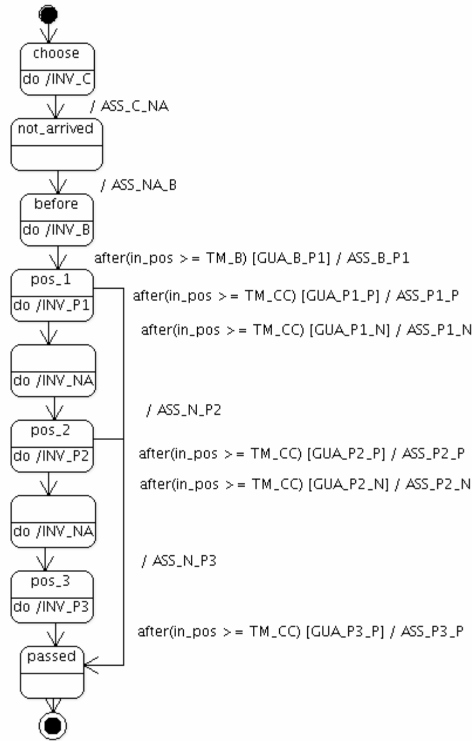

Рисунок 50 - Компонента position

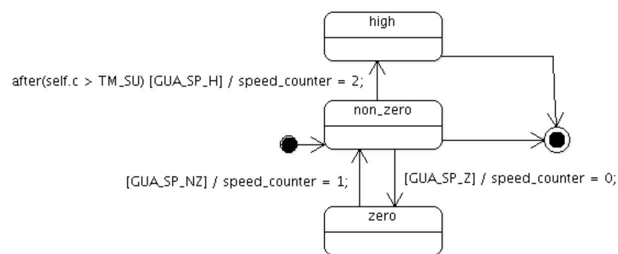

Рисунок 51 - Компонента speed

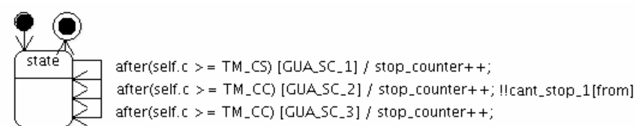

Рисунок 52 – Компонента stop_comp

Макроопределения, обозначенные на представленных рисунках идентификаторами, состоящими из заглавных букв, расшифровываются следующим образом (Таблица 6):

Таблица 6 -- Макроопределения

<table>
<tr><td>Имя</td><td>Значение</td></tr>
<tr><td>ASS_C_NA</td><td>from=random(); to=random(); pos_counter = 0; stop_counter = 0; intention_counter = 0; driver_counter = 0; speed_counter = 0; in_pos = 0; proc_time = 0; stop_time = 0;</td></tr>
<tr><td>ASS_NA_B</td><td>pos_counter++; !!arrive[from]</td></tr>
<tr><td>GUA_B_P1</td><td>!(speed_counter == 1) && (place[from] == true) && (stop_counter > 0)</td></tr>
<tr><td>ASS_B_P1</td><td>in_pos = 0; pos_counter++; !!occupy[from + 4 * from]</td></tr>
<tr><td>GUA_P1_N</td><td>!(to == 0) && !(speed_counter == 1) && (place[(from - 3) % 4] == true) && (stop_counter > 1)</td></tr>
<tr><td>ASS_P1_N</td><td>in_pos = 0; !!free[from % 4 + 4 * from]</td></tr>
<tr><td>ASS_N_P2</td><td>pos_counter++; !!occupy[(from - 3) % 4 + 4 * from]</td></tr>
<tr><td>GUA_P2_N</td><td>(to == 2) && !(speed_counter == 1) && (place[(from - 2) % 4] == true) && (stop_counter > 2)</td></tr>
<tr><td>ASS_P2_N</td><td>in_pos = 0; !!free[(from – 3) % 4 + 4 * from]</td></tr>
<tr><td>ASS_N_P3</td><td>pos_counter++; !!occupy[(from – 2) % 4 + 4 * from]</td></tr>
<tr><td>GUA_P3_P</td><td>!(speed_counter == 1)</td></tr>
<tr><td>ASS_P3_P</td><td>pos_counter++; !!free[(from – 2) % 4 + 4 * from]</td></tr>
<tr><td>GUA_P1_P</td><td>(to == 0) && !(speed_counter == 1)</td></tr>
<tr><td>ASS_P1_P</td><td>pos_counter++; !!free[from % 4 + 4 * from]</td></tr>
<tr><td>GUA_P2_P</td><td>(to == 1) && !(speed_counter == 1)</td></tr>
<tr><td>ASS_P2_P</td><td>pos_counter++; !!free[(from - 3) % 4 + 4 * from]</td></tr>
<tr><td>GUA_SP_H</td><td>intention_counter == 4</td></tr>
<tr><td>GUA_SP_NZ</td><td>(intention_counter == 0) || (intention_counter == 4)</td></tr>
<tr><td>GUA_SP_Z</td><td>!(intention_counter == 0) && !(intention_counter == 4) && (stop_counter <= pos_counter - 1)</td></tr>
<tr><td>GUA_SC_1</td><td>(pos_counter > 0) && (stop_counter == 0) && !(speed_counter == 1) && ((intention_counter == 0) || (intention_counter == 4))</td></tr>
<tr><td>GUA_SC_2</td><td>!(to == 0) && (stop_counter == 1) && !(speed_counter == 1) && ((intention_counter == 0) || (intention_counter == 4))</td></tr>
</table>

<table>
<tr><td>GUA_SC_3</td><td>(to == 2) && (stop_counter == 2) && !(speed_counter == 1) && ((intention_counter == 0) || (intention_counter == 4))</td></tr>
<tr><td>INV_C</td><td>assume(in_pos <= 0)</td></tr>
<tr><td>INV_B</td><td>assume(!(speed_counter == 2) && (stop_counter == 0) || (in_pos <= 4))</td></tr>
<tr><td>INV_P1</td><td>assume(!(speed_counter == 2) && (stop_counter == 1) || (in_pos <= 1))</td></tr>
<tr><td>INV_P2</td><td>assume(!(speed_counter == 2) && (stop_counter == 2) || (in_pos <= 1))</td></tr>
<tr><td>INV_P3</td><td>assume(!(speed_counter == 2) && (stop_counter == 3) || (in_pos <= 1))</td></tr>
</table>

Состояние choose диаграммы position соответствует недетерминированному выбору машиной направления подъезда к перекрестку и направления движения. Состояние not_arrived соответствует ситуации, когда направления уже определены, но машина еще не появилась на перекрестке. Состояние passed соответствует ситуации, когда машина выехала с перекрестка. Остальные именованные состояния соответствуют относительным положениям машины на перекрестке. Неименованные состояния диаграммы position используются для технических целей, а именно в связи с тем, что одна дуга не может быть помечена двумя различными синхронизациями.

Состояния non_zero, zero и speed_up диаграммы speed соответствуют ненулевой, нулевой и максимальной скорости машины.

Компонент, описывающий поведение управляющего устройства бортового компьютера, представлена на рисунке 53, ее компоненты «водитель» (driver) и «обработчик» (processor) – на рисунках 54, 55, соответственно.

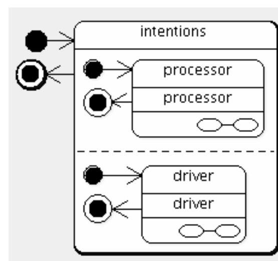

Рисунок 53 - Компонент, описывающий поведение управляющего устройства бортового компьютера

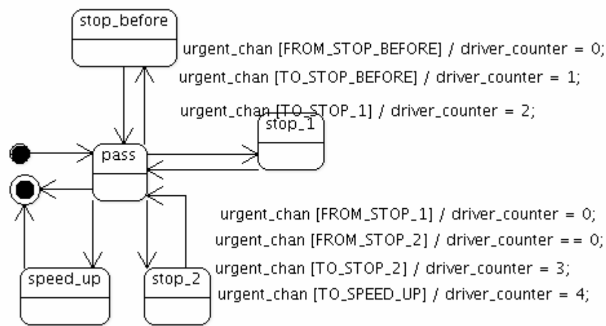

Рисунок 54 - Компонента «Водитель» (driver)

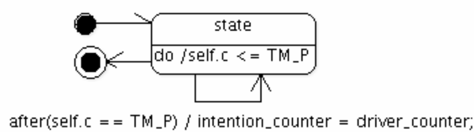

Рисунок 55 - Компонента «Обработчик» (processor)

Макроопределения, обозначенные на рисунке идентификаторами, состоящими из заглавных букв, расшифровываются следующим образом:

Таблица 7 – Расшифровка макроопределений

Имя	Значение
FROM_STOP_BEFORE	cars_n2[(from + 1) % 4] == 0
TO_STOP_BEFORE	(pos_counter > 0) && (cars_n2[(from + 1) % 4] > 0) && (stop_counter == 0)
TO_STOP_1	(pos_counter > 0) && !(to == 0) && (stop_counter <= 1) && ((cars_b[(from - 1) % 4] > 0) || (place[(from – 1) % 4] == false))
FROM_STOP_1	(cars_b[(from - 1) % 4] == 0) && (place[(from – 1) % 4] == true)
FROM_STOP_2	(cars_b[(from - 2) % 4] == 0) && (place[(from – 2) % 4] == true)
TO_STOP_2	(pos_counter > 0) && (to == 2) && (stop_counter <= 2) && ((cars_b[(from - 2) % 4] > 0) || (place[(from – 2) % 4] == false))
TO_SPEED_UP	(pos_counter > 0) && (!(to == 0) && (cars_b[(from - 1) % 4] > 0) && (stop_counter > 1) || (to == 2) && (cars_b[(from - 2) % 4] > 0) && (stop_counter > 2) || (pos_counter == 2 + to))

Состояния pass, stop_before, stop_1, stop_2 и speed_up диаграммы speed соответствуют намерению водителя проехать перекресток, остановиться, не выезжая на перекресток,

остановиться на первой проезжаемой секции перекрестка, на второй секции и проехать перекресток на максимальной скорости.

3.1.5 Проверка свойств модели

Разрабатываемый нами алгоритм трансляции позволяет по предложенной модели построить сеть плоских временных автоматов, используемых средством верификации UPPAAL. Целью трансляции в сеть плоских временных автоматов является проверка выполнимости спецификаций сети, выраженных на языке формул темпоральной логики CTL_{-X}. При этом множество формул, синтаксически допустимых в средстве UPPAAL, достаточно широко, чтобы проверять важные свойства системы, такие как отсутствие тупиков и свойства безопасности и живости системы, которые можно охарактеризовать, соответственно, как «в системе не произойдет ничего плохого» и «в системе обязательно будет происходить что-то хорошее».

Представленные в следующей таблице свойства проверялись на описанной системе с двумя машинами. Следует отметить, что проверка свойств осуществлялась на маломощном вычислительном устройстве класса нетбук. Для простоты записи в обозначенных свойствах имена плоских временных автоматов, получаемых в результате трансляции, заменены именами диаграмм в исходной модели (Таблица 8).

Таблица 8 – Проверяемые свойства

Свойство	Время проверки	Выполнимость
A[] not deadlock	8 секунд	выполнено
car1_position.before **imply** car1_position.passed	1 секунда	не выполнено
car1_position.pos_1 **imply** car1_position.passed	6 секунд	выполнено
A[](car1_from == 0 && car2_from == 1 → (**not** car1_position.pos_1 **or not** car2_position.pos_2))	3 секунды	выполнено
A[] not places.accident	2 секунды	выполнено

Первое свойство является свойством отсутствия тупиков: все вычисления в модели являются бесконечными; иначе говоря, какой бы конфигурации (множества активных состояний, значений переменных и значений таймеров) мы ни достигли, всегда можно или выполнить какой-либо переход, или увеличить время. Данное свойство, помимо прочего, позволяет оценить размер пространства состояний построенной модели.

Второе свойство является свойством живости: если машина подъехала к перекрестку, то она обязательно его проедет. Однако в описанной модели такое свойство живости не выполняется. Можно привести контрпример к данному свойству со следующим содержательным описанием. Машина появляется на перекрестке, после чего бесконечно долго сбавляет скорость, не останавливаясь и не выезжая на пересечение дорог. В терминах диаграмм, автомат position может бесконечно долго оставаться в состоянии before.

Третье свойство также можно считать свойством живости: если машина выехала непосредственно на пересечение дорог, то она обязательно проедет перекресток. Его выполнимость показывает, что контпример, приведенный для второго свойства, является, по сути, единственным.

Четвертое и пятое свойства являются свойствами безопасности.

Четвертое свойство является свойством взаимного исключения (mutual exclusion) и может быть сформулировано следующим образом: если одна машина едет справа, тогда как вторая – снизу, то они не могут одновременно занимать правую верхнюю секцию. Аналогичные свойства можно сформулировать для любых направлений и любой секции перекрестка, и все они также выполняются.

Выполнимость пятого свойства подтверждает недостижимость «аварийного» состояния в модели, т.е., согласно содержательному описанию модели и условию перехода в «аварийное состояние», отсутствие аварий на перекрестке. Пятое свойство фактически совпадает со всеми аналогами четвертого свойства, взятыми в совокупности, однако формально отличается, формулируется более кратко и проверяется за меньшее время.

Представляет интерес проверка свойств и для модели, содержащей большее число машин. Однако проверка свойств уже для модели с тремя машинами требует больших вычислительных мощностей. Так, время проверки свойства отсутствия тупиков в модели с тремя машинами на том же вычислительном устройстве заняло около часа. Такое увеличение времени проверки связано как с увеличением пространства состояний сети плоских временных автоматов, так и с увеличением размера представления состояний, что выражается в падении скорости обработки состояний в десятки раз. Такое увеличение размера представления состояний связано с существенным увеличением числа процессов сети при добавлении в модель машины.

3.2 Эксперименты по сравнению сред моделирования РВС РВ

Как было показано в разделе 1.1 компонент LRC базовой версии системы CERTI состоял из двух процессов – процесса федерата, участника моделирования, и процесса RTIA, предоставляющего набор сервисов стандарта HLA этому федерату. При этом всё взаимодействие между процессами осуществлялось с помощью механизма передачи сообщений, требующего предобработки данных, и отрицательно сказывающемся на производительности среды выполнения.

На данном этапе настоящей работы был разработан прототип MT-CERTI (Multi-Threaded CERTI) с многопоточной реализацией локального компонента LRC. Более точно, процесс RTIA был реализован как дополнительный поток управления в контексте процесса федерата. Тем самым, взаимодействие составных частей LRC было перенесено с уровня процессов на уровень потоков, предоставляющий более эффективные механизмы обмена. Так как потоки выполняются в едином контексте процесса и имеют общее адресное пространство, то для передачи данных достаточно передать лишь указатель на эти данные, не прибегая к дополнительной конвертации или копированию памяти.

В данном разделе показатели производительности оригинальной системы CERTI сравниваются с аналогичными показателями прототипа MT-CERTI. Раздел содержит описание методики экспериментального исследования, результаты проведённого на данном этапе работы тестирования и их всесторонний анализ.

3.2.1 Методика исследования

Методика сравнения двух систем моделирования в силу сложности этих систем должна учитывать множество разнородных факторов. В настоящей работе предлагается методика, учитывающая следующие параметры:

1. Аспекты тестирования – набор характеристик среды выполнения, важных для её эффективного функционирования. Примером аспектов тестирования могут служить время отклика, ограничивающее минимальный размер допустимого директивного интервала и, как следствие, диапазон задач реального времени, которые система способна решать.
2. Тестовые сценарии – набор имитационных моделей, точечно нагружающих отдельные компоненты среды выполнения и позволяющие оценить выделенные показатели производительности сред выполнения. Например, набор моделей для

тестирования масштабируемости систем должен включать модели с разным количеством участников.

3. Тестовый стенд – включает в себя описание используемого во время прогона тестов программного и аппаратного окружения. Используемое окружение должно соответствовать тестовому сценарию и быть одинаковым для всех исследуемых сред выполнения.
4. Измеряемые параметры – множество показателей производительности системы, которые измеряются в процессе тестирования. Если аспекты тестирования – характеристики среды выполнения, как единой системы, то измеряемые параметры – показатели её отдельных составляющих. Примерами измеряемых параметров могут служить загрузка процессора, использование памяти на конкретной машине, интенсивность сетевой передачи данных.

3.2.2 Аспекты тестирования

Время отклика

В общем случае одной из основных характеристик среды выполнения является её среднее *время отклика* – размер временного интервала с момента передачи события среде выполнения и до завершения обработки этого события. В случае распределённой RTI, описанной стандартом HLA, временем отклика можно считать время, прошедшее с момента отправки сообщения федератом-издателем и до момента его доставки всем федератам-подписчикам. Таким образом, время отклика определяет скорость реакции системы моделирования на изменения в состоянии модели.

При выполнении задач моделирования реального времени, решение которых является одной из первостепенных целей настоящей работы, время отклика модели накладывает ограничения на минимальный размер директивных интервалов. Тем самым, время отклика существенно влияет на диапазон имитационных задач, которые могут быть решены с использованием заданной среды выполнения.

Пропускная способность

Другим важным параметром среды выполнения является её *пропускная способность*. Участники моделирования могут рассматривать среду выполнения как транспортную среду, обеспечивающую доставку передаваемых данных от отправителя к получателю. Действительно, распределённая инфраструктура RTI скрывает от федератов детали сетевого взаимодействия и представляет им интерфейс для передачи сообщений. Поэтому RTI, как и

любую транспортную среду, можно охарактеризовать её пропускной способностью – объёмом данных, которые она способна передать в единицу времени.

Во время выполнения имитационной модели пропускная способность среды выполнения ограничивает интенсивность взаимодействия участников моделирования. Пусть несколько участников моделирования обмениваются данными, причём их обмены не зависят друг от друга, и поэтому не зависят от времени отклика модели. Если обмен данными будет достаточно интенсивным, то RTI с низкой пропускной способностью не будет успевать их обрабатывать и станет «узким местом» системы.

Нагрузка на логические компоненты

На логическом уровне любая распределённая среда выполнения состоит из набора локальных компонентов LRC и, возможно, центрального компонента CRC. На более низком уровне абстракции каждый из этих логических компонентов может быть образован несколькими процессами, пусть даже работающими на разных машинах. Например, компонент LRC системы CERTI состоит из процесса RTIA и библиотеки libRTI, выполняющейся в контексте процесса федерата.

Анализ производительности среды выполнения на уровне крупных логических блоков позволяет определить «узкие места» системы и понять, каким образом её можно оптимизировать на уровне макроэлементов. Например, при интенсивном обмене данными между участниками моделирования узким местом может стать компонент CRC, в случае CERTI образованный процессом RTIG. Поэтому целесообразным является исследование возможностей для снижения нагрузки на RTIG, например, построение среды выполнения с каскадной архитектурой.

Масштабируемость

Время отклика и пропускная способность среды выполнения могут значительно изменяться в зависимости от размера выполняемой имитационной модели. Чем больше имитационная модель, тем сложнее поддерживать её в согласованном состоянии, и тем менее производительной становится среда выполнения. Динамика подобных изменений называется *масштабируемостью* среды выполнения.

В случае распределённой среды RTI, описанной спецификациями стандарта HLA, вопрос масштабируемости можно рассматривать сразу с нескольких позиций:

1. Масштабируемость среды выполнения **на уровне федератов** предполагает исследование зависимости эффективности среды выполнения от количества участников моделирования и их характеристик. Например, нагрузка на RTI от

участника-сборщика, продвигающего своё логическое время крупными шагами, и получающего сообщения лишь определённого вида, будет гораздо меньше, чем от активно взаимодействующего федерата.

2. Масштабируемость **на уровне инструментальных машин** изучает вопрос производительности среды выполнения в зависимости от используемых хостов. Вместе с наращиванием мощности задействованного оборудования показатели производительности системы должны возрастать, однако рост распределённой системы приводит к соответствующему росту расходов на её синхронизацию и снижению эффективности использования ресурсов каждой отдельной машины. Принципиальным здесь является вопрос совместного размещения федератов: что выгоднее: использовать больше инструментальных машин или увеличивать количество участников моделирования на существующих машинах, какие виды взаимодействующих федератов выгодно размещать вместе, а какие раздельно.
3. Стандарт HLA предполагает возможность использования одной и той же среды выполнения для одновременного проведения сразу нескольких имитационных экспериментов. Масштабируемость среды выполнения **на уровне федераций** рассматривает целесообразность такого её использования. Важно исследовать вопрос взаимного влияния моделей друг на друга, определить динамику падения производительности системы при увеличении числа исполняемых моделей, найти максимальное количество моделей, одновременное выполнение которых может быть целесообразным.

3.2.3 Тестовые сценарии

Основными характеристиками производительности среды выполнения являются время отклика и пропускная способность модели. В самом деле, загрузка логических компонентов может быть определена одновременно с измерением времени отклика и пропускной способности, а масштабируемость изучает зависимость данных величин относительно размера имитационной модели и используемого аппаратного комплекса. Поэтому для настоящей работы достаточным было разработать только два типа моделей.

Имитационную модель можно рассматривать как одну распределённую программу, состоящую на логическом уровне из инфраструктуры RTI и множества подключённых к ней участников моделирования – федератов. Однако в рамках данного раздела нужно исследовать характеристики среды выполнения RTI, минимизировав их корреляции со

свойствами используемых при этом множеством федератов. Поэтому разработанные были максимально упрощены.

Каждая из созданных моделей состоит из федератов двух типов – терминала и вычислителя. Каждый терминал передаёт заданное количество сообщений вычислителям. В модели «Лавина» вычислитель лишь регистрирует полученные сообщения. В модели «Пинг-Понг» вычислитель дополнительно отвечает на каждое сообщение собственным сообщением с тем же телом, а терминал не передаёт новых сообщений, пока не получит уведомление от вычислителя. Несмотря на простоту описанных моделей, аналогичные им имитационные задачи часто используются в практике исследования систем [8]. Аналогичные модели входят, например, в пакет тестирования «RTINGv6-Benchmarks», использовавшимся разработчиками системы моделирования DMSO.

Для измерения пропускной способности среды выполнения в рамках настоящей работы будет использоваться модель «Лавина». В рамках этого тестового сценария участники моделирования работают полностью асинхронно: терминал лишь генерирует поток данных, а вычислитель только отмечает доставленные ему данные. Поэтому скорость моделирования напрямую зависит от количества данных, которые система может передать в единицу времени – пропускной способности системы. Таким образом, значение пропускной способности может быть получено в виде отношения количества переданных данных ко времени выполнения модели.

Модель «Пинг-Понг» хорошо подходит для измерения времени отклика системы. Действительно, терминал не посылает новых сообщений, пока не получит уведомления о доставке предыдущего сообщения от вычислителя, а вычислитель не отправляет это уведомление, пока не получит сообщения от терминала. Однако время отклика среды выполнения в рамках распределённой системы моделирования определяется как размер временного интервала с момента передачи сообщения от участника-отправителя до момента его доставки участником-получателем.

Принимая время обработки поступившего сообщения вычислителем пренебрежимо малым, можно считать, что время между отправкой сообщения терминалом и получением ответа от вычислителя равно удвоенному времени отклика среды выполнения. Если время обработки уведомления от вычислителя также пренебрежимо мало, то время работы терминала с момента отправки им своего первого сообщения и до получения последнего уведомления от вычислителя равно произведению времени отклика среды выполнения на удвоенное количество отправленных терминалом сообщений.

В качестве передаваемого сообщения на данном этапе работы был использован единственный целочисленный параметр, значение которого устанавливалось до начала выполнения модели и уменьшалось на единицу при каждой передаче. Терминал и вычислитель завершали своё выполнение, когда передаваемый параметр достигал нулевого значения. Упаковка единственного параметра в необходимый формат требует минимальных затрат, а уменьшение значения на единицу позволяет легко контролировать корректность передаваемых данных. В дальнейшем, однако, необходимо дополнительно исследовать зависимость показателей производительности системы моделирования от количества, структуры и размера передаваемых между участниками сообщений. Излишняя сложность формата передачи данных, предусмотренного стандартом HLA, может негативно сказываться на производительности системы моделирования.

3.2.4 Режимы работы

Система «CERTI» реализует спецификации стандарта моделирования HLA, и изначально разрабатывалась как распределённая система. При этом разработчики не уделяли особенного внимания случаю её нераспределённого использования, когда все участники моделирования запускались бы на единственной инструментальной машине. При работе на одной машине данная система обеспечивает взаимодействие участников моделирования с использованием тех же самые механизмов, что и при распределённом моделировании.

Среда «Стенд ПНМ» способна работать в нескольких режимах: жёсткого РВ, мягкого РВ, вне РВ и режиме отладки. Режим отладки замедляет работу системы и не годится для тестирования производительности. В режиме «вне РВ» система «Стенд ПНМ» способна работать лишь на одной инструментальной машине. При этом для взаимодействия нескольких участников моделирования, работающих на одной машине, используются механизмы, гораздо более эффективные, чем средства сетевого взаимодействия. Таким образом, сравнение производительности системы «CERTI» и «Стенда ПНМ» в режиме «вне РВ» нецелесообразно.

В режимах жёсткого и мягкого РВ система «Стенд ПНМ» может использовать несколько инструментальных машин, однако при этом производит синхронизацию логического времени программных моделей со временем астрономическим. На действия участников моделирования отводятся фиксированные промежутки физического времени. Если программный компонент модели отрабатывает за меньший срок, то его выполнение

искусственно задерживается. В тоже время оригинальная версия системы «CERTI» способна работать лишь в режиме AFAP (As Fast As Possible).

Для решения сложившейся проблемы исходный код системы «Стенд ПНМ» был модифицирован таким образом, чтобы исключить искусственные синхронизационные задержки при использовании нескольких инструментальных машин. В результате была получена версия системы «Стенд ПНМ», логика которой идентична системе «CERTI».

3.2.5 Результаты исследования

На данном этапе работы было проведено сравнение времени выполнения моделей различными версиями системы «CERTI» и системой «Стенд ПНМ» с отключёнными временными задержками. Время работы каждого участника моделирования измерялось с момента начальной синхронизации модели и до завершения его выполнения. Итоговое время выполнения модели считалось как среднее арифметическое от полученных значений. Так как основной интерес для исследования на данном этапе представляют собой оригинальная система CERTI и её многопоточная версия MT-CERTI, то эти системы участвовали в большем количестве экспериментов.

Эксперимент 1: единственный сервер

Тестовый стенд первого эксперимента состоял из единственной инструментальной машины (Intel Xeon 2,4GHz, 10Gb RAM), на которой были запущены все компоненты системы моделирования. Результаты измерения времени выполнения моделей (Таблица 9) показывают, что прирост производительности MT-CERTI по сравнению с оригинальной версией CERTI достигает 30%.

С увеличением числа передаваемых сообщений время выполнения модели «Пинг-Понг» растёт линейно, в то время как модель «Лавина» демонстрирует экспоненциальный рост. Такое поведение связано с используемым CERTI распределённым механизмом временной синхронизации участников моделирования. В модели «Лавина» логическое время федерата-терминала не зависит от логического времени других федератов, поэтому он может генерировать сообщения с произвольной скоростью. В данном случае скорость генерации превышает скорость обработки сообщений федератом-получателем. Поэтому инфраструктура RTI вынуждена буферизовать сообщения внутри себя. Вместе с ростом числа сообщений в буферах, скорость их обработки падает ещё больше, а размер буферов

растёт ещё быстрее. Этот замкнутый круг приводит к экспоненциальному росту времени выполнения модели.

Описанная проблема может быть решена как на уровне системы моделирования, так и на уровне имитационной модели. С одной стороны можно ограничить размер внутреннего буфера сообщений RTI и останавливать федерат-отправитель, если его сообщения заполнили отведённую квоту. С другой стороны внутрь федерата-отправителя можно встроить дополнительные временные задержки, искусственно ограничивающие скорость генерации сообщений. Практическая эффективность предложенных способов, однако, существенно зависит от конкретной имитационной задачи, и подбор наилучшего размера внутреннего буфера сообщений инфраструктуры RTI или оптимального времени задержки между передачей сообщений требует дополнительных исследований.

Таблица 9 - Зависимость времени выполнения моделей от числа передаваемых сообщений при использовании единственной инструментальной машины, мс

Число сообщений	Лавина		Пинг-Понг	
	CERTI	MT-CERTI	CERTI	MT-CERTI
10	3,8	2,4	6,8	4,6
100	35,4	19,4	73,1	45,6
1000	334	201,4	734,4	475,1
10000	3202,8	1888,9	7172,1	4728,6
100000	57335,9	50286,7	72134,8	49386,4

Эксперимент 2: кластер

Для проведения второго эксперимента использовался кластер из двух серверов (Intel Core2Duo 2,6GHz, 2Gb RAM), на каждом из которых выполнялся свой компонент имитационной модели. В случае систем «CERTI», на одной из них так же выполнялся и процесс RTIG. Результаты распределённого эксперимента (Таблица 10) показывают, что многопоточная MT-CERTI так же выигрывает до 30% у оригинальной версии системы, но в то же время проигрывает специализированной системе «Стенд ПНМ» примерно в 3 раза. Таким образом, система «Стенд ПНМ» способна выполнять значительно более сложные модели с меньшими директивными интервалами.

Таблица 10 - Зависимость времени выполнения моделей от числа передаваемых сообщений при использовании комплекса из двух инструментальных машин, мс

Число сообщений	Лавина			Пинг-Понг		
	CERTI	MT-CERTI	Стенд ПНМ	CERTI	MT-CERTI	Стенд ПНМ
10	4,1	2,8	1,6	10,2	6,3	2,3
100	38,1	26,1	7,6	94,4	65,2	22,8
1000	399,7	269	84,8	884,6	666,2	228
10000	6063	3015,2	1127,6	8770,7	6570,6	2280
100000	60601	30182,4	11722,1	87643,2	66524,8	22800

Эксперимент 3: Географическая удалённость

Аналогично предыдущему, данный эксперимент использовался комплекс из двух машин (AMD Opteron 2,4GHz, 12Gb RAM; Intel Core i7 1,6GHz, 4Gb RAM), но на этот раз они были соединены через сеть Интернет (среднее время ping-а 13,6 мс). Так как разница

между скоростями работы CERTI и MT-CERTI невелика по сравнению со временем передачи сообщения через сеть, то в данном эксперименте принимала участие только система CERTI. Более того, отсутствие какие-либо механизмов контроля качества на пути передачи данных приводит к существенному дрожанию результатов. Таблица 11 показывает минимальное, среднее и максимальное время выполнения моделей среди проведённых проб.

Сравнение приведённых результатов с экспериментом 2, в котором комплекс из двух машин был объединён локальной сетью, показывает, что динамика увеличения времени выполнения модели «Пинг-Понг» выше, чем аналогичный показатель модели «Лавина». Эта закономерность объясняется разным числом сетевых сообщений, которые передаются между компонентами моделей.

Таблица 11 - Зависимость времени выполнения моделей системой CERTI от числа передаваемых сообщений при использовании двух удалённых друг от друга машин, мс

Число сообщений	**Лавина**			**Пинг-Понг**		
	Минимум	**Среднее**	**Максимум**	**Минимум**	**Среднее**	**Максимум**
10	11,5	14,88889	18,5	146	161,6667	176
100	62,5	84,9	222	1328	1487,45	2395,5
1000	597	1752,15	5955,5	13443	14703,6	16833
10000	5951,5	9651,19	28786,5	137689,5	149603,2	165342

Эксперимент 4: одновременное выполнение

Стандарт моделирования HLA IEEE 1516 2000 предусматривает возможность одновременного проведения сразу нескольких экспериментов с использование одной и той же инфраструктуры RTI [4]. Поэтому в рамках данного эксперимента модели «Пинг-Понг» «Лавина» запускались как последовательно, так и параллельно на одной и той же инфраструктуре RTI. При этом использовался тот же аппаратный комплекс, что и во время описанного выше эксперимента 2.

Как видно из результатов тестирования (Таблица 12), падение скорости при одновременном запуске тестовых моделей значительно меньше времени их выполнения. Таким образом, одновременное проведение нескольких экспериментов с использованием единственного программно-аппаратного комплекса может быть целесообразным. Данный вопрос, однако, требует дальнейшего и более тщательного изучения.

Таблица 12 - Зависимость времени выполнения моделей от числа передаваемых сообщений при их последовательном и параллельном запуске, мс

Число сообщений	Лавина		Пинг-Понг	
	Последовательно	Параллельно	Последовательно	Параллельно
10	9	10,4	19,5	20,7
100	96,1	87,4	182,2	209,8
1000	889,3	1253,9	1794,3	2332,4

3.2.6 Выводы

Полученные результаты

Согласно результатам проведённого исследования, созданная на данном этапе работы многопоточная среда «MT-CERTI» обрабатывает события быстрее, чем оригинальная версия этой системы: средний прирост скорости составляет более 30%. При этом различие в производительности двух систем заметно как при использовании единственной инструментальной машины, так и комплекса из нескольких машин, объединённых локальной сетью. В то же время многопоточная среда выполнения «MT-CERTI», так же как и оригинальная «CERTI» отстаёт от системы «Стенд ПНМ».

Совместное моделирование географически удалённых участников, объединённых сетью Интернет, приводит к существенному дрожанию результатов эксперимента, что не позволяет сравнивать между собой оригинальную и модифицированную версию CERTI.

Одну и ту же инфраструктуру RTI может быть целесообразно использовать для одновременного проведения сразу нескольких имитационных экспериментов.

Дальнейшая деятельность

В дельнейшем необходимо провести измерения не только времени выполнения моделей, но и нагрузки, которую оказывает система моделирования на аппаратуру: расход оперативной памяти, затраченное процессорное время, нагрузка на сеть передачи данных. Некоторые из перечисленных характеристик не могут быть получены непосредственно из кода моделей, и потребуют модификации кода среды выполнения или использования внешних средств мониторинга программ.

В ходе проведённого исследования был выявлен экспоненциальный рост времени выполнения имитационной модели при перегрузке инфраструктуры RTI интенсивными потоками сообщений. Для борьбы с подобными перегрузками был предложено два

ортогональных механизма, каждый из которых, однако, нуждается в дополнительном исследовании.

Проведённое исследование практически не задаётся вопросом масштабируемости среды выполнения. Оно лишь поверхностно рассматривает возможность использования одновременного проведения нескольких имитационных экспериментов с помощью одной единственной инфраструктуры RTI. В дальнейшем необходимо детально исследовать динамику изменения производительности среды выполнения в зависимости от количества участников моделирования и числа используемых инструментальных машин.

3.3 Эксперименты с модифицированным транслятором UML в исполняемые модели, совместимые со стандартом HLA

В данном разделе приведены функциональные тесты, показывающие корректную работу генератора исходного кода моделей, совместимых с HLA, на основе UML диаграмм состояний. Модификация генератора исходного кода заключалась в добавлении возможности описания внутренней логики работы федерата (составной части распределённой модели) при помощи аппарата конечных автоматов. Подробное описание спецификации данного автомата можно изучить в разделе 1.3.

Далее будет приведен эксперимент генерации простой модели на основании UML диаграмм состояний. Эксперимент будет описываться поэтапно в соответствии со всеми стадиями генерации исходного кода модели. Все этапы генерации исхожного кода показывает Рисунок 56.

Рисунок 56 - Полная схема генерации исходного кода модели.

3.3.1 Диаграмма состояний в ArgoUML

Для создания UML диаграмм состояний нами был использован редактор UML ArgoUML. Обоснованность выбора данного редактора описаны в обзоре UML редакторов, с которым можно ознакомиться в отчете по третьему этапу данной работы [3]. Стоит отметить, что в общем случае подойдет любой редактор UML с возможности получать на выходе XMI представление UML диаграмм.

В качестве эксперимента была выбрана задача написания простейшего клиент-серверного приложения: Пусть есть два федерата (первый – отправитель, второй - получатель); отправитель с заданной периодичностью шлет сообщения, содержащие одно целое число, получатель получает данное число. При этом обмен между федерата происходит при помощи интерфейсов HLA RTI, а логики работы каждого федерата описывается при помощи конечного автомата.

UML диаграмму состояний, описывающую данную федерацию, показывает Рисунок 57.

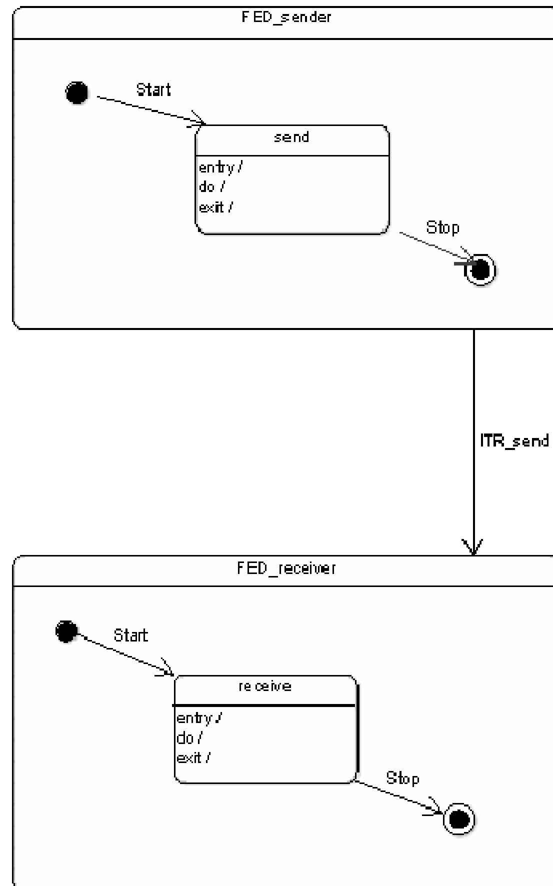

Рисунок 57 - UML диаграмма состояний экспериментальной модели.

3.3.2 XMI представление диаграммы состояний

После создания UML диаграммы состояний представляющую модель, необходимо сохранить данную модели в специализированом XML представлении - XMI. Данная функциональность является стандартной для использованного нами редактора UML. Пример XMI представления приведен на рисунке 58.

```
1 <?xml version = '1.0' encoding = 'UTF-8' ?>
2 <XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Thu Apr 28 13:55:44 YEKST 2011'>
3   <XMI.header>    <XMI.documentation>
4       <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
5       <XMI.exporterVersion>0.32.2(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan 2010) $ </XMI.exporterVersion>
6     </XMI.documentation>
7     <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
8   <XMI.content>
9     <UML:Model xmi.id = '-87--2-66-26-3a16eb0e:12ded3f9afb:-8000:0000000000000961'
10      name = 'BCS' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
11      isAbstract = 'false'>
12      <UML:Namespace.ownedElement>
13        <UML:Class xmi.id = '-87--2--3-111--79bec435:12dfc06f9f9:-8000:00000000000000AE3'
14          name = 'C_1' visibility = 'public' isSpecification = 'false' isRoot = 'false'
15          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
16          <UML:Namespace.ownedElement>
17            <UML:SignalEvent xmi.id = '-87--2-66-26-1cf3bef8:12e00a917b7:-8000:0000000000000D37'
18              name = 'LEAVE' isSpecification = 'false'/>
19            <UML:SignalEvent xmi.id = '-87--2-66-26-513d4ebd:12e000b3ec8:-8000:0000000000000C16'
20              name = 'SENSOR_STATUS_WORD' isSpecification = 'false'/>
21            <UML:SignalEvent xmi.id = '-87--2-66-26-7161222:12e001d3885:-8000:0000000000000C1C'
22              name = 'C2_STATUS_WORD' isSpecification = 'false'/>
23            <UML:SignalEvent xmi.id = '-87--2-66-26-7161222:12e001d3885:-8000:0000000000000C2A'
24              name = 'C3_STATUS_WORD' isSpecification = 'false'/>
25            <UML:SignalEvent xmi.id = '-87--2-66-26-7161222:12e001d3885:-8000:0000000000000C39'
26              name = 'C4_STATUS_WORD' isSpecification = 'false'/>
27            <UML:SignalEvent xmi.id = '10-2-0-116--5449f0e6:12dfd0cbc77:-8000:0000000000008B01'
28              name = 'T1' isSpecification = 'false'/>
29            <UML:SignalEvent xmi.id = '10-2-0-116--5449f0e6:12dfd0cbc77:-8000:0000000000008B02'
30              name = 'T2' isSpecification = 'false'/>
31            <UML:SignalEvent xmi.id = '-87--2-66-26-798a98ac:12dedfaf9bf:-8000:0000000000000B1D'
32              name = 'ACT' isSpecification = 'false'/>
33            <UML:SignalEvent xmi.id = '-87--2-66-26-798a98ac:12dedfaf9bf:-8000:0000000000000B1F'
34              name = 'DEACT' isSpecification = 'false'/>
35            <UML:StateMachine xmi.id = '-87--2-66-26-3a16eb0e:12ded3f9afb:-8000:0000000000000962'
36              name = 'C_1' isSpecification = 'false'>
37              <UML:StateMachine.context>
38                <UML:Class xmi.idref = '-87--2--3-111--79bec435:12dfc06f9f9:-8000:00000000000000AE3'/>
39              </UML:StateMachine.context>
40              <UML:StateMachine.top>
41                <UML:CompositeState xmi.id = '-87--2-66-26-3a16eb0e:12ded3f9afb:-8000:0000000000000963'
42                  name = 'top' isSpecification = 'false' isConcurrent = 'false'>
```

Рисунок 58 - XMI представление экспериментальной модели.

Однако данный формат не является удобным для понимания и представления конечного автомата для описания внутренней логики. В связи с этим был выбран специализированный XML формат описания диаграмм состояний – SCXML. А так же были разработаны и реализованы трансляторы их XMI в SCXML и наоборот.

3.3.3 SCXML представление диаграммы состояний

Как было сказано выше данный формат является наиболее удобным для описание диаграмм состояний и конечных автоматов, так как в своей структуре оперируется необходимыми параметрами и примитивами для описания данных структур. Пример SCXML представления экспериментальной модели показывает Рисунок 59.

```
<?xml·version="1.0"·encoding="UTF-8"?>¶
<!--·car_alarm.xml·-·state·chart·XML·file·-->¶
<53·initialstate="Federation">¶
··<state·id="FED_sender">¶
····<transition·event="sender">¶
······<target·next="receiver"/>¶
······<parametr·name=·"par1"></parametr>¶
····</transition>¶
····<parametr·name=·"par1"·type="int"·initval=".0"></parametr>¶
····<attribute·name="regulating"></attribute>¶
··</state>¶
···<state·id="FED_receiver">¶
····<attribute·name="constrained"></attribute>¶
··</state>¶
·</53>¶
```

Рисунок 59 - SCXML представление экспериментальной модели.

3.3.4 Cheetah шаблоны для генерации исходных кодов модели

Полученное на предыдущем этапе SCXML представление модели подается в ход генератору исходных кодов на основе библиотеки шаблонов cheetah. Подробное описание работы с шаблонами cheetah описанной в отчетах за предыдущие этапы проекта [2],[3]. В данном подразделе будет описана только модификация генератора для создания исходного кода внутренней логики работы генератора на основе конечных автоматов.

Для описания внутренней логики работы каждому федерату в федерации ставится в соответствие автомат (данный автомат разрабатывается на этапе создания UML представления модели). Для генерации исходного кода по данному автомату для генератора был создан специализированный cheetah шаблон. Подробный разбор данного шаблона представлено далее.

```
int  ${state.id}_StateChart::run(void)
{
  InitializeSM();
  /* main cycle */
  while (1)
  {
    (this->*state_table[curr_state])();
    //DecrementTimer();
    /* insert you code - if needed */
  }
  return 0;
};
int  ${state.id}_StateChart::make_step(void)
{
  //state_table[curr_state]();
  (this->*state_table[curr_state])();
  //DecrementTimer();
  /* insert you code - if needed */

  return 0;
};
```

Рисунок 60 - Функции запуска автомата внутренней логики работы федерата.

На рисунке 60 представлены описание двух базовых функции по управлению конечным автоматом внутренней логики. Функция run() предназначена для автономной работы с автоматов (для тестовых запусков федерата вне федерации), функция make_step() предназначена для запуска внутри федерации. Во втором случае бесконечный главный цикл модели описывается в шаблоне, отвечающим за генерацию кода федерата внутри федерации.

```
void·${state.id}_StateChart::InitializeSM(void)¶
{¶
··curr_state·=·(State_Type)0;¶
··#if·$state.__dict__.has_key('parametr')·:¶
······#for·$parametr·in·$state.parametr·:¶
··${parametr.name}·=·${parametr.initval};¶
······#end·for¶
··#end·if¶
··#if·$state.__dict__.has_key('var')·:¶
```

Рисунок 61 - Функции инициализации автомата внутренней логики работы федерата.

На рисунке 61 представлено описание функции инициализации автомата внутренней логики. Устанавливаются значения всех параметров федерата, и текущие состояние автомата федерата указывает на начальное состояние.

```
void ${headState.id}_StateChart::${state.id}_func(void)
{
        #if $state.__dict__.has_key('transition') :
        #for $transition in $state.transition:
           #if $transition.__dict__.has_key('parametr')==0 :
              #if $transition.__dict__.has_key('cond') :
   if (${transition.cond})
   {
                 #if $transition.__dict__.has_key('event') :
      ${transition.event};
                 #end if
      curr_state = ${transition.target.upper()};
   }
              #end if
           #end if
        #end for
        #end if
}
```

Рисунок 62 - Функции состояний автомата внутренней логики работы федерата.

На рисунке 62 представлено описание шаблона функции для каждого состояния автомата внутренней логики. При вызове функции, отвечающей за конкретное состояний, происходит проверка возможность перехода в другое состояние (соблюдение сторожевых условий перехода) и изменения значений параметров, связанных с данным переходом. В конце каждой такой функции обязательно смена значения указателя на текущее состояние.

3.3.5 Генерация исходного кода

На выходе генератор исходного кода по шаблонам предоставляет исходные коды на языке C++ описанный на UML (на первой стадии генерации) распределенной модели. В данных исходниках полностью отображена функциональность внутренней логики (описанной при помощи конечного автомата), и прописаны интерфейсы для взаимодействия федератов внутри федерации через HLA RTI. Пример исходных кодов представлен на рисунке 63.

```
 2 #ifndef SM_STATE_FED_ARMED_H
 3 #define SM_STATE_FED_ARMED_H
 4
 5 #include "sm_state.h"
 6 #include "federate.h"
 7
 8 class SM_State_FED_Armed : public SM_State, Federate
 9 {
10 public:
11
12   //--------------------------------------------------------------------
13   SM_State_FED_Armed (std::wstring _federationName, std::wstring _fddName);
14
15   //--------------------------------------------------------------------
16   ~SM_State_FED_Armed ();
17
18   //--------------------------------------------------------------------
19
20
21   //--------------------------------------------------------------------
22
23   void onentry ();
24
25 protected:
26
27 private:
28   //--------------------------------------------------------------------
29          void LockDoors();
30
31   //--------------------------------------------------------------------
32   // copy constructor not implemented
33   SM_State_FED_Armed( const SM_State_FED_Armed& );
34
35   // assignment operator not implemented
36   SM_State_FED_Armed& operator=( const SM_State_FED_Armed& );
37
38
39   //--------------------------------------------------------------------
40
41   void getHandles();
42   void customInit();
43   void loop();
44
45
46   //--------------------------------------------------------------------
47 }; // SM_State_FED_Armed
48
```

```
12
13 //--------------------------------------------------------------------
14
15 #incluce "sm_state_FED_Armed.h"     // class SM_Stat_FED_Armed
16
17
18
19 #incluce <iostream>
20
21
22
23 //--------------------------------------------------------------------
24 //
25 // Class Name : SM_State_FED_Armed
26 //
27 //--------------------------------------------------------------------
28
29 SM_State_FED_Armed::SM_State_FED_Armed ()
30 {}
31
32 //--------------------------------------------------------------------
33 SM_State_FED_Armed::~SM_State_FED_Armed ()
34 {}
35
36 //--------------------------------------------------------------------
37 void SM_State_FED_Armed::onentry ()
38 {
39   LockDoors();
40 }
41
42 //--------------------------------------------------------------------
43 void SM_State_FED_Armed::LockDoors ()
44 {
45
46   std::cout << "SM_State_FED_Armed::LockDoors\n";
47
48
49 }
50
51 //--------------------------------------------------------------------
52
```

Рисунок 63 - Пример исходных кодов на С++ полученных на выходе генератора на основе UML представления модели.

3.3.6 Выводы

В ходе проведения были получены корректные исходные кода на языке С++ для работы в среде HLA RTI. Описание модели (создателем модели) в терминах UML существенной уменьшает времени разработку модели.

Подбор удобных форматов для представления данных на промежуточных этапа генерации кода делает удобным тестирование разработанного средства. А также способствует удобной ручной коррекции данных на различных этапах генерации исходного кода.

Наблюдаемые результаты соответствуют ожидаемым, следовательно, функциональное тестирование проведено успешно.

3.4 Эксперименты по восстановлению параметров модели по контрпримеру в UPPAAL.

Ниже приведены несколько трасс, построенных на созданных ранее моделях. На них можно видеть формат трассы UML, получаемой в результате построения, описанного в разделе 1.6.

Каждый шаг обозначается строкой со словом Step и порядковым номером. Далее указаны переходы, сделанные на данном шаге, с указанием, в каком автомате это произошло. После идет значения переменных и таймеров. Для удобства указаны значения только тех переменных и таймеров, которые изменились по сравнению с предыдущим шагом.

Светофор

Первый пример, на котором рассмотрено преобразование трасс UPPAAL в трассы UML – система управления уличным движением на перекрестке, подробно описанная в предыдущих отчетах. Эта система призвана осуществлять правильное управление сигналами двух светофоров на перекрестке проспекта и улицы. Светофоры работают под управлением контроллера. В обычных условиях светофоры согласованно изменяют свет так, чтобы периодически открывать движение по обеим магистралям. Но контроллер также способен обнаруживать приближение к перекрестку автомобиля скорой помощи. В этом случае контроллер переходит в особый режим работы, чтобы обеспечить как можно более быстрое, но вместе с тем безопасное, пересечение перекрестка автомобилем скорой помощи.

На рисунках ниже приведена UML-диаграмма системы.

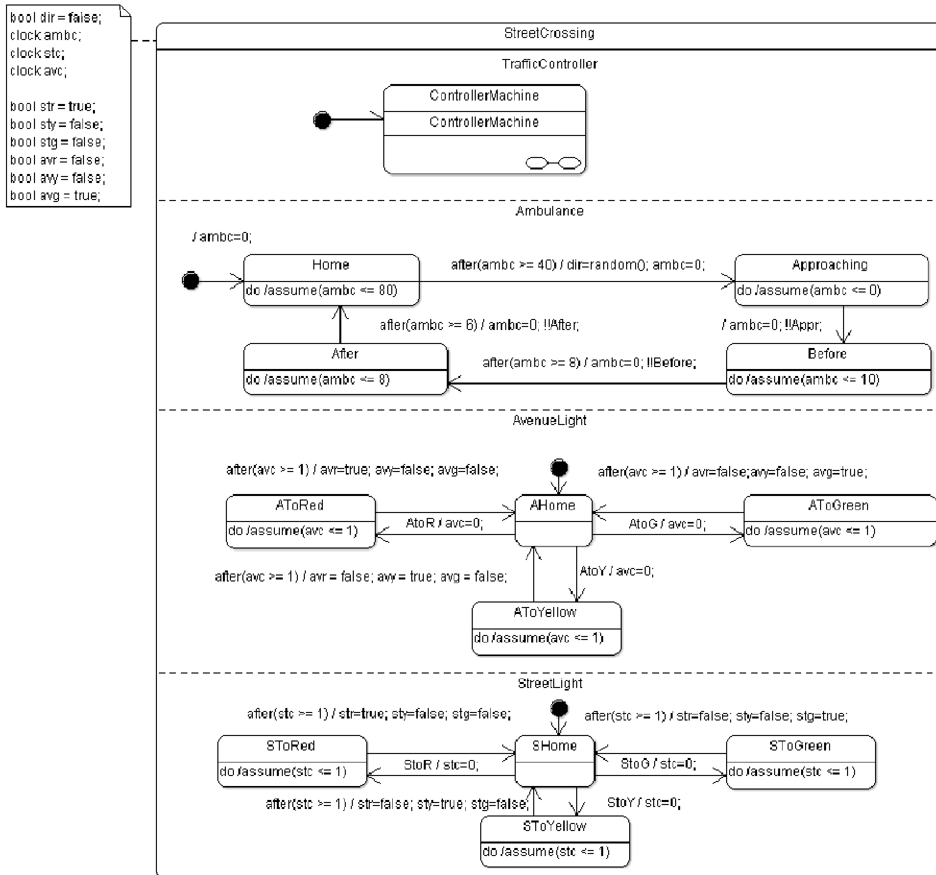

Рисунок 64 - UML-диаграмма системы (1)

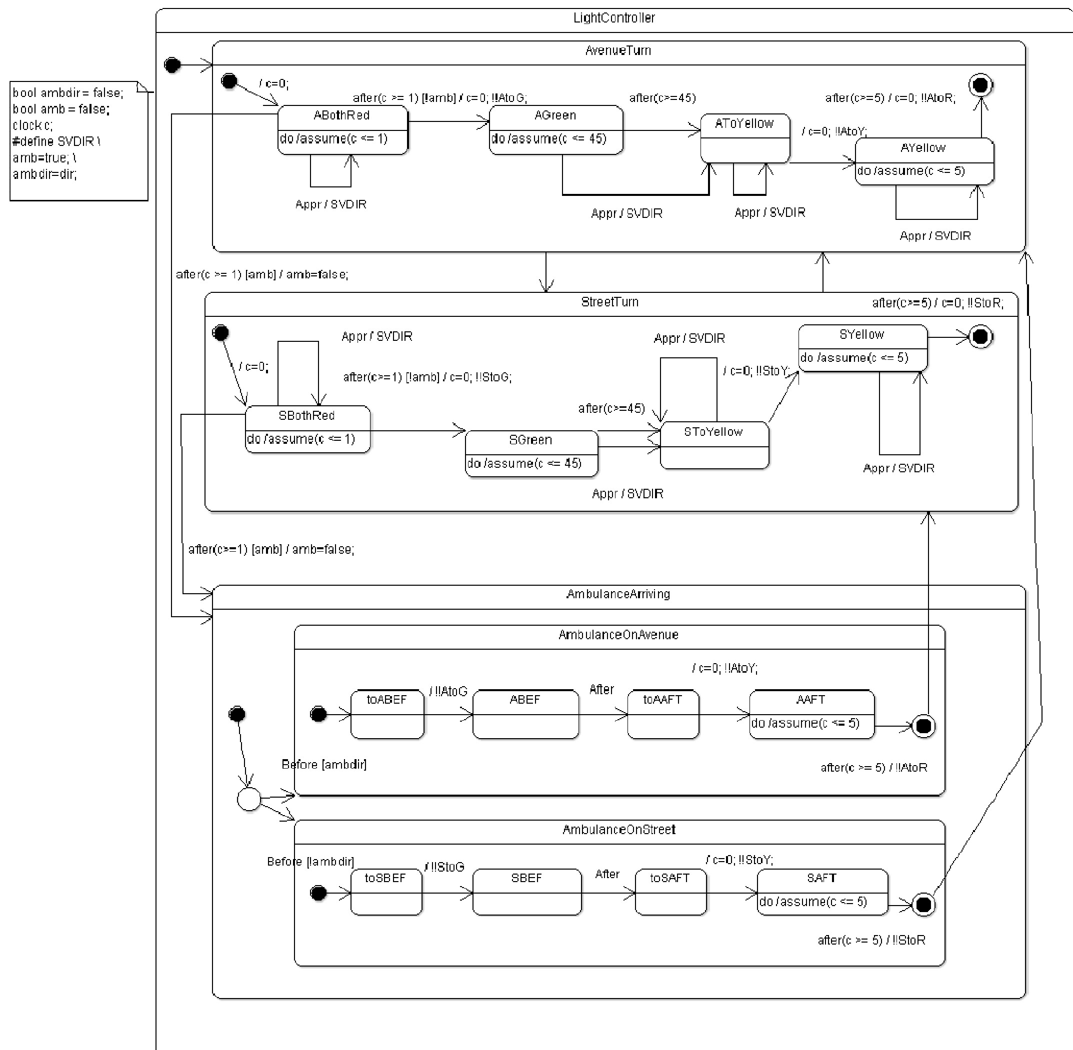

Рисунок 65 - UML-диаграмма системы (2)

Приведенная ниже трасса описывает три проезда скорой помощи через перекресток при разных комбинациях света на светофорах.

```
Step 1
TrafficController_process :: AvenueTurn.ABothRed --> AvenueTurn.AGreen
Step 2
AvenueLight_process :: AvenueLight.AHome --> AvenueLight.AToGreen
c >= 0
c <= 1
stc >= 1
ambc >= 1
avc >= 0
avc <= 1
ambc <= 2
stc <= 2
Step 3
```

```
AvenueLight_process :: AvenueLight.AToGreen --> AvenueLight.AHome
c >= 1
c <= 45
stc >= 2
ambc >= 2
avc >= 1
avc <= 45
ambc <= 46
stc <= 46
Step 4
Ambulance_process :: Ambulance.Home --> Ambulance.Approaching
dir = 1
c >= 39
ambc >= 0
ambc <= 0
stc >= 40
avc >= 39
Step 5
TrafficController_process :: AvenueTurn.AGreen --> AvenueTurn.AToYellow
Step 6
Ambulance_process :: Ambulance.Approaching --> Ambulance.Before
ambdir = 1
amb = 1
ambc <= 10
c <= 55
avc <= 55
stc <= 56
Step 7
Ambulance_process :: Ambulance.Before --> Ambulance.After
ambc <= 8
c <= 63
c >= 47
avc <= 63
stc >= 48
stc <= 64
avc >= 47
Step 8
Ambulance_process :: Ambulance.After --> Ambulance.Home
ambc <= 80
c <= 143
c >= 53
avc <= 143
stc >= 54
stc <= 144
avc >= 53
Step 9
Ambulance_process :: Ambulance.Home --> Ambulance.Approaching
dir = 0
c >= 93
ambc <= 0
stc >= 94
avc >= 93
Step 10
Ambulance_process :: Ambulance.Approaching --> Ambulance.Before
ambdir = 0
ambc <= 10
c <= 153
avc <= 153
stc <= 154
Step 11
TrafficController_process :: AvenueTurn.AToYellow --> AvenueTurn.AYellow
Step 12
AvenueLight_process :: AvenueLight.AHome --> AvenueLight.AToYellow
c >= 0
c <= 1
```

```
avc >= 0
avc <= 1
Step 13
Ambulance_process :: Ambulance.Before --> Ambulance.After
ambc <= 1
stc >= 102
stc <= 155
Step 14
TrafficController_process :: AvenueTurn.AYellow --> StreetTurn.SBothRed
Step 15
TrafficController_process :: StreetTurn.SBothRed --> AmbulanceArriving.UNNAMED7
amb = 0
c >= 1
avc >= 1
Step 16
AvenueLight_process :: AvenueLight.AToYellow --> AvenueLight.AHome
avy = 1
avg = 0
ambc <= 8
c <= 9
stc <= 162
avc <= 9
Step 17
Ambulance_process :: Ambulance.After --> Ambulance.Home
ambc <= 80
c <= 89
stc <= 243
stc <= 242
c >= 6
stc >= 108
avc <= 89
avc >= 6
Step 18
Ambulance_process :: Ambulance.Home --> Ambulance.Approaching
c >= 46
stc >= 148
ambc <= 0
avc >= 46
Step 19
Ambulance_process :: Ambulance.Approaching --> Ambulance.Before
ambc <= 10
c <= 99
stc <= 253
stc <= 252
avc <= 99
Step 20
TrafficController_process :: AmbulanceArriving.UNNAMED7 --> AmbulanceOnStreet.toSBEF
Step 21
Ambulance_process :: Ambulance.Before --> Ambulance.After
ambc <= 8
c <= 107
stc <= 261
stc <= 260
c >= 54
stc >= 156
avc <= 107
avc >= 54
Step 22
TrafficController_process :: AmbulanceOnStreet.toSBEF --> AmbulanceOnStreet.SBEF
Step 23
StreetLight_process :: StreetLight.SHome --> StreetLight.SToGreen
stc >= 0
stc <= 1
Step 24
StreetLight_process :: StreetLight.SToGreen --> StreetLight.SHome
```

```
stg = 1
str = 0
stc >= 1
ambc >= 1
stc <= 8
c >= 55
avc >= 55
Step 25
TrafficController_process :: AmbulanceOnStreet.SBEF --> AmbulanceOnStreet.toSAFT
Step 26
Ambulance_process :: Ambulance.After --> Ambulance.Home
ambc >= 0
ambc <= 80
c <= 187
stc <= 133
stc <= 88
c >= 60
avc <= 187
avc >= 60
```

DrTesy

Модель системы DrTesy, разработанная на предыдущем этапе данной работы [3], состоит из четырех вычислителей C_1, C_2, C_3 и C_4, подключенных к общей шине. Процессоры C_2 и C_3 имеют общую память. Каждый из вычислителей выполняет свой круг задач. Вычислитель C_1 – главный процессор, управляющий вычислениями и передачей данных во всей системе. Процессор C_2 вычисляет параметры полета и готовит данные для датчиков. Процессор C_3 определяет положение самолета по показаниям датчиков и обеспечивает перемещение по требуемому маршруту, а также управляет сенсорами. Процессор C_4 обрабатывает информацию с радара и управляет полетом самолета на малой высоте.

На приведенной ниже трассе показана инициализация системы и несколько первых шагов, в том числе вход в критическую секцию для работы с общей памятью.

```
Step 1
region2_process :: SECTION1_from_SECTION1.SB100 --> SECTION1_from_SECTION1.SB101
PAUSE_from_SECTION4_timeout >= 0
CHECK_C3_from_SB104A_timeout >= 0
CHECK_C4_from_SB104A_timeout >= 0
UNNAMED26_timeout >= 0
S3_timeout >= 0
S1_from_sb109ref_timeout >= 0
S1_from_SB106A_timeout >= 0
S1_from_sb109ref_from_SECTION5_timeout >= 0
S1_from_sb107_timeout >= 0
CHECK_C4_timeout >= 0
S3_from_SECTION4_timeout >= 0
COLLECT_SENSOR_timeout >= 0
TIMER_timeout >= 0
T1_timeout >= 0
CHECK_S_from_SB104A_timeout >= 0
CHECK_C2_from_SB104A_timeout >= 0
```

```
CHECK_S_timeout >= 0
S1_timeout >= 0
PAUSE_from_SECTION3_timeout >= 0
S1_from_SECTION5_timeout >= 0
S1_from_SB109_timeout >= 0
S1_from_SECTION3_timeout >= 0
S1_from_SB107C_timeout >= 0
CHECK_C2_timeout >= 0
PAUSE_timeout >= 0
REPLY_timeout >= 0
S3_from_SECTION3_timeout >= 0
CHECK_C3_timeout >= 0
WAIT_from_SECTION2_timeout >= 0
S1_from_SECTION4_timeout >= 0
T2_timeout >= 0
Step 2
C2_region1_process :: C2_region1.Compute --> C2_region1.Lock
Step 3
C3_region3_process :: RW3_from_AM21.WAIT --> RW3_from_AM21.LOCKING
Step 4
C3_region4_process :: C3_region4.Compute --> C3_region4.LOCK
critical_C3 = 1
sharedMemory = 1
Step 5
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
PAUSE_from_SECTION4_timeout >= 10
CHECK_C3_from_SB104A_timeout >= 10
CHECK_C4_from_SB104A_timeout >= 10
UNNAMED26_timeout >= 10
S3_timeout >= 10
S1_from_sb109ref_timeout >= 10
S1_from_SB106A_timeout >= 10
S1_from_sb109ref_from_SECTION5_timeout >= 10
S1_from_sb107_timeout >= 10
CHECK_C4_timeout >= 10
S3_from_SECTION4_timeout >= 10
COLLECT_SENSOR_timeout >= 10
TIMER_timeout >= 10
T1_timeout >= 10
CHECK_S_from_SB104A_timeout >= 10
CHECK_C2_from_SB104A_timeout >= 10
CHECK_S_timeout >= 10
S1_timeout >= 10
PAUSE_from_SECTION3_timeout >= 10
S1_from_SECTION5_timeout >= 10
S1_from_SB109_timeout >= 10
S1_from_SECTION3_timeout >= 10
S1_from_SB107C_timeout >= 10
CHECK_C2_timeout >= 10
PAUSE_timeout >= 10
REPLY_timeout >= 10
S3_from_SECTION3_timeout >= 10
CHECK_C3_timeout >= 10
WAIT_from_SECTION2_timeout >= 10
S1_from_SECTION4_timeout >= 10
T2_timeout >= 10
Step 6
C3_region3_process :: RW3_from_AM21.LOCKING --> RW3_from_AM21.EXIT
critical_C3 = 0
Step 7
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 8
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 9
region1_process :: region1.CP --> TIMER_INTER_from_TIMER_INTER.UNNAMED19
```

```
Step 10
region2_process :: SECTION1_from_SECTION1.SB101 --> SECTION1_from_SECTION1.SB25
Step 11
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 12
region2_process :: SECTION1_from_SECTION1.SB25 --> SECTION1_from_SECTION1.UNNAMED21
Step 13
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 14
C2_region1_process :: C2_region1.Lock --> C2_region1.Compute
Step 15
C3_region3_process :: RW3_from_AM21.EXIT --> STS17.SB8
Step 16
C3_region4_process :: C3_region4.LOCK --> C3_region4.Compute
in_C_3_Compute = 1
in_C_2_Compute = 1
sharedMemory = 0
Step 17
region2_process :: SECTION1_from_SECTION1.UNNAMED21 --> SECTION1_from_SECTION1.SB102
Step 18
C2_region2_process :: C2_region2.WAIT --> RW2_top.WAIT
REQUEST_C2 = 1
Step 19
C3_region3_process :: STS17.SB8 --> STS17.UNNAMED133
Step 20
C2_region1_process :: C2_region1.Compute --> C2_region1.Lock
Step 21
C2_region2_process :: RW2_top.WAIT --> RW2_top.LOCKING
Step 22
C3_region4_process :: C3_region4.Compute --> C3_region4.LOCK
in_C_3_Compute = 0
critical_C2 = 1
in_C_2_Compute = 0
sharedMemory = 1
Step 23
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 24
C2_region2_process :: RW2_top.LOCKING --> RW2_top.EXIT
critical_C2 = 0
REQUEST_C2 = 0
Step 25
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 26
region2_process :: SECTION1_from_SECTION1.SB102 --> SECTION1_from_SECTION1.SB13
Step 27
C3_region3_process :: STS17.UNNAMED133 --> RW3_from_AM21.WAIT
Step 28
C2_region1_process :: C2_region1.Lock --> C2_region1.Compute
Step 29
C2_region2_process :: RW2_top.EXIT --> C2_region2.SB23
Step 30
C3_region4_process :: C3_region4.LOCK --> C3_region4.Compute
in_C_3_Compute = 1
in_C_2_Compute = 1
sharedMemory = 0
Step 31
region2_process :: SECTION1_from_SECTION1.SB13 --> SECTION1_from_SECTION1.SB103
Step 32
C2_region2_process :: C2_region2.SB23 --> RW2_top_from_RW3.WAIT
REQUEST_C2 = 1
Step 33
C3_region4_process :: C3_region4.Compute --> SMART_CONTROL_C3_from_SMART_CONTROL_C3.UNNAMED145
in_C_3_Compute = 0
Step 34
C2_region1_process :: C2_region1.Compute --> SMART_CONTROL_C2.UNNAMED79
```

```
in_C_2_Compute = 0
Step 35
C2_region2_process :: RW2_top_from_RW3.WAIT --> RW2_top_from_RW3.LOCKING
critical_C2 = 1
sharedMemory = 1
Step 36
C2_region2_process :: RW2_top_from_RW3.LOCKING --> RW2_top_from_RW3.EXIT
critical_C2 = 0
REQUEST_C2 = 0
Step 37
C2_region2_process :: RW2_top_from_RW3.EXIT --> C2_region2.SB24
Step 38
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 39
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 40
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 41
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 42
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
Step 43
region2_process :: SECTION1_from_SECTION1.SB103 --> SECTION1_from_SECTION1.UNNAMED24
Step 44
C3_region1_process :: C3_region1.TIMER --> C3_region1.TIMER_2
Step 45
region2_process :: SECTION1_from_SECTION1.UNNAMED24 --> SB104.WAIT
Step 46
C3_region1_process :: C3_region1.TIMER_2 --> C3_region1.TIMER
```

3.5 Эксперименты со средствами оценки наихудшего времени выполнения

В данном разделе проведены результаты экспериментальных исследований, посвященных оценке наихудшего времени выполнения (WCET). Целью проведенных экспериментов является проверка точности работы анализатора WCET. Проверялась точность как оценки максимального времени выполнения заданных линейных участков, так и оценка наихудшего времени выполнения некоторых задач, взятых из наборов тестов, подготовленных специально для проверки инструментов оценки наихудшего времени выполнения.

3.5.1 Методика исследования

Для проверки точности исследования проводились в два этапа:

1. Запуск тестов на целевом вычислителе.

   Проводилась серия запусков подготовленных тестовых наборов на целевом вычислителе. Каждый тест прогонялся на вычислителе несколько раз, на каждом

запуске вычислялось время выполнения теста в тактах работы процессора. Среди полученных результатов выбирался запуск с наибольшим временем работы.

2. Сравнение результатов запуска с результатами, выданными анализатором наихудшего времени выполнения.

   Основным требованием на данном этапе является то, что оценка, выданная анализатором, не должна быть ниже оценки, полученной при прогоне на вычислителях. Если хоть одна полученная оценка не удовлетворяет данному требованию, то анализатор в общем случае нельзя считать применимым для оценки наихудшего времени выполнения. Дополнительно проводилось сравнение точности показаний оценки анализатора и максимального времени прогона.

3.5.2 Исследуемые задачи

Для проведения экспериментов использовалось два вида тестов:

1. Программы, состоящие из одного линейного участка.

   Данные тесты использовались для проверки точности оценки максимального времени выполнения линейных участков. Выполнялся прогон на тестах с постепенным увеличением размеров линейного участка.

2. Шаблонные задачи для оценки наихудшего времени выполнения.

   Данные тесты были взяты из специально подготовленного набора для инструментов оценки наихудшего времени выполнения. Данные тесты расположены по URL http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. Среди тестов выбраны те, которые удовлетворяют функциональности анализатора, т.е. не содержат операций с числами с плавающей точкой. Некоторые тесты не получилось запустить на целевой архитектуре, поэтому эксперименты проведены над следующими задачами: adpcm, bs, fac, fibcall, insertsort, janne_complex, jfdctint.

3.5.3 Целевой вычислитель

При выполнении этапа прогона тестов на целевом вычислителе был выбран процессор avr с восьмибитной архитектурой. Описание процессора можно прочитать в работе [58]. В микроконтроллерах AVR обычно используется двухуровневый конвейер, отсутствует кэш, что приводит к более точным результатам тестирования. Для получения исполняемого файла использовался кросс-компилятор avr-gcc, генерировался код без оптимизации.

При анализе оценки наихудшего времени выполнения использовалась готовая модель конвейера и памяти вычислителя avr8bit. Некоторые из тестов, содержащие большое пространство возможных состояний вычисления, при анализе аварийно завершились с ошибкой нехватки памяти (использовался вычислитель с операционной системой с 32битной архитектурой и оперативной памятью 2 гигабайта).

3.5.4 Результаты тестирования

Результаты тестирования оценки времени выполнения линейных участков приведены в Таблица 13. Все полученные анализатором оценки превышают реально полученные времена выполнения тестов, тем самым удовлетворяют поставленному требованию. При этом точность получаемой оценки падает при увеличении размера линейного участка (в общем случае разность между оценками не превышает 40%).

Результаты тестирования оценки максимального времени выполнения шаблонных задач приведены в Таблица 13. Некоторые тесты аварийно завершились при работе анализатором WCET (такие тесты обозначены знаком «-»). Данные тесты характеризуются наличием нескольких циклов (до 10) с максимальным количеством итераций до 30. При данных характеристиках возникает слишком большое пространство состояний, не помещающееся в памяти (при использовании 32-битной архитектуры).

Таблица 13 - Результаты тестирования времени выполнения линейных участков

Номер теста	Наихудшее время прогона на целевом вычислителе (в тактах работы процессора)	Наихудшее время, полученное анализатором WCET (в тактах работы процессора)	Размер линейного участка (строк кода)
1.	13	13	3
2.	54	55	6
3.	106	144	11
4.	178	255	17
5.	326	470	34
6.	622	980	60

Таблица 14 - Результаты тестирования времени выполнения шаблонных задач

Наименование теста	Наихудшее время прогона на целевом вычислителе (в тактах работы процессора)	Наихудшее время, полученное анализатором WCET (в тактах работы процессора)
adpcm	52531	-
bs	155	166
fac	828	906
fibcall	1806	2434
insertsort	1718	2872
janne_complex	369	-
jfdctint	47661	54383

3.5.5 Выводы

В результате проведенных экспериментальных исследований выяснено, что анализатор наихудшего времени выполнения выдает оценку с приемлемой точностью и удовлетворяет требованию превышения оценки WCET реального времени выполнения. При этом анализатор можно использовать как для оценки времени выполнения линейных участков, так и для оценки некоторых небольших задач (пространство состояний которых позволяет анализатору успешно посчитать оценку).

3.6 Эксперименты со средствами оптимизации надежности РВС РВ

В данном разделе приведено описание и результаты апробации схемы интеграции средств синтеза архитектур (построения расписаний) со средой имитационного моделирования. Апробация проводилась на примере средства выбора МОО РВС РВ (раздел 1.9).

3.6.1 Представление конфигурации РВС РВ в виде расписания общего вида.

Для применения описанной в разделе 2.2 схемы интеграции средств синтеза архитектур и построения расписаний и среды моделирования к средству сбалансированного

выбора МОО РВС РВ (раздел 1.9) необходимо осуществить переход от внутреннего представления конфигурации РВС РВ к расписанию общего вида.

Покажем, что каждой конфигурации РВС РВ, определенной в разделе 1.9.2 можно однозначно поставить в соответствие расписание, определенное в разделе 2.2.1 как множество S троек *(v,m,n)*, где *v* – задание программы, *m* – процессор, на котором задание выполняется, *n* – порядковый номер задания на процессоре. Каждой конфигурации $\{H_i^{F_i}, S_i^{F_i}, F_i\}$ модуля U_i ставятся в соответствие элементы расписания S следующим образом, в зависимости от F_i:

1. F_i=*None* $\Rightarrow$ $H_i^{F_i} = \{H_i^{F_i}(1)\}$, $S_i^{F_i} = \{S_i^{F_i}(1)\}$. Данной конфигурации соответствует элемент расписания $s_{i1} = (S_i^{F_i}(1), H_i^{F_i}(1), 1)$

2. F_i=*NVP/0/1* $\Rightarrow$ $H_i^{F_i} = \{H_i^{F_i}(1)\}$, $S_i^{F_i} = \{S_i^{F_i}(1), S_i^{F_i}(2), S_i^{F_i}(3)\}$. Данной конфигурации соответствуют элементы расписания $s_{i1} = (receiver_i, H_i^{F_i}(1), 1)$, $s_{i2} = (S_i^{F_i}(1), H_i^{F_i}(1), 2)$, $s_{i3} = (S_i^{F_i}(2), H_i^{F_i}(1), 3)$, $s_{i4} = (S_i^{F_i}(3), H_i^{F_i}(1), 4)$, $s_{i5} = (voter_i, H_i^{F_i}(1), 5)$.

3. F_i=*NVP/1/1* $\Rightarrow$ $H_i^{F_i} = \{H_i^{F_i}(1), H_i^{F_i}(2), H_i^{F_i}(3)\}$, $S_i^{F_i} = \{S_i^{F_i}(1), S_i^{F_i}(2), S_i^{F_i}(3)\}$. Данной конфигурации соответствуют элементы расписания $s_{i1} = (receiver_i, H_i^{F_i}(1), 1)$, $s_{i2} = (S_i^{F_i}(1), H_i^{F_i}(1), 2)$, $s_{i3} = (S_i^{F_i}(2), H_i^{F_i}(2), 1)$, $s_{i4} = (S_i^{F_i}(3), H_i^{F_i}(3), 1)$, $s_{i5} = (voter_i, H_i^{F_i}(1), 3)$.

4. F_i=RB/*1/1* $\Rightarrow$ $H_i^{F_i} = \{H_i^{F_i}(1), H_i^{F_i}(2)\}$, $S_i^{F_i} = \{S_i^{F_i}(1), S_i^{F_i}(2)\}$. Данной конфигурации соответствуют элементы расписания $s_{i1} = (receiver_i, H_i^{F_i}(1), 1)$, $s_{i2} = (S_i^{F_i}(1), H_i^{F_i}(1), 2)$, $s_{i3} = (test_i, H_i^{F_i}(1), 3)$, $s_{i4} = (recovery_i, H_i^{F_i}(1), 4)$, $s_{i5} = (S_i^{F_i}(2), H_i^{F_i}(1), 5)$, $s_{i6} = (test_i, H_i^{F_i}(1), 6)$, $s_{i7} = (sender_i, H_i^{F_i}(1), 7)$, $s_{i8} = (S_i^{F_i}(1), H_i^{F_i}(2), 1)$, $s_{i9} = (test_i, H_i^{F_i}(2), 2)$, $s_{i10} = (recovery_i, H_i^{F_i}(2), 3)$, $s_{i11} = (S_i^{F_i}(2), H_i^{F_i}(2), 4)$, $s_{i12} = (test_i, H_i^{F_i}(2), 7)$. Отметим, что в рамках поставленной задачи необходимо получить максимальное время завершения работы программ, поэтому считается, что результат работы первой версии программного компонента всегда отвергается, происходит откат и запуск второй версии.

Никаких других элементов, кроме описанных выше, в расписании нет.

В ходе выполнения заданий $receiver_i$ происходит прием данных от других модулей; $sender_i$ – отправка данных; $voter_i$ – выбор результата, выдаваемого большинством версий программного компонента, и отправка данных; $test_i$ – проверка результата с помощью

контрольного теста; $recovery_i$ – откат к начальному состоянию. Если в i-ом модуле не используется МОО, то действия по приему и отправке данных входят в задание s_{i1}.

Опишем зависимости между элементами построенного расписания:

- Если F_i=*NVP/0/1* или F_i=*NVP/1/1*, то элементы s_{i2}, s_{i3}, s_{i4} непосредственно зависят от s_{i1}; s_{i5} – от s_{i2}, s_{i3}, s_{i4}.
- Если F_i=RB*/1/1*, то s_{i2}, и s_{i8} непосредственно зависят от s_{i1}; s_{ij} – от s_{ij-1} для j={3,4,5,6,9,10,11,12}; s_{i7} – от s_{i9} и s_{i12}.
- Если модуль U_i непосредственно зависит по данным от модуля U_j, то s_{i1} непосредственно зависит либо от s_{j1} (F_j=*None*), либо от s_{j5} (F_j=*NVP/0/1* или F_j=*NVP/1/1*), либо от s_{j7} (F_j=*RB/1/1*).

Директивные сроки всех заданий, соответствующих модулю U_i, равны D_i, временем завершения работы программного компонента модуля U_i считается время завершения работы задания, осуществляющего передачу данных.

3.6.2 Апробация схемы интеграции средств моделирования со средствами оптимизации надёжности РВС РВ

На данном этапе исследовались временные характеристики работы средств. При запуске средства выбора МОО РВС РВ, интегрированного со средой моделирования, на простейшей модели ВС, состоящей из одного модуля, содержащих по 3 варианта программных и аппаратных компонентов, был получен следующий результат: общее время работы средства составило 15 минут 42 секунды. Все время работы при подсчете времени с помощью функции самого средства выбора МОО РВС РВ (без запуска имитационных экспериментов) составляло менее одной секунды.

Полученные результаты показали, что проведение имитационных экспериментов требует больших вычислительных затрат и средство выбора МОО РВС РВ нужно модифицировать таким образом, чтобы по возможности сократить количество обращений к процедуре моделирования.

Также исследовалось соотношение времен работы отдельных этапов схемы интеграции средства выбора МОО РВС РВ и среды моделирования. Для этого проводилось несколько запусков схемы для одной и той же конфигурации, измерялись времена работы и результаты усреднялись (Таблица 15). На рисунке 64 эти данные отображены в виде

графиков. Количество федератов в данном случае равно количеству используемых вариантов аппаратных компонентов (процессоров).

Таблица 15 - Время работы этапов интеграции программных средств, усредненное по 5 запускам

Кол-во федератов	Генерация SCXML-модели (сек.)	Генерация кода федератов (сек)	Получение исполняемых файлов (сек.)	Выполнение моделей в среде CERTI (сек)	Итого (сек.)
1	<1	1	3	11	15
2	<1	5	9	29	43
3	<1	13	14	44	71
4	<1	16	18	51	85
5	<1	13	21	59	93
6	1	24	29	77	131
7	1	98	45	66	210
8	2	209	62	65	338
9	2	101	57	77	237
10	3	272	82	117	474
11	1	114	59	109	283
12	4	403	112	104	623
13	4	291	107	126	528
14	2	99	62	111	274
15	7	323	168	145	643
16	4	428	143	126	701
17	5	460	161	138	764
18	7	360	198	174	737
19	5	615	183	168	971
20	8	508	231	208	955

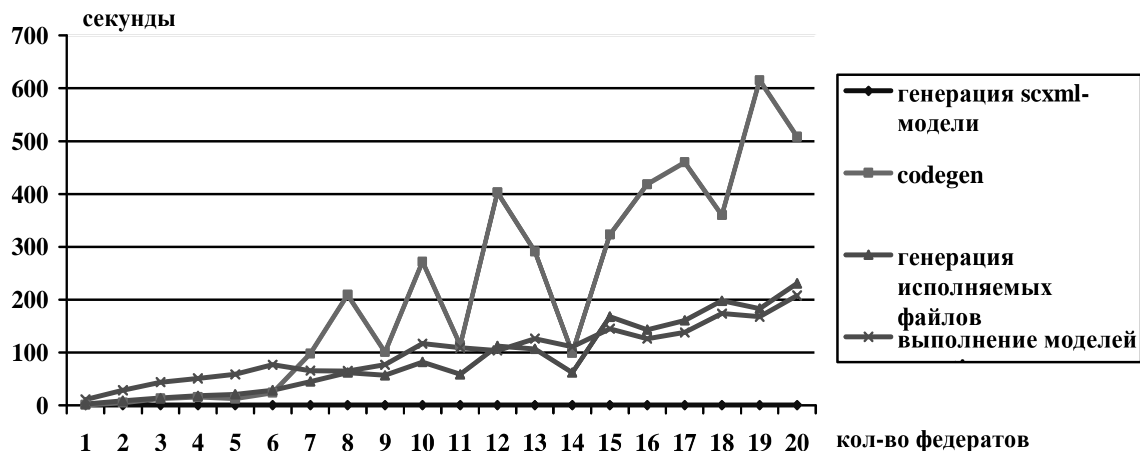

Рисунок 66 - Сравнение времени работы отдельных этапов интеграции средства выбора МОО РВС и среды моделирования

Результаты сравнения времен работы отдельных компонентов схемы интеграции показали, что с увеличением количества федератов (используемых вариантов аппаратных компонентов) быстрее всего увеличивается время работы средства генерации кода федератов по SCXML-модели (раздел 1.2). Кроме того, время работы этого средства сильно зависит от структуры расписания: для близких по количеству федератов моделей времена работы средства генерации кода сильно различаются. Данный эффект можно объяснить тем, что при построении кода федератов по диаграммам состояний SCXML в худшем случае поиск зависимостей между федератами имеет экспоненциальную сложность (для каждого состояния одного автомата рассматриваются все состояния всех остальных автоматов). Однако для диаграмм состояний, соответствующих расписаниям, обладающих сильной степенью связности, зависимости находятся за малое количество шагов.

3.7 Эксперименты со средствами трассировки моделей и внесения неисправностей

3.7.1 Введение

В разделе 1.3 были рассмотрены возможные схемы трассировки имитационных экспериментов в разрабатываемой среде моделирования РВС РВ. Для реализации были

выбраны централизованная схема «федерат-сборщик» и распределенная схема на основе использования RTI-интерфейса для сбора трассы для каждого отдельного федерата.

В разделе 1.11 было рассмотрены варианты построения средства внесения неисправностей и выбрана схема с одним федератом-перехватчиком сообщений, который позволяет перехватывать сообщения и портить значения переменных.

Целью данного экспериментального исследования является оценка влияния процесса трассировки и внесения неисправностей на процесс моделирования. В качестве критерия сравнения выбрано время моделирования в зависимости от количества событий (переданных сообщений), то есть, на сколько увеличивается время при включении опции трассировки событий или внесения неисправностей. По итогам предыдущих этапов работы в качестве формата хранения трассы был выбран формат OTF [3].

3.7.2 Методика проведения экспериментов

Для проведения экспериментов использовались модели «Лавина» и «Пинг-Понг», описанные в разделе 3.2. Для каждой из них была добавлена опция выбора схемы трассировки и опции для автоматизации проведения экспериментов (с указанием количества запусков эксперимента и записью результатов экспериментов в виде таблицы в файл с расширением csv).

Для каждой модели эксперименты проводятся следующим образом:

1. Запускается модель без трассировки с 10, 100, 1000, 10000 сообщениями (по 100 запусков). Фиксируется среднее время выполнения модели.
2. Аналогично запускается модель со схемой трассировки «сборщик-федерат».
3. Запускается модель со схемой трассировки на основе использования RTI-интерфейса.

Нужно отметить, что при определении времени выполнения модели с последней схемой трассировки не учитывается время на сбор полной трассы.

Эксперименты проводились на единственной инструментальной машине, на которой были запущены все компоненты распределённой системы моделирования. Компьютер, на котором проводились эксперименты имел следующие характеристики: процессор Core i5 560M 2660 Мгц, 4 Гб DDR3 оперативной памяти, SSD жесткий диск, 128 Гб, операционная система Ubuntu 10.10.

Аналогичные эксперименты для сравнения производительности системы производительность среды выполнения без перехватчика и среды выполнения с

перехватчиком были выполнены для средства внесения неисправностей. Для этого была взята тестовая модель «Лавина». Тестирование проводилось с перехватчиком и без. Для этого модель в обоих случаях запускали с параметром N = 500, 1000, 1500, …, 5000, где N – число сообщений, передаваемых отправителем. Для справедливости эксперимента запуск с каждым параметром проводился несколько раз и полученные значения усреднялись. Запуски производились на компьютере Core 2 Duo 2660 Мгц, 4 Гб DDR2 оперативной памяти, жесткий диск, 320 Гб, операционная система Ubuntu 12.

3.7.3 Результаты экспериментального исследования

Результаты экспериментального исследования выполнения моделей «Лавина» и «Пинг-Понг» с различными схемами трассировки приведены в таблицах 16 и 17.

Таблица 16. Среднее время выполнения модели Лавина с различными схемами трассировки, мкс

Количество событий	Время выполнения модели Лавина, мкс.		
	без трассировки	с федератом-сборщиком	с трассировкой каждого федерата
10	5	6	5
100	45	48	47
1000	508	581	548
10000	4703	5192	5023

Таблица 17. Среднее время выполнения модели Пинг-Понг с различными схемами трассировки, мкс

Количество событий	Время выполнения модели Пинг-Понг, мкс.		
	без трассировки	с федератом-сборщиком	с трассировкой каждого федерата
10	8	9	8
100	84	95	87
1000	887	1010	962
10000	8706	9934	9385

Результаты экспериментального исследования выполнения моделей «Лавина» с федератом-перехватчиком и без него приведены в таблице 18.

Таблица 18. Среднее время выполнения модели Лавина без федерата-перехватчика и с федератом-перехватчиком, мкс

Количество событий	Время выполнения модели Лавина, мкс.	
	без федерата-перехватчика	с федератом-перехватчиком
500	239	405
1000	421	825
1500	654	1214
2000	854	1639
2500	1074	2085
3000	1327	2479
3500	1473	2846
4000	1692	3321
4500	1971	3867
5000	2198	4458

3.7.4 Выводы

По результатам проведенных экспериментов со схемами трассировки можно сделать следующие выводы:

- Использование централизованной схемы с федератом-сборщиком увеличивает время выполнения модели от 7 до 15%.
- Использование распределенной схемы трассировки на основе использования RTI-интерфейса увеличивает время выполнения модели от 3 до 9 %.

Таким образом, результаты проведенных экспериментов показали, что рассматриваемая распределенная схема трассировки оказывает наименьшее влияние на процесс моделирования. Однако, можно сделать вывод и о том, что требуется проведение дополнительного экспериментального исследования:

- на большем количестве моделей;
- для трассировки разнообразных типов событий;
- для трассировки моделей, размещенных на различных машинах, взаимодействующих по сети.

Также есть предположение, что в зависимости от количества и расположения моделей на хостах напрямую зависит скорость трассировки и эффективность той или иной схемы.

Как видно из результатов экспериментов со средством внесения неисправностей, при увеличении числа сообщений, время выполнения модели с перехватчиком значительно больше времени выполнения модели без перехватчика. В модели с перехватчиком каждое сообщение от отправителя сначала идет перехватчику, а затем отправителю, то есть фактически число сообщений, проходящих через RTIG, удваивается. Как было сказано в [2], CERTI имеет централизованную архитектуру, поэтому процесс RTIG является узким местом системы, что объясняет полученные результаты.

4 Подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 36 академических часов

По курсам «Технологии разработки встроенных систем», «Математические методы спецификации и верификации ПО» и «Алгоритмы планирования вычислений в системах реального времени» были подготовлены учебные материалы, представленные в приложении А.

По курсу «Технологии разработки встроенных систем» были разработаны материалы в виде слайдов для презентации объёмом 4 академических часа. Подготовлены материалы для следующих лекций:

- средства разработки: средства отслеживания проблем и измененийсредства версионного контроля;
- средства разработки: средства компиляции и линковки программ.

По курсу «Математические методы спецификации и верификации ПО» были разработаны материалы в виде текста объёмом 2 академических часа. Подготовлены материалы для практического занятия:

- применение систем верификации моделей программ для проверки распределенных программ с неограниченным числом процессов;

По курсу «Алгоритмы планирования вычислений в системах реального времени» были разработаны материалы в виде текста лекций объёмом 30 академических часа. Подготовлены материалы для следующих лекций:

- задачи условной оптимизации;
- классические задачи комбинаторной оптимизации;
- задачи, возникающие при проектировании вычислительных систем реального времени;
- алгоритмы случайного поиска;
- алгоритмы имитации отжига;
- алгоритм имитации отжига для решения задачи построения статических многопроцессорных расписаний с минимальным временем выполнения на заданном числе процессоров;

- параллельный алгоритм имитации отжига для построения статических многопроцессорных расписаний;
- генетические и эволюционные алгоритмы;
- модификации генетических алгоритмов;
- генетический алгоритм для решения задачи определения минимально необходимого числа процессоров и построения расписания выполнения функциональных задач со временем выполнения не превышающим заданный директивный срок;
- алгоритмы дифференциальной эволюции;
- муравьиные алгоритмы;
- муравьиные алгоритмы для решения задачи построения статико-динамических расписаний;
- муравьиный алгоритм для решения задачи построения расписания обменов по шине с централизованным управлением;
- алгоритмы, использующие аппроксимирующую модель целевой функции.

Заключение

В результате выполнения работ по четвёртому этапу НИР были выполнены следующие работы.

В соответствии с п. 4.1 календарного плана и требованиями 4.1.2, 4.1.3 и 4.1.11 ТЗ разработаны методы и инструментальных средств поддержки анализа и разработки РВС РВ второй очереди. В рамках данного направления была модифицирована среда выполнения моделей, внесены изменения в средства трансляции из формата UML в формат SCXML и из формата SCXML в UML, проведён обзор схем трассировки моделей и разработано средство трассировки моделей, разработано средство внесения неисправностей, обоснована корректность алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов, приведены предложения по минимизации временных автоматов, предложен и реализован алгоритм восстановления параметров модели по контрпримеру в UPPAAL, проведён обзор методов оценки наихудшего времени выполнения и реализаован метод оценки наихудшего времени выполнения, проведён обзор моделей процессоров для оценки наихудшего времени выполнения, а также содержит описание разработанного метода оптимизации надёжности РВС РВ, включающего запуск имитационной модели для оценки времени выполнения компонентов РВС РВ.

В соответствии с п. 4.2 календарного плана и требованиями 4.1.1, 4.1.2, 4.1.3 и 4.1.11 ТЗ проведено экспериментальное исследование методов и инструментальных средств поддержки анализа и разработки РВС РВ второй очереди. В рамках данного направления была разработана модель поведения бортовых компьютеров автомобилей и проведено её экспериментальное исследование, проведено экспериментальное сравнение модифицированной и штатной среды выполнения моделей, проведены эксперименты с модифицированным средством трансляции из формата UML в формат SCXML, проведено экспериментальное исследование различных схем трассировки моделей, проведено экспериментальное исследование алгоритма восстановления параметров модели по контрпримеру в UPPAAL, также проведено экспериментальное исследование средств оценки наихудшего времени выполнения программ и средства оптимизации надёжности РВС РВ.

В соответствии с п. 4.3 календарного плана и требованиями 4.1.6 и 4.1.11 ТЗ проведена интеграция разработанных на предыдущих этапах работы средств, таких как среда моделирования, средства верификации и анализа результатов моделирования. Также была проведена интеграция с методами и инструментальными средствами поддержки анализа и

разработки РВС РВ второй очереди такими как средства оценки наихудшего времени выполнения и средства оптимизации надёжности РВС РВ.

В соответствии с п. 4.4 календарного плана и требованиями 4.1.8, 4.1.9 и 4.1.11 ТЗ разработаны научно-методические материалы для учебных материалов по курсам «Технологии разработки встроенных систем», «Алгоритмы планирования вычислений в системах реального времени» и «Математические методы спецификации и верификации ПО».

Список использованных источников

1. Отчёт о научно-исследовательской работе «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)» (Этап 1) // М.:, 2010. - Стр. 65

2. Отчёт о научно-исследовательской работе «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)» (Этап 2) // М.:, 2011. - Стр. 189

3. Отчёт о научно-исследовательской работе «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)» (Этап 3) // М.:, 2011. - Стр. 162

4. Simulation Interoperability Standards Committee of the IEEE Computer Society IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Federate Interface Specification. 2000.

5. Noulard E., Rousselot J.-Y., CERTI, an Open Source RTI, why and how // Spring Simulation Interoperability Workshop. San Diego, USA, 2009.

6. Chemeritskiy, E.V., Savenkov, K.O. Towards a real-time simulation environment on the edge of current trends // In Proceedings of the 5-th Spring/Summer Young Researchers' Colloquium on Software Engeneering, SYRCoSE-2011, Yekaterinburg, Russia, may 12-13 2011, pp. 128-133.

7. Чемерицкий Е.В., Волканов Д.Ю., Смелянский Р.Л. Оценка применимости среды CERTI для моделирования РВС РВ // Пятая всероссийская научно-практическая конференция по имитационному моделированию и его применению в науке и промышленности, ИММОД-2011, Санкт-Петербург, 19-21 октября 2011, т.1, стр. 409-413.

8. Karlsson M., Karlsson P. An In-Depth Look at RTI Process Models // In Proceedings of 2003 Spring Simulation Interoperability Workshop, Stockholm, Sweden, 2003.

9. B. d’Ausbourg, P. Siron, and E. Noulard, “Running Real Time Distributed Simulations under Linux and CERTI,” European Simulation Interoperability Workshop, Edimburgh, Scotland, 2008.

10. Stokes J. Introduction to Multithreading, Superthreading and Hyperthreading [ARS] (http://arstechnica.com/old/content/2002/10/hyperthreading.ars).

11. Möller B., Karlsson M. Making RTI Tuning Easy with Performance Profiles // In Proceedings of 2005 Spring Simulation Interoperability Workshop, Toulouse, France, 2005.

12. Williams A. The Boost Thread Library // [HTML] (http://www.boost.org/doc/libs/1_47_1/libs/boost_thread/boost_thread.htm), 2011.

13. Patrick P. C. Lee, Tian Bu, Girish Chandranmenon A lock-free, cache-efficient shared ring buffer for multi-core architectures // In proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'09), Princeton, USA, October 19-20 2009, pp. 78-79.

14. L. Malinga and WH. Le Roux, HLA RTI Performance Evaluation // European Simulation Interoperability Workshop, Istanbul, Turkey, 2009, pp. 1-6.

15. Robert C. Martin , UML Tutorial: Finite State Machines // Engineering Notebook Column C++ Report, June 1998

16. *State Chart XML* (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft 26 April 2011 [HTML] (http://www.w3.org/TR/SCXML/)

17. *Cheetah Users' Guide* [PDF] (http://www.cheetahtemplate.org/docs/users_guide.pdf)

18. *Антоненко В.А., Волканов Д.Ю., Чистолинов М.В.* Средство генерации имитационной модели, совместимой со стандартом HLA. — Санкт-Петербург, 2011. — т.1, стр. 331-335.

19. Norman Wilde, Sharon Simmons, Dennis Edwards and L. Pounds. But Where Does It DO That? Locating Features in a Distributed Simulation. The 2002 Fall Simulation Interoperability Workshop, Paper number 02F-SIW-088, 2002.

20. Mr. Jerry Black. Data Collection in an HLA Federation, 1999.

21. Балашов В.В., Бахмуров А.Г., Волканов Д.Ю., Смелянский Р.Л., Чистолинов М.В., Ющенко Н.В. Стенд полунатурного моделирования для разработки встроенных вычислительных систем. Труды Третьей Всероссийской научной конференции <<Методы и средства обработки информации>> (МСО-2009, 6-8 октября 2009 года). - М.: Издательский отдел факультета ВМиК МГУ имени М.В.Ломоносова; МАКС-Пресс, 2009, с.16-24

22. Heng-Jie Song, Zhi-Qi Shen, Chun-Yan Miao, Ah-Hwee Tan, Guo-Peng Zhao. The Multi-Agent Data Collection in HLA-based Simulation System. 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07), 2007.

23. A. David, M.O. Moller, W. Yi. Verification of UML Statechart with Real-time Extensions // Uppsala: Department of Information Technology, Uppsala University. IT Technical Report 2003-009, 2003.

24. E.M. Clarke, O. Grumberg, D. Peled. Model Checking // The MIT Press. 1999.

25. R. Alur, C. Courcoubetis, D. Dill Model-checking for real-time systems. Proceedings of the 5-th IEEE Symposium on Logic in Computer science, 1990, p. 414-425.

26. R. Alur, D. Dill. Automata for modeling real-time systems. Proceedings of the 17-th ICALP, 1990, p. 322-335.

27. Reinhard Wilhelm, Jakob Engblom The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools // 2008.

28. Reinhold Heckmann, Christian Ferdinand Worst-Case Execution Time Prediction by Static Program Analysis // 2004.

29. Daniel Sandell, Andreas Ermedahl Static Timing Analysis of Real-Time Operating System Code // 2004.

30. Прус В.В. Эффективный алгоритм перебора кратчайших путей в графе //Труды Всероссийской научно-технической конференции "Методы и средства обработки информации" (МСО-2003). М.: Издательский отдел факультета ВМиК МГУ, 2003. С. 474.

31. Alexandre Davide, John Håkansson, Kim G. Larsen, and Paul Pettersson *Model Checking Timed Automata with Priorities using DBM Subtraction* // 2006

32. Armin Biere, Alessandro Cimatti, Edmund M. Clarke Bounded Model Checking // Advances in Computers. 2003. Vol. 58. P. 118-149.

33. Sungjun Kim, Hiren D. Patel, Stephen A. Edwards Using a Model Checker to Determine Worst-case Execution Time // 2009.

34. Reinhard Wilhelm Time analysis and timing predictability // 2005

35. Ющенко Н.В. Оценка времени выполнения программ статико-динамическим методом // «Программные системы и инструменты»: Тематический сборник факультета ВМиК МГУ им. ЛомоносоваN2/Под ред. Л.Н.Королева. М.:Издательский отдел факультета ВМиК МГУ, 2001. С.157-167.

36. Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, Ren R. Hansen, Kim G. Larsen, METAMOC: modular execution time analysis using model checking, 2010

37. Shaw, A. C. Reasoning About Time in Higher-Level Language Software. IEEE Transactionson Software Engineering // 1989.

38. Andreas Engelbredt Dalsgaard, Mads Christian Olesen, Martin Toft, Ren Rydhof Hanseneand Kim, Guldstrand Larsen, WCET Analysis of ARM Processors using Real-Time Model Checking // 2009.

39. Xianfeng Li, Yun Liang, Tulika Mitra, Abhik Roychoudhury, Chronos: a Timing Analyzer for Embedded Software // 2008.

40. Todd Austin, Eric Larson, Dan Ernst, SimpleScalar: An Infrastructurefor Computer System Modeling // 2002.

41. Peter J. Ashenden, The VHDL Cookbook // 1990.

42. Reinhard Wilhelm, Run-Time Guarantees for Real-Time Systems // 2009.

43. Marc Schlickling, Markus Pister, Semi-Automatic Derivation of Timing Models for WCET Analysis // 2010.

44. Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom, Clustered Worst-Case Execution-Time Calculation // 2005.

45. Niklas Holsti, Sami Saarinen, Status of the Bound-T WCET Tool // 2002.

46. Савенков К.О., Ющенко Н.В. Методика описания поведения процессора для оценки времени выполнения программы // Труды Всероссийской научной конференции «Методы и средства обработкиинформации» (1 октября – 3 октября 2003 г., г. Москва). М.: Издательский отдел факультета ВМиК МГУ, 2003. С. 486-491.

47. N. Wattanapongsakorn and Levitan S.P. Reliability Optimization Models of Fault-tolerant distributed systems // Reliability and Maintainability Symp. (RAMS), Philadelphia, PA, Jan. 22-25, 2001, pp. 193-199

48. A.G. Bakhmurov, V.V. Balashov, A.B Glonina, V.N. Pashkov, R.L. Smeliansky, D.Yu. Volkanov. Simulation Modeling Based Method For Choosing An Effective Set Of Fault Tolerance Techniques For Real-Time Avionics Systems // Proc. 4th EUCASS European Conference for Aerospace Sciences, St. Petersburg, Russia, 2011. - Накопитель (Flash).

49. Benso & P. Prinetto, Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, 2004

50. Leveugle R. Fault Injection in VHDL Descriptions and Emulation // IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'00). Yamanashi, Japan, 2000. P. 414.

52. Волканов Д.Ю., Шаров А.А. Программное средство автоматического внесения неисправностей для оценки надежности вычислительных систем реального времени с использованием имитационного моделирования // Методы и средства обработки информации. Труды второй Всероссийской научной конференции. - М.: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова, 2005. - С.457-464.

53. Smart, Julian; Hock, Kevin; Csomor, Stefan. Cross-Platform GUI Programming with wxWidgets, 5 AuguWst 2005, Prentice Hall, pp.744

54. Калашников А.В., Костенко В.А. Параллельный алгоритм имитации отжига для построения многопроцессорных расписаний // Известия РАН. Теория и системы управления. 2008. № 3. С. 101-110

55. Зорин Д.А. Способ представления и преобразования расписаний в итерационных алгоритмах структурного синтеза вычислительных систем реального времени // Программные системы и инструменты. Тематический сборник № 12, М.: Изд-во факультета ВМК МГУ, 2011., С. 1-1

56. State Chart XML (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft 26 April 2011 [HTML] (http://www.w3.org/TR/SCXML/)

57. Г.С. Осипов. Методы искусственного интеллекта // М.: ФИЗМАТЛИТ. – 2011.

58. Микроконтроллеры семейства AVR. [ASPX] (http://www.atmel.com/products/microcontrollers/avr/default.aspx).