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 № госрегистрации Инв. №	УТВЕРЖДАЮ декан академик РАН ____________ Е.И. Моисеев «___» ____________ 2011 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 № 3)

Наименование этапа: «Разработка среды выполнения моделей компонентов РВС РВ»

Руководитель работ, д.ф.-м.н. ______________ Смелянский Р.Л.
профессор подпись, дата

Москва 2011

Обозначения и сокращения

АРМ	Автоматизированное рабочее место
БИ	Бортовые интерфейсы
БПМ	Библиотека поддержки моделирования
БРЭО	Бортовое радиоэлектронное оборудование
КБО	Контур бортового оборудования
ЛВК	Лаборатория вычислительных комплексов
ЛЖ	Линия жизни
МКИО	Мультиплексный канал информационного обмена
НИР	Научно-исследовательская работа
ПНМ	Полунатурное моделирование
ПЧМ	Последовательная частная модель
РВС РВ	Распределённая вычислительная система реального времени
РИМ	Распределённое имитационное моделирование
РЧМ	Распределённая частная модель
СМ	Система моделирования
СММ	Система математического моделирования
ТЗ	Техническое задание
ЧМ	Частная модель
ЭО ТПК	Экспериментальный образец типового программного комплекса
ЯОМ	Язык описания моделей

ACID	Атомарность, Согласованность, Изоляция и Долговечность (Atomicity, Consistency, Isolation and Durability)
API	Интерфейс программирования приложений (Application Programming Interface)
DDS	Сервис распределения данных (Data Distribution Service)
DOM	Программный интерфейс для доступа к документа (Document Object Model)
EPL	Общественная лицензия Eclipse (Eclipse Public License)
FOM	Федеративная объектная модель (Federation Object Model)
HLA	Высокоуровневая архитектура (High Level Architecture)
JDK	Комплект разработчика приложений на языке Java (Java Development Kit)
RCTL	Темпоральная логика реального времени (Real-time Computation Tree Logic)
RTI	Среда имитационного моделирования
SCXML	Расширяемый язык разметки для диаграмм состояний (State Chart eXtensible Markup Language)
STL	Стандартная библиотека шаблонов (Standard Template Library)
TCP	Протокол управления передачей (Transmission Control Protocol)
UDP	Протокол пользовательских дейтаграмм (User Datagram Protocol)
UML	Универсальный язык разметки (Universal Markup Language)
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 третьего этапа НИР была разработка среды выполнения моделей компонентов РВС РВ. Основное содержание работ по третьему этапу следующее: разработка среды выполнения моделей компонентов РВС РВ; экспериментальное исследование и оптимизация среды выполнения моделей компонентов РВС РВ; разработка модели РВС РВ; подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 64 академических часов.

Результатом работы по третьему этапу являются: промежуточный отчёт о НИР за третий этап. Промежуточный отчёт о НИР за третий этап включает в себя: описание разработанных элементов среды выполнения моделей компонентов РВС РВ; описание разработанной модели РВС РВ; описание результатов экспериментального исследования и основных оптимизаций среды выполнения моделей компонентов РВС РВ; разработанные учебные материалы.

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

Содержание

Введение ... 6

1 Разработка среды выполнения моделей компонентов РВС РВ ... 7

1.1 Общее описание архитектуры среды выполнения моделей ... 7

1.2 Проблемы построения сред моделирования РВС РВ ... 8

1.3 Сравнительная оценка инструментов UML-моделирования с открытым исходным кодом 17

1.4 Трансляция UML в SCXML и SCXML в UML ... 27

1.5 Разработка средства визуализации Vis4 с поддержкой формата OTF ... 36

2 Разработка модели РВС РВ ... 51

2.1 Общее описание модели ... 51

2.2 Процессор C_1 ... 55

2.3 Процессор C_2 ... 68

2.4 Процессор C_3 ... 71

2.5 Процессор C_4 ... 78

3 Экспериментальное исследование и оптимизация среды выполнения моделей компонентов РВС РВ ... 82

3.1 Исследование применимость CERTI для моделирования РВС РВ ... 82

3.2 Доработка инфраструктуры CERTI RTI ... 87

3.3 Оптимизация шаблона Cheetah для трансляции в исполняемые модели, совместимые со стандартом HLA ... 103

3.4 Обоснование корректности алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов ... 108

3.5 Оптимизация алгоритма трансляции UML во временные автоматы ... 127

3.6 Оптимизация средств трансляции UML во временные автоматы ... 142

3.7 Экспериментальное исследование модели РВС РВ Dr Tesy ... 145

4 Подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 64 академических часов ... 153

Заключение ... 155

Список использованных источников ... 159

Введение

Настоящий документ представляет собой научно-технический отчет по третьему этапу НИР «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)». Документ содержит отчет по пунктам 3.1-3.4 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 3.1 календарного плана ТЗ приводится описание разработанных элементов среды выполнения моделей компонентов РВС РВ.

Во второй главе в соответствии с пунктом 3.3 календарного плана ТЗ приводится описание разработанной модели РВС РВ.

В третьей главе в соответствии с пунктом 3.2 календарного плана ТЗ описываются результаты экспериментального исследования и основных оптимизаций среды выполнения моделей компонентов РВС РВ.

В четвёртой главе в соответствии с пунктом 2.3 календарного плана ТЗ приводятся описание разработанных учебных материалов.

В заключении изложены основные результаты третьего этапа НИР.

1 Разработка среды выполнения моделей компонентов РВС РВ

В данном разделе описываются разработанные на третьем этапе средства, входящие в состав среды выполнения моделей компонентов распределённых вычислительных систем реального времени (РВС РВ). В разделе 1.1 приводится общее описание архитектуры среды выполнения. В разделе 1.2 описываются проблемы, возникающие при построении среды моделирования компонентов РВС РВ. Раздел 1.3 содержит сравнение инструментов с открытым исходным кодом для построения моделей на языке UML. В разделе 1.4 описывается средство трансляции из формата UML в формат SCXML. Раздел 1.5 содержит описание разработанного средства анализа и визуализации трасс Vis4 с поддержкой формата OTF, являющегося открытым форматом.

1.1 Общее описание архитектуры среды выполнения моделей

На предыдущих этапах данной работы [1,2] была предложена общая схема архитектуры среды моделирования. В соответствии с этой схемой ЭО ТПК разделяется на две подсистемы: среду выполнения частных моделей (ЧМ) и средства АРМ инженера-экспериментатора.

Среда выполнения ЧМ включает следующий набор средств (см. Рисунок 1):

- библиотека поддержки моделирования,
- средства регистрации и трассировки событий моделирования,
- средства обеспечения взаимодействия ЧМ с аппаратным обеспечением,
- средства оперативной (динамической) проверки выполнения спецификаций интерфейсов и поведения.

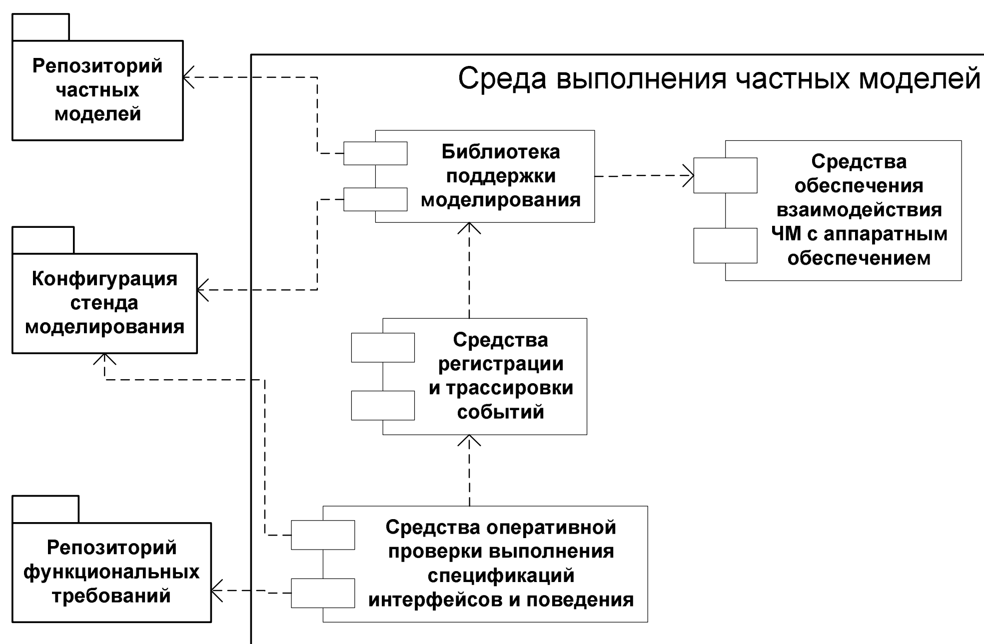

Рисунок 1. Состав средств среды выполнения ЧМ.

На третьем этапе необходимо было решить ряд задач в рамках разработки среды моделирования, которые не были решены на втором этапе. Во-первых, это выбор инструмента для построения UML-моделей. Во-вторых, это средство трансляции из формата UML в SCXML, используемое при трансляции в исполняемую модель, совместимую со стандартом HLA. В-третьих, это средство анализа и визуализации трасс Vis4 с поддержкой формата OTF.

1.2 Проблемы построения сред моделирования РВС РВ

1.2.1 Разработка моделей РВС

Удобный пользовательский язык описания моделей

Одна из основных целей разработчиков системы моделирования заключается в разработке единого средства для изучения свойств целого класса систем или явлений по их имитационным моделям. Для достижения этой цели система моделирования и, в первую очередь, язык моделирования должны работать в терминах, достаточно абстрактных для описания любой модели системы из заданного класса. Однако исследуемые системы даже из близких предметных областей часто оперируют разными понятиями. Поэтому разработчикам моделей приходиться переосмысливать интуитивное представление изучаемой системы, описывая её поведение в неестественных и неудобных для этого терминах.

Современные системы моделирования стараются решить описанную проблему, предлагая своим пользователям средства для автоматизации процесса разработки моделей. Подобная поддержка может быть реализована многими разными способами: поставкой набора готовых шаблонов элементов модели, дополнительными библиотеками, содержащими наборы наиболее часто используемых функций, специальными языками моделирования или же комбинацией перечисленных средств [3]. Универсальные системы моделирования, предназначенные для решения широкого диапазона имитационных задач, обычно работают на высоком уровне абстракции, а их среды выполнения предоставляют низкоуровневый интерфейс, с которым тяжело работать напрямую. Для подобных систем важны, прежде всего, гибкость, расширяемость и простота внесения изменений, что трудно совместимо с фиксированной структурой формально построенного языка описания моделей. Поэтому автоматизация разработки в универсальных системах обычно достигается за счёт использования стандартного набора библиотечных функций. Такой подход хоть избавляют пользователя от необходимости в самостоятельном описании наиболее частных моделей, но в то же время вынуждает его работать на уровне интерфейсов системы моделирования, что неудобно. В случае моделирования РВС, это означает, например, работу не на уровне устройств и каналов передачи данных, а на уровне взаимодействующих логических процессов.

Если система моделирования постоянно используется для решения одного типа задач, то набор необходимых операций фиксирован и известен заранее. Таким образом, становится оправданным использование узкоспециализированного языка описания моделей, который позволит разработчикам избавиться от синтаксического сахара и перейти непосредственно к описанию логики моделей. В результате разработчики получают удобный и интуитивно понятный инструмент для работы в конкретной прикладной области. Например, графический язык описания моделей, позволяющий описывать модель в терминах решаемой задачи, может эффективно использоваться людьми, не обладающими программистскими навыками.

Таким образом, разработчики современных систем моделирования постоянно находятся в поисках компромисса между шириной диапазона решаемых имитационных задач и удобством практического использования их средства. На данный момент наиболее прогрессивной представляется двухуровневая система автоматизации построения модели. На верхнем ярусе находится высокоуровневый язык описания моделей, работающий в терминах решаемой задачи. Модели, написанные на этом языке, транслируются в более

низкоуровневые программы, работающие в терминах вызовов среды выполнения. На этом же уровне абстракции разработчикам модели предоставляется гибкая и расширяемая библиотека функций, позволяющая использовать систему для решения новых классов имитационных задач на новых, непредугаданных ранее уровнях детальности.

Средства визуализации и отладки

Имитационные модели, как и любые компьютерные программы, требуют отладки. Отладка имитационной модели затрудняется многими факторами, связанными со сложной многокомпонентной структурой имитационной программы и выполняющей её распределённой среды моделирования. Например, для каждого выполнения модели необходимо запустить одну и более программ на каждом из участвующих в моделировании компьютеров. Кроме того, процесс отладки существенно затрудняется сложностью наблюдения за работой распределённого комплекса и сбора соответствующей статистической информации.

Современные системы моделирования пытаются упростить процесс отладки моделей, предоставляя их разработчикам средства визуализации процесса моделирования, позволяющие вести наблюдение за изменениями состояния модели. Кроме того, часто средства визуализации предоставляют функциональность для управления выполнением модели и динамического изменения её состояния. Например, с их помощью становится возможной временная приостановка выполнения модели и модификация значения её параметров.

Часто средства оперативной визуализации поставляются в виде отдельного средства поддержки моделирования. Однако в ряде случаев возможна его эргономичная интеграция с другими средствами разработки моделей. Это актуально, в частности, если для описания модели используется специализированный графический язык. В таком случае наиболее удобным способом отображения состояния модели будет его указание на той же схеме модели, на которой ведётся её проектирование. Такой подход позволяет разработчикам не переключаться между двумя разными средствами моделирования и соответствующими им представлениями его модели.

1.2.2 Интерфейс модели со средой прогона

Любая имитационная программа состоит из двух основных частей: имитационной модели, отражающей логику работы изучаемой системы, и среды прогона моделей, осуществляющей контроль над поведением модели и обеспечивающей взаимодействие её

компонентов. Данный раздел подробно описывает зоны ответственности каждой из частей имитационной программы и рассматривает основные принципы их взаимодействия. Особое внимание уделяется вопросам продвижения модельного времени и контролю над выполнением имитационной модели.

Концепция сервиса моделирования

Современные системы моделирования всё ближе подходят к концепции сервиса, когда отдельные компоненты имитационные модели просто подключаются к системе через стандартный интерфейс и начинают работать, аналогично подсоединяемым к энергосети электроприборам. Таким образом, система моделирования реализует набор сервисов и предоставляет участникам моделирования интерфейс для доступа к этим сервисам. При этом, любые взаимодействия участников моделирования происходят внутри инфраструктуры системы моделирования и контролируются. В результате система способна поддерживать модель в согласованном состоянии автоматически.

Кроме того, концепция сервисов сводит к минимуму зависимости между компонентами имитационной модели и позволяет рассматривать их в виде опциональных модулей. Каждый модуль обладает стандартным интерфейсом для подключения к системе моделирования и взаимодействует с другими компонентами модели исключительно через её сервисы. Поэтому он обладает свойством переносимости и может использоваться повторно при решении существенно отличающихся имитационных задач. Таким образом, отдельный модуль имитационной модели становится законченным самостоятельным продуктом. С одной стороны он может работать совместно с любой системой моделирования, предоставляющей подходящий интерфейс. В то же время его можно использовать в любой имитационной модели, построенной на описанных принципах. Примером подобного модуля может служить, например, модель, имитирующая логику работы авиационного высотомера или канала передачи данных Ethernet.

Многие модели, особенно лежащие в близких областях, включают в свой состав похожие по функциональности компоненты, поэтому библиотеки реализующих их переносимых модулей могут значительно упростить процесс разработки и поддержки. Использование подобных библиотек позволяет реализовывать только те компоненты модели, для которых пока ещё нет готовых решений, повторно используя разработанные ранее и уже проверенные модули для остальных компонентов. Вместе с увеличением числа решённых имитационных задач будет расширяться и библиотека готовых модулей. Таким образом,

концепция модульного проектирования потенциально позволяет свести задачу разработки модели к её конструированию из созданных ранее блоков.

Концепция сервисов системы моделирования требует коренной переработки стиля написания имитационных программ. Теперь разработчикам можно сосредоточиться непосредственно на описании логики работы каждого отдельного компонента имитационной модели, используя для взаимодействия с другими компонентами модели и средой выполнений предоставленные ей сервисы.

Сервисы системы моделирования должны быть построены на базе особых механизмов взаимодействия, позволяющих максимально ослабить зависимость отдельных компонентов модели друг от друга. Примером подобного механизма может выступать передача данных по схеме издатель-подписчик, успешно применённой, например, в стандарте инфраструктуры передачи данных OMG DDS [4] и стандарте распределённого моделирования HLA [5]. Согласно этой модели издатель публикует данные определённого типа, не заботясь о том, кто будет их использовать в дальнейшем. Подписчик, в свою очередь, может получить доступ к любым данным заданного типа и не заботится при этом об их происхождении. Таким образом, сервис передачи данных позволяет издателям и подписчикам сохранить анонимность и не привязываться друг к другу.

Кроме того, концепция сервисов системы моделирования предполагает, что среда выполнения взаимодействует с подключёнными к ней участниками в соответствии со схемой клиент-сервер. Роль серверов здесь играют предоставляемые средой выполнения службы и сервисы, а множество клиентов – образуется участниками моделирования. Каждый запрос к сервису распределённой среды выполнения потенциально способен привести к нескольким сетевым обменам между её компонентами, поэтому синхронное взаимодействие клиента и сервера неэффективно. Отсюда появляется необходимость построения асинхронных моделей на основе идей их параллельного многопоточного выполнения.

Сервисы продвижения времени

Центральной проблемой распределённого имитационного моделирования является проблема согласования работы отдельных компонентов модели в едином модельном времени. Соответствующий механизм согласования процессов называется *алгоритмом временной синхронизации* или *алгоритмами продвижения времени* модели [6,7]. Эффективные алгоритмы продвижения времени обладают сложной логикой, а их реализация требует больших усилий. Реализация алгоритмов продвижения времени в виде сервиса

системы моделирования позволяет облегчить процесс разработки и уменьшить число ошибок в создаваемых моделях.

Алгоритмы временной синхронизации можно условно разделить на две группы: *консервативные* и *оптимистические* [6]. Консервативная схема продвижения модельного времени является более традиционной и реализована подавляющим большинством современных систем моделирования. Основная идея данной схемы заключается в том, система моделирования должна приостанавливать выполнение компонента модели до тех пор, пока существуют факторы, способные повлиять на результат его работы. Такой подход автоматически обеспечивает согласованность имитационной модели между собой на любом этапе её выполнения.

В отличие от алгоритмов консервативной схемы, оптимистические алгоритмы позволяют выполнять компоненты имитационной модели при любых условиях. Но такой подход может приводить к ошибкам выполнения, поэтому система должна предусматривать возможность отката и исправления ошибок. Резюмируя вышесказанное, оптимистические алгоритмы сложнее для реализации, требуют от имитационной программы возможности сохранения и восстановления её промежуточных состояний и, соответственно, потребляют много памяти, а возможность отката нарушает свойство *предсказуемости* системы, тем самым, затрудняя её использование для решения имитационных задач с привязкой к реальному времени. Однако за счёт отсутствия постоянной синхронизации между отдельными компонентами модели алгоритмы данной группы позволяют эффективнее использовать вычислительные ресурсы распределённой системы моделирования. Подобные алгоритмы широко используются, например, при моделировании работы крупных компьютерных сетей, включающих сотни тысяч узлов [8].

Помимо рассмотренных консервативных и оптимистических алгоритмов продвижения времени исследователи часто выделяют ещё одну группу. Алгоритмы временной синхронизации, позволяющие объединять между собой компоненты модели, работающие по правилам консервативной схемы, с элементами модели, работающими в соответствии с оптимистической схемой, называются *смешанными*. При умелом применении, смешанные алгоритмы на практике часто позволяют совместить преимущества каждой из рассмотренных схем продвижения времени. В то же время неправильное сопряжение разных алгоритмов продвижения времени может привести к существенному падению производительности. Поэтому практическое применение смешанных алгоритмов

требует дополнительной поддержки процесса разработки модели со стороны системы моделирования – анализатора связей между компонентами модели.

Так как каждая из перечисленных схем продвижения времени обладает своими достоинствами и недостатками, то среда выполнения модели должна предоставлять подключённым к ней участникам моделирования сервис для управления продвижением её модельного времени в соответствии с правилами любой этих схем. Кроме того, для эффективного использования предоставленного функционала система моделирования должна включать дополнительные средства анализа имитационной модели, способные давать рекомендации по выбору и настройке наиболее подходящей схемы продвижения времени.

Сервисы управления имитационной моделью

Существует два принципиально отличающихся способа управления процессом выполнения имитационной модели: *централизованное* и *децентрализованное* управление. При централизованном управлении на этапе разработки имитационной модели среди участников моделирования выделяется главный участник, который единолично координирует работу остальных участников: производит начальную инициализацию модели и определяет момент завершения моделирования, устанавливает барьеры синхронизации модели, руководит процессом сохранения и восстановления её промежуточного состояния.

Второстепенные участники моделирования не принимают никакого участия в принятии этих решений, что упрощает логику их работы и, следовательно, упрощает процесс их создания. Централизованную схему управления в чистом виде удобно применять, например, в процессе отладки модели, когда все доступные средства контроля поведения модели необходимо сосредоточить на пульте оператора, следящего за её выполнением. Основной проблемой централизованного управления является невозможность изменения запланированного поведения модели. Например, добавление новых участников моделирования часто требует изменения логики главного участника моделирования, даже если логика отдельных элементов модели при этом изменений не требует.

В отличие от централизованного управления, децентрализованное управление требует участия в принятии решений всех зависящих от этого участников моделирования. Таким образом, реализация децентрализованной схемы требует использования более сложных методов согласования нескольких равноправных участников моделирования с использованием техники реплицирования и контроля когерентности данных. Тем не менее, применение децентрализованной схемы управления предпочтительнее с точки зрения

концепции сервисов моделирования и идеи модельного проектирования, так как позволяет динамически подстраивать поведение модели под вновь подключившихся участников. Децетрализованная схема управления широко используется в работе многих сетевых протоколов [9].

1.2.3 Распределённое моделирование

Унификация и стандартизация

Требования существующих задач имитационного моделирования к современным средам выполнения моделей намного превышают текущие возможности персональных компьютеров. Например, изучение свойств беспроводных сенсорных сетей требует высокого уровня детализации модели, потенциально включающей в себя сотни тысяч устройств [8]. Уже долгое время для исследований подобных моделей используются разнообразные параллельные и распределённые системы моделирования, построенные на базе многопроцессорных и многомашинных архитектур [10].

Основная задача интерфейса взаимодействия между отдельными компонентами имитационной модели и объединяющей их средой выполнения сводится к созданию абстракции над аппаратной платформой, используемой для проведения моделирования. Он позволяет скрыть от разработчиков модели детали практической реализации системы и сосредоточить свои усилия непосредственно на описании модели. Кроме того, модели, написанные с использованием стандартного интерфейса, легко переносимы на другие системы моделирования. Например, сегодня перспективным направлением исследований в области моделирования является моделирования на современных облачных платформах [11].

Поддержка политик качества сервиса QoS

Одним из сценариев использования разрабатываемой среды выполнения является полунатурное моделирование РВС РВ, когда часть компонентов имитационной модели реализованы программой, а часть – представляет собой физическое оборудование. Подключаемые аппаратные устройства могут налагать дополнительные требования к имитационной модели. Например, поведение физического оборудования может требовать соблюдения жёстких директивных интервалов, в течение которых модель должна ответить на поступивший от устройства запрос. Для гарантированного обеспечения реакции в заданном доверительном интервале система реального времени должна обладать свойством *предсказуемости* – то есть должна существовать возможность получения оценки наихудшего времени выполнения заданного задания [12].

Последняя доступная версия стандарта HLA IEEE-1516 не предусматривает своего использования для моделирования реального времени. Предположительно, такая возможность появится в спецификациях следующей версии стандарта HLA – HLA NG, разработка которого активно ведётся в настоящее время. Таким образом, на данный момент совместимая со стандартом HLA модель не вправе требовать какого-либо контроля качества от RTI.

В текущей версии стандарта HLA можно выделить следующие основные проблемы, препятствующие возможности проводить с его помощью моделирования систем реального времени, где производительности и предсказуемость системы имеют первостепенное значение:

1. Стандарт HLA не предоставляет необходимых интерфейсов для задания требуемых параметров качества сервиса. Поэтому для проверки удовлетворения директивным интервалам требуется расширить существующий интерфейс или же разработать надстройку, предоставляющую возможности, необходимые для моделирования реального времени;
2. Программная прослойка RTI работает поверх существующей операционной системы и не может предоставить средств, необходимых для управления распределением ресурсов инструментальной машины. При таком положении, операционная система может, например, неожиданно откачать из оперативной памяти страницы процессов моделирования или дать другим существующим задачам больший приоритет в использовании процессорного времени. В то же время системы моделирования, не использующие стандарт HLA, позволяют интегрироваться с ОС [15];
3. В случае распределённого имитационного моделирования стандарт HLA поддерживает лишь два типа передачи данных: надежный и ненадежный (обычно выраженных в реализациях CERTI в виде использования протоколов транспортного уровня TCP и UDP соответственно). Однако таких каналов недостаточно для проведения моделирования реального времени.

Описанные ограничения справедливы и для реализации CERTI. Одним из возможных путей решения проблем может стать использование дополнительного программного слоя для передачи данных с поддержкой необходимых для моделирования реального времени политик QoS. Существует множество стандартов, обладающих необходимой

функциональностью, и множество соответствующих им реализаций. Один из наиболее распространённых и прогрессивных стандартов в данной области – стандарт OMG DDS [4].

Спецификации OMG DDS определяют стандарт взаимодействия процессов, применимый к широкому классу распределённых систем реального времени и встроенных систем (DRE). В основе DDS лежит ориентированная на данные модель с архитектурой вида издатель-подписчик (DCPS). Процессы, использующие DDS, могут производить чтение и запись типизированных данных. При этом модель DCPS образует прослойку, которая позволяет читающим данные процессам получить доступ к самой последней их версии. В рамках DCPS определяются глобальное пространство данных, которые можно читать и изменять, и глобальное пространство имён, позволяющее создавать новые и искать существующие разделяемые объекты. Стандарт DDS определяет большое количество политик QoS и способов обмена данными между взаимодействующими процессами.

Суммируя вышесказанное, разрабатываемая среда выполнения имитационных моделей в итоге должна быть сформирована вокруг концепций заложенных в стандарт распределённого моделирования HLA. Из-за предъявляемых спектром предполагаемых задач требований, таких как поддержка разнообразных QoS и не описанных спецификациями стандарта HLA, в новую среду выполнения необходимо добавить дополнительный уровень для передачи данных и расширить функциональность, предоставляемую сервисами RTI с помощью системы, основанной на стандарте DDS.

1.3 Сравнительная оценка инструментов UML-моделирования с открытым исходным кодом

В рамках данного проекта возникает необходимость задействования графических средств разработки частных моделей, по сути представляющих собой конечные автоматы, описывающие поведение некоторых компонентов системы реального времени. В РВС РВ как правило несколько компонент может выполняться одновременно, поэтому это накладывает некоторые дополнительные требования на графический язык их моделирования. Кроме того, необходимо обеспечить возможность последующей верификации разработанных моделей. В силу популярности и распространенности, было решено в качестве языка для графического описания частных моделей использовать подмножество языка UML [2], позволяющее описывать диаграммы состояний. Таким образом, возникает задача выбора средства UML-моделирования в наибольшей степени подходящего под указанные нужды.

1.3.1 Основные критерии сравнения инструментов UML-моделирования

Диаграммы состояний, описываемые в стандарте языка UML2 в полной мере, позволяют описывать параллельно выполняющиеся компоненты распределенной системы реального времени. Поэтому, одним из важнейших критериев в процессе сравнения рассматриваемых средств была *степень их близости к выполнению всех требований стандарта*. В наибольшей степени здесь становится критичным наличие/отсутствие возможности создания параллельных регионов на диаграммах состояний, а также полнота поддержки разметки переходов сторожевыми условиями и триггерами.

При разработке большинства достаточно крупных систем возникает необходимость импорта/экспорта данных из одного используемого инструмента проектирования в другой. Касательно проектирования на языке UML это означает необходимость в обеспечении взаимодействия выбранного средства проектирования с другими, посредством стандартного формата XMI. В частности, данные в формате XMI поступают на вход верификатора моделей, поэтому следующим критерием сравнения рассматриваемых средств является *поддержка ими импорта/экспорта в этот формате*.

При разработке распределенных систем реального времени сложность внутреннего устройства их компонентов достаточно велика и элементарное неудобство и необходимость выполнения огромного числа лишних действий для создания требуемого элемента диаграммы может существенно затруднить создание необходимых моделей. Поэтому *удобство и наглядность* также являются одними их важнейших критериев при сравнении средств UML-моделирования. В частности, при рассмотрении интерфейса для создания диаграмм состояний в большинстве из рассматриваемых средств, удобство и наглядность в наибольшей степени зависит от наличия/отсутствия таких возможностей как:

- Быстрое создание перехода из состояния в состояние (без использования его переноса мышью из списка доступных элементов)
- Быстрая запись сторожевых условий и триггеров прямо на переходах из состояния в состояние.
- Поддержка операции отмены совершенного действия.

1.3.2 Сравниваемые средства

В рамках задачи сравнения средств UML-моделирования были проанализированы следующие средства с открытым исходным кодом:

- Papyrus
- Moskitt
- VioletUML
- TinyUML
- ArgoUML
- Topcased
- BOUML

Сравнение проводилось в два этапа, на первом из которых все перечисленные средства исследовались на предмет наличия в них, по крайней мере потенциальной возможности создания диаграмм описанного раннее вида, а на втором лишь те из перечисленных средств, что прошли отбор на первом этапе. На рисунках 2-8 представлены основные окна рассматриваемых средств.

Практически сразу были исключены из рассмотрения средства: VioletUML и TinyUML. Первые два оказались слишком примитивными и скорее подходят для обучения, чем для разработки моделей.

Таким образом, более детально будут рассмотрены следующие четыре инструмента: Papyrus, Moskitt, Topcased, BOUML и ArgoUML.

1.3.3 Сравнительная оценка средств

1.3.3.1 Papyrus

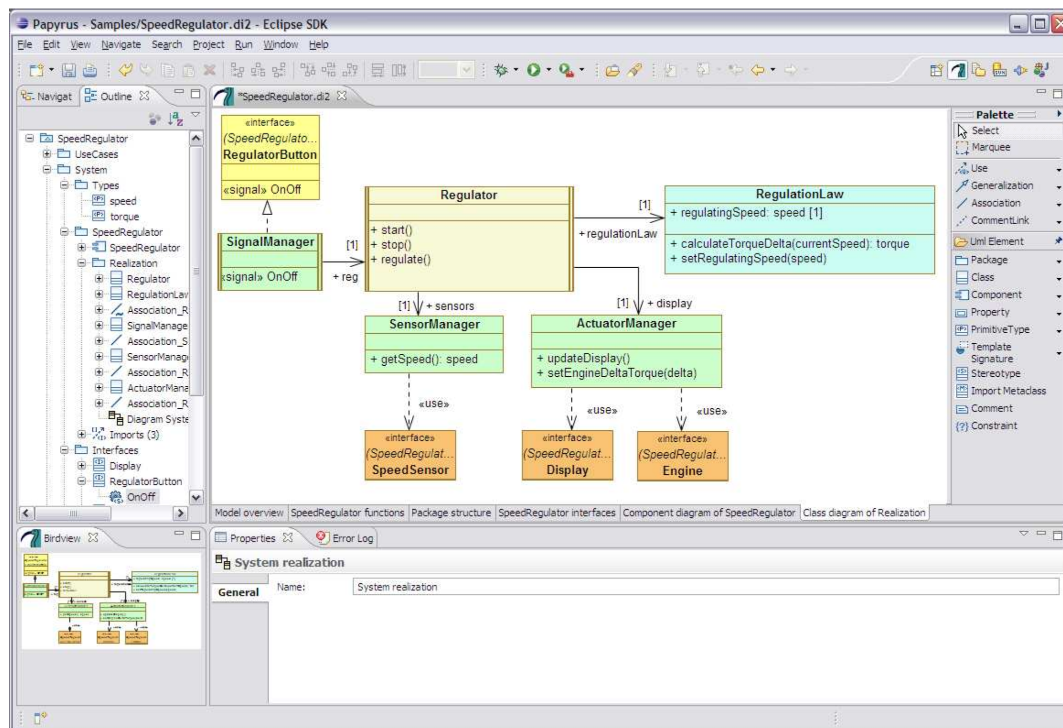

Рисунок 2. Основное окно средства Papyrus.

Проект перешел в 2008-2009 году под управление Eclipse Foundation и с тех пор началась его тотальная переработка. Актуальная версия для Eclipse Helios сейчас находится в состоянии инкубации, поэтому в ней очень много дефектов. Рабочая же версия есть только для Eclipse Galileo. Основным недостатком данного инструмента является отсутствие поддержки композитных состояний и невозможность создания триггеров и сторожевых условий, удовлетворяющих нуждам систем реального времени. Экспорт в XMI не реализован, однако созданная модель может храниться в данном формате. Графический редактор диаграмм состояний имеет множество недостатков: большое число артефактов отображения элементов, отсутствует возможность быстрого создания переходов, велика вероятность совершения необратимого действия, которое может сделать неработоспособной всю модель. Кодогенерация заявлена, но не работает. Имеются наглядные видео ролики и документация.

1.3.3.2 Moskitt

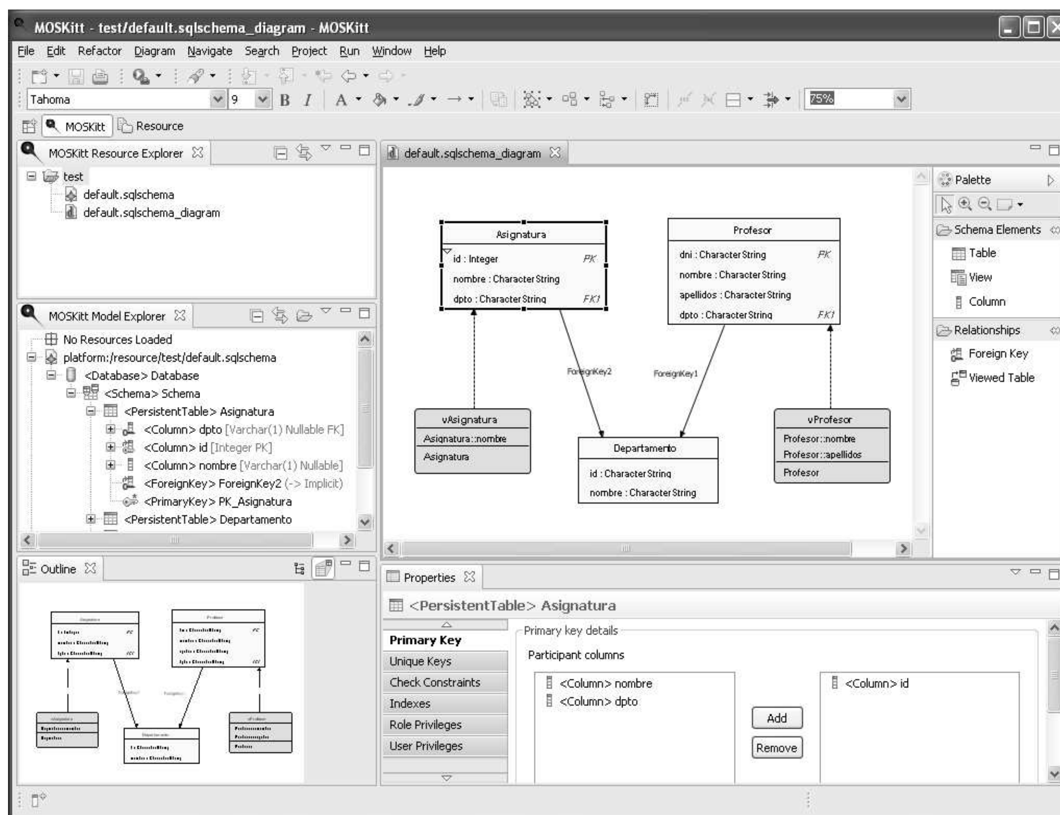

Рисунок 3. Основное окно средства Moskitt.

UML редактор с проработаным графическим интерфейсом, однако, кодогенерация работает только для создания схемы БД. Так же стоит отметить, что стандарт UML2 поддерживается не полностью. Существенно отсутствие поддержки экспорта в XML.

Положительным моментом является наличие наглядных видео материалов и подробная документация.

1.3.3.3 VioletUML

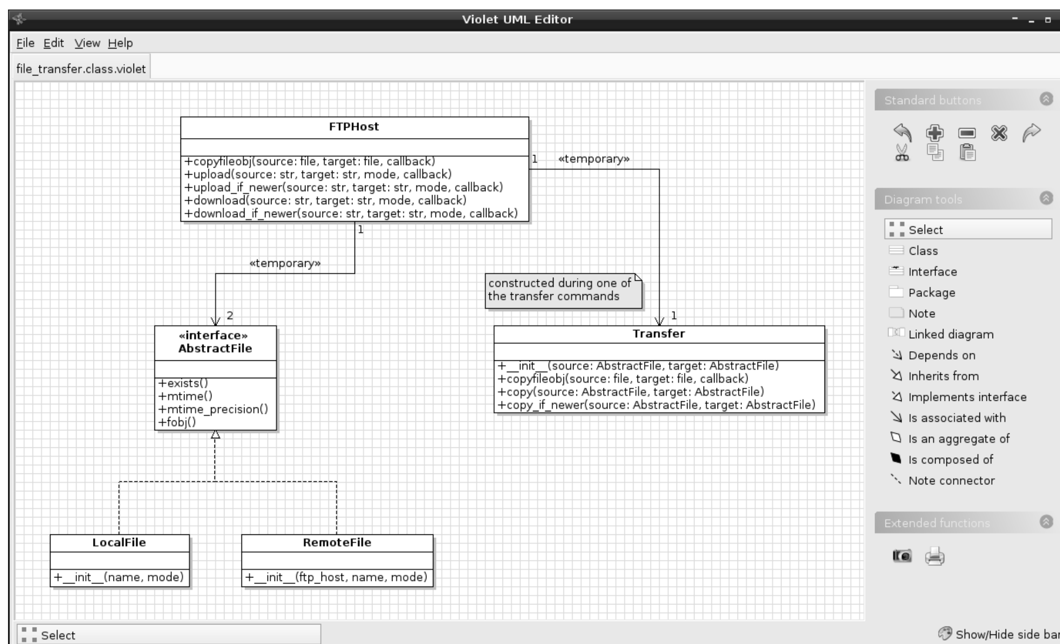

Рисунок 4. Основное окно средства VioletUML.

Данный редактор реализован в виде плагина к Eclipse, поддерживает версии eclipse 3.1.1, 3.2, 3.3. Поддерживает локальное сохранение eclipse, тем самым позволяя восстанавливаться ранее созданные диаграммы. Поддерживает drag'n'drop функции для java классов, с целью их представления в виде диаграмм последовательностей UML с последующим редактированием и переводов назад в исходный код.

Список полезных особенностей редактора:

- возможность работы с несколькими документами одновременно;
- возможность связывать несколько диаграмм;
- поддерживается зуммирование;
- функция отменить действие/ восстановить действие;
- функции вырезать / копировать / вставить;
- выделение сразу нескольких элементов диаграмм и поддержка drag'n'drop;
- возможность прятать все панели инструментов с целью увеличения рабочего пространства для создания диаграмм;
- возможность преставление диаграммы в виде изображения (например, для переноса в word документ).

Однако существует несколько функций, которые не реализованы в данном UML редакторе, из них:

- генерация исходного кода;
- генерация диаграмм по исходному коду;
- семантический анализатор диаграмм;
- экспортирование и импортирование в формат XMI.

1.3.3.4 TinyUML

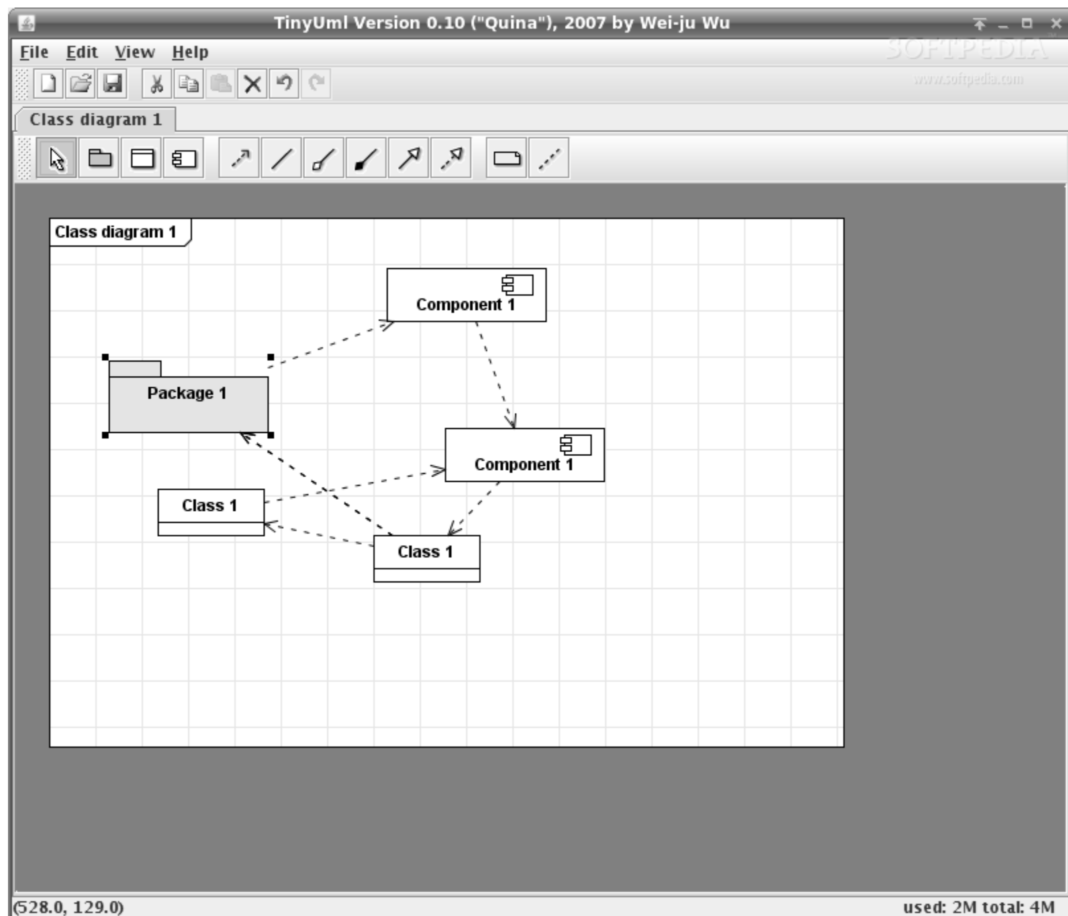

Рисунок 5. Основное окно средства TinyUML.

TinyUML UML 2 редактор диаграмм, основной акцент делается на создании приятного вида (читаемость) диаграмм и простоте использования. Распространяется под лицензией GPL, реализована на java. Стоит отметить, что данный проект создан для создания и редактирования небольших диаграмм. Следовательно, скорей всего примени в образовательных целях нежели в имитационном моделировании.

1.3.3.5 ArgoUML

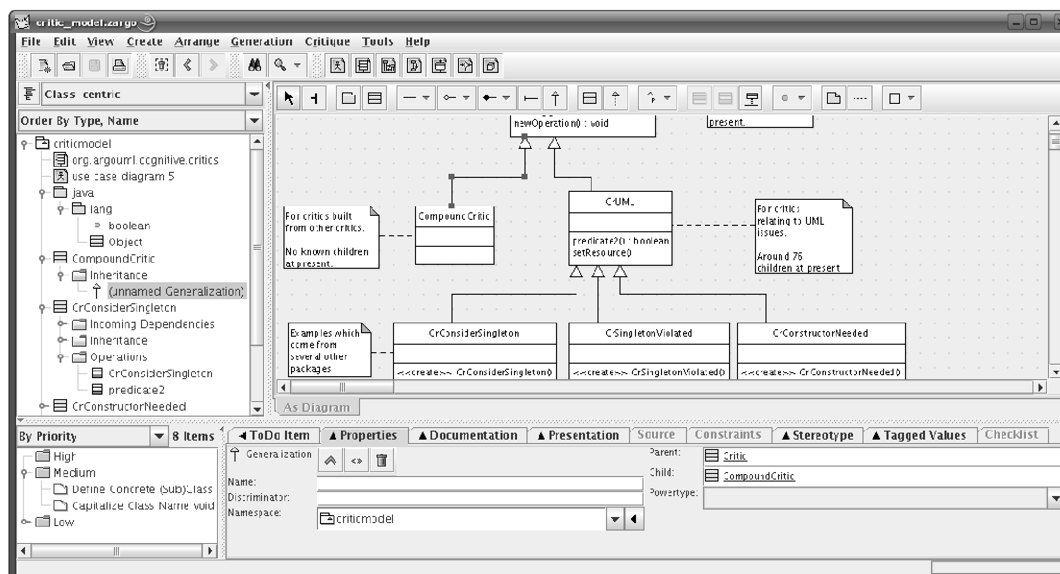

Рисунок 6. Основное окно средства ArgoUML.

ArgoUML полностью написан на Java. ArgoUML является открытым программным обеспечением. Распространяется под лицензией BSD. ArgoUML имеет интуитивно понятный и насыщенный пользовательский графический интерфейс.

Из полезных особенностей редактора:

- поддержка спецификаций UML 1.3, 1.4;
- экспортирование и импортирование в формат XMI 1.0, 1.1, 1.2;
- генерация исходного кода Java, C++, C# и PHP
- обратный инжиниринг из исходного кода и байткода Java
- автоматическую верификацию модели UML (design critics).

Нереализованные функции UML редактора:

- отсутствие функции обратного проектирования (нет возможности создавать модели из имеющегося кода на Java);
- нет импорта файлов, созданных в других пакетах для работы с UML.

1.3.3.6 Topcased

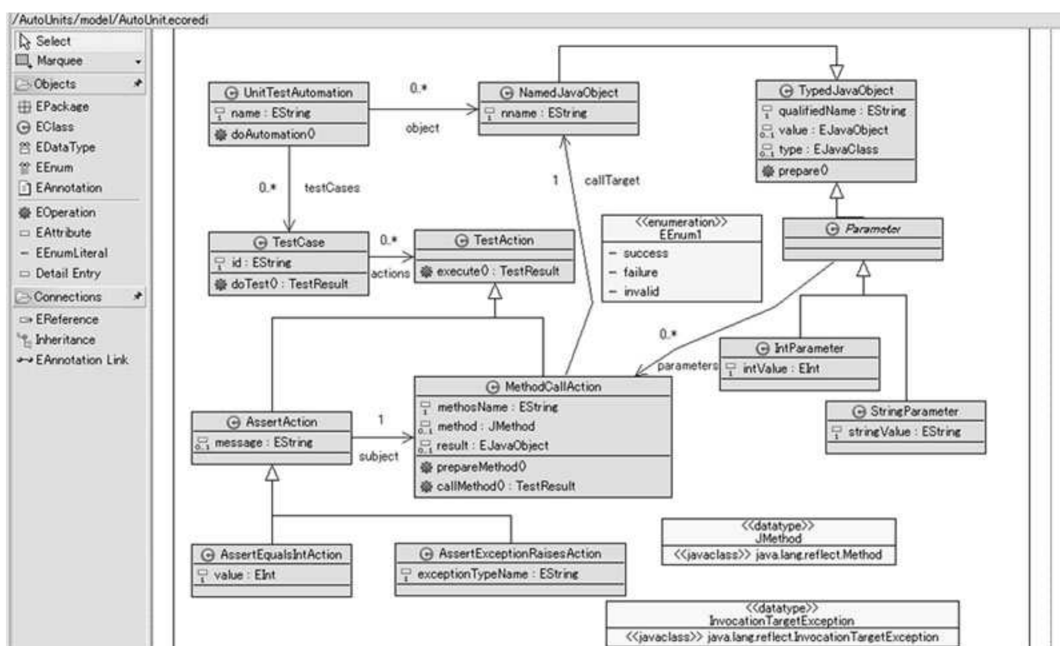

Рисунок 7. Основное окно средства Topcased.

TOPCASED строго ориентирована на модели: не только сама TOPCASED предоставляет редакторы моделей, средства проверки и трансформации моделей, но и сама создана на основе модели и генерации кода. Одной из нотации, поддерживаемых данным средством является UML.

Система TOPCASED активно используется при проектировании различных системы такими компаниями, как Airbus, Siemens VDO, CNES, Sopra Group, EADS Astrum, Ellidiss Technologies и др., при производстве критичных приложений реального времени.

1.3.3.7 BOUML

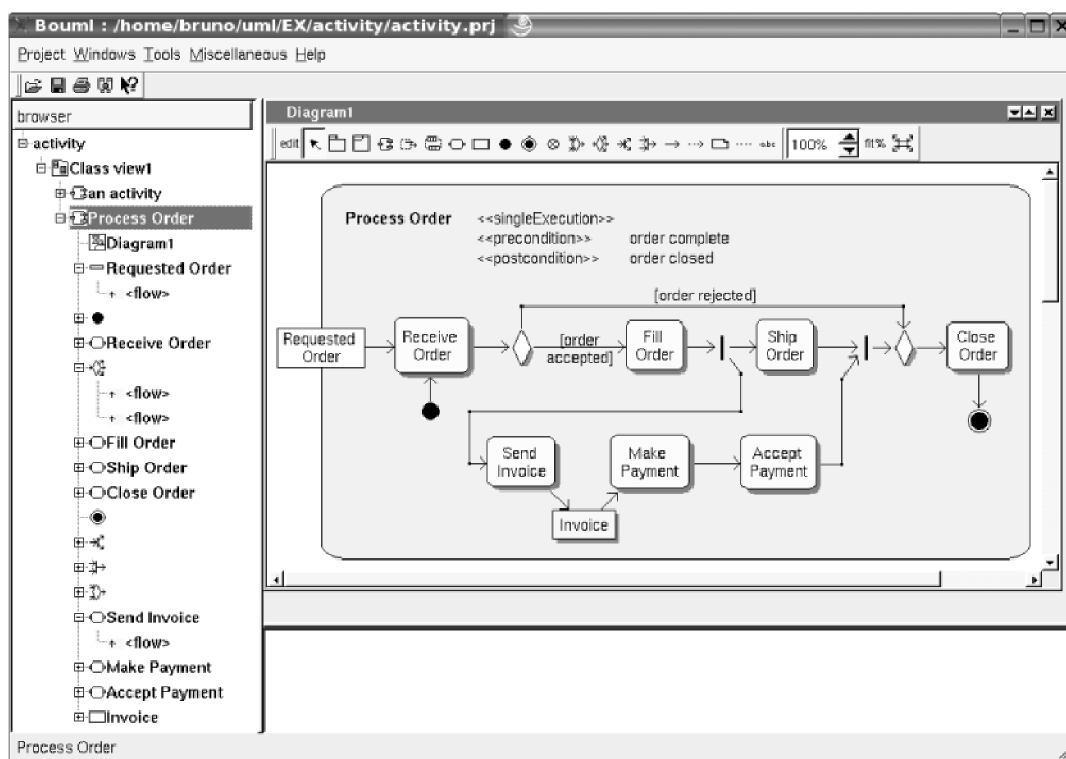

Рисунок 8. Основное окно средства BOUML.

BOUML это бесплатный UML 2 инструмент, позволяет специфицировать и генерировать код на C + +, Java, IDL, Php и Python. BOUML работает под Unix/Linux/Solaris, MacOS X и Windows.

Список полезных особенностей редактора:

- позволяет рисовать диаграммы в соответствии со стандартом UML 2.0;
- генерация код на следующие языки программирования: C++, Java, Php, Python;
- генерация диаграмм UML по исходного коды с языков программирования;
- кроссплатформенность (благодаря библиотеке QT)
- поддержка XMI.

1.3.4 Заключение

Таблица 1. Результаты сравнения рассмотренных редакторов.

Название средства	Лицензия	XMI	State Charts	ЯП	Доку-ментация	Кодо-генерация	Удобство использо-вания
Papyrus	EPL	-	+	Java	+ (Видео, PDF)	-	1 из 10
Moskitt	EPL	-	+	Java	+ (Видео, PDF)	-/+	4 из 10
VioletUML	GPL	-	+	Java	+/-	-	7 из 10
TinyUML	ACE/TAO/ CIAOLicense	-	-	Java	-	-	5 из 10
ArgoUML	EPL	+	+	Java	+ (HTML)	Java, C++, C#, PHP	8 из 10
BOUML	GPL	+	+	Java	+ (Видео, HTML)	Java, C++, C#, PHP, Python	4 из 10

В таблице 1 представлены результаты сравнения рассмотренных редакторов. Ни один из редакторов полностью не удовлетворяет всем предъявленным требованиям. Это связано в основном с отсутствием гибкой настройки генераторов кода, что делает невозможным написания собственных шаблонов для генерации кода федерации и федератов HLA. Стоит отметить, что полная поддержка UML2, в результате оказалась не необходимым условием, так как данная версия стандарта не вносит существенных изменений в диаграммы состояний UML.

В результате обзора был выбран UML редактор ArgoUML. Определяющими факторами выбора стали удобство интерфейса и поддержка экспорта в XMI. Возможность кодогенерации по диаграммам состояний отошла на второй план, так как в рамках проекта реализован собственный генератор кода, имеющий необходимую функциональность для создания федерации и федератов HLA по XMI представлению диаграмм состояний.

1.4 Трансляция UML в SCXML и SCXML в UML

Формат SCXML предназначен для описания диаграмм состояний. В отличие от XMI, использовавшегося ранее, он не содержит лишней информации и имеет очень простую структуру, что позволяет легко работать с файлами в этом формате. Формат SCXML было решено применять в средствах моделирования и верификации наравне с XMI. Если рисовать диаграммы удобнее в графическом редакторе с экспортом в XMI, то дальше работать

удобнее с файлами в более простом формате SCXML. В связи с этим возникает задача реализовать транслятор из XMI в SCXML.

1.4.1 Описание формата SCXML

SCXML основан на XML и содержит следующие теги.

<state> – состояние. Имеет атрибуты id – имя, onentry – инвариант. Может содержать внутри себя другие состояния (state, parallel, final) и переходы (transition). Если состояние содержит другие состояния, то добавляется атрибут initial, в котором записывается id начального состояния.

<parallel> - AND-состояние. Содержит внутри себя теги state, соответствующие дочерним состояниям.

<final> - выходное состояние.

<transition> - переход. Переход начинается в том состоянии, в которое вложен данный тег. Имеет атрибуты: target – id состояния, в которое делается переход, cond – предусловие, event – действие. Содержимое контейнера transition – имя принимаемого сигнала.

1.4.2 Трансляция

Реализован экспорт и импорт в формат SCXML для объектов класса UMLStateMachine. Так как SCXML не поддерживает ссылки на вложенные диаграммы, предварительно внутреннее представление диаграммы состояний предварительно проходит через процедуру Normalize, использовавшуюся и ранее как один из этапов трансляции UML/XMI в UPPAAL.

Ниже, на рисунках 9-11 приведен пример UML-диаграммы, представляющей собой усложненный вариант диаграммы из отчета по второму этапу, иллюстрировавшей синтаксис UML.

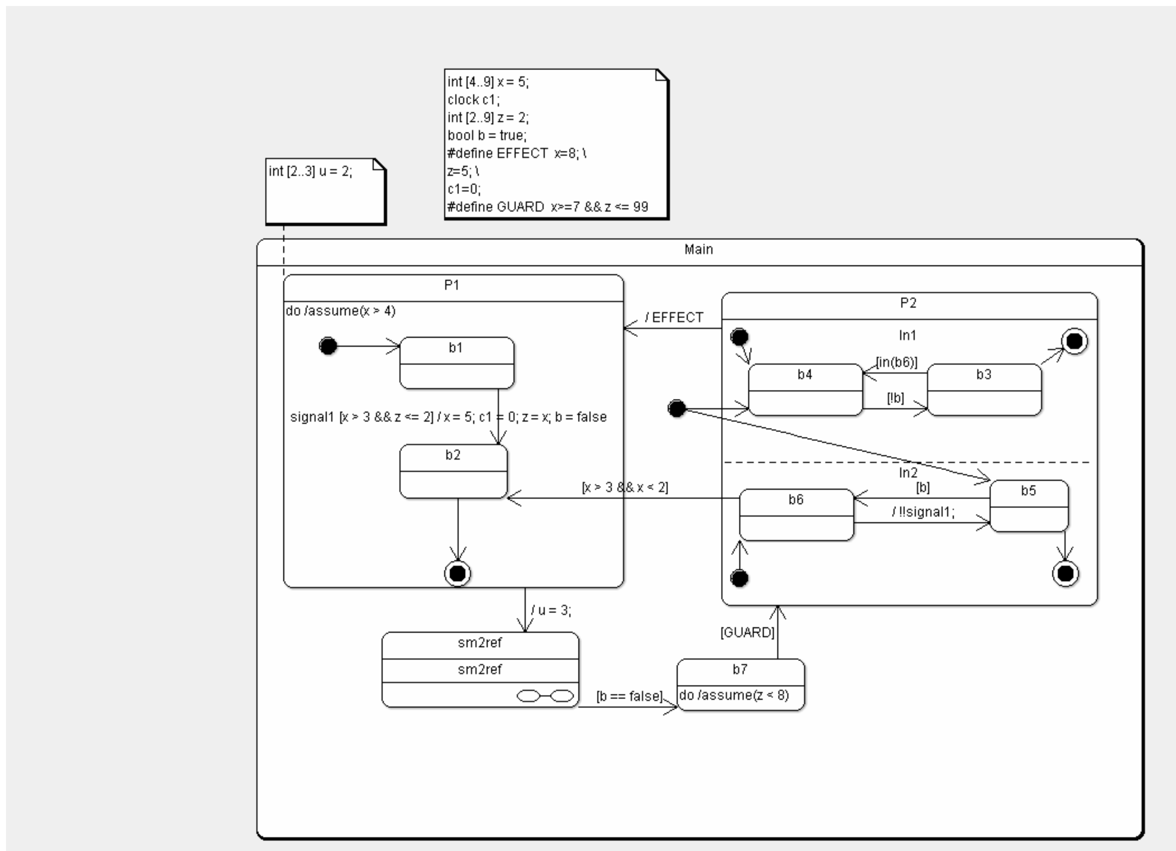

Рисунок 9. Пример UML диаграммы.

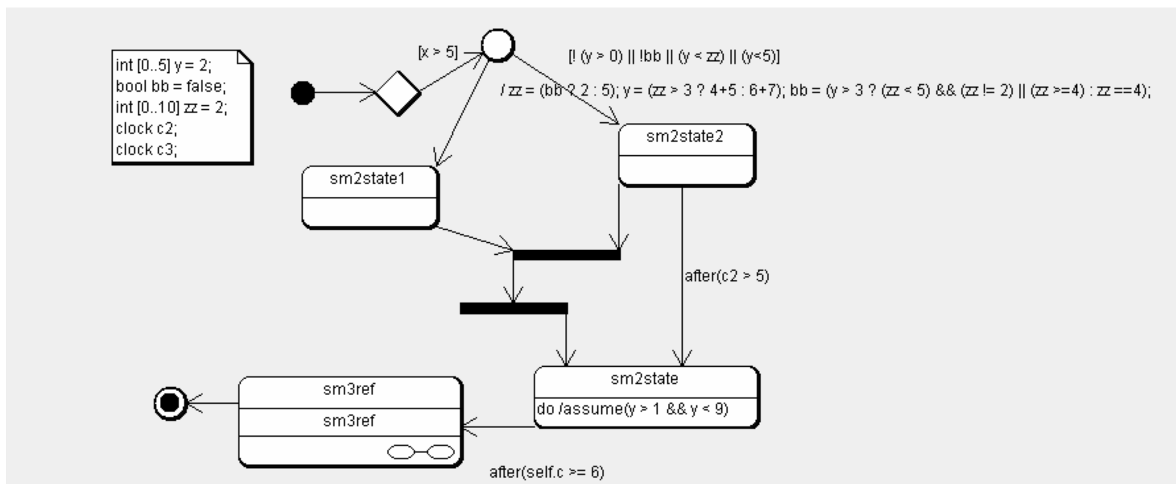

Рисунок 10. Пример UML диаграммы.

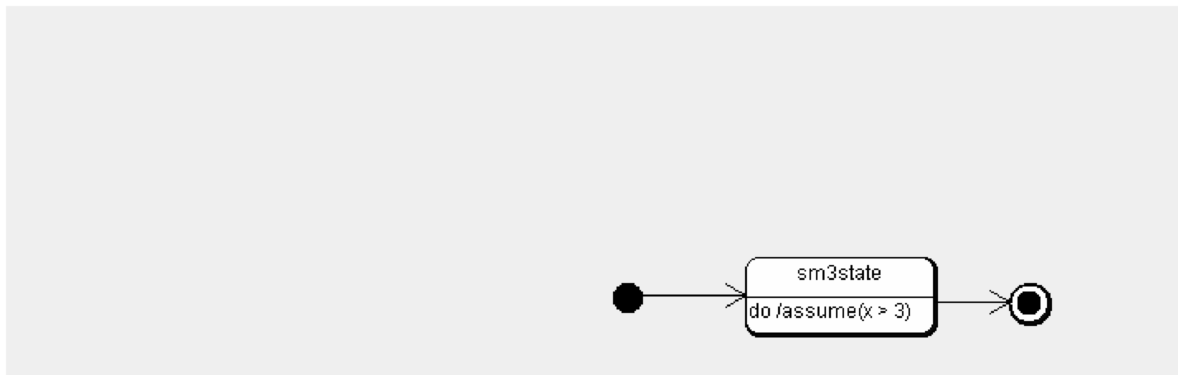

Рисунок 11. Пример UML диаграммы.

Данная диаграмма автоматически преобразуется в следующий SCXML-файл.

```
<?xml version="1.0" ?>
<scxml>
     <!--bool b = true;
bool bb = false;
clock c3;
int [2..3] u = 2;
int [0..1] in_b6 = 0;
int [0..10] zz = 2;
clock c2;
int [0..5] y = 2;
int [4..9] x = 5;
clock c1;
int [2..9] z = 2;
-->
     <state id="top">
          <state id="Main" initial="UNNAMED1">
               <state id="UNNAMED1">
                    <transition
target="b4_extra_entry_extra_entry"/>
                    <transition
target="b5_extra_entry_extra_entry"/>
               </state>
               <state id="b7">
                    <onentry>
```

```
                         assume(z&lt;8)
                    </onentry>
                    <transition cond="x&gt;=7&amp;&amp;z&lt;=99"
target="P2_entry"/>
               </state>
               <parallel id="P2" initial="P2_entry">
                    <state id="In1" initial="e5">
                         <state id="e5">
                              <transition target="b4"/>
                         </state>
                         <state id="b4">
                              <transition cond="!(b)" target="b3"/>
                         </state>
                         <state id="b3">
                              <transition target="e6"/>
                              <transition cond="in_b6 == 1"
target="b4"/>
                         </state>
                         <final id="e6">
                              <transition target="P2_exit"/>
                         </final>
                         <state id="b4_extra_entry">
                              <transition target="b4"/>
                         </state>
                    </state>
                    <state id="In2" initial="e7">
                         <state id="e7">
                              <transition target="after_entry_b6"/>
                         </state>
                         <state id="b6">
                              <transition
event="!!signal1;in_b6=0;" target="b5"/>
```

```
                                    <transition
cond="x&gt;3&amp;&amp;x&lt;2" event="in_b6=0;"
target="b6_extra_exit"/>
                              </state>
                              <state id="b5">
                                    <transition target="e8"/>
                                    <transition cond="b" event="in_b6=1;"
target="b6"/>
                              </state>
                              <final id="e8">
                                    <transition target="P2_exit"/>
                              </final>
                              <state id="after_entry_b6">
                                    <transition event="in_b6=1;"
target="b6"/>
                              </state>
                              <state id="b5_extra_entry">
                                    <transition target="b5"/>
                              </state>
                              <final id="b6_extra_exit">
                                    <transition
target="b6_extra_exit_extra_exit"/>
                              </final>
                        </state>
                        <state id="P2_entry">
                              <transition target="e5"/>
                              <transition target="e7"/>
                        </state>
                        <final id="P2_exit">
                              <transition event="x=8;z=5;c1=0;"
target="b1"/>
                        </final>
                        <state id="b4_extra_entry_extra_entry">
                              <transition target="b4_extra_entry"/>
```

```
                    </state>
                    <state id="b5_extra_entry_extra_entry">
                         <transition target="b5_extra_entry"/>
                    </state>
                    <final id="b6_extra_exit_extra_exit">
                         <transition target="b2"/>
                    </final>
               </parallel>
               <state id="b1">
                    <onentry>
                         assume(x&gt;4)
                    </onentry>
                    <transition cond="x&gt;3&amp;&amp;z&lt;=2"
event="x=5;c1=0;z=x;b=false;" target="b2">
                         signal1
                    </transition>
               </state>
               <state id="b2">
                    <onentry>
                         assume(x&gt;4)
                    </onentry>
                    <transition event="u=3;" target="junction1"/>
               </state>
               <state id="junction1">
                    <transition cond="x&gt;5" target="choice1"/>
               </state>
               <state id="choice1">
                    <transition
cond="!(y&gt;0)||!(bb)||y&lt;zz||y&lt;5" target="sm2state2"/>
                    <transition
event="zz=(bb?2:5);y=((zz&gt;3)?(4+5):(6+7));bb=((y&gt;3)?(((zz&lt
;5)&amp;&amp;(zz!=2))||(zz&gt;=4)):(zz==4));" target="sm2state1"/>
               </state>
               <state id="b1_from_sm2ref">
```

```
                    <transition target="join1"/>
               </state>
               <state id="join1">
                    <transition target="sm2state"/>
               </state>
               <state id="sm2state">
                    <onentry>
                         assume(y&gt;1&amp;&amp;y&lt;9)
                    </onentry>
                    <transition target="sm3state"/>
                    <transition target="sm3state">
                         after(self.c&gt;=6)
                    </transition>
               </state>
               <state id="sm2state2">
                    <transition target="b1_from_sm2ref"/>
                    <transition target="sm2state">
                         after(c2&gt;5)
                    </transition>
               </state>
               <state id="sm2state1">
                    <transition target="b1_from_sm2ref"/>
               </state>
               <state id="sm3state">
                    <onentry>
                         assume(x&gt;3)
                    </onentry>
                    <transition cond="!b" target="b7"/>
               </state>
          </state>
     </state>
</scxml>
```

На рисунке 12 приведена визуализация данной диаграммы в редакторе scxmlgui

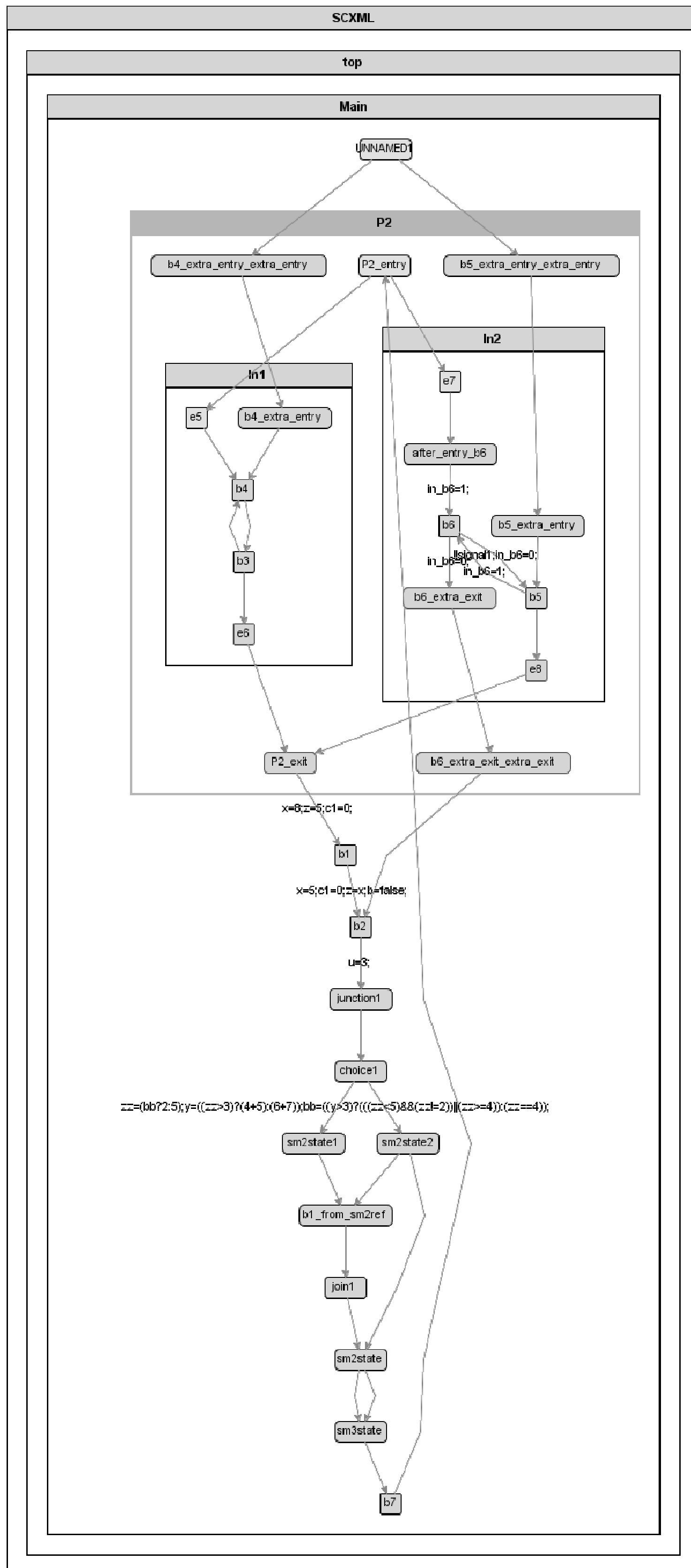

Рисунок 12. SCXML-диаграмма, соответствующая диаграмме с рисунков 9-11.

1.5 Разработка средства визуализации Vis4 с поддержкой формата OTF

1.5.1 Формат трасс OTF и средство визуализации и анализа трасс Vis3

В рамках первого и второго этапов НИР было проведено теоретическое и экспериментальное исследование существующих форматов трасс, средств визуализации и анализа трасс. На основе полученных результатов наиболее предпочтительным для трассировки поведения РВС РВ оказался открытый формат OTF (и в частности формат OTFz с сжатием) с точки зрения расширяемости, гибкости, эффективности чтения/записи и хранения трасс [2]. Для анализа и визуализации трасс в формате OTF существует два средства:

- Vampir 7.3 - проприетарное ПО с достаточно широкими возможностями анализа трасс;
- ViTE 1.2 (Visual Trace Explorer) – развивающееся средство с открытыми исходными кодами, имеющее очень ограниченный функционал и не очень удачный интерфейс.

Таким образом, в настоящее время не существует хорошего, с точки зрения функциональности, открытого программного средства для анализа и визуализации трасс в формате OTF. Поэтому в рамках НИР актуальной стала задача разработки такого программного средства.

Для решения поставленной задачи было предложено взять за основу одно из существующих открытых и доступных программных средств, работающих с трассами в других форматах, и внедрить в него поддержку формата OTF. В качестве такого средства было выбрано средство Vis3, предназначенное для работы с трассами в формате TRC [2]. Vis3 - визуализатор временных диаграмм, разработанный в лаборатории вычислительных комплексов факультета ВМК МГУ для анализа и визуализации трасс функционирования комплексов бортового оборудования (КБО). Vis3 хорошо себя зарекомендовал в течение нескольких лет работы в рамках стенда математического моделирования КБО (СММ КБО).

Данный раздел отчета по НИР посвящен разработке средства визуализации и анализа трасс Vis4 с поддержкой формата OTF и включает в себя:

- Анализ архитектуры программного средства Vis3;
- Обзор систем сборки и выбор системы управления сборкой для Vis4;
- Разработка Vis4 с поддержкой формата OTF;

- Выводы и результаты работы.

1.5.2 Анализ архитектуры Vis3

Средство визуализации и анализа трасс Vis3 является одним из модулей стенда СММ КБО, поэтому тесно взаимосвязано с другими модулями. Диаграмма зависимостей Vis3 от других модулей стенда представлена на рисунке 13.

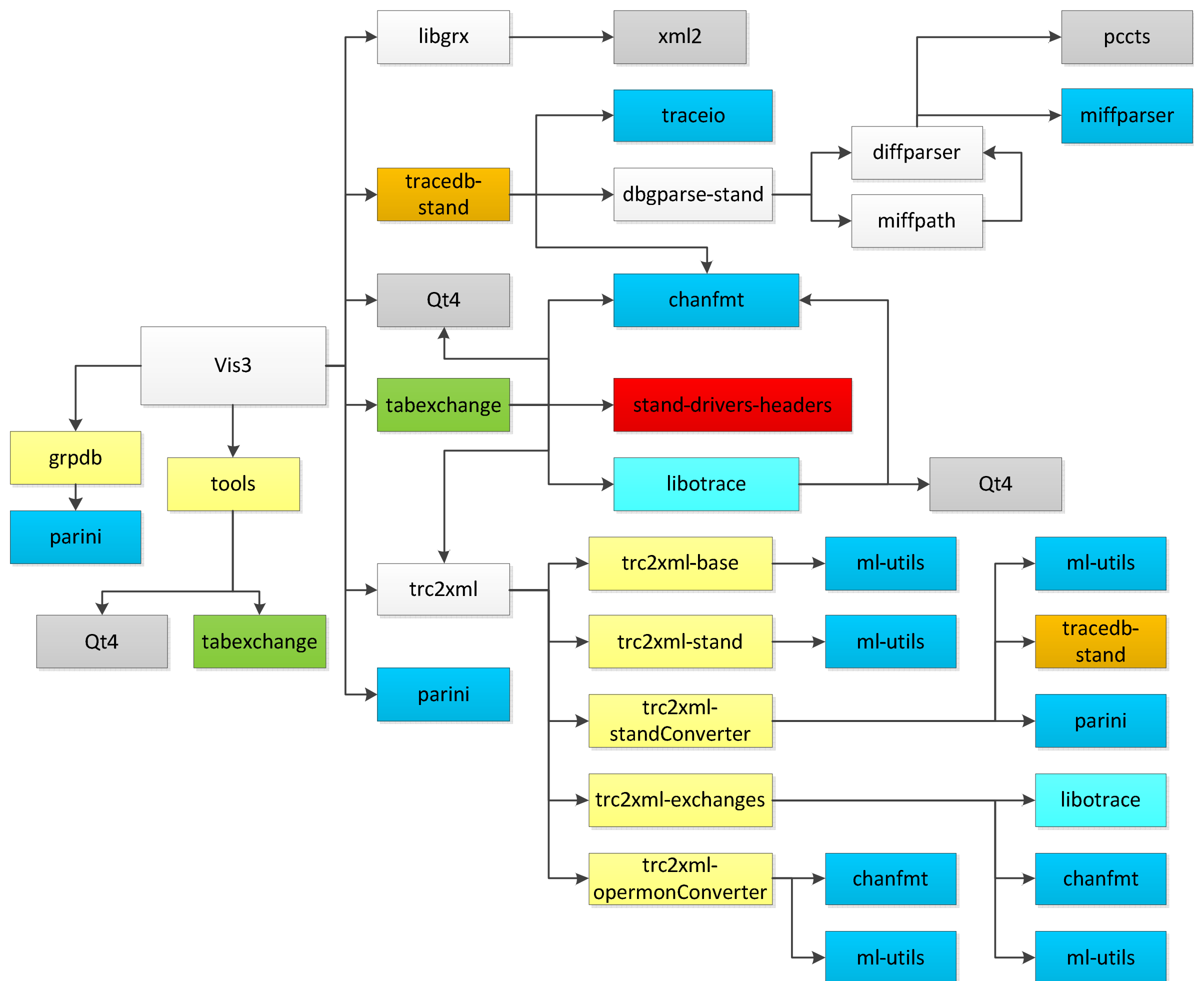

Рисунок 13. Диаграмма зависимости Vis3 от других модулей стенда.

Из рисунка 13 видно, что Vis3 взаимодействует с 21-м модулем стенда и требует наличия 3-х внешних библиотек (pccts, xml2, Qt4). Таким образом, одна из проблем Vis3 – сильная связность с другими модулями стенда.

1.5.1.1 Общая структура Vis3

На самом верхнем уровне находится класс MainWindow, определяющий общий вид окна временной диаграммы Vis3. Он содержит панель инструментов (toolbar), область показа диаграммы (canvas), и область для инструментов (sidebar) (рисунок 14).

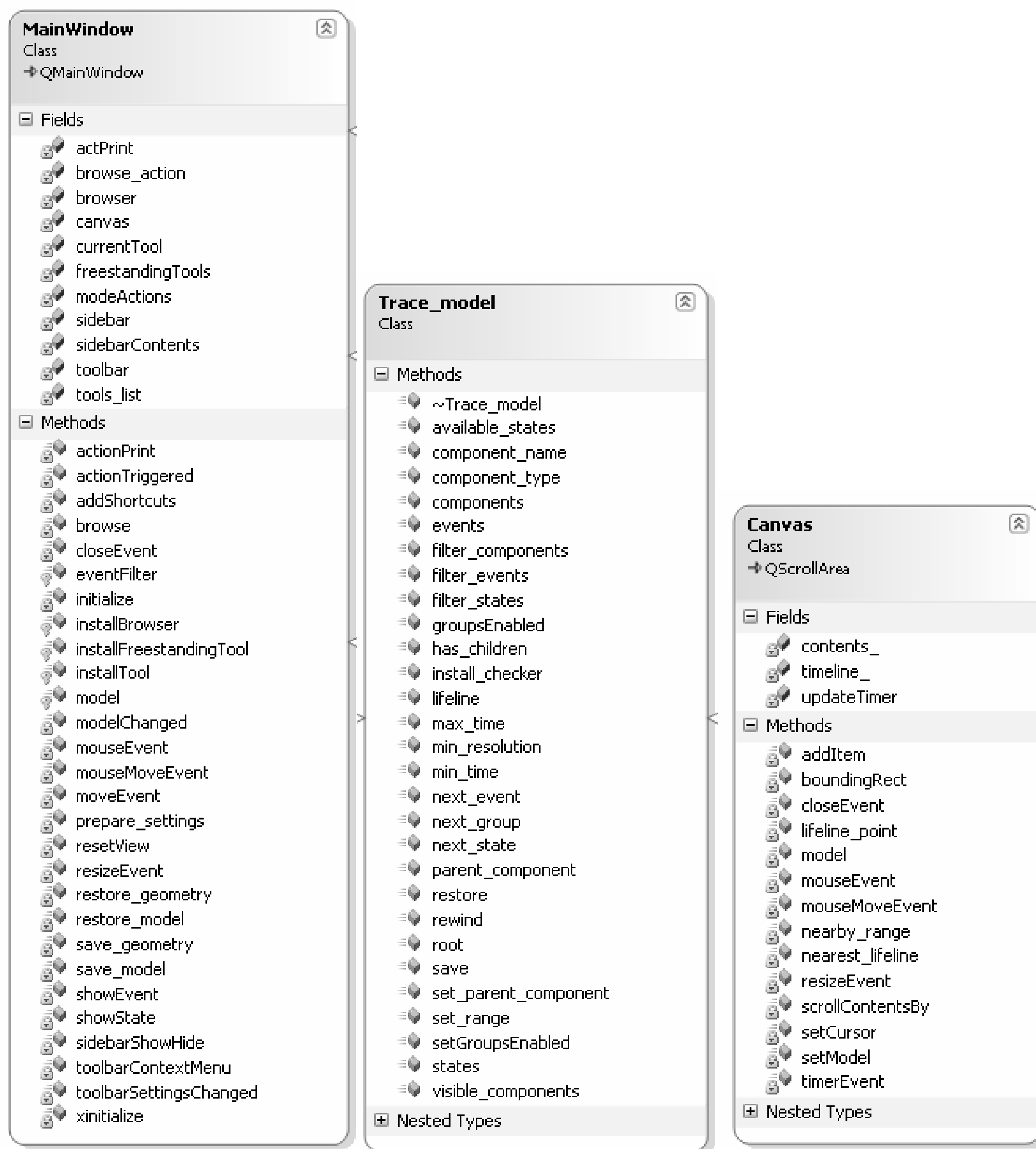

Рисунок 14. Основные классы Vis3.

Классу MainWindow после объявления передается объект класса Trace_model, который описывает начальный вид трассы. После этого создается область показа диаграммы,

набор доступных инструментов, и дальнейшее взаимодействие с пользователем определяется инструментами.

Класс Trace_model – базовый класс, на основе которого для каждого проекта создается дочерний класс, определяющий, откуда брать трассу. Набор инструментов для работы с трассой определяется виртуальным методом initialize. Доступ к трассе осуществляется через абстрактный интерфейс класса Trace_model. Основная функция класса – последовательно выдавать события и состояния, которые необходимо отображать в визуализаторе.

1.5.2.2 Визуализация трассы

За визуализацию трассы в Vis3 отвечает класс Canvas. Он хранит текущий показываемый объект класса Trace_model, и показывает его в виде временной диаграммы. Класс отвечает за визуализацию событий, состояний, и групповых событий (в виде стрелочек). Кроме этого, он предоставляет интерфейс, через который инструменты могут добавлять графические объекты поверх диаграммы, например, выделить найденное событие.

1.5.2.3 Инструменты для работы с трассой

Инструменты для работы с трассой в Vis3 можно разделить на два типа: "управляемые" и независимые. Управляемые элементы отображаются в правой части приложения (sidebar), и в каждый момент времени отображается только один инструмент. На панели инструментов (toolbar) в верхней части Vis3 есть набор кнопок, позволяющих переключать управляемые инструменты.

Независимые инструменты - это кнопки в отдельной области панели инструментов (toolbar) сверху. Обработка нажатия на подобные кнопки задаются автором инструмента, например, может появляться дополнительное окно с графиками.

Взаимодействие между инструментами и визуализатором происходит следующим образом. Объект класса Canvas хранит текущий объект класса Trace_model. Все инструменты имеют к нему доступ. Если инструмент не изменяет текущий вид трассы (например «измеритель расстояний»), то он просто считает трассу, возможно добавляя к Canvas дополнительные графические элементы. Если инструмент в результате работы изменяет текущий вид трассы, то он создает новый объект класса Trace_model и передает его в Canvas. При этом генерируется сигнал, который ловится всеми инструментами и приводит к

обновлению всех элементов графического интерфейса. Например, если изменяется глобальный фильтр событий, то инструмент «поиск событий» делает глобально скрытые события недоступными для поиска.

1.5.2.4 Механизмы расширения Vis3

Одним из достоинств Vis3 является наличие механизмов расширения, при помощи которых архитектура средства может быть расширена для новых проектов и работы средства с другими форматами трасс. Таким образом, для внедрения нового формата трасс в Vis3 в первую очередь было необходимо:

1. определить новый класс, реализующий интерфейс Trace_model. Интерфейс достаточен для показа временной диаграммы, но может быть недостаточен для реализации специфичных возможностей, например показа параметров или поиска по параметрам. В этом случае, в унаследованный класс могут быть добавлены иные методы, необходимые для конкретного проекта.
2. Расширить следующие классы с помощью наследования:
    - класс MainWindow, определяющий общий вид окна временной диаграммы, и переопределить в нём метод initialize.
    - класс Event_model, определяющий информацию о событии, и переопределить в нём метод detailWidget.
    - Класс State_model, определяющий информацию о состоянии компонента РВС РВ.
    - класс Tool, реализующий «управляемый инструмент», и переопределить в нём методы activate и deactivate.

Унаследованные класс, как правило, должны использовать интерфейс класса Canvas для реализации графических элементов. В каждый из унаследованных классов могут быть добавлены дополнительные методы, используемые специфичными инструментами или учитывающие особенности формата трассы.

1.5.3 Обзор систем сборки и выбор системы управления сборкой для Vis4

Для автоматизации сборки проектов традиционно используют системы управления сборкой, такие как make на Unix подобных системах и nmake для компиляторов Microsoft. Для сборки Vis3, как одного из модулей стенда СММ КБО, используется система управления сборкой Boost.build. Однако следует отметить, что для сборки остальных модулей стенда, от которых зависит Vis3, используется система cvslvk, разработанная в лаборатории вычислительных комплексов в 1999. В настоящее время cvslvk устарела, не поддерживается и используется только для работы с уже существующими проектами, поэтому её использование в рамках НИР нецелесообразно. Таким образом, возникла проблема выбора единой системы управления сборкой для получения независимой от стенда версии Vis3 и её дальнейшем использовании при разработке Vis4. К системе сборки можно сформулировать следующие требования:

- Открытость и доступность
- Кроссплатформенность
- Поддерживаемость
- Используемость
- Простота написания скриптов для сборки
- Способность кеширования собираемых файлов для ускорения сборки
- Поддержка языков C, C++
- Поддержка конфигураций сборки
- Автоматический поиск зависимостей в системе

Также, нужно отметить, что могут возникнуть проблемы при совместном использовании в программных средствах НИР нескольких систем сборки, поэтому желательно для всех средств НИР выбрать единую систему управления сборкой.

Ниже будет рассмотрен ряд наиболее известных и используемых в последнее время систем управления сборкой проектов:

- **GNU make**
- **Boost.build**
- **CMake**
- **SCons**

1.5.3.1 GNU make

Последняя версия: 3.82 (28 июля, 2010)

Make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы. Существует несколько версий make. GNU make — входит в большинство дистрибутивов Linux и часто используется в сочетании с GNU build system [13].

Правила. Основное содержание входного файла make – набор правил сборки. Правило языка make задаёт отношение зависимости между файлом-целью (target) и исходными файлами для сборки цели (prerequisites или dependencies в англоязычной документации), а также действие (recipe) - последовательность команд, необходимую для построения цели, и выполняемую, когда хотя бы один из исходных файлов был модифицирован позже файла цели.

Имеется возможность задавать как явные правила для конкретных имен файлов, так и шаблонные неявные правила, позволяющие определить зависимости между файлами с различными суффиксами (расширениями), например, задать правила для получения объектных файлов из Си-файлов.

Переменные входного языка make делятся на:

- определяемые пользователем;
- предопределённые;
- встроенные (например, список "устаревших" относительно цели исходных файлов).

Возможна как просто подстановка значения переменной в правило или действие, так и преобразование этого значения с помощью нескольких предопределённых функций, а также функций, заданных пользователем. Среди предопределённых функций имеется функция eval, которая вызывает разбор и выполнение значения своего аргумента как текста на входном языке make.

Действие представляет собой последовательность строк командного языка shell. Есть ряд специальных возможностей по поддержке: рекурсивного вызова make (выбор нужной версии make, передача значений переменных, подсчёт уровня вложенности). Возможно также параллельное выполнение действий для разных целей, если соответствующая опция задана в командной строке.

Недостатки make. Хотя make и позволяет собирать достаточно сложные проекты (с подкаталогами), на практике для крупных проектов применяются надстройки над make-средствами, генерирующие входные файлы для make – например, GNU autoconf для настройки на целевую платформу и сборки ПО с открытым исходным кодом, CMake, и т.д. Входной язык make содержит "наслоения" многих лет развития.

1.5.3.2 Boost.build

Последняя версия: 1.48.0 (15 ноября, 2011)

Boost — собрание библиотек, расширяющих C++, свободно распространяющихся по лицензии Boost Software License вместе с исходным кодом [14]. Проект был создан после принятия стандарта C++, когда многие были недовольны не включением в стандарт некоторых библиотек. Проект является своего рода «испытательным полигоном» для различных расширений языка и часть библиотек являются кандидатами на включение в следующий стандарт C++.

Средство Boost.build включает в себя:

- язык описания проектов Boost.Jam версии 2 и интерпретатор этого языка "Boost.Build Engine" (исполняемый файл b2);
- набор предопределённых правил сборки (файл Boost-build.jam).

Язык Boost.Jam поддерживает следующие основные конструкции языка:

- правило (rule) – по сути, определение функции; Основное (но не единственное) назначение правила – задать зависимость целевого файла от исходных файлов с помощью правил DEPENDS и INCLUDES.
- вызов правила;
- действие (action) - связывается с правилом и задаёт конкретные команды при применении правила для построения цели из исходных файлов;

- управляющие конструкции if, for;
- переменные (значения их – списки строк); набор операций присваивания, сравнения, и др.
- свойства (features) – которые используются например для задания путей к заголовочным файлам, значений переменных препроцессора, варианта сборки для отладки или оптимизации, используемого компилятора.
- встроенные (builtin) переменные и правила.

Большинство правил сборки в Boost.build имеют следующие аргументы:

- имя цели;
- список исходных файлов и других целей (например, библиотек для исполняемого файла);
- список требований (набор значений свойств, которые должны присутствовать для того, чтобы правило сработало);
- набор значений свойств, принимаемых по умолчанию, если не заданы явно;
- набор значений свойств (usage-requirements), которые устанавливаются при выполнении правила и распространяются на подцели.

Отличительными чертами системы сборки Boost.build являются:

- **Простое и высокоуровневое описание процесса сборки проекта.** В большинстве случаев достаточно указать набор исходных файлов и цель сборки.
- **Переносимость** между разными платформами.
- **Варианты сборки.** Boost.build поддерживает возможность одновременного создания нескольких сборок в разных директориях (например, сборки release и debug за один вызов).
- **Глобальные зависимости.** Независимо от того, в какой директории собирается проект, Boost.build проверяет все зависимости во всем проекте, предотвращая тем самым проблему несогласованности бинарных файлов. Поэтому удобно использовать один Boost.build-проект в рамках другого.
- **Использование требований.** В Boost.build поддерживается возможность уточнения свойств сборки цели, когда это необходимо. Эти свойства применяются всякий раз при использовании данной цели.

- **Автономность.** Boost.build зависит только от компилятора C, поэтому Boost.build можно включать в разрабатываемый проект.

1.5.3.3 CMake

Последняя версия: 2.8.6 (4 октября, 2011)

Система сборки CMake – кроссплатформенная система сборки с открытым исходным кодом, разработана и поддерживается компанией Kitware. CMake позволяет задать параметры процесса сборки в текстовом файле и на основе этого файла генерирует входной файл для выбранной системы сборки (Unix make, Microsoft Visual Studio, Borland make). Входной язык CMake содержит набор примитивов, ориентированных на задачи сборки, в частности:

- объявить проект;
- добавить в проект исполняемый файл с указанием списка исходных файлов;
- добавить в проект файл библиотеки с указанием списка исходных файлов;
- включить подкаталог в проект;
- задать свойства (например, опции компиляции) для одного или нескольких файлов.

В CMake поддерживается возможность использовать макропеременные, присваивать им строковые значения и списки строк. CMake поддерживает набор предопределённых переменных, среди которых можно выделить две важные группы: переменные, определяющие имена файлов и пути в файловой системе, например, CMAKE_BINARY_DIR, и переменные, определяющие целевую платформу, например, WIN32.

Во входном языке CMake предусмотрены управляющие операторы if и foreach. Имеется также возможность задания и вызова макроопределений. Примечание. Набор базовых операторов входного языка может быть расширен за счет модификации исходного текста CMake.

1.5.3.4 SCons

Последняя версия: 2.1.0 (9 сентября, 2011)

SCons — это инструмент для автоматизации сборки программных проектов, разработанный как замена утилиты make с интегрированной функциональностью

аналогичной autoconf/automake. SCons автоматически анализирует зависимости между исходными файлами и требования адаптации к операционной системе исходя из описания проекта, и генерирует конечные бинарные файлы для установки на целевую ОС. SCons предоставляет следующие возможности:

- SCons использует язык Python в качестве основы, поэтому конфигурация проектов и инструменты для управления процессом сборки являются сценариями на Python.
- Встроенная поддержка C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, Objective-C.
- Для языков C, C++ и Fortran автоматически анализируются зависимости.
- Сборка из репозиториев систем контроля версий. Встроенная поддержка получения исходных кодов из SCCS, RCS, CVS, Subversion, BitKeeper и Perforce.
- Поддержка проектов Microsoft Visual Studio .NET и более ранних версий Visual Studio, с возможностями генерации файлов .dsp, .dsw, .sln и .vcproj.
- Обнаружение изменения содержимого файлов по контрольным суммам MD5, наряду с традиционным обнаружением изменений по времени записи файла.
- Возможность параллельной сборки.
- Встроенная возможность поиска необходимых для сборки файлов (#include файлы, библиотеки, и т. д.).
- Способность кеширования собираемых файлов для ускорения параллельной сборки — подобно ccache, но для любых типов файлов.
- Изначальная поддержка кросплатформенности. В настоящее время работает в Linux и др. POSIX-системах, Windows NT, Mac OS X, OS/2.

1.5.3.5 Вывод

Сравнительный анализ обозначенных систем сборки приводится в таблице 2.

Таблица 2. Сравнительный анализ систем управления сборкой.

Критерий	make	Boost.Build	CMake	Scons
сведения о разработчике (правообладателе)	Free Software Foundation	David Abrahams and Vladimir Prus	Компания Kitware	Steven Knight
Открытость (Лицензия)	Лицензия GNU GPL версии 3.	Лицензия Boost Software License версия 1	Лицензия BSD	Лицензия MIT
Используемость и апробированность средства в эксплуатации	Известно более 30 лет, широко используется для сборки ПО как с открытым, так и с закрытым кодом	Известно с 2002 г.	Известно не менее 6 лет. Успешно используется для сборки таких крупных проектов как KDE, MySQL, **CERTI**	Известен с 2008 года. Успешно используется для сборки таких проектов как:VMWare, Google Chrome, Blender и др.
Кросс-платформенность	да	да	да	да
1. Языковые возможности				
1.1. Переменные	Строковые, списки строк	Строковые, списки строк	Строковые, списки строк	Строковые, списки строк
1.2. Управляющие структуры	if в теле файла управления сборкой, if и foreach в функциях	if, foreach	if, foreach	if, foreach
1.3. Средства задания зависимостей	Явные правила для конкретных файлов, неявные правила для шаблонов имен файлов	Гибкие средства задания зависимостей, в том числе управляемое пользователем связывание цели с подцелью	Набор правил для предопределённых целей, окончательный список зависимостей формируется после генерации файла управления сборкой; есть конструкция для задания определяемых пользователем типов целей.	Гибкие средства задания зависимостей
1.4. Средства задания действий	Последовательность команд на языке shell	Императивный язык программирования	Набор предопределённых действий.	Сценарии на языке Python
2. Возможность параллельного выполнения задач сборки	Есть	есть	Есть	Есть
3. Оценка удобства поддержки сложных проектов	Малая степень удобства	Большая степень удобства	Большая степень удобства	Большая степень удобства

4. Сборка проекта с иерархией подпроектов	Есть поддержка рекурсии по каталогам. Но для практической работы нужно предварительное конфигурирование.	Автоматическое распространение параметров сборки по подцелям; удобство ссылок на подпроекты	Поддерживается, автоматический поиск CMakeList	Поддерживается
5. Поддержка различных конфигураций сборки	Для реальной работы нужно предварительное конфигурирование Make-файлов дополнительными средствами (GNU autoconf)	Поддерживается	Поддерживается	Поддерживается

На основе приведённого анализа, можно сделать следующий вывод

- Система сборки make значительно уступает boost.build, cmake и scons.
- Boost.build не получил широкого распространения в других проектах;
- Cmake и scons очень близки по поддерживаемым возможностям;
- Для сборки CERTI используется Cmake, который также широко используется для сборки многих крупных проектов и достаточно прост в понимании и освоении.

Таким образом, для Vis4 было решено использовать систему управления сборкой CMake и рекомендовано использовать её для других программных средств в рамках НИР.

1.5.4 Разработка Vis4 с поддержкой формата OTF

Для работы с трассами в формате OTF средство Vis4 использует библиотеку OTF 1.10 (от 15.11.2011). На основе механизмов расширения Vis3, описанных в разделе 1.5.2.4, для Vis4 был разработаны класс OTF_trace_model, реализующий методы виртуального класса Trace_model, дочерние классы OTFMainWindow, OTF_event_model, OTF_state_model и незначительно изменён ряд классов для работы с трассами в формате OTF.

Средство Vis4 тестировалось на небольших сгенерированных тестовых трассах в формате OTF и трассах Test_2010_06_01, Test_2010_10_28, Pohod88, Kingstown в формате OTF, полученных после конвертирования реальных трасс в формате TRC. Примеры визуализации трасс Pohod88 и Kingstown в Vis4 приведены на рисунках 15,16.

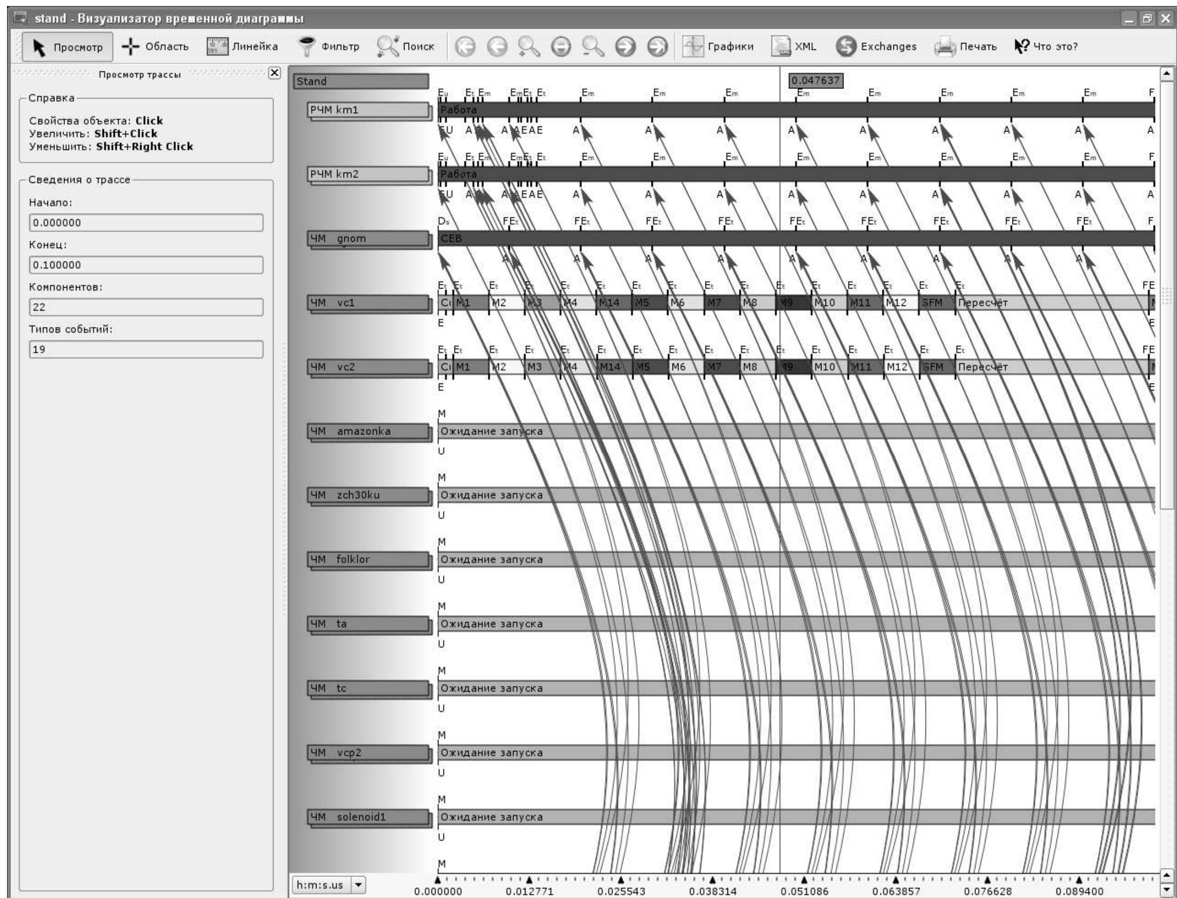

Рисунок 15. Визуализация трассы pohod88.otf в Vis4.

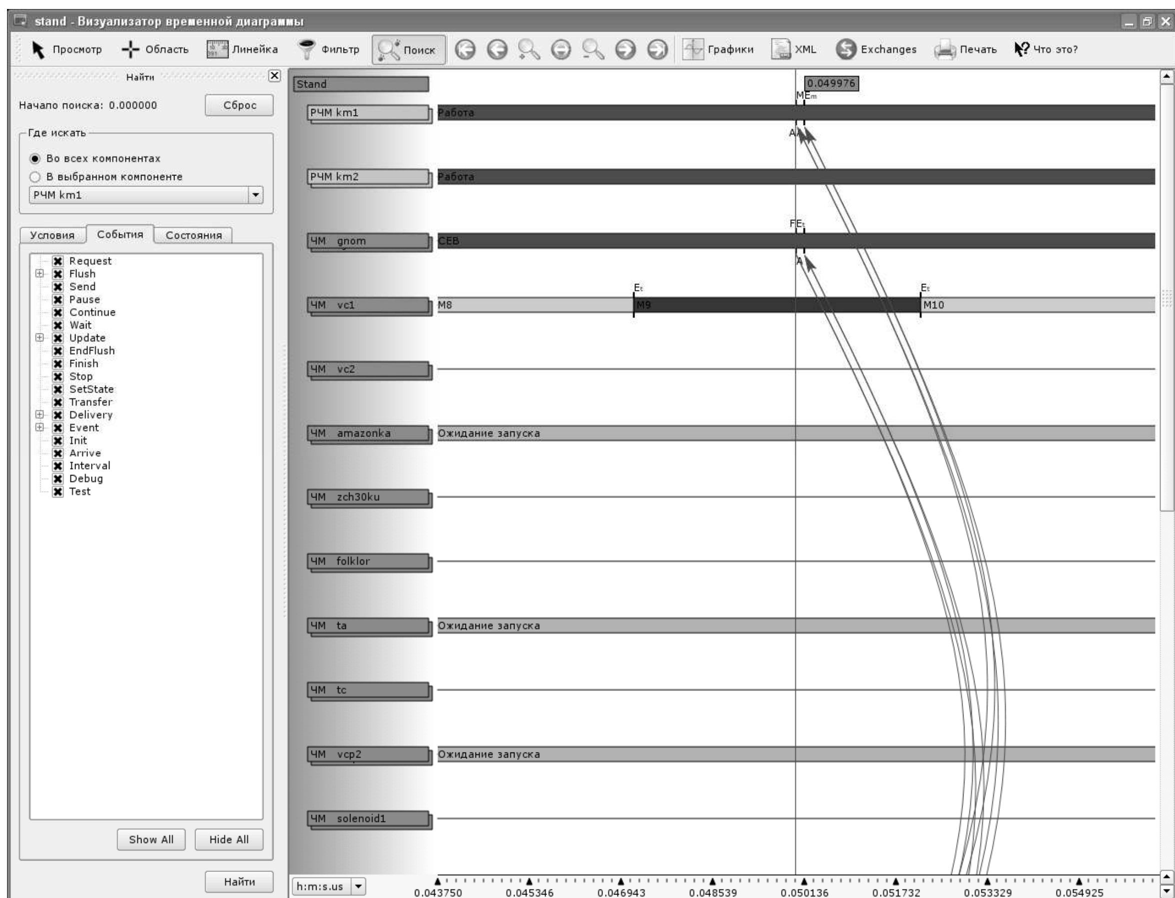

Рисунок 16. Визуализация трассы Kingstown.otf в Vis4.

1.5.5 Выводы и результаты работы

В ходе данного этапа НИР был было принято решение о разработке средства Vis4 для визуализации и анализа трасс (на основе средства Vis3) с поддержкой открытого формата OTF. Для решения поставленной задачи:

- была проанализирована архитектура программного средства Vis3 и её возможности по встраиванию поддержки новых форматов трасс;
- был проведен обзор систем управления сборкой проектов и в качестве единой для всех программных средств НИР (в том числе и для Vis4) рекомендовано использовать систему CMake;
- было разработано программное средство Vis4, поддерживающее формат OTF.

2 Разработка модели РВС РВ

В данном разделе описывается модель РВС РВ, разработанная на данном этапе работы. В разделе 2.1 приведено общее описание модели. Разделы 2.2, 2.3, 2.4 и 2.5 содержат описание вычислителей C_1, C_2, C_3 и C_4, входящих в состав модели.

2.1 Общее описание модели

РВС РВ 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.

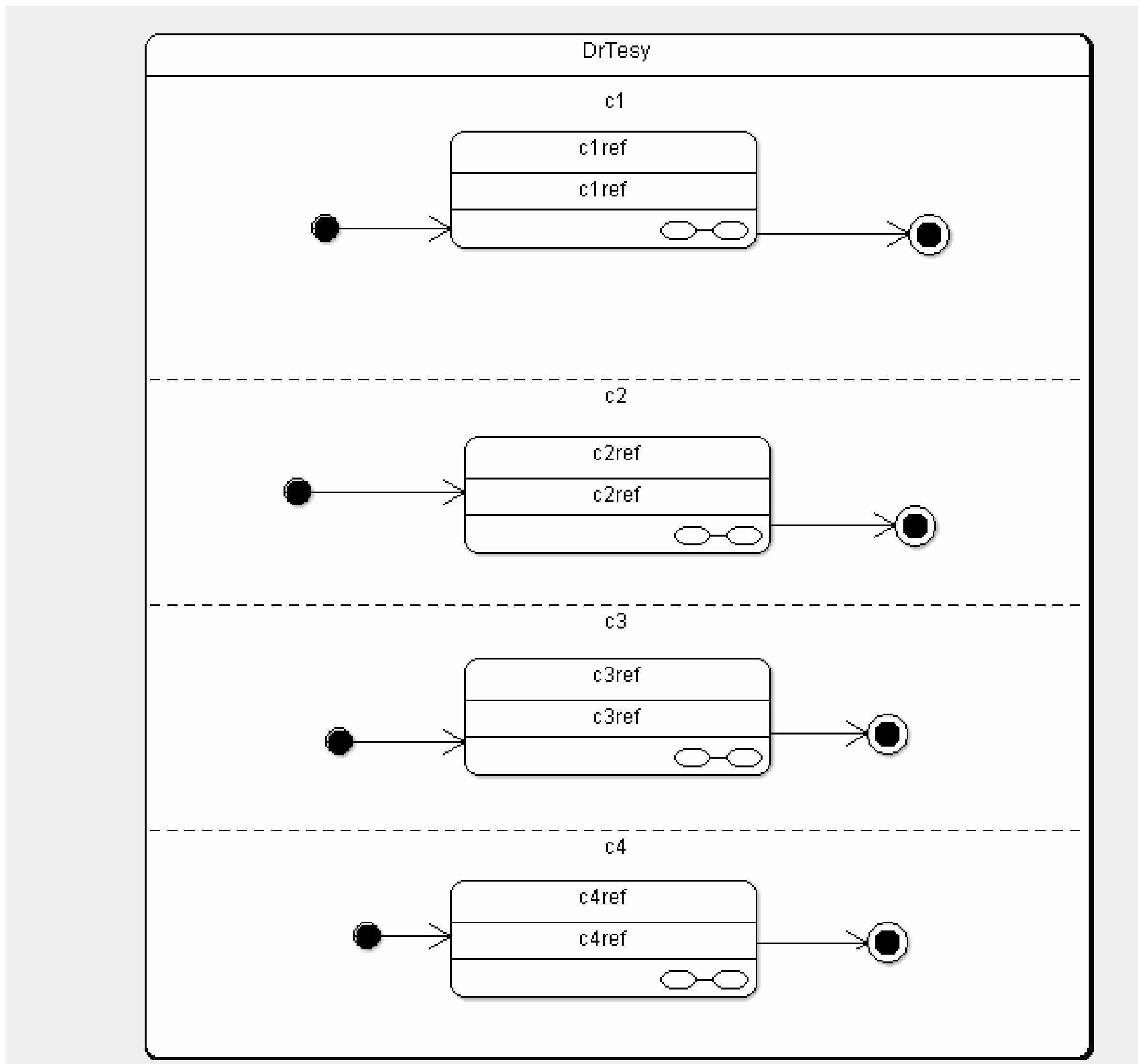

Рисунок 17. Глобальная диаграмма.

Глобальная диаграмма модели представлена на рисунке 17. Данная модель абстрагируется от конкретных параметров полета и специфицирует преимущественно на коммуникации между процессорами. Поведение программы при вычислениях задано в виде набора состояний, соответствующих различным операциям, но их функционирование более детально не определяется.

Типы данных, используемых в рассматриваемой распределенной системе таковы.

Com_Inter_C2 – прерывания для процессора C_2.

Send_status2 = 0

Send_Param1 = 1

Send_Param2 = 2

Receive_param = 3

Init_C2 = 4

Com_Inter_C3 – прерывания для процессора C_3.

Send_status3 = 0

TVS_com = 1

HVP_com = 2

FLR_com = 3

Init_C3 = 4

Com_Inter_C4 – прерывания для процессора C_4.

Send_status4 = 0

Send_data = 1

Receive_data = 2

AAR_radiate = 3

Permiss_radiate1 = 4

Permiss_radiate2 = 5

Init_C4 = 6

Com_Inter_Sensor – прерывания для сенсоров.

Send_data = 0

Send_statusSensor = 1

Mode1 – режим 1.

Cancel = 0

FLR_TVS = 1

HVP = 2

Mode2 – режим 2.

SRNS = 0

Next_WP = 1

Online_WP = 2

Manual = 3

Mode3 – режим 3.

Horizontal = 0

Aux_PRA = 1

Main_AAR = 2

None = 3

Глобальные переменные

```
int [0..4] COMC2_value = 0;
int [0..4] COMC3_value = 0;
int [0..6] COMC4_value = 0;
int [0..10] C4_data1 = 0;
int [0..10] C4_data2 = 0;
int [0..10] C4_data3 = 0;
int [0..10] SENSOR_value = 0;
int [0..10] Sensor_data = 0;
int [0..2] R1_data = 0;
int [0..3] R2_data = 0;
int [0..3] R3_data = 0;
bool sharedMemory = false;
bool REQUEST_C2 =  false;
bool REQUEST_C3 =  false;
```

2.2 Процессор C_1

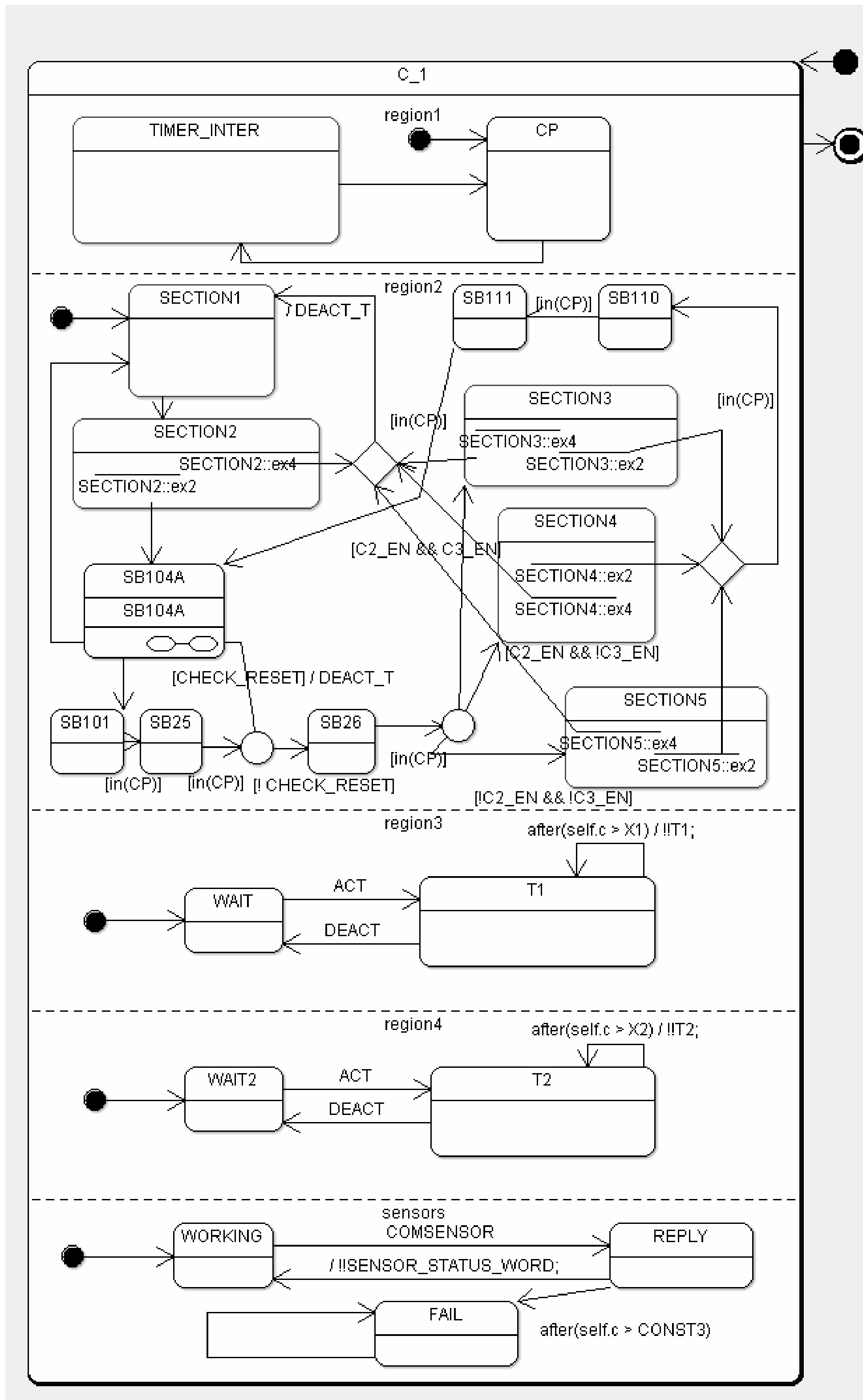

Рисунок 18. Общая диаграмма C_1.

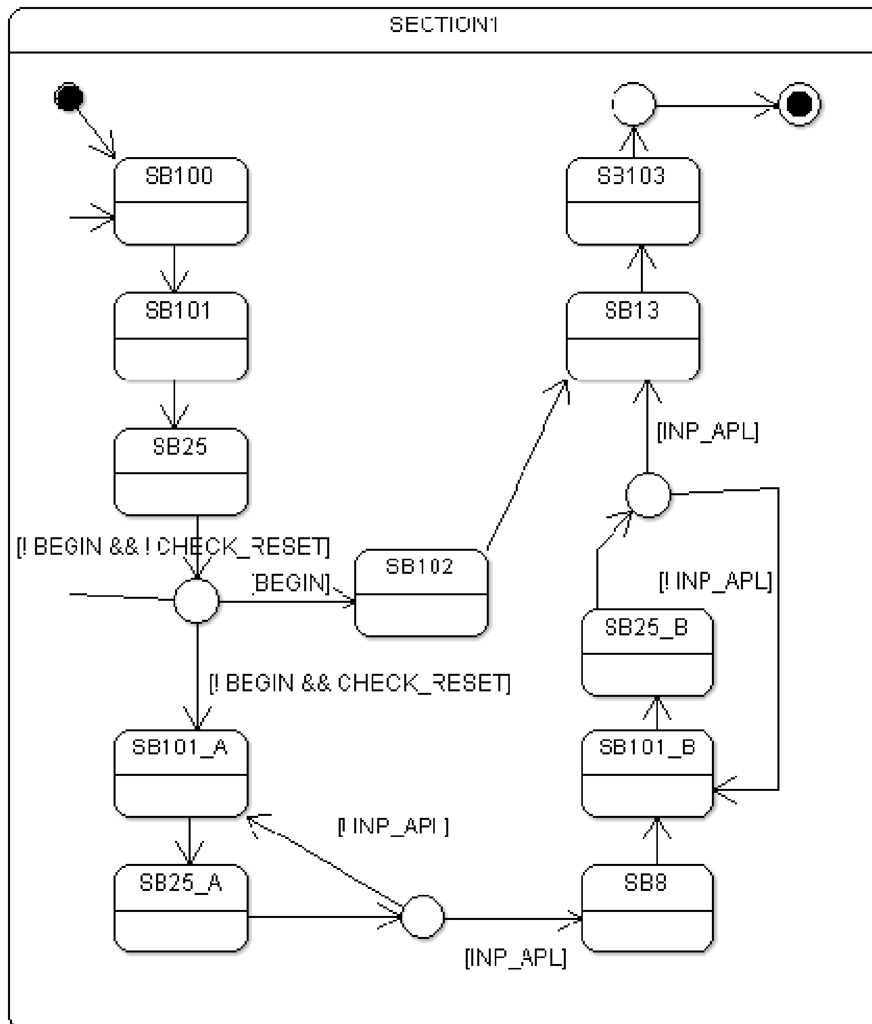

Рисунок 19. Первая секция главного цикла C_1.

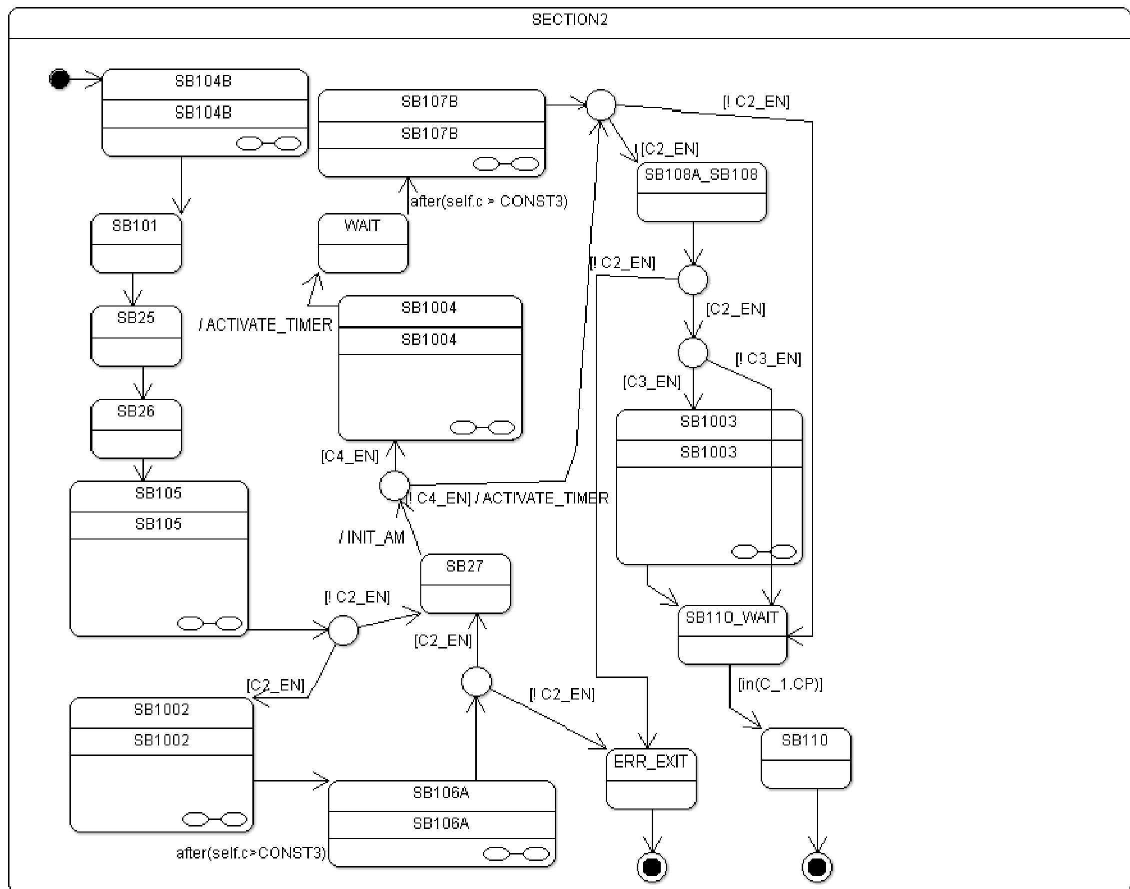

Рисунок 20. Вторая секция главного цикла C_1.

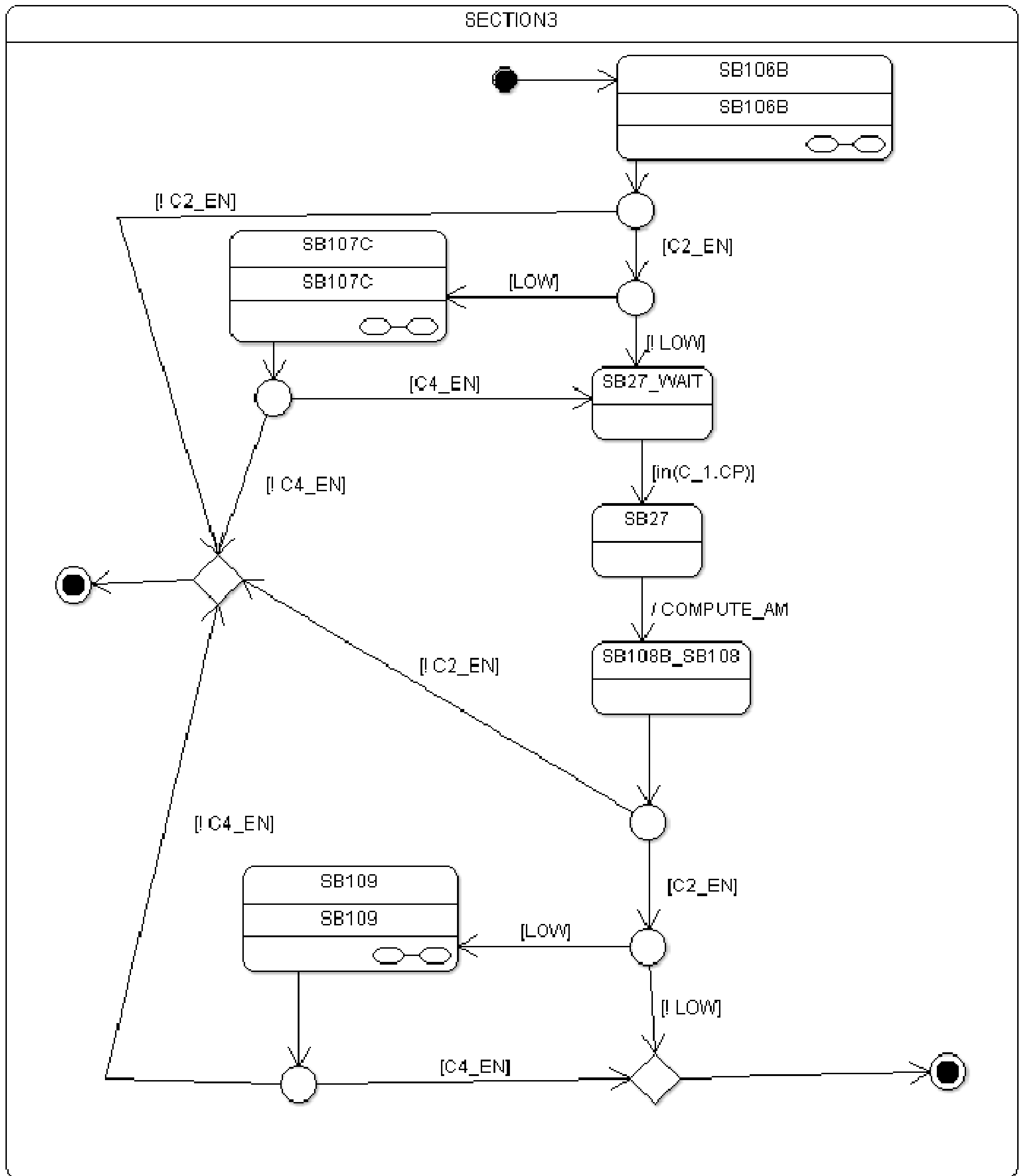

Рисунок 21. Третья секция главного цикла C_1.

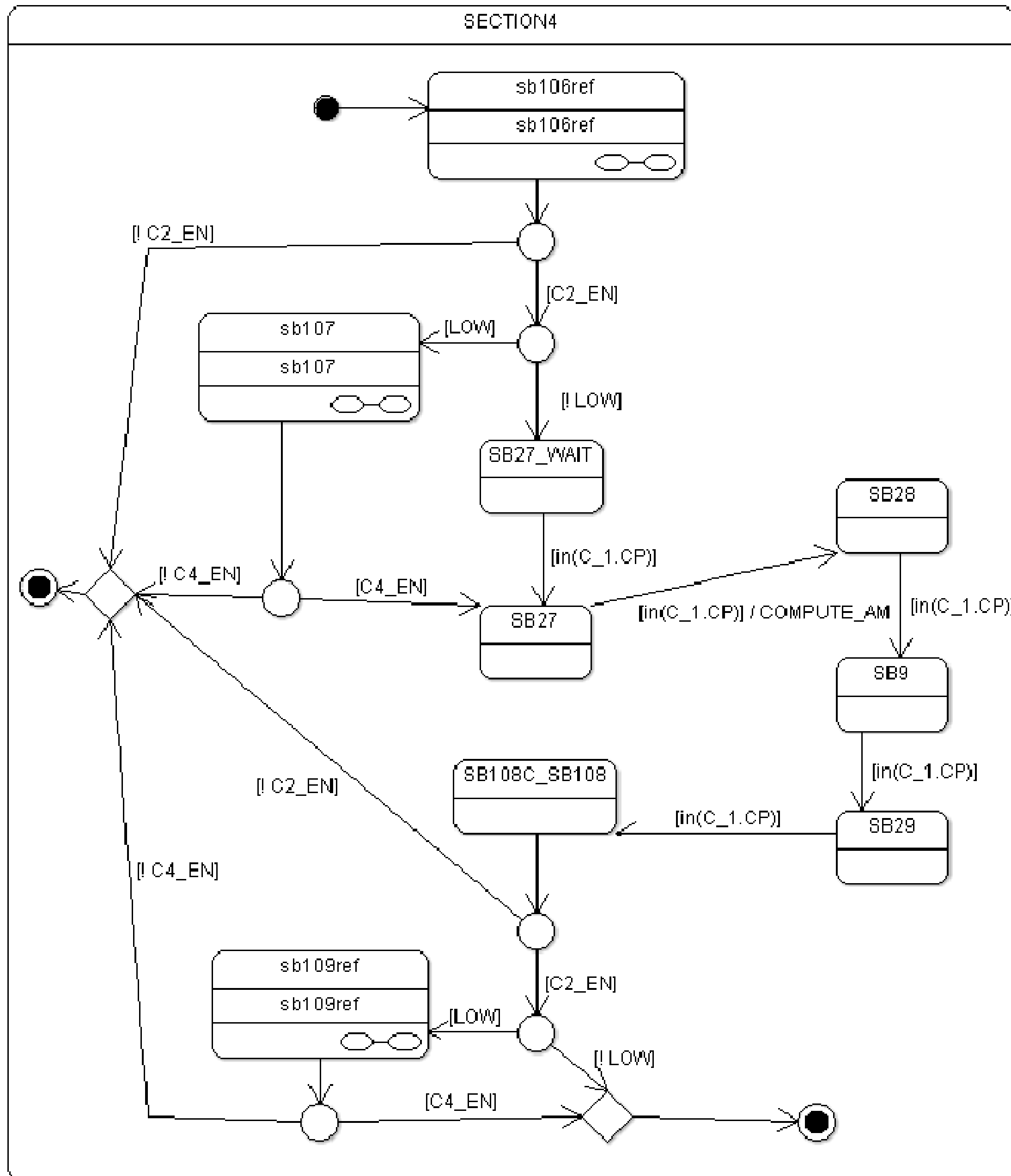

Рисунок 22. Четвертая секция главного цикла C_1.

Рисунок 23. Пятая секция главного цикла C_1.

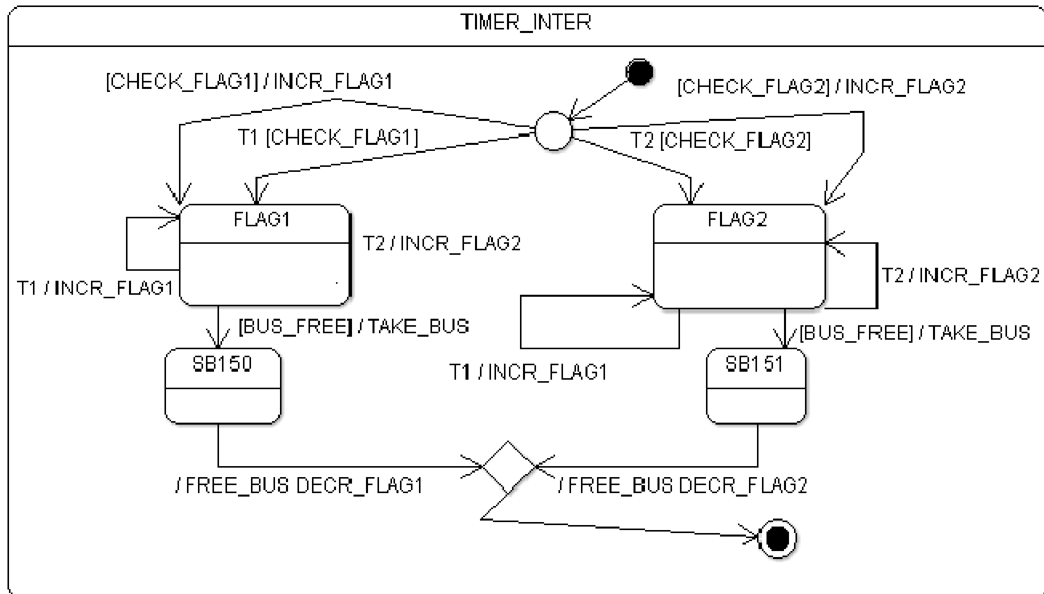

Рисунок 24. Прерывания по таймеру в C_1.

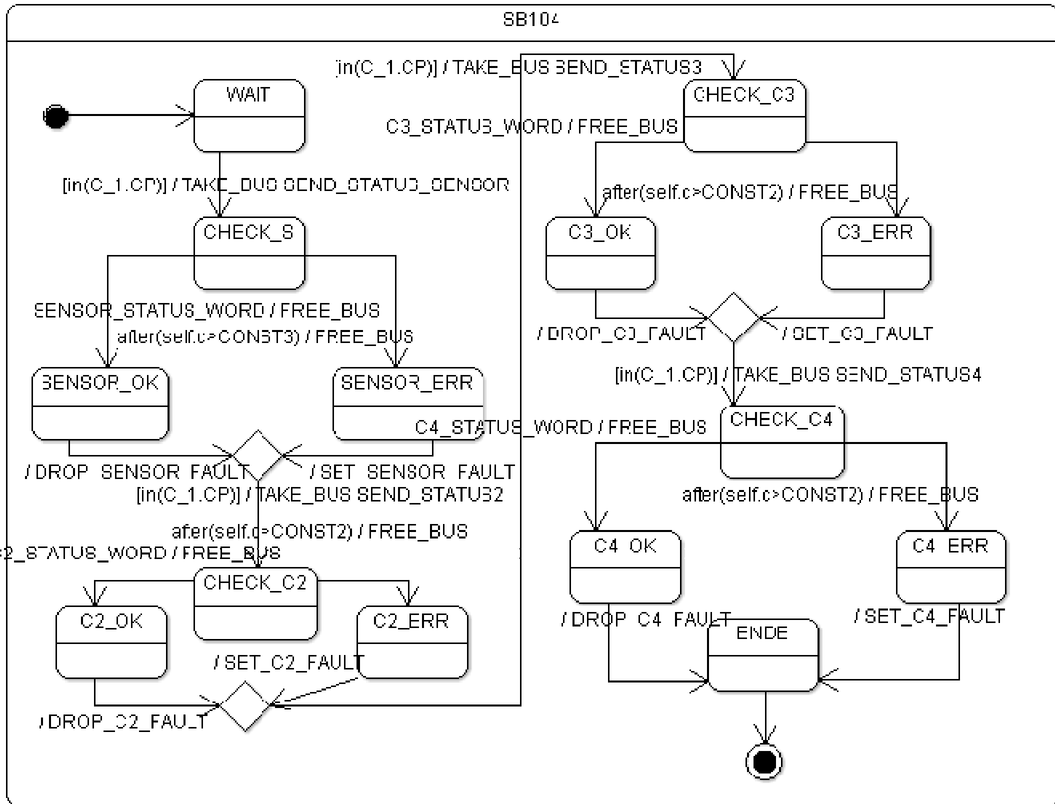

Рисунок 25. Специальный блок 104.

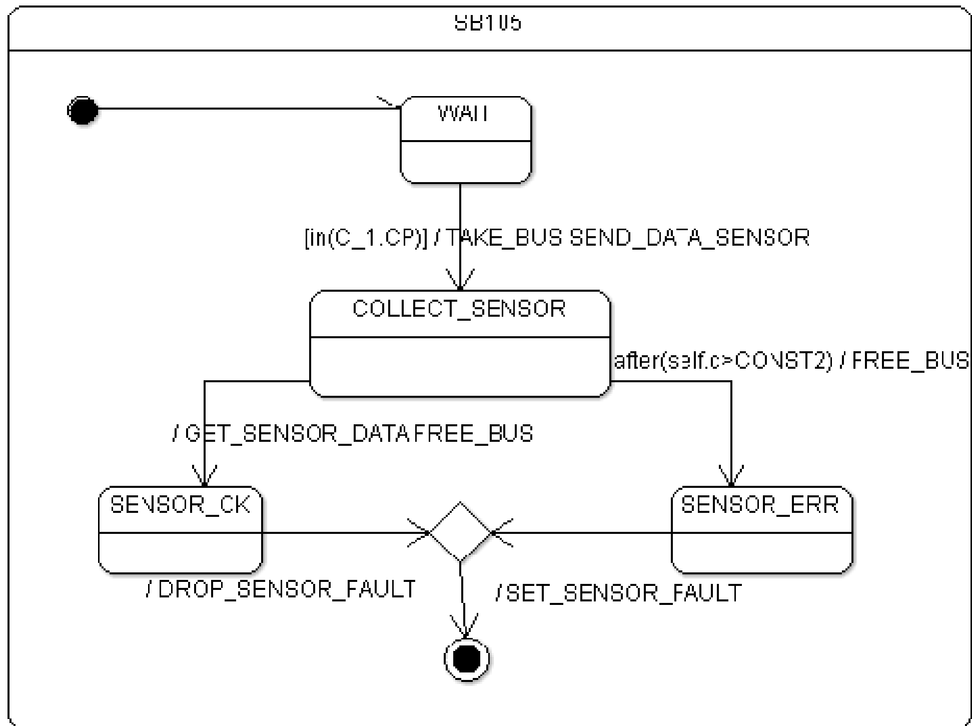

Рисунок 26. Специальный блок 105.

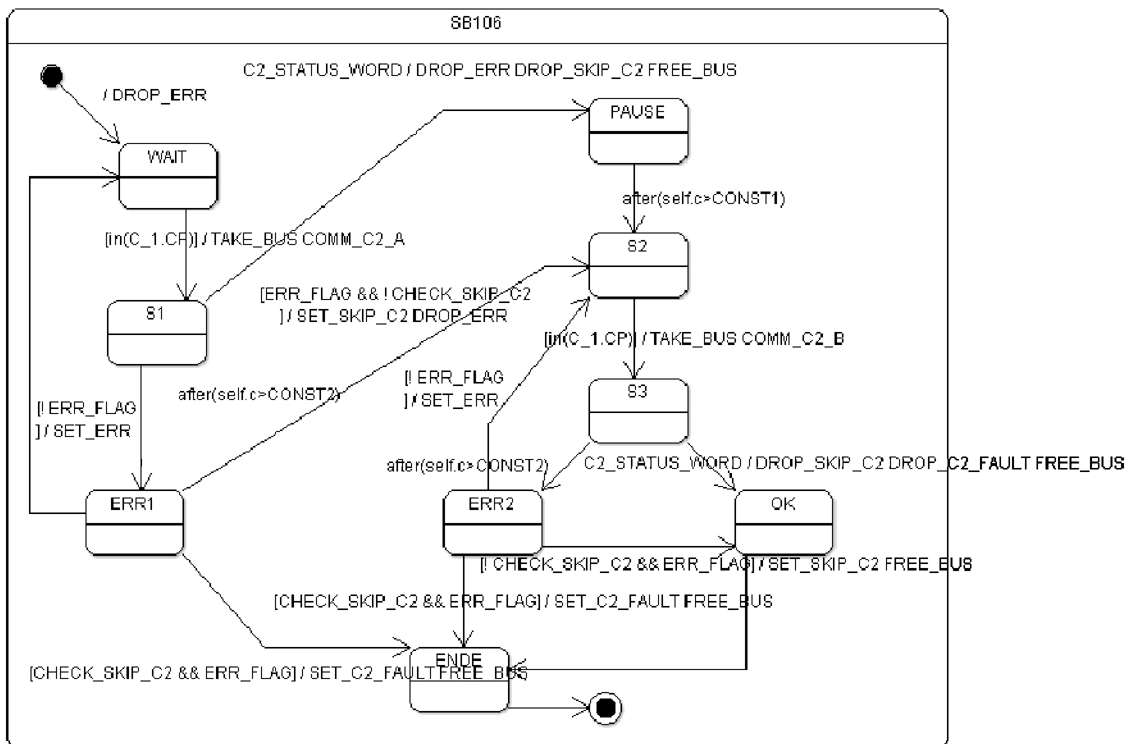

Рисунок 27. Специальный блок 106.

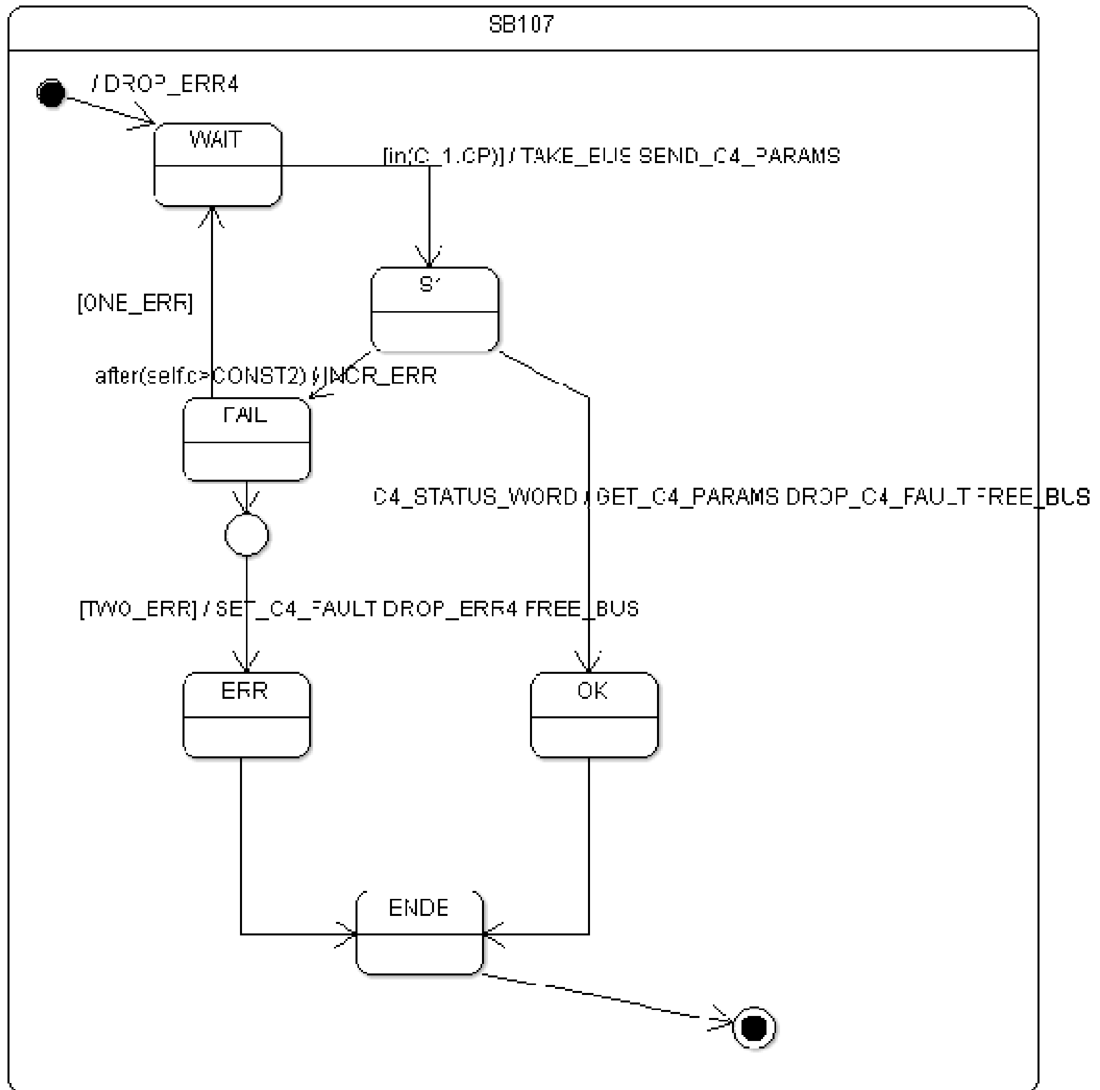

Рисунок 28. Специальный блок 107.

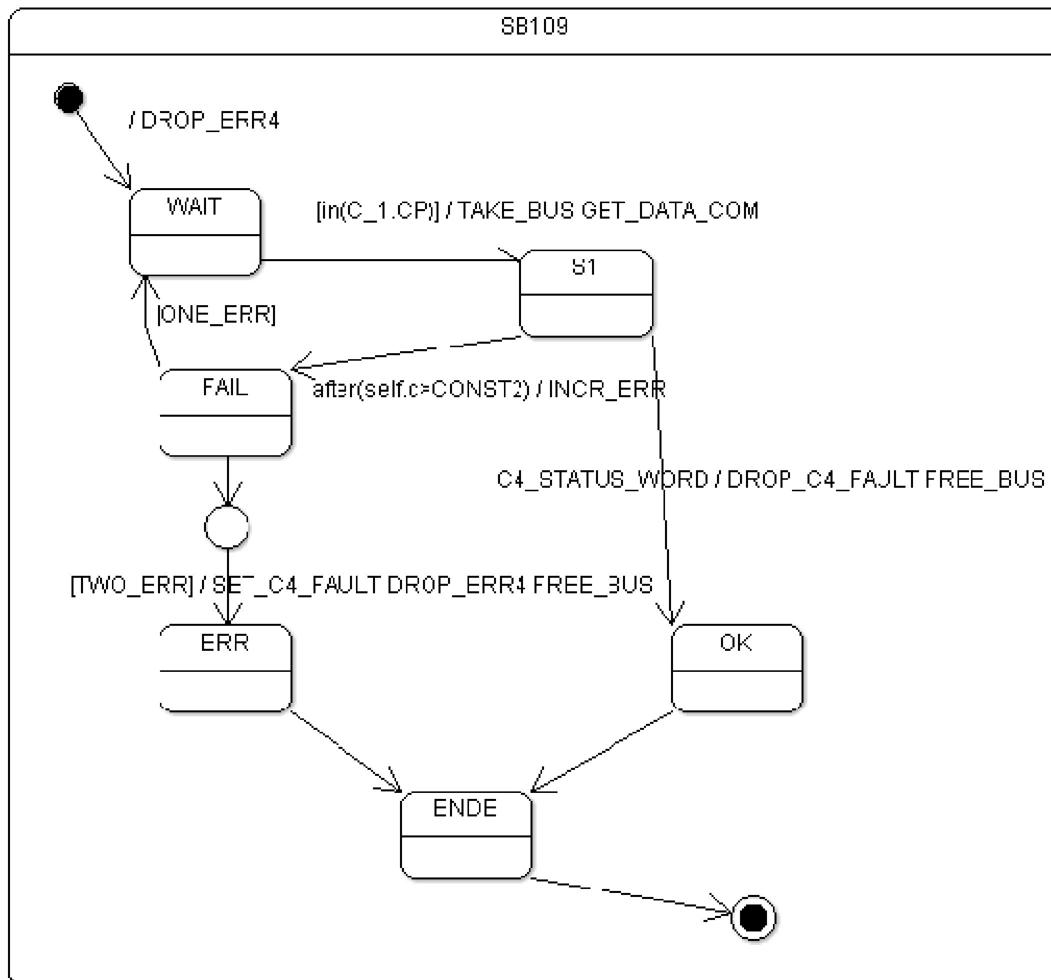

Рисунок 29. Специальный блок 109.

Модель процессора C_1 представлена на рисунках 18-29. После инициализации процессор C_1 работает в бесконечном цикле. Параллельно с основным циклом выполняется процесс, изображенный на первом из параллельных регионов. Этот процесс определяет, в каком режиме находится процессор: в рабочем (CP) или в режиме прерывания (TIMER_INTER). Благодаря этому процессу можно промоделировать прерывание по таймеру: ко всем переходам в основном цикле добавлено предусловие in(C_1.CP), гарантирующее, что процессор находится в обычном режиме. Если процессор обрабатывает прерывание, то основной цикл дожидается окончания обработчика. Процессор C_1 содержит два таймера, которые инициируют прерывания с частотой, определяющейся константами X1 и X2 соответственно. Значения констант равны 1 и 50 миллисекундам.

Главный цикл содержит пять основных секций, которые описывают различные сценарии поведения в зависимости от того, работают ли другие вычислители. Секции 1 и 2 (Section1, Section2) используются для загрузки данных и инициализации удаленных вычислителей. В экстренном случае, когда C_2 или C_3 теряют работоспособность, C_1 берет на себя их вычислительные задачи. Соответствующие алгоритмы содержатся в секциях 3, 4, 5. Переход в эти секции происходит при выполнении соответствующих предусловий (C2_EN, C3_EN), а флаги в них устанавливаются в блоках, отвечающих за коммуникации (SB106-109).

Возможен также перезапуск всей системы. Соответствующий флаг проверяется в главном цикле. Если система перезапускаются, таймеры также сбрасываются сигналом DEACT_T.

Большинство состояний на диаграммах названы по шаблону SB + номер, где SB – специальный блок (specific block). Такие блоки можно рассматривать как отдельные шаги алгоритма. Конкретные вычисления на диаграммах не указаны, если они не влияют на коммуникации между процессорами. Специальные блоки, отвечающие за передачу данных между процессорами, описаны отдельными диаграммами (SB104-SB109), ссылки на которые есть в диаграммах главного цикла.

Прерывание в процессоре C_1 происходит по сигналам T1 и T2 от таймеров или при установке флагов FLAG1 и FLAG2. Эти флаги означают, что после завершения текущей операции на шине необходимо передать сообщения на другие процессоры. Первый флаг указывает на необходимость запустить специальный блок 150, передающий управляющие сигналы с C_3 на приборы для наблюдения. Второй флаг означает необходимость запустить специальный блок 151, блокирующий некоторые сенсоры в ожидании сигнала AAR radiation.

Для инициализации передачи в режиме прерывания необходимо, чтобы шина была доступна, поэтому перед обработкой прерываний проверяется флаг bus_free, и в случае успешного захвата он сбрасывается, предотвращая попытки получения шины другими процессорами. Так как во время ожидания освобождения шины могут возникнуть новые прерывания, специальные счетчики Flag1 и Flag2 хранят количество поступивших прерываний каждого типа. После завершения обработки прерывания (сигнал LEAVE) шина освобождается, а соответствующий счетчик уменьшается.

Секция 1 представляет собой простую последовательность блоков, на порядок выполнения влияют только значения флага перезагрузки и кнопки ввода.

Секция 2 содержит инициализацию компьютеров (SB1002-1004), проверку логических характеристик (SB104, SB105), и передачу сообщений на C_2, C_3 и C_4 (SB106-108). Также активируются внутренние таймеры (ACTIVATE_TIMER). Операция деактивации таймера (DEACT_T) вызывается, если коммуникации оканчиваются неудачно и необходимо перезапустить главный цикл с SB100. Паузы между событиями в этой секции обеспечиваются таймаутами.

Специальный блок 104 проверяет состояние компьютеров C_2, C_3 и C_4 и сенсоров (сенсоры считаются единым внешним устройством). Коммуникация с каждым из удаленных устройств происходит следующим образом. Сначала проверяется, что компьютера C_1 работает в обычном режиме, затем происходит захват шины и отправка сигнала-запроса о состоянии. Если в течение заданного количества миллисекунд не получен ответ, устанавливается флаг ошибки, иначе этот флаг сбрасывается. После этого шина освобождается.

Специальный блок 105 собирает данные с доступных сенсоров. Сначала проверяется, что компьютер C_1 работает в обычном режиме, затем происходит захват шины и отправка сигнала-запроса о состоянии. Если в течение 12 миллисекунд не получен ответ, устанавливается флаг ошибки SENSOR_FAULT, иначе этот флаг сбрасывается. После этого шина освобождается.

Специальный блок 106 задает коммуникации с процессором C_2 и отвечает за получение основных параметров полета. Допускается одна повторная попытка установления соединения, если на первый запрос на соединение нет ответа в течение 12 миллисекунд. Флаг skip_data используется, чтобы разрешить работать с данными, полученными во время предыдущего обмена в случае отсутствия ответа. Если ответа нет два раза подряд, то считается, что процессор C_2 отказал, и устанавливается соответствующий флаг (C2_EN).

Специальный блок 107 отвечает за получение данных с процессора C_4. Допустимы две ошибки, связанные с отсутствием соединения. Счетчик Err_C4 содержит число ошибок. Когда счетчик достигает 2, считается, что C_4 отказал.

Специальный блок 109 отвечает за передачу данных на процессор C_4. Обмен сообщениями аналогичен специальному блоку 107.

Секция 3 выполняется, если процессоры C_2 и C_3 работают. Обмен данными с процессором C_4 происходит только если самолет летит на низкой высоте (предусловие LOW). В специальном блоке 27 вычисляется текущее состояние, используемое в алгоритмах

(схема COMPUTE_AM). Если передача данных на процессоры C_2 и C_4 заканчивается ошибкой, секция завершается и происходит выход в главный цикл.

Секция 4 выполняется, если процессор C_2 работает, а C_3 отказал. В ней происходят вычисления блоков 28, 9 и 29. В них не происходит коммуникаций, поэтому они не конкретизируются.

Секция 4 выполняется, если оба процессора C_2 и C_3 отказали. Передача сообщений на процессор C_2 (SB106, SB108) опускается, и выполняется дополнительный блок 36.

2.3 Процессор C_2

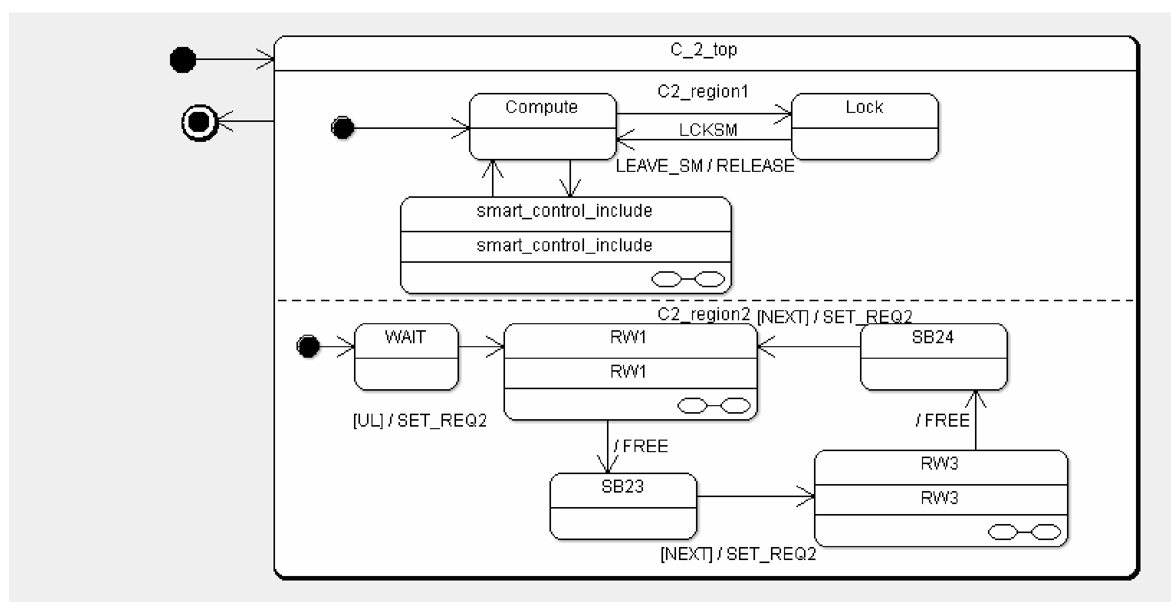

Рисунок 30. Общая диаграмма C_2.

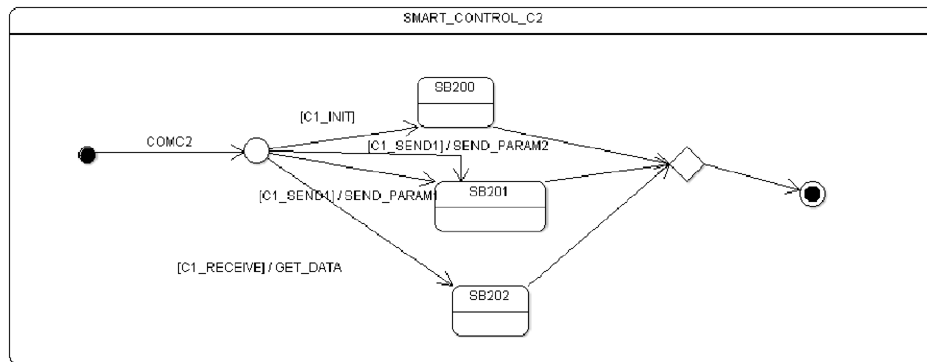

Рисунок 31. Обработка сообщений в C_2.

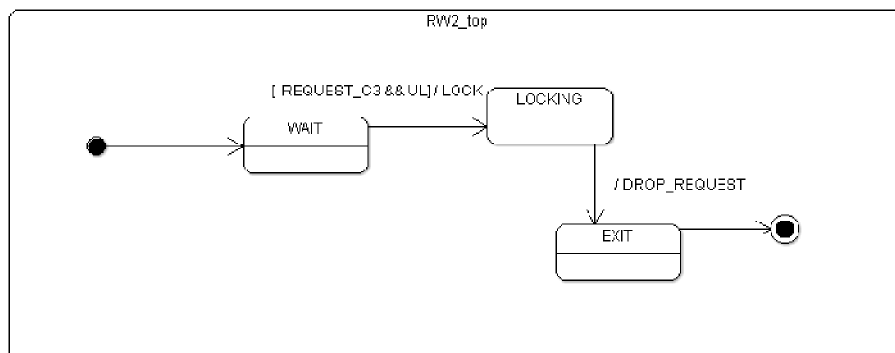

Рисунок 32. Доступ к разделяемой памяти в C_2.

Модель процессора C_2 представлена на рисунках 30-32. На процессоре C_2 в бесконечном цикле выполняются специальные блоки 23 и 24. Выполнение прерывается, когда C_1 запрашивает прием или передачу данных. Процессоры C_2 и C_3 имеют доступ к общей памяти, и цикл может быть также приостановлен, если память используется другим процессором.

На основной диаграмме два параллельных региона. Один из них описывает текущее состояние процессора, другой – текущий выполняемый блок. Процессор имеет три режима:

нормальный (COMPUTE), ожидание освобождения общей памяти (LOCK) и прерывание для взаимодействия по шине с процессором C_1 (SMART_CONTROL_C2).

Процессор C_2 выполняет по очереди две задачи: расчет основных параметров полета (блок 23) и вывод данных на индикаторы (блок 24). Между этими блоками выполняется операция чтения /записи в общую память. Переход к очередному блоку возможет, если процессор работает в нормальном режиме и общая память не занята (предусловие NEXT). Перед операцией с общей памятью выставляется флаг запроса (SET_REQ), а после окончания отправляется сигнал FREE_SM, переводящий процессор из режима ожидания в нормальный режим.

В режиме прерывания процессор C_2 обменивается данными с C_1. Возможны три случая: инициализация процессора C_2 (специальный блок 200), отправка основных параметров полета (SB201) и получение данных от C_1 (SB202).

Работа с общей памятью необходима по завершении каждого специального блока. На диаграммах состояний она моделируется специальным состоянием, помещенным между блоками. Эти состояния, имена которых начинаются с RW, реализуют механизм семафоров, предотвращающий одновременный доступ двух процессоров к памяти.

2.4 Процессор C_3

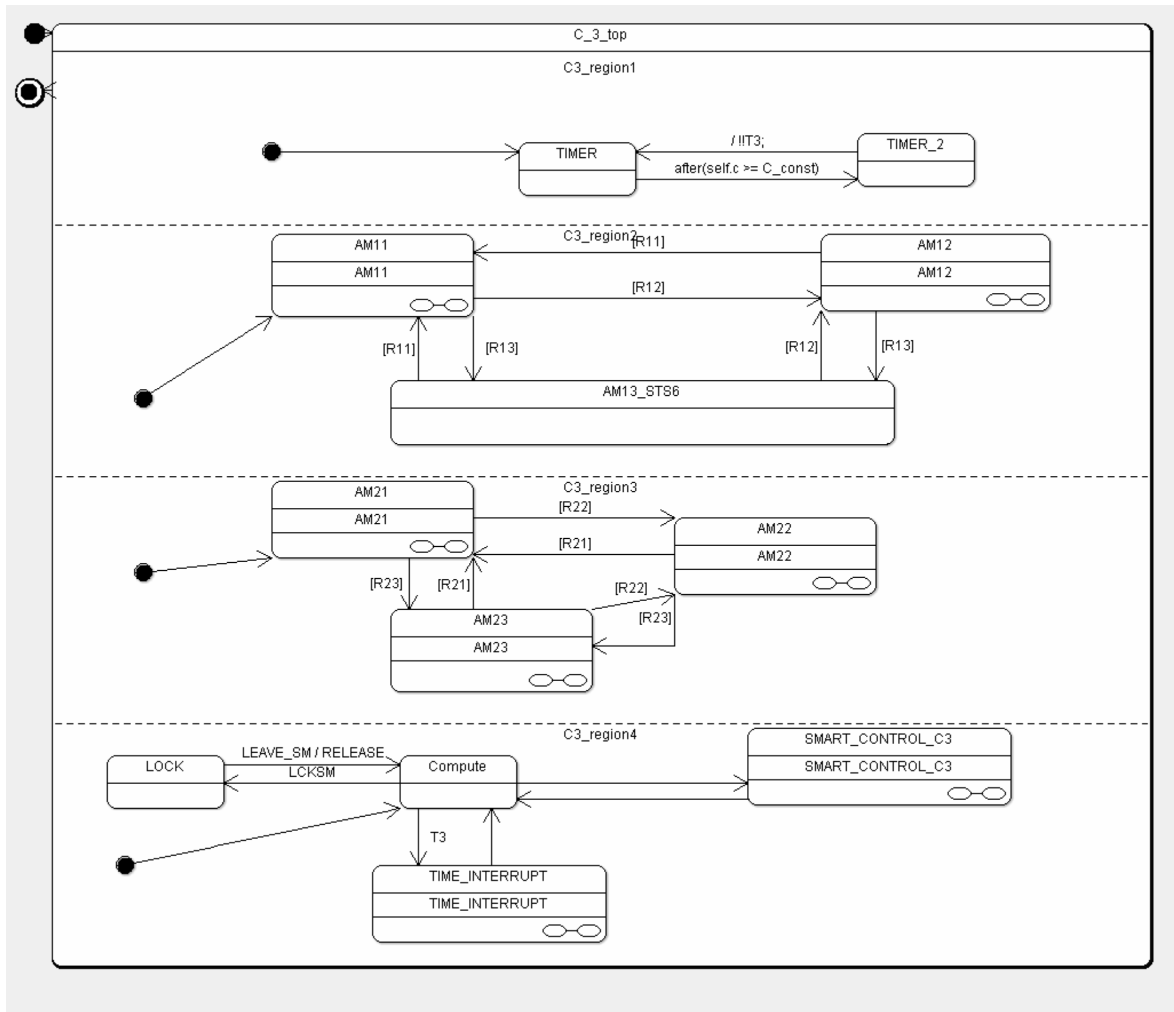

Рисунок 33. Общая диаграмма C_3.

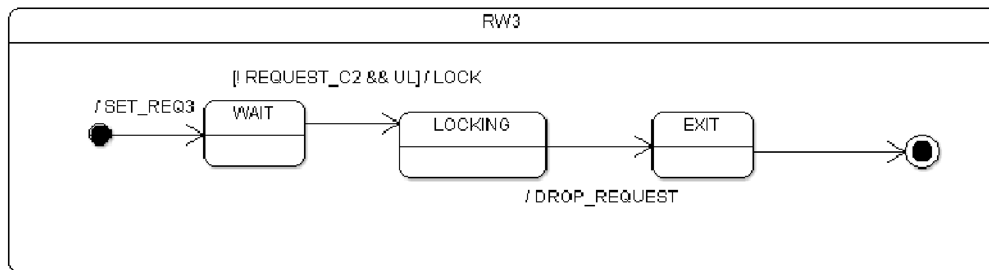

Рисунок 34. Доступ к разделяемой памяти в C_3.

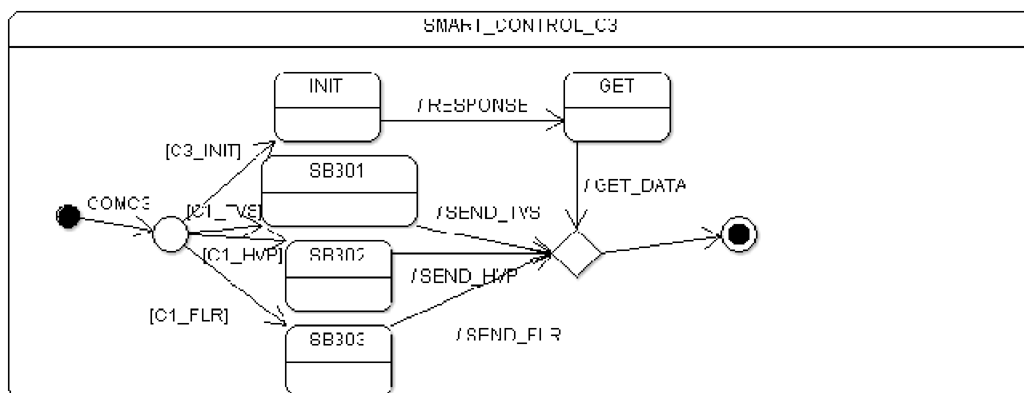

Рисунок 35. Обработка сообщений в C_3.

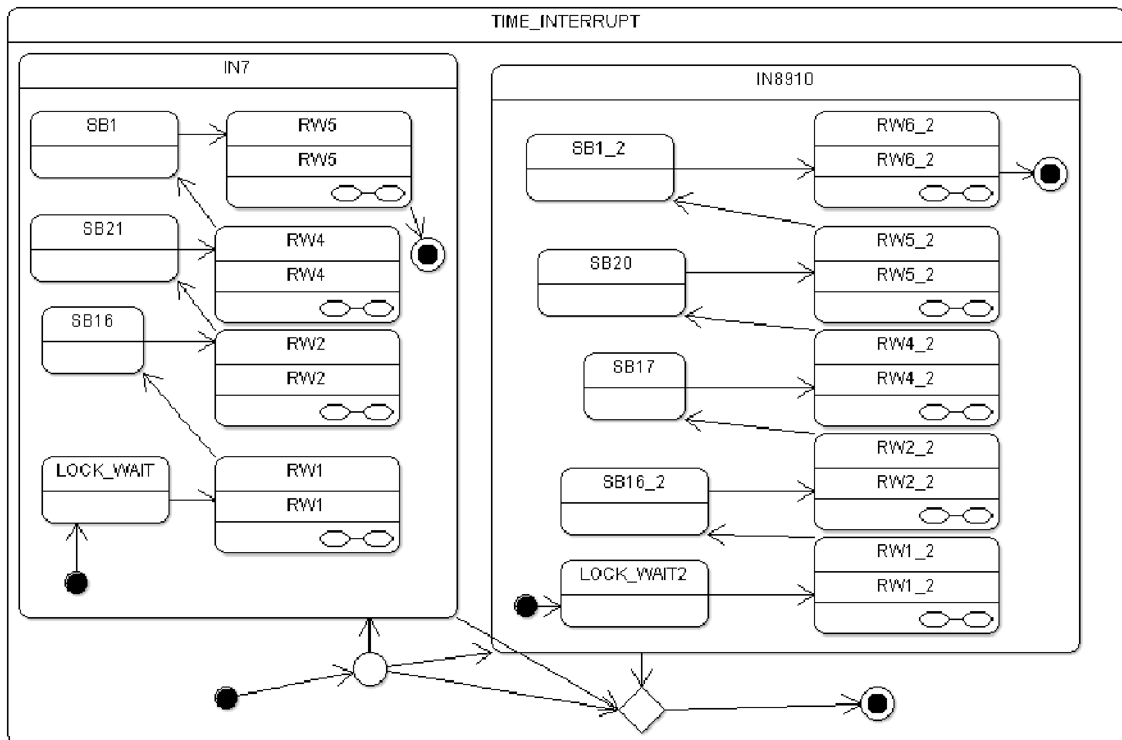

Рисунок 36. Обработка прерываний по таймеру в C_3.

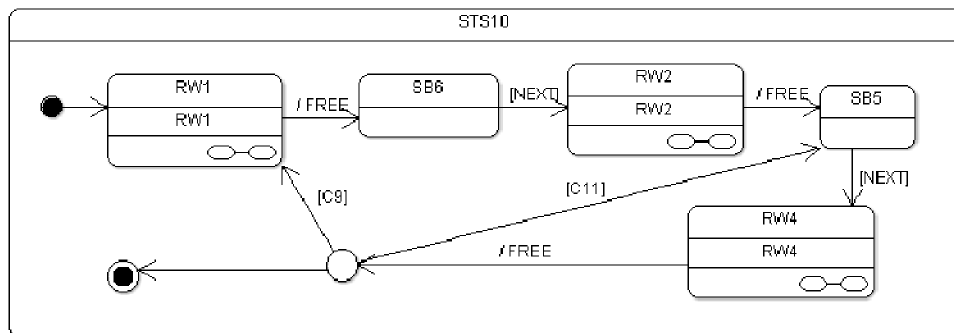

Рисунок 37. Блок STS10.

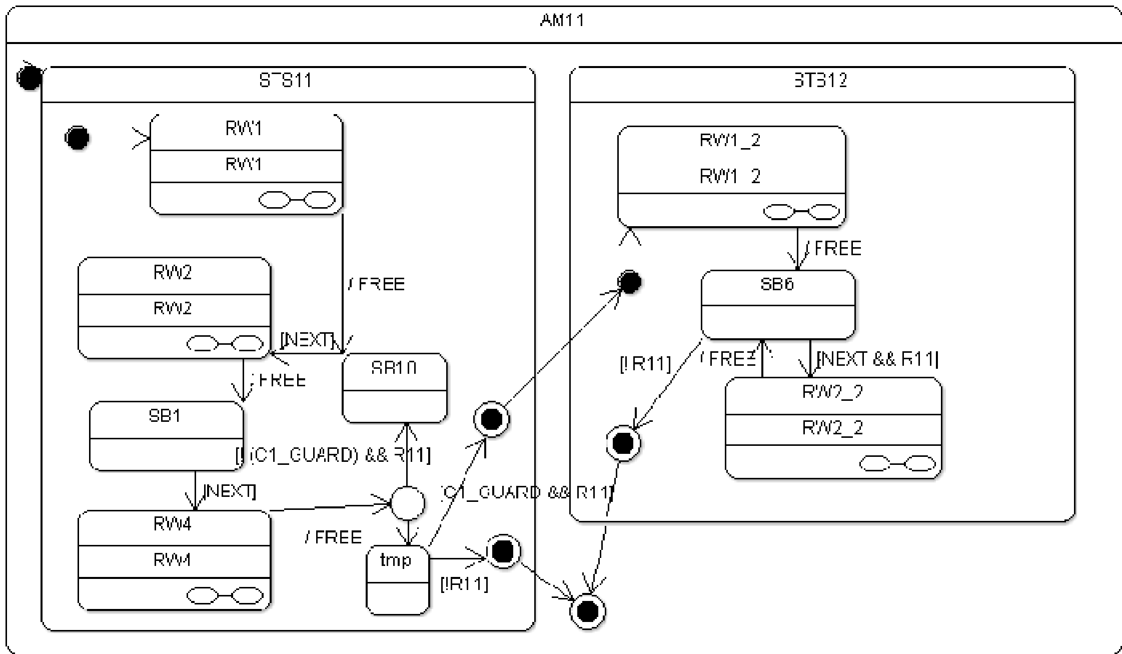

Рисунок 38. Вычислительный блок AM11.

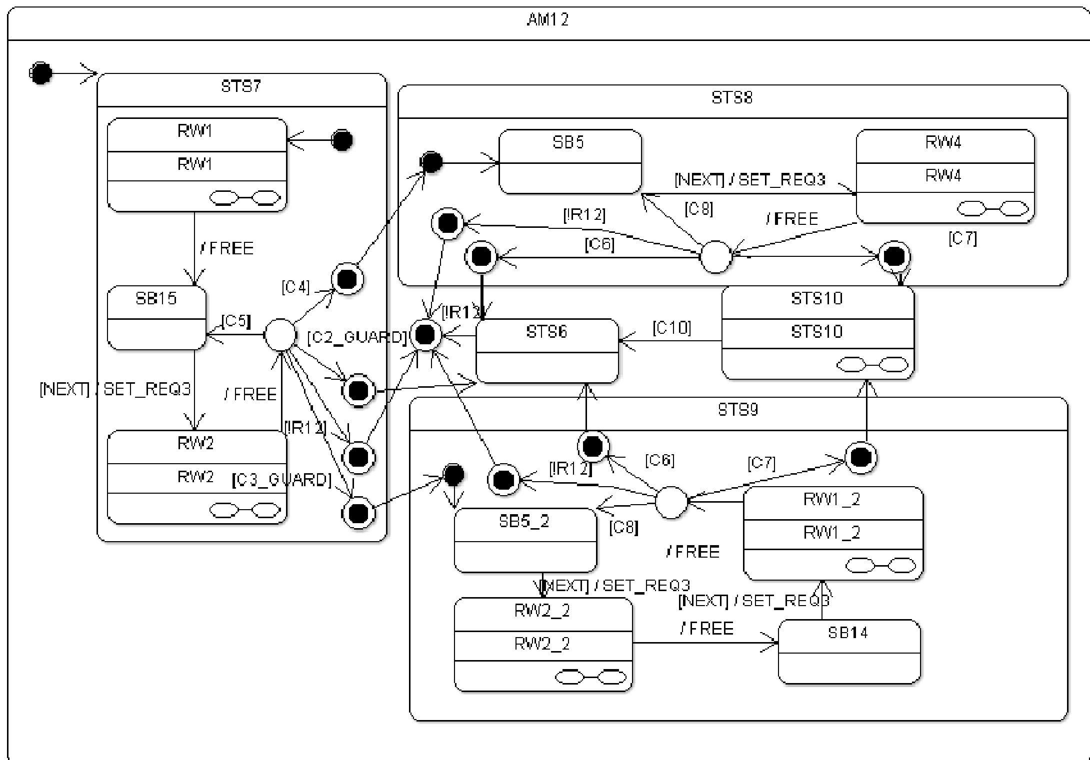

Рисунок 39. Вычислительный блок AM12.

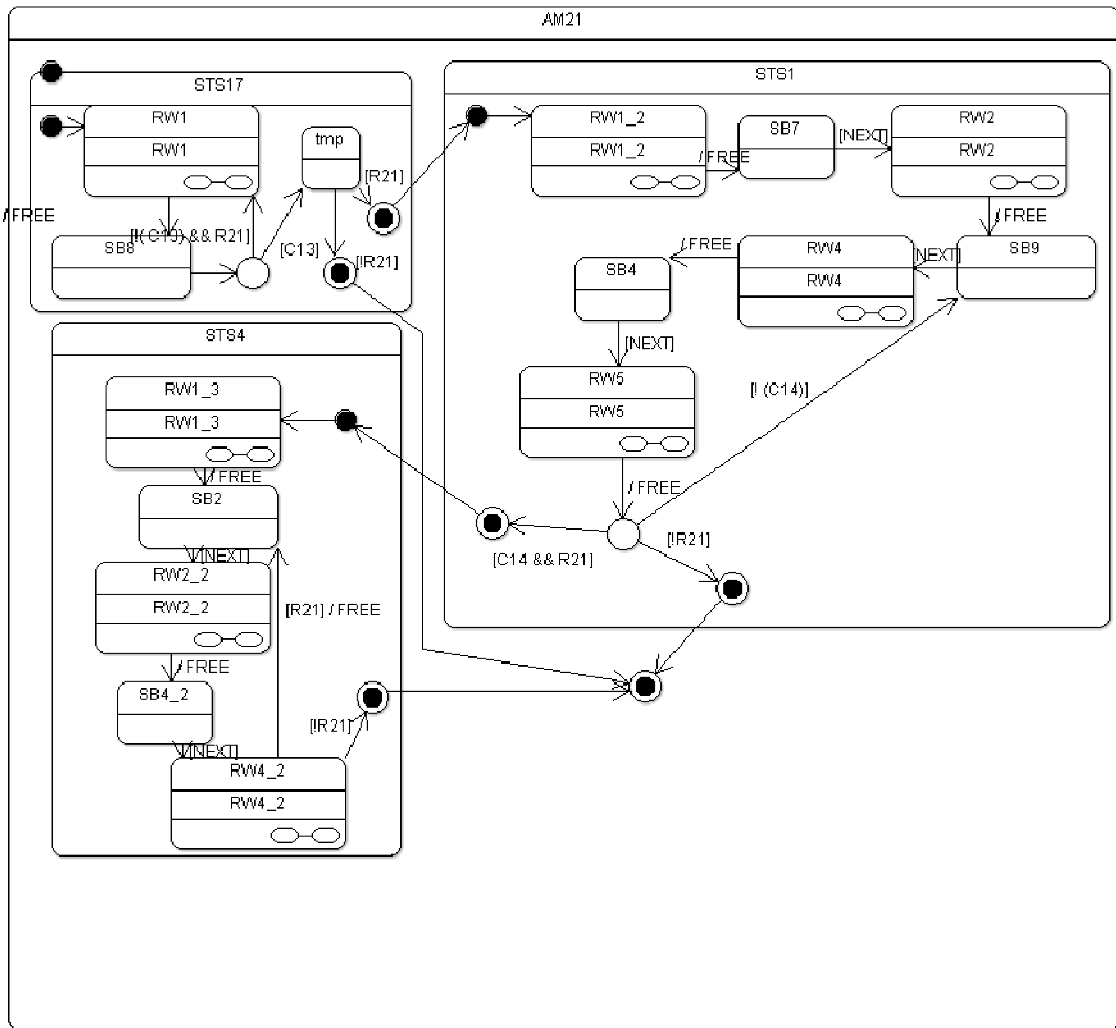

Рисунок 40. Вычислительный блок AM21.

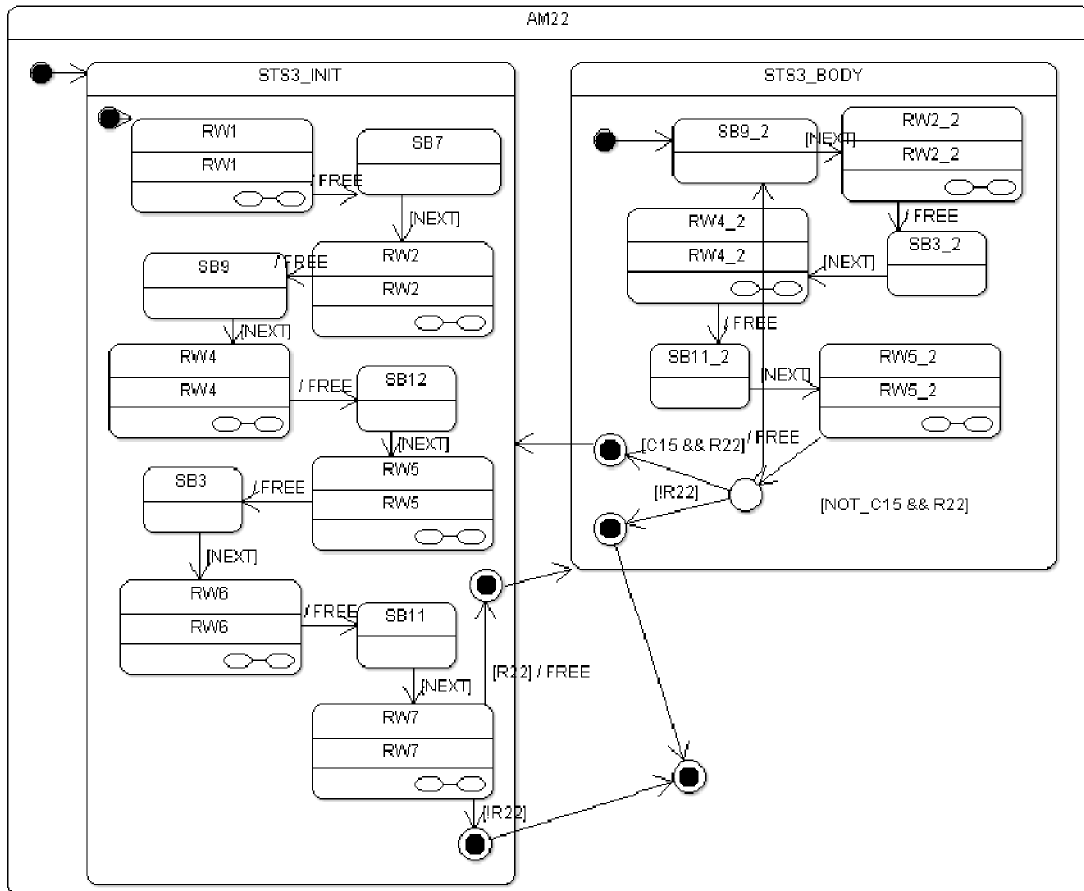

Рисунок 41. Вычислительный блок AM22.

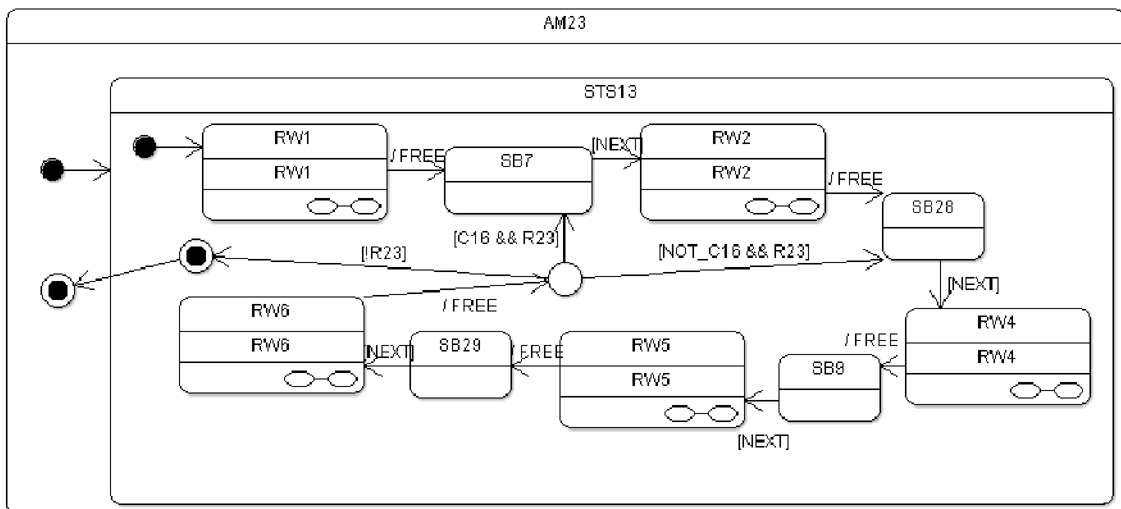

Рисунок 42. Вычислительный блок AM23.

Модель процессора C_3 представлена на рисунках 33-42. Процессор C_3 выполняет в цикле вычисления, определяемые блоками 1-22. Состав и порядок выполняемых блоков зависят от текущего режима работы системы, который передается на процессор C_3 с процессора C_1.

Выполнение главного цикла прерывается по таймеру для пересчета значений, связанных с некоторыми сенсорами (FLR, TVS, HVP), а также при необходимости провести обмен данными по шине с C_1. Прерывания обрабатываются в специальных блоках 301-303. C_3 подключен к общей памяти, поэтому выполнение также может быть приостановлено, если память занята другим процессором.

У процессора C_3 есть четыре режима: нормальный (COMPUTE), ожидание освобождения памяти (LOCK), прерывание по таймеру (TIME_INTERRUPT) и прерывание для коммуникаций по шине (SMART_CONTROL_C3).

Выделяется четыре возможных причины прерывания для коммуникации: отправка трех параметров (FLR, TVS, HVP) и запрос о статусе процессора (C3_STATUS_WORD).

На процессоре C_3 выполняются параллельно два вида вычислений. Какой именно блок выполняется, определяется значениями переменных R1 и R2.

2.5 Процессор C_4

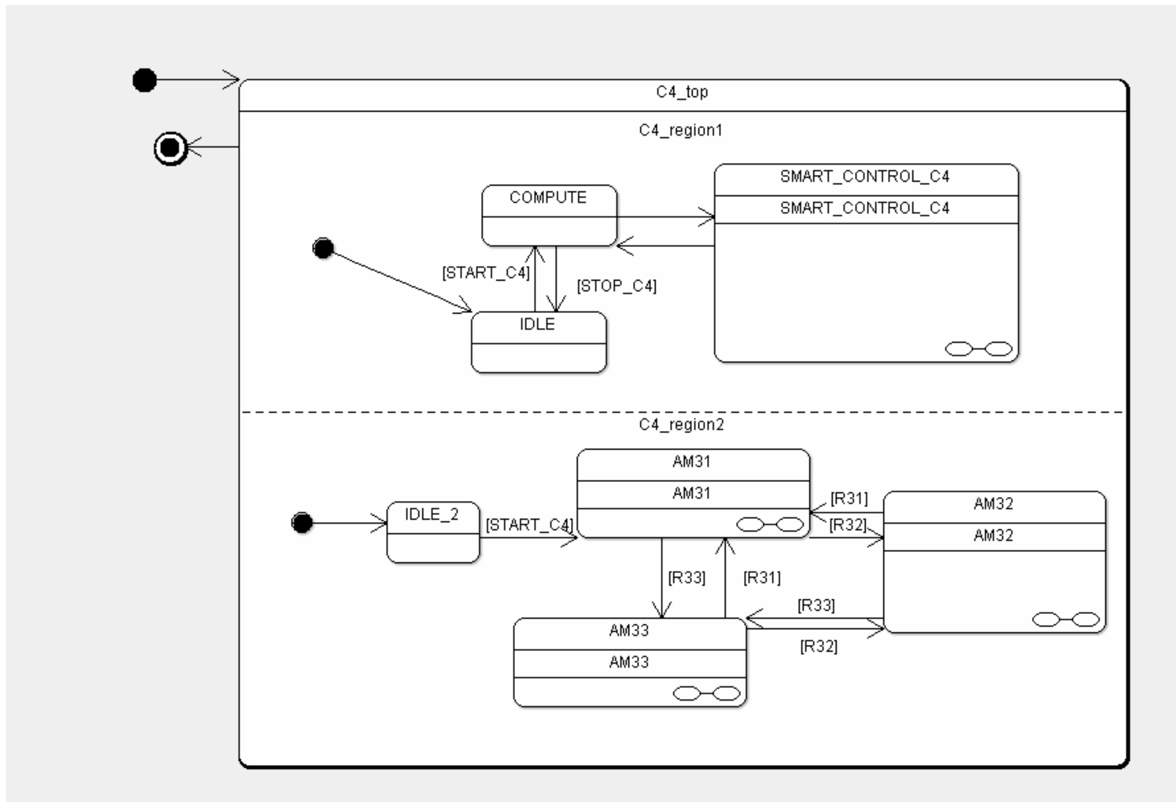

Рисунок juan42. Общая диаграмма C_4.

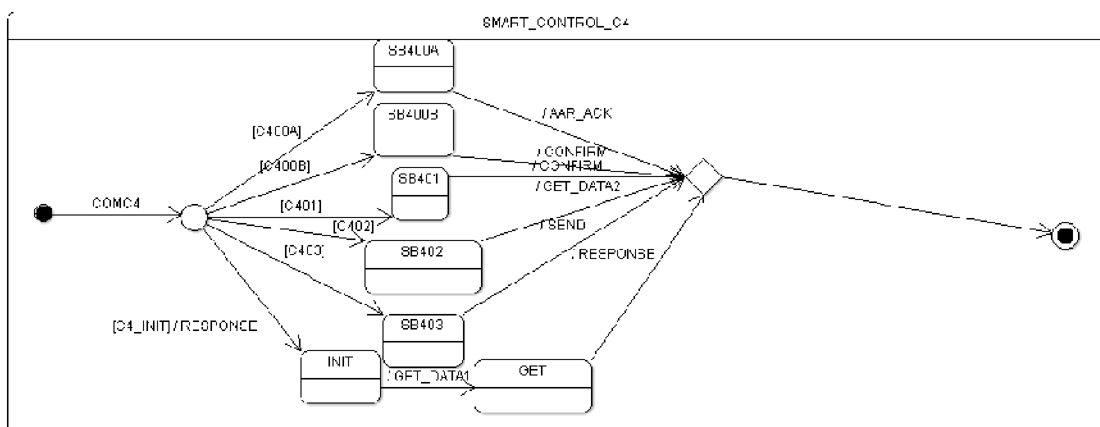

Рисунок 44. Обработка сообщений в C_4.

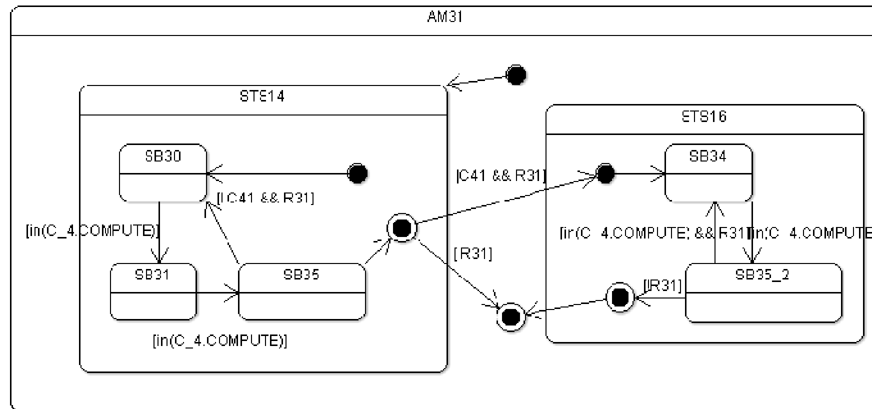

Рисунок 45. Вычислительный блок AM31.

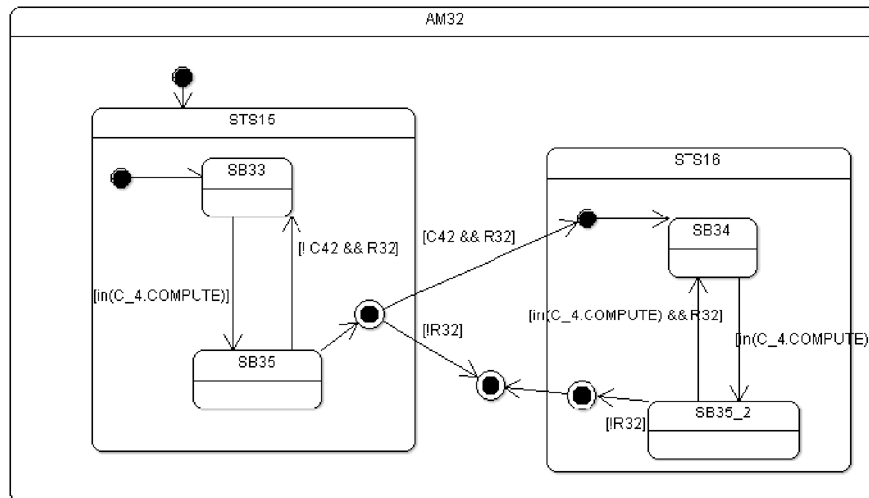

Рисунок 46. Вычислительный блок AM32.

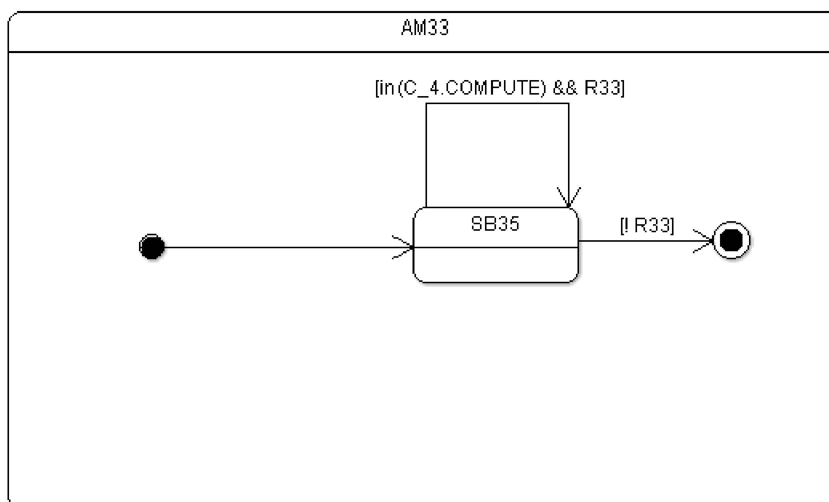

Рисунок 47. Вычислительный блок АМ33.

Модель процессора С_4 представлена на рисунках 43-47. Процессор С_4 работает, когда необходим расчет параметров полета на низкой высоте или когда включается радар, предотвращающий столкновения. В зависимости от состояния процессора, которое присылается с С_1, компьютер С_4 выполняет алгоритмы, заданные специальными блоками 31-35. Выполнение прерывается в случае, если нужно получить или передать данные с процессора С_1.

Так же, как и для других процессоров, поведение процессора С_4 определяется диаграммой с двумя параллельными регионами, один из которых определяет режим работы процессора, а второй задает непосредственно вычисления. У процессора С_4 три режима: ожидание (IDLE), выполнение (COMPUTE) и прерывание (SMART_CONTROL_C4). Процессор С_4 переходит из режима ожидания в режим выполнения, когда включается радар (AAR_on == true).

В режиме прерывания процессора С_4 обрабатывает запросы в соответствие с кодом текущей операции (COMC4).

Вычисления представлены последовательностью простых состояний. Так как процессор C_4 не использует общую память, при переходах между состояниями дополнительных операций не требуется.

3 Экспериментальное исследование и оптимизация среды выполнения моделей компонентов РВС РВ

В данном разделе описывается экспериментальное исследование и оптимизация среды выполнения моделей компонентов РВС РВ. В разделе 3.1 приводится описание исследования применимости CERTI для моделирования РВС РВ. В разделе 3.2 описывается доработка инфраструктуры CERTI RTI по результатам исследований, приведённых в разделе 3.1. В разделе 3.3 приводится описание оптимизаций шаблона Cheetah для трансляции в исполняемые модели, совместимые со стандартом HLA. Раздел 3.4 содержит обоснование корректности алгоритма трансляции UML-диаграмм в сеть плоских временных автоматов. В разделе 3.5 приведено описание оптимизации алгоритма трансляции UML во временные автоматы. В разделе 3.6 приведено описание оптимизации средства трансляции UML во временные автоматы. Раздел 3.7 содержит описание экспериментального исследования модели РВС РВ Dr Tesy, описанной в разделе 2.

3.1 Исследование применимость CERTI для моделирования РВС РВ

3.1.1 Цели исследования

На предыдущем этапе настоящей научно-исследовательской работы было проведено экспериментальное исследование применимости инфраструктуры CERTI RTI [17] для решения задачи моделирования распределённых вычислительных систем в реальном времени. Для этого показатели производительности системы CERTI сравнивалась с аналогичными результатами системы Стенд ПНМ [16], предназначенной специально для решения подобных задач. Полученные характеристики системы CERTI значительно проигрывали характеристикам системы Стенд ПНМ, однако продемонстрированный результат был достаточным для решения поставленных экспериментальных задач. Таким образом, была показана принципиальная применимость системы CERTI для целей настоящего проекта.

В ходе анализа результатов проведённого исследования было высказано несколько предположений о причинах отставания CERTI, однако для подтверждения предположений экспериментальных данных было собрано недостаточно. В результате, на текущем этапе настоящей научно-исследовательской работы была поставлена задача получения более

точных экспериментальных результатов, и повторного анализа выдвинутых ранее гипотез на основе этих данных.

3.1.2 Методика проведения экспериментов

Одной из основных численных характеристик, определяющих возможность использования конкретной системы для решения задач моделирования в реальном времени является среднее время обработки события. В подобных задачах важно гарантировать наступление каждого события строго внутри заданного директивного интервала. Таким образом, среднее время обработки события заведомо должно быть меньше верхней границы интервала. Кроме того, меньшее время обработки означает возможность дополнительного усложнения задачи и меньшую требовательность к вычислительным ресурсам.

В рамках проведённого экспериментального исследования среднее время обработки события может быть получено в виде отношения времени выполнения модели без ограничений реального времени к числу обработанных при этом событий. Так как системы моделирования выполняют одну и ту же модель и передают одинаковое число сообщений, то общее время обработки событий является показателем, эквивалентным среднему времени обработки события. Поэтому в дальнейшем под показателем производительности будет пониматься общее время обмена сообщениями.

Выводы о применимости системы CERTI для решения задачи моделирования распределённых вычислительных систем в реальном времени были построены на основе сравнения её результатов с аналогичными показателями специализирующейся на них системы «Стенд ПНМ». В качестве тестовых задач при этом были использованы две простейших модели из пакета тестирования системы «Стенд ПНМ». Обе задачи моделируют поведение комплекса, состоящего из терминала и вычислителя. Терминал итеративно передаёт вычислителю сообщения с единственным числовым параметром, уменьшающимся на единицу при каждой следующей пересылке, пока он не достигнет нулевого значения. В задаче «BasicTest» вычислитель лишь регистрирует поступившие сообщения. В задаче «BCVMTest» вычислитель отвечает на каждое сообщение собственным сообщением с тем же телом, а терминал не передаёт новых сообщений, пока не получит соответствующее уведомление от вычислителя.

Модель «BCVMTest» практически не нагружает элементы системы, поэтому хорошо подходит для измерения среднего *времени отклика* модели – времени обработки одного события или передачи одного сообщения. Модель «BasicTest», напротив, создаёт

значительную пиковую нагрузку, поочерёдно перегружая компоненты модели на пути от отправителя к получателю. В результате, выполнение модели является хорошим тестом *пропускной способности* системы моделирования.

3.1.3 Экспериментальный стенд

На предыдущем этапе настоящей работы исследования проводились с участием единственной инструментальной машины, на которой были запущены все компоненты распределённой системы моделирования. При такой конфигурации экспериментального оборудования система Стенд ПНМ способна поддерживать исполняемую модель в согласованном состоянии, используя лишь средства локальной синхронизации процессов и отказываясь от сетевой передачи данных. При этом система CERTI продолжала использовать более «дорогие» механизмы сетевой синхронизации. В то же время на практике для выполнения моделей обычно используется комплекс из нескольких инструментальных машин. Поэтому в рамках данного этапа исследований необходимо было измерить производительность систем в режиме работы, использующем механизмы сетевой синхронизации.

Кроме того, нескольких компонентов имитационной программы выполняющихся на одной машине неминуемо конкурируют за ресурсы вычислительной системы и оказывают друг на друга косвенное влияние, учесть которое не представляется возможным. Поэтому полученные численные результаты систем моделирования могут быть в значительной степени искажены и отличаться от результатов их реального применения на практике.

Опыт предыдущих исследований был учтён, и на этот раз для постановки экспериментов использовался комплекс из двух машин с идентичной конфигурацией: процессор Intel Core 2 Duo 2.60 Ггц, 2 Гб оперативной памяти, операционная система Debian 5.0.8 Lenny. При этом на каждом из компьютеров выполнялся свой компонент имитационной модели.

Таблица 3. Время выполнения моделей BasicTest и BCVMTest системами моделирования CERTI и Стенд ПНМ в зависимости от числа переданных сообщений, мкс

Число сообщений	BasicTest		BasicTestM	BCVMTest	
	CERTI	Стенд ПНМ	CERTI	CERTI	Стенд ПНМ
10	5	1	5	10	4
100	61	5	49	94	25
1000	611	49	487	924	249
10000	9297	499	4843	9123	2500
100000	1139762	5000	48575	90034	24999

Результаты нового эксперимента, приведённые в таблице 3, показывают, что система CERTI отстаёт от системы «Стенд ПНМ» в несколько раз. При этом отставание растёт линейно относительно числа сообщений. Лавинообразный рост времени выполнения задачи «BasicTest» системой CERTI объясняется особенностями её механизма продвижения модельного времени, его причины описываются в следующем разделе работы.

3.1.4 Анализ результатов

Анализ полученных в ходе исследования результатов является предметом для более глубоко анализа и требует более детального изучения рассмотренных систем. Коренное отличие инфраструктуры CERTI RTI от среды выполнения системы «Стенд ПНМ» заключается в степени их параллелизма. Корни стандарта HLA, на основе которого была построена система CERTI, уходят к виртуальным средам – играм и тренажёрам, позволяющим географически разделённым участникам использовать общую модель игрового мира, обеспечивая при этом достаточную степень её интерактивности [7]. В отличие от CERTI, система моделирования «Стенд ПНМ» изначально создавалась как вычислительный, многомашинный комплекс, вычислительные узлы которого находятся в одном помещении.

Поэтому системы, к которым предъявлялись различные требования, были построены на различных принципах. В основу системы моделирования «Стенд ПНМ» была заложена идея общих часов – во время выполнения модели производится высокоточная синхронизация вычислительных узлов комплекса. Отдельные участники моделирования при этом выполняются в соответствии с общесистемным временем. Таким образом, согласованность имитационной модели обеспечивается автоматически.

Идея использования существенно удалённых узлов не позволяет применить аналогичный подход в системах, реализующих стандарт HLA. В соответствии с его спецификациями отдельные участники моделирования должны использовать своё собственное модельное время, называемое *логическим временем*. Логическое время продвигается участниками моделирования независимо друг от друга с помощью сервисов инфраструктуры RTI. При этом задача поддержания модели в согласованном состоянии решается внутри RTI с помощью одного из распределённых алгоритмов синхронизации [7]. Система CERTI предлагает на выбор два таких алгоритма [18].

Описанные подходы продвижения времени обладают своими сильными и слабыми сторонами, и выбор лучшего из них существенно зависит от решаемой имитационной задачи. Использование распределённых алгоритмов синхронизации требует больших накладных расходов. Кроме того, при неосторожном использовании данный механизм может привести к значительному падению производительности. Например, этим объясняются столь плохие результаты системы CERTI на задаче «BasicTest». В данной задаче терминал не зависит от внешних факторов, поэтому он неограниченно продвигает своё модельное время и мгновенно забрасывает RTI сообщениями. Инфраструктура не успевает обрабатывать поступающий поток сообщений, и накапливает их в соответствующем буфере. С ростом размера буфера манипуляции с его участием требуют всё больше ресурсов, поэтому обработка сообщений замедляется, а рост буфера, напротив, ускоряется. В итоге происходит лавинообразный рост время выполнения модели. Описанная ситуация легко исправляется с помощью замедления терминала или ограничения размера буфера – время выполнения изменённой модели приведено в колонке «BasicTest M».

С другой стороны отказ от идеи общего времени на более сложных реальных задачах часто приводит к росту производительности за счёт упреждающего выполнения инструкций. Например, если бы терминал совершал сложные вычисления после передачи каждого пятого сообщения, то опережение остальных участников моделирования позволяло бы ему рассчитывать на большее процессорное время.

Что касается общей динамики результатов проведённого исследования, то она не изменилась по сравнению с результатами предыдущего исследования. Система моделирования CERTI по-прежнему в несколько раз проигрывает системе Стенд ПНМ. Учитывая проведённые изменения топологии экспериментального стенда, можно сделать вывод о верности высказанного на предыдущем этапе предположения – проблемы CERTI связаны, прежде всего, с её архитектурой.

3.1.5 Выводы

Результаты, показанные реализацией CERTI RTI на простейших тестовых задачах, в несколько раз уступают результатам системы «Стенд ПНМ» и не позволяют использовать её для решения того же диапазона задач без предварительных модификаций. Таким образом, вновь полученные экспериментальные данные подтверждают выводы предыдущего исследования. Система CERTI проигрывает в производительности из-за своей неэффективной архитектуры, показатели которой, однако, могут быть значительно улучшены. Следовательно, на основе системы CERTI может быть построено средство для эффективного решения задач моделирования РВС РВ в реальном времени.

3.2 Доработка инфраструктуры CERTI RTI

На предыдущем этапе настоящей научно-исследовательской работы в архитектуре данной системы моделирования был выделен ряд проблем, касающихся нерационального распределения функциональности инфраструктуры RTI и использования неэффективных механизмов взаимодействия между ними. Для решения этих проблем было предложено несколько возможных путей развития существующей архитектуры. Все разработанные правки имели различный потенциал и требовали разного уровня затрат. Проведённый анализ показал, что наиболее целесообразной модификацией на данном этапе настоящей работы является повышение эффективности взаимодействия между несколькими процессами, работающими на одном и том же узле распределённой системы моделирования.

3.2.1 Модель управления потоков системы CERTI

Стандарт распределённого моделирования HLA IEEE 1516-2000 [19] не предъявляет строгих требований к способу практической реализации описанных в нём служб. В частности, не предписывается использование какой-либо *модели потоков управления*, разделяющей задачи федератов и связывающей их инфраструктуры RTI между процессами распределённой имитационной программы. В результате описанная в стандарте функциональность на практике может быть реализована многими разными способами. В то же время, выбор модели потоков управления является одним из центральных факторов, существенно влияющих на общие показатели производительности системы моделирования.

Рассмотрим модель потоков управления распределённой системы CERTI RTI [17], выбранной в рамках настоящей научно-исследовательской работы в качестве базы для

создания новой среды выполнения. В общем виде архитектура CERTI RTI может быть представлена в виде совокупности из одного глобального процесса RTI Gate (RTIG), отвечающего за управление выполнением имитационной модели, нескольких локальных процессов RTI Ambassador (RTIA), и библиотеки libRTI, обеспечивающей взаимно-однозначную связь участвующего в моделировании федерата с соответствующим ему процессом RTIA (рисунок 48). При этом взаимодействие между глобальным процессом RTIG и локальными процессами RTIA происходит через сокеты сетевого протокола TCP/IP, а взаимодействие каждого процесса RTIA с соответствующим ему процессом федерата – через локальные каналы передачи данных операционной системы (сокеты UNIX).

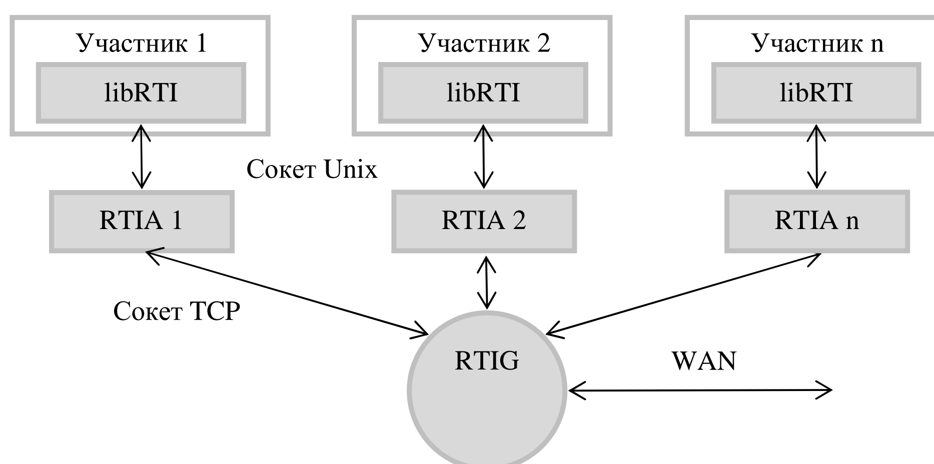

Рисунок 48. Архитектура RTI реализации CERTI.

Запуск RTI происходит в несколько этапов. Прежде всего, запускается глобальный процесс RTIG, который будет координировать работу подключённых компонентов RTI. В общем случае каждый участник моделирования может представлять собой независимую программу. При подключении нового участника к федерации, слинкованная с ним библиотека libRTI создаёт на инструментальной машине новый процесс RTIA, и связывается с ним через сокет UNIX. Далее созданный процесс RTIA устанавливает TCP соединение с глобальным процессом RTIG. Таким образом, локальная модель потоков управления системы CERTI состоит из двух взаимодействующих процессов – процесса федерата и соответствующего ему процесса RTIA.

3.2.2 Потоки против процессов

Любая распределённая программа предполагает наличие, как минимум, одного потока управления на каждом узле-исполнителе. На практике же распределённые программы часто представлены на каждой локальной машине сразу несколькими потоками,

соответствующими одному или нескольким процессам. В частности, распределённая инфраструктура RTI системы моделирования CERTI, выбранная в рамках настоящего проекта в качестве основы для создания новой среды выполнения моделей, использует два таких процесса.

Процессы и потоки представляют собой два разноуровневых механизма для написания параллельных программ: в контексте одного процесса могут выполняться сразу несколько взаимодействующих друг с другом потоков. Обычно взаимодействие нескольких потоков, разделяющих общее адресное пространство, может быть более эффективно, чем взаимодействие нескольких независимых процессов, требующее использования более сложных средств синхронизации операционной системы. Например, в текущей архитектуре CERTI, федерат и соответствующий ему процесс RTIA обмениваются данными с помощью передачи сообщений через сокет UNIX. При этом параметры должны быть сериализованы перед передачей и декодированы после неё. Использование же потоков позволяет передавать данные с помощью простого механизма семафоров, сохраняя возможность прямого обращения к параметрам запроса и не требуя их предварительной конвертации.

Кроме того, потоки способны более эффективно использовать доступные вычислительные ресурсы. До недавнего времени все процессоры были одноядерными, и использование только полновесных процессов не приводило к существенному падению производительности. Однако с появлением многоядерных архитектур, способных эффективно выполнять несколько нитей одновременно, потери стали более ощутимыми [20]. Таким образом, при построении новой модели потоков управления целесообразно рассматривать лишь те из них, которые используют нескольких потоков.

3.2.3 Локальные модели потоков управления

Раздел стандарта моделирования HLA версии IEEE 1516-2000, посвящённый спецификаций интерфейсов [19], задаёт правила взаимодействия инфраструктуры RTI и набора подключённых к ней участников моделирования – федератов. В данном разделе описываются интерфейсы, через которые федераты могут использовать доступные службы инфраструктуры RTI. Здесь же описывается набор методов федерата, которые инфраструктура RTI может использовать в своей работе.

Далее будем различать две категории вызовов, в зависимости от их направления: пусть *прямые вызовы* используются при обращении федерата к службам инфраструктуры RTI, а *обратные вызовы* – при обращении инфраструктуры RTI к методам подключённого к

ней федерата. При этом обратный вызов может быть осуществлён инфраструктурой RTI только после получения соответствующего указания от подключённого к ней федерата. У федерата существует два принципиально разных способа сигнализации о готовности обработать обратный вызов: обратиться к соответствующей службе инфраструктуры RTI «evokeCallback», которая совершит обратный вызов, или явно дать разрешение на получение асинхронных вызовов. В последнем случае, федерату, возможно, придётся работать с несколькими потоками управления одновременно, поэтому его разработчики не должны допускать возможных состояний гонки, взаимной блокировки или удушения отдельных потоков.

Далее в данном разделе будут подробно рассмотрены три актуальных на данный момент модели взаимодействия федерата и инфраструктуры RTI в рамках локальной машины: *однопоточная*, *асинхронная* и *многопоточная* [21].

Однопоточная модель управления

На рисунке 49 изображена модель федерата, совершающего прямой вызов функции «evokeCallback». Обрабатывая этот вызов, инфраструктура RTI способна совершить обратный вызов методов федерата. При этом вся обработка (как прямого, так и обратного вызовов) производится в одном и том же потоке управления. Однопоточная модель применяется в одной из первых реализаций инфраструктуры RTI – DMSO [22], где одна единственная нить разделялась между федератом и инфраструктурой RTI.

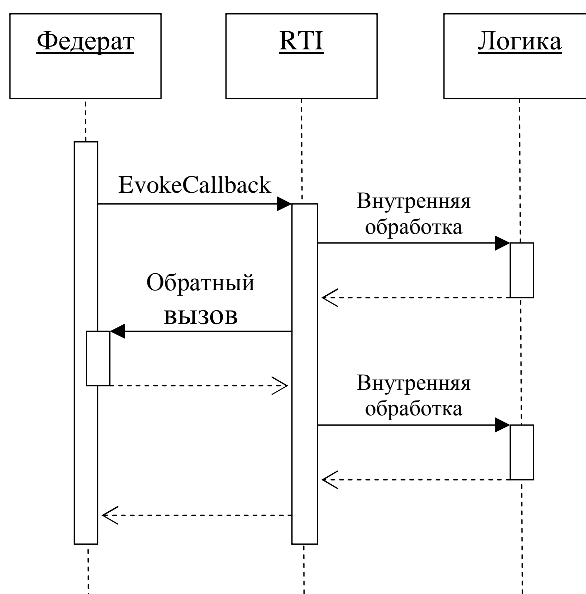

Рисунок 49. Схема работы однопоточной модели управления.

В данной модели инфраструктура RTI получает процессорное время только во время обработки вызовов, поступивших от подключённого к ней федерата. Поэтому для корректной работы имитационной программы федерат должен постоянно совершать вызовы «evokeCallback», тем самым, давая инфраструктуре RTI возможность для поддержания своей внутренней логики работы, например, временной синхронизации федератов. Главной сложностью при этом является неочевидность выбора лучшего момента для совершения вызова. На практике он зависит от массы самых разнообразных факторов: логики имитационной модели, реализации выполняемого федерата и эффективности используемой RTI, программной и аппаратной платформы машины-исполнителя, сетевой топологии распределённой системы и других причин, способных повлиять на время внутренней обработки поступившего вызова.

К преимуществам однопоточной модели можно отнести простоту реализации и малую задержку при выполнении вызовов – данный подход не требует дополнительной синхронизации потоков управления. Такое свойство потенциально полезно при решении задач моделирования в реальном времени, когда разработчики не желают допускать лишних переключений контекста процессора.

Основным недостатком данной модели является появление дополнительных условий, выполнение которых необходимо для корректного функционирования модели. Чрезмерно большой интервал между вызовами может привести к ошибке в работе инфраструктуры RTI, поэтому разработчики должны избегать «лобового» выполнения сложных операций, разбивая их на несколько небольших подзадач, перемежающихся с вызовами метода «evokeCallback».

Асинхронная модель управления

В асинхронной модели (рисунок 50) инфраструктура RTI использует один или несколько внутренних потоков, которые выполняются независимо от потоков подключённого федерата. Таким образом, федерату не нужно заботиться о специальном выделении процессорного времени для работы инфраструктуры RTI. Вызовы со стороны федерата, однако, обрабатываются в данном случае так же, как и в простой однопоточной модели. Асинхронная модель управления потоков может использоваться в одной из конфигураций улучшенной версии инфраструктуры DMSO – RTI:NG [22]. Кроме того, эта модель применяется в реализации RTI от компании Pitch – pRTI [23].

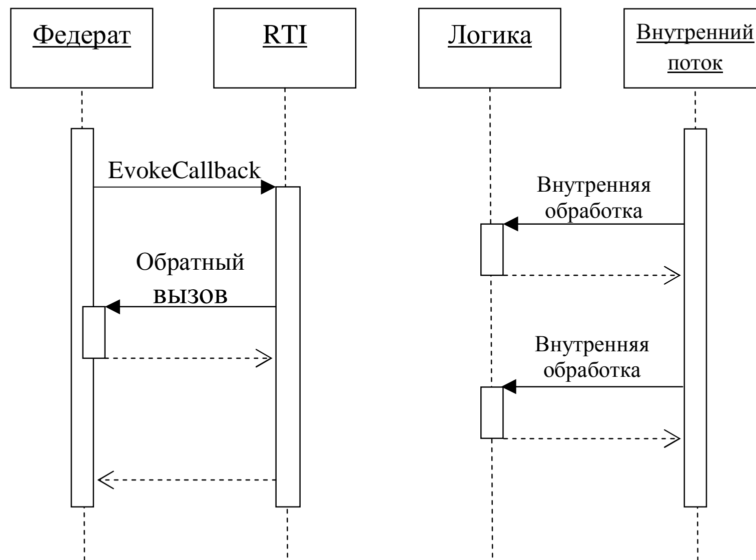

Рисунок 50. Схема работы асинхронной модели управления.

В отличие от однопоточной модели управления, асинхронная модель позволяет инфраструктуре RTI работать независимо от поведения федерата. Тем самым, снимается ограничение на временной интервал между вызовами «evokeCallback». Федерат должен вызывать этот метод, только если необходим обратный вызов от инфраструктуры RTI. Таким образом, асинхронная модель удобнее для разработчиков имитационной модели.

Так как вызов «evokeCallback» не возвращает управление федерату до тех пор, пока не истечёт заданный временной интервал, то поток управления федерата способен бесцельно простаивать в ожидании обратного вызова от инфраструктуры RTI. Данный недостаток становится особенно заметным в случае, когда внутри федерата существуют собственные задачи, требующие значительного процессорного времени.

Многопоточная модель управления

В многопоточной модели управления (как и в уже рассмотренной асинхронной), инфраструктура RTI использует один или несколько внутренних потоков, выполняемых независимо от потоков федерата. Внутренние потоки отвечают за корректную работу инфраструктуры RTI. Кроме того, инфраструктура использует один специальный поток управления специально для выполнения обратных вызовов – Federate Service Thread (FST) (рисунок 51).

Использование потока FST, даёт возможность совершения обратных вызовов в любой момент времени, и делает ненужным вызов «evokeCallback». Такая прогрессивная модель

используется по умолчанию в системе pRTI [23]. Однако подобное поведение инфраструктуры RTI усложняет процесс проектирования федератов, заставляя его разработчиков думать о предотвращении гонок между потоками управления системы.

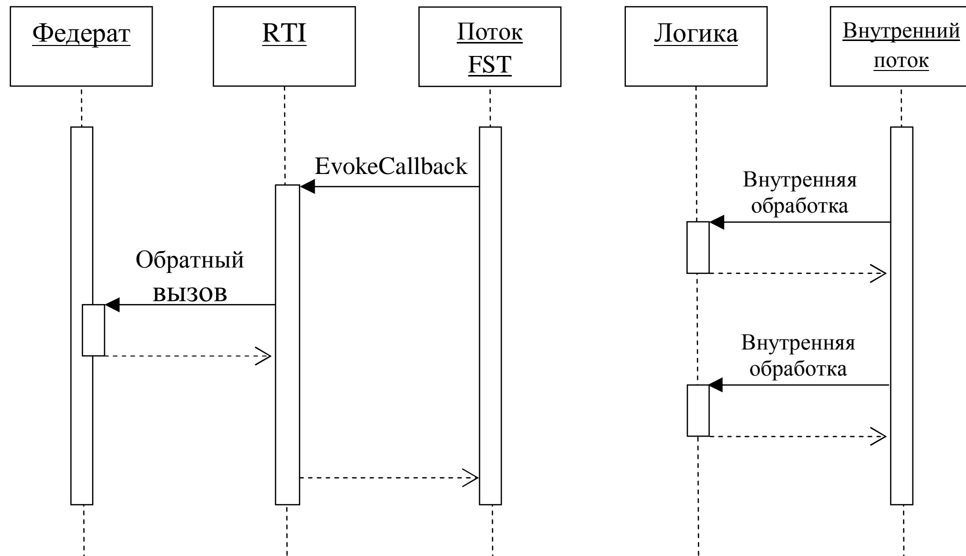

Рисунок 51. Схема работы многопоточной модели управления.

3.2.4 Выбор новой модели управления для системы CERTI

Описанные модели потоков управления обладают своими преимуществами и недостатками. Эффективность их практического применения во многом зависит от круга решаемых имитационных задач, требований заказчиков и стратегии разработчика.

Однопоточная модель теоретически позволяет достичь максимальной эффективности использования процессора, так как при её оптимальном использовании требуется минимальное число переключений контекста при своём оптимальном использовании. Однако на практике такое использование модели затруднено или даже невозможно, так как её управление требует учитывать множество разнообразных и не всегда зависящих от разработчика параметров. Таким образом, разработка и дальнейшая поддержка эффективных моделей с использованием однопоточной модели управления требует неоправданно больших затрат.

Асинхронная модель лучше подходит для практического использования. Уступая однопоточной модели в теоретических оценках производительности для лучшего случая, она обладает достаточной эффективностью для решения большинства задач. При этом

асинхронная модель практически не требует дополнительной настройки при изменении условий решаемой задачи или параметров используемой системы.

Многопоточная модель наиболее устойчива к изменениям, и позволяет достичь неплохих показателей производительности без какой-либо дополнительной настройки. Таким образом, уменьшаются трудозатраты на разработку, развёртывание и поддержку имитационной модели. Кроме того, многопоточная модель позволяет ослабить связи между аппаратной платформой, используемой системой моделирования и выполняемой с их помощью имитационной моделью. Однако такой подход заставляет разработчиков федератов писать более сложные многопоточные программы.

В рамках настоящего проекта наиболее оправданной для внедрения представляется асинхронная модель потоков управления. С одной стороны её использование позволит достигать неплохих показателей производительности, не требуя при этом сложной дополнительной настройки имитационной модели. С другой стороны в отличие от многопоточной модели, асинхронная модель потоков управления способна корректно работать с любыми федератами и не требует потокобезопасного кода. Кроме того, асинхронная модель потоков управления наиболее похожа на модель процессов, используемую в системе CERTI, где процесс федерата отвечает за осуществление прямых и обратных вызовов, а процесс RTIA – за корректную работу инфраструктуры RTI. Таким образом, реализация асинхронной модели потребует минимального количества изменений в исходном коде данной реализации.

В перспективе развития настоящей работы целесообразным можно считать создание системы моделирования, поддерживающей сразу несколько моделей управления потоками в зависимости от заданной конфигурации: асинхронную и многопоточную. Такой подход позволит совмещать преимущества двух моделей потоков. Если федераты имитационной модели поддерживают возможность многопоточной обработки, то соответствующая модель управления может быть более эффективна для её выполнения. Если же федераты пользовательская модель этим свойством не обладает – она всё равно может быть выполнена с помощью, возможно, менее эффективной асинхронной модели. Более того, компоненты RTI, расположенные на различных узлах распределённой системы, могут иметь разную реализацию. Таким образом, становится возможным совмещение компонент, построенных на основе разных моделей потоков управления, внутри одной и той же имитационной модели.

3.2.5 Влияние стратегий выполнения на производительность системы

Значительное влияние на производительность систем, использующих в своей реализации несколько потоков управления, оказывают механизмы их взаимодействия и синхронизации. Существуют две основных стратегии: *активное ожидание* или *пассивное ожидание* [21]. В настоящем разделе рассматривается эффективность их применения к рассмотренным выше моделям потоков управления, при решении простейших задач имитационного моделирования.

Поток, выполняющийся в соответствии с правилами активного ожидания, циклически переходит из спящего состояния, выполняет появившиеся за время его сна задачи и снова засыпает на заданный временной интервал. Пребывая в состоянии пассивного ожидания, поток ждёт появления новой задачи, выполняет её и сразу же засыпает.

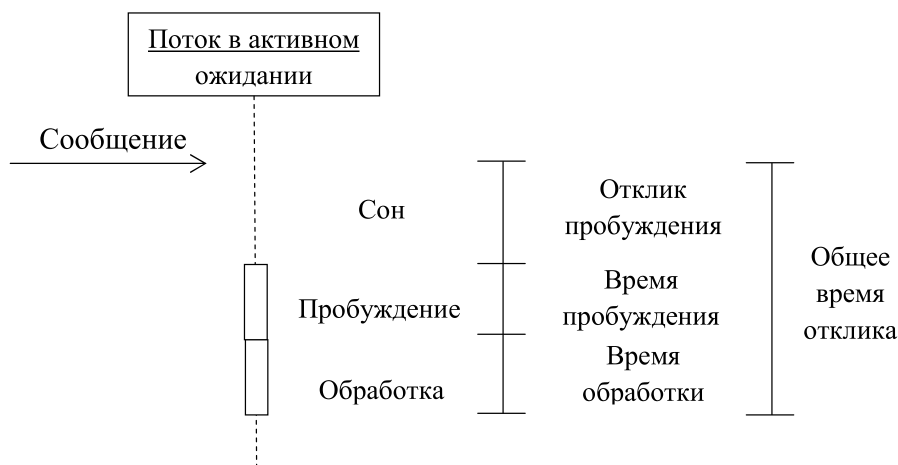

Рисунок 52. Отклик потока при активном ожидании.

Стратегии выполнения потоков можно так же изобразить графически. На рисунке 52 изображён процесс обработки поступившего сообщения в зависимости от выбранной стратегии. Время отклика потока управления в активном ожидании можно разделить на три части: время отклика пробуждения, время пробуждения, время обработки. При использовании пассивного ожидания время отклика пробуждения всегда равно нулю (рисунок 53).

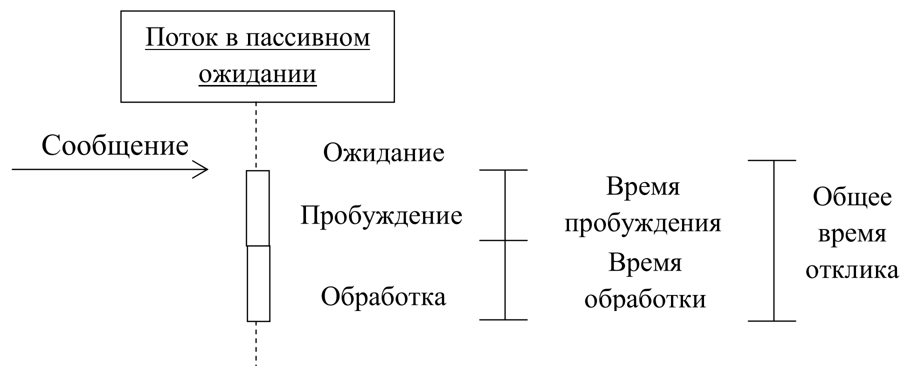

Рисунок 53. Отклик потока при пассивном ожидании.

Во время обработки одного сообщения оба рассмотренных механизма ожидания потребляют примерно одинаковое количество процессорного времени. При этом пассивное ожидание обычно обладает меньшим временем отклика. Однако реальная разница между двумя подходами проявляется при рассмотрении быстрой передачи нескольких сообщений. При активном ожидании поток управления может некоторое время не обрабатывать появившиеся сообщения, а затем обработать их все сразу. Поэтому система становится менее отзывчивой, но потребляет при этом меньше процессорного времени, экономя на переключении контекста процессора. Применение пассивного ожидания даёт обратную картину: система более отзывчива, так как обрабатывает вновь поступившие сообщения так быстро, как только это возможно, но потребляет при этом больше ресурсов процессора.

Разные модели потоков управления могут использовать различные комбинации механизмов ожидания для поддержания внутренней логики инфраструктуры RTI и обработки обратных вызовов. Однопоточная модель потоков управления всегда использует активное ожидание. В асинхронной модели для выполнения обратных вызовов используется активное ожидание, а для обработки внутренних задач может применяться любая из двух описанных стратегий. Многопоточная модель управления, использующая дополнительный поток FST, может реализовывать любую стратегию ожидания как для обработки внутренних задач RTI, так и для совершения обратных вызовов.

Время отклика модели

Рассмотрим имитационную модель BCVMTest, в которой два федерата итеративно обмениваются друг с другом сообщениями по правилам игры в пинг-понг, замеряя при этом время каждой итерации. Реализация инфраструктуры RTI, использующая механизм активного ожидания, продемонстрирует в данном тесте большое время отклика, так как её

внутренний поток управления будет засыпать после обработки каждого сообщения. Пусть, например, внутренний поток управления засыпает на 2*N секунд. Тогда между прибытием сообщения и его обработкой в среднем пройдёт N секунд.

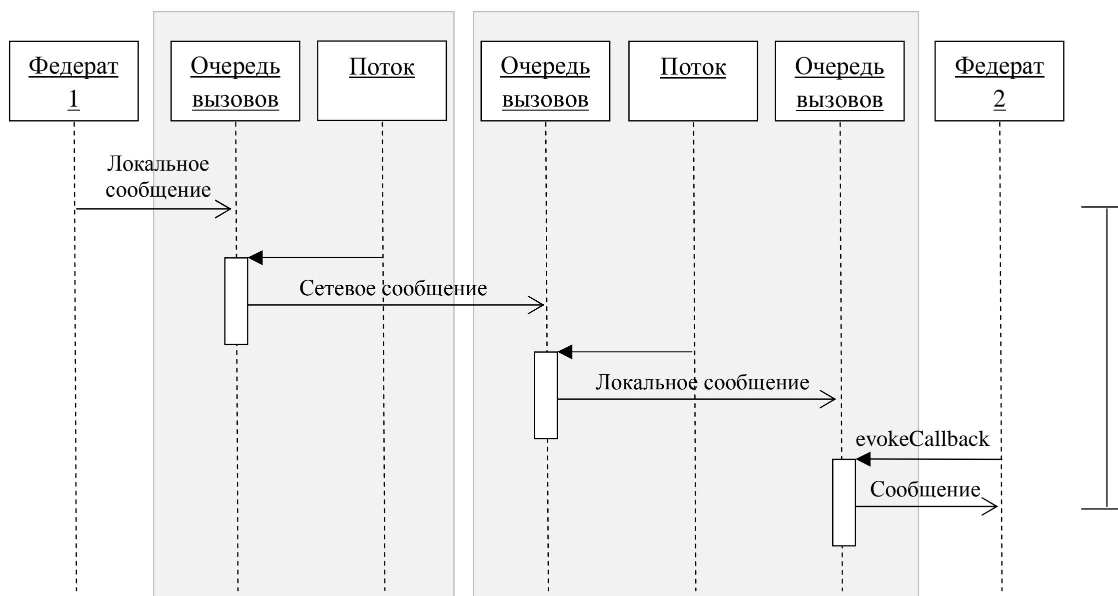

Рисунок 54. Отклик асинхронной модели при использовании активного ожидания.

На рисунке 54 изображена передача сообщения между двумя федератами в RTI, использующей асинхронную модель потоков управления и активное ожидание. Сообщение генерируется первым федератом и помещается в очередь процесса RTI, работающего на локальной машине. Локальный процесс проверяет очередь сообщений через регулярные промежутки времени. Если при этом обнаруживается новое сообщение, то оно пересылается (возможно, не напрямую) на машину, обрабатывающую федерат получатель, где заносится в очередь сообщений для обработки. Локальный компонент RTI, работающей на этой машине, регулярно проверяет эту очередь и перемещает новые сообщения для федерата в очередь обратных вызовов. В результате, исходное сообщение будет доставлено федерату, как только он сможет его обработать.

Реализация инфраструктуры RTI, построенная на основе механизма пассивного ожидания, будет обладать меньшим временем отклика, так как внутренняя обработка сообщения будет производиться сразу после его получения. Если же в реализации используется не асинхронная, а многопоточная модель управления, то сообщение будет доставлено федерату-получателю без дополнительного ожидания.

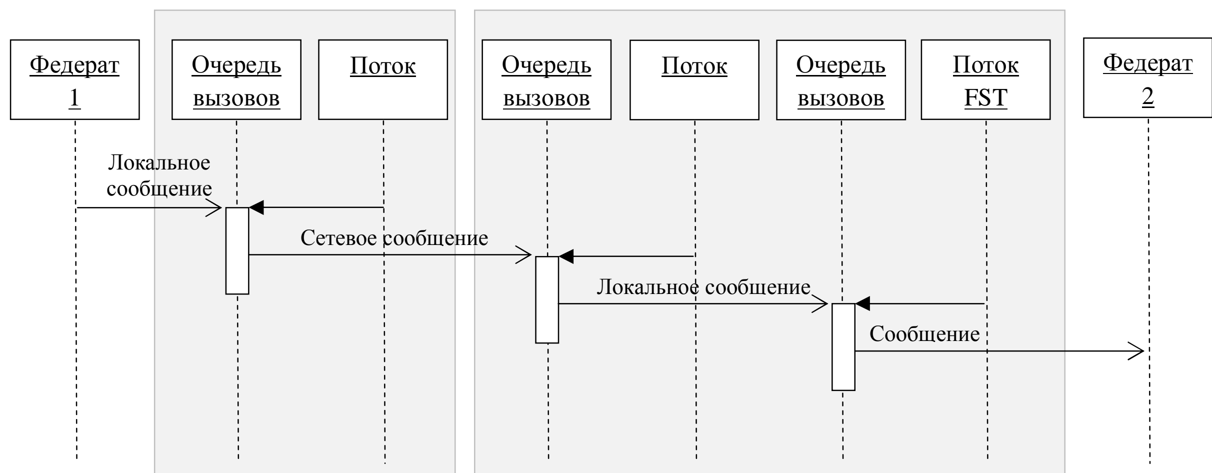

Рисунок 55. Отклик многопоточной модели при использовании пассивного ожидания.

На рисунке 55 изображена передача сообщения между двумя федератами, использующими механизм пассивного ожидания и многопоточную модель управления. Сообщение генерируется первым федератом и помещается в очередь процесса RTI, работающего на локальной машине. Локальный процесс сразу же проверяет очередь сообщений и пересылает его на машину, обрабатывающую федерат получатель. Там сообщение заносится в очередь для обработки. Сразу после этого пробуждается соответствующий локальный процесс RTI, который перемещает сообщение в очередь обратных вызовов. Так как для обратных вызовов инфраструктура RTI использует специальный поток управления FST, то сообщение тут же доставляется федерату. Таким образом, использование многопоточной модели управления и механизма пассивного ожидания обеспечивает минимальное время передачи одного сообщения.

Пропускная способность системы моделирования

Рассмотрим модель BasicTest, служащая для оценки пропускной способности системы. По сценарию данного теста один федерат быстро пересылает большое количество разнородных сообщений другому федерату. При этом значение пропускной способности рассчитывается в виде отношения переданного объёма данных ко времени, которое заняла его пересылка.

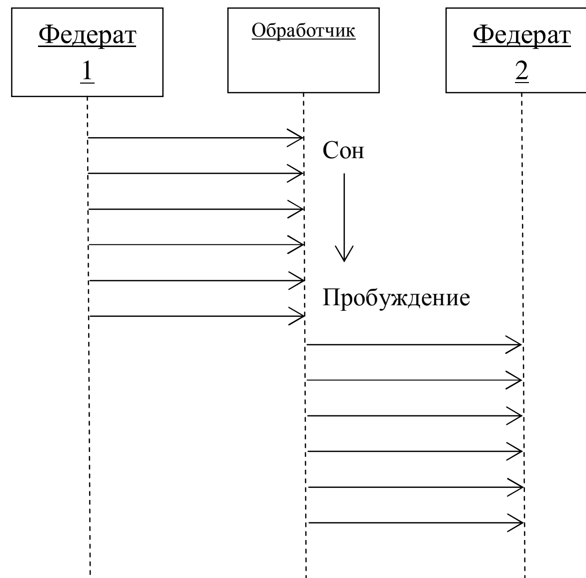

Рисунок 56. Пропускная способность активного ожидания, лучший случай.

На рисунке 56 изображена схема выполнения теста BmThruput инфраструктурой RTI, реализующей механизм активного ожидания. Изначально локальный процесс RTI находится в спящем состоянии, а все входящие сообщения федерата помещаются в очередь. Затем процесс пробуждается, и сразу обрабатывает все накопленные сообщения, отправляя их второму федерату. Таким образом, при передаче сообщений появляется задержка, но на несколько сообщений приходится единственное переключение контекста.

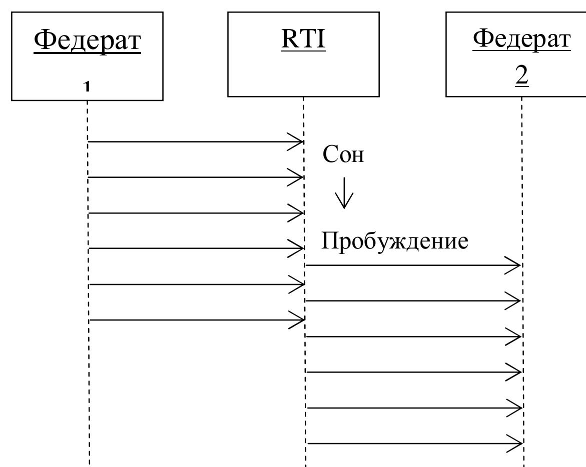

Рисунок 57. Пропускная способность пассивного ожидания, лучший случай.

На рисунке 57 изображена схема выполнения того же теста с использованием механизма пассивного ожидания в оптимальной для этого ситуации. Локальный процесс инфраструктуры RTI обрабатывает все поступающие сообщения за одно переключение

контекста и возвращается в состояние ожидания. Таким образом, пассивное ожидание теоретически позволяет достигнуть лучших результатов, так как обладает меньшим временем отклика при тех же затратах процессорного времени.

Федерат 1

Обработчик

Федерат 2

Сон

Пробуждение

Сон

Пробуждение

Рисунок 58. Пропускная способность активного ожидания, худший случай.

Теперь рассмотрим эффективность работы каждой конфигурации в худшем для них случае. На рисунке 58 изображена работа системы, реализующей механизм активного ожидания. Пусть на этот раз федерат отправляет свои сообщения с меньшей скоростью. Локальный процесс RTI пробуждается, обнаруживает несколько ожидающих просмотра сообщений, быстро их обрабатывает и снова засыпает. При этом он, однако, не успевает обнаружить сообщения, пришедшие немного позднее, и они откладываются до следующей итерации обработки. Таким образом, среднее время задержки перед обработкой сообщений значительно увеличивается.

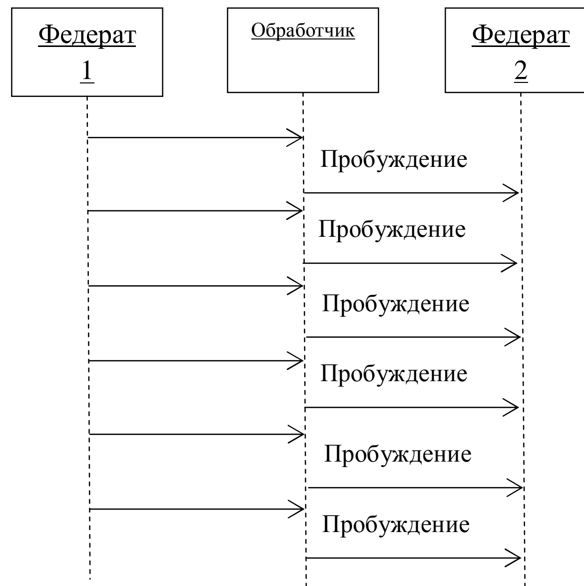

Рисунок 59. Пропускная способность пассивного ожидания, худший случай.

На рисунке 59 изображена схема работы системы, построенной на основе механизма пассивного ожидания, в худшем случае. Пусть первый федерат снова пересылает сообщения с меньшей скоростью. Тогда локальный процесс RTI может успевать обработать вновь поступившее сообщение и снова уйти в режим ожидания. Таким образом, переключение контекста будет происходить при обработке каждого поступившего сообщения. С другой стороны, среднее время отклика такой системы будет меньше, чем системы на основе механизма активного ожидания. Теоретически использование пассивного ожидания всегда даёт лучшие показатели производительности, но затрачивает больше процессорного времени.

3.2.6 Выбор стратегии выполнения потоков управления

Компромисс между расходом процессорного времени и временем отклика модели должен строиться индивидуально на основе требований, предъявляемых каждой конкретной имитационной моделью. Иногда целесообразно пойти на увеличение времени отклика, чтобы сохранить процессорное время для других задач.

Использование активного ожидания при его правильной настройке позволяет расходовать меньше процессорного времени. Однако оптимизация его настроек учитывать большое количество разнообразных параметров, и крайне неустойчива к их изменениям. Поэтому в рамках настоящей работы более перспективным механизмом синхронизации представляется пассивное ожидание. Хоть оно и расходует больше процессорного времени,

но позволяет при этом достичь меньшего времени отклика модели – что принципиально при решении задачи моделирования систем реального времени. К тому же, пассивное ожидание практически не нуждается в настройке и дополнительной пользовательской оптимизации.

3.2.7 Практическая реализация

На данном этапе настоящей научно-исследовательской работы было осуществлено частичное внедрение выбранных модели потоков управления и стратегии их выполнения в существующую архитектуру инфраструктуры CERTI RTI. При этом была сохранена возможность сборки традиционной для данного средства централизованной модели, не использующей многопоточные процессы. Для этого к опциям системы сборки был добавлен специальный конфигурационный параметр: RTIA_BUILD_THREADS. В случае его истинности, во время инициализации инфраструктуры RTI, библиотека libRTI вместо запуска независимого процесса RTIA запускает вспомогательный поток, обладающий идентичной функциональностью.

Для работы с потоками данных была использована библиотека boost_thread [24], написанная на языке C++ и, де факто, являющаяся стандартным решением для подобного круга задач. Как и система CERTI, данная библиотека кроссплатформенная. Таким образом, код, написанный с её помощью, может быть использован сразу несколькими операционными системами, и не сужает первоначальной области применения системы CERTI.

Выбранная асинхронная модель потоков управления во многом схожа с традиционной процессной моделью системы CERTI, поэтому большую часть кода процесса RTIA удалось адаптировать для запуска в рамках потока. Однако значительная часть его логики реализована на основе использования средств межпроцессного взаимодействия, применение которых нерационально для синхронизации нескольких потоков в рамках одного процесса. В частности, процесс RTIA обменивается данными с процессом федерата через сокет UNIX, передача через который требует предварительной сериализации и последующего декодирования данных.

В текущей версии прототипа потоки, хоть они и находятся в одном адресном пространстве, но используют для синхронизации тот же самый локальный канал передачи UNIX. В то же время данные между потоками могут передаваться по указателю. При этом отпадает необходимость не только в дополнительной конвертации передаваемых данных, но и в дополнительном их копировании. Таким образом, на последующих этапах настоящей

научно-исследовательской работы канал передачи должен быть заменён более эффективными средствами синхронизации – разделяемыми данными и семафорами.

Внедрение синхронизации потоков на основе семафором, однако, повлечёт за собой изменения в логике процесса RTIA. Текущая реализация процесса RTIA использует встроенные в операционную систему средства обработки мультиплексированного ввода данных, которые позволяют ему опрашивать очереди сообщений, как со стороны процесса федерата, так и со стороны центрального процесса RTIG одновременно. Однако средств, позволяющих проводить синхронизацию со вспомогательным потоком, работающим в контексте того же процесса, и внешним процессом одновременно, к сожалению не существует. Поэтому для синхронизации потока RTIA «на два фронта» на практике придётся или использовать механизм активного ожидания, регулярно проверяющий очереди поступивших сообщений, или же ещё один дополнительный поток управления. Реализация любого из предложенных решений потребует примерно эквивалентных трудозатрат, однако второй вариант позволяет использовать пассивное ожидание и, в свете проведённого исследования, кажется более предпочтительным.

Ещё один вопрос, который возникает при отказе от использования локального канала передачи данных – вопрос их временного хранения. Так как потоки могут работать с разной скоростью, то возникает необходимость буферизации передаваемых между ними данных. На следующих этапах настоящей работы нужно будет провести исследование эффективности применения для этой задачи различных структур хранения данных, минимизировав накладные расходы, связанные с излишним копированием, выделением и освобождением динамической памяти.

3.3 Оптимизация шаблона Cheetah для трансляции в исполняемые модели, совместимые со стандартом HLA

Средство для генерации кода имитационных моделей на основании диаграмм состояний UML (Генератор) было реализовано на языке Питон. Для создания и редактирования диаграмм состояний UML использовалось средство ArgoUML. Однако генератор не работает напрямую с диаграммой UML, а только с её XML представлением. В качестве XML представления был выбран специализированный XML формат, предназначенный для работы с диаграммами состояний -   State Chart XML. Данный формат позволяет описывать конечные автоматы в общем виде на основе диаграмм состояний Харела. Используя SCXML можно описать различные типы структур конечных автоматов. В

качестве примера можно привести такие случаи, как вложенность, параллельность, синхронизация или параллельность подавтоматов.

Для работы с шаблонами генерации кода использовалась специализированная библиотека шаблонов Cheetah. Суть работы данной библиотеки следующая, шаблон компилируется с помощью cheetah-компилятора в представление шаблонов на языке Питон. Данный файл используется загрузчиком питона для генерации выходного файла. Шаблон Cheetah состоит из комбинации выходного текста и кода, похожего на исходный код языка Питон. На рисунке 60 приведен пример шаблона для генерации C++ класса.

```
\#ifndef EVENTS_H
\#define EVENTS_H
enum EVENTS
{
#set $events = [ ]
#for $state in $stateChartXML.state:
  #if $state.__dict__.has_key('transition'):
    #for transition in state.transition:
      #if $events.count($transition.event) == 0:
        #silent $events.append($transition.event)
      #end if
    #end for
  #end if
#end for
#silent $events.sort()
#for $event in $events:
Event_${event},
#end for
};
//-------------------------------------------------------------------------
#end if
```

Рисунок 60. Шаблон для генерации C++ класса.

На вход генератору подается диаграмма состояний языка UML и шаблон для генерации кода. Далее диаграмма переводится во внутренние представление языка Питон. После создания SCXML файла, он подаётся на вход непосредственно генератору кода по шаблону, который генерирует **.h .cpp .fed** файлы. Для генерации кода федератов (с правильными интерфейсами для подключения к RTI) были созданы особые шаблоны: отдельно для **.h, .cpp** и **.fed** файлов. На рисунке 61 представлена схема работы генератора кода моделей совместимых со стандартом HLA.

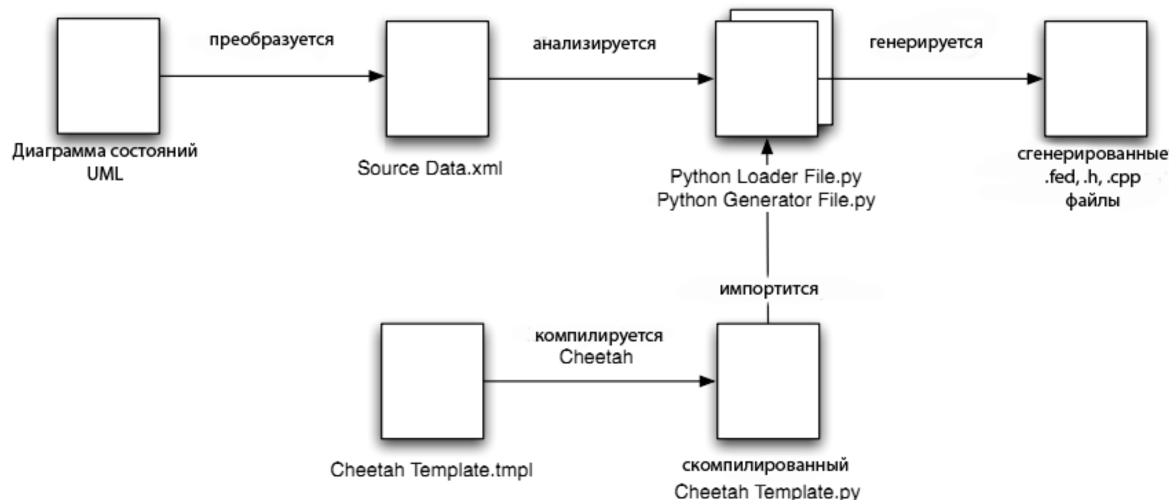

Рисунок 61. Схема работы генератора кода.

Архитектура имитационной модели обладает своей спецификой и определяет набор используемых и интегрированных библиотек. Ниже перечислены требования к поддержке автогенерации кода:

- должна быть обеспечена возможность генерации программных компонент реализующих взаимодействие федератов в федерации, согласно стандарту HLA;
- код должен содержать специфические для подключения к HLA RTI интерфейсы (передаваемые параметры и сообщения между федератами);
- код должен быть правильно отформатирован;
- должны быть учтены различные варианты генерации кода:
  - создание федерата
  - описание внутренней логики модели (внутри федерата)
  - добавление реализованного компонента

Это означает, что базовых функций среды моделировании не достаточно для генерации кода. Как было написано выше в данном проекте предложено использовать функциональность библиотеки Cheetah для генерации кода имитационной модели по заданным шаблонам, однако, представленные на втором этапе шаблоны не позволяли описывать полный спектр федератов в соответствии со стандартом HLA. Данная проблема частично связанна со спецификой XML представления диаграмм состояний (более подробное описание в следующем разделе).

3.3.1 Оптимизация шаблона федерата

В ходе работ на данном этапе, был оптимизирован текст шаблона для генерации структуры федерата имитационной модели, без реализации внутренней логики работы федерата.

В частности, был добавлен код получения сообщения от федератов, для которых обозначено взаимодействие в файле управления федерации (рисунок 62).

Основная проблема заключалась в следующем: в диаграммах состояний (statechart) состояние на которое указывает переход, не имеет информации о том, что в него есть переход. Такой подход представлял проблему при генерации кода федерата, так как в коде федерата, с которым взаимодействуют, необходимо определять специальные конструкции (согласно стандарту HLA), для передачи параметра определенного типа. Для решения описанной проблемы в шаблоне потребовалось организовать еще один проход по XML представлению имитационной модели, чтобы проверять, является ли данный федерат, тем на который указывает другой федерат, что позволило вставить в код федерата необходимую информацию.

Так же был добавлен код обработки атрибутов федерата, и определение типов передаваемых параметров при обмене между федератами, подписанными на параметры друг друга (рисунок 63). Предыдущие версии шаблонов генерации кода федерата не учитывали возможность задания нефиксированного числа атрибутов федерата, для описания его свойств. Новая версия позволяет вставлять атрибуты в специальный тэг в XML представлении имитационной модели, что позволяет определять практически весь спектр федератов стандартизированных HLA.

```
#for $otherstate in $stateChartXML.state:
   #if $otherstate.__dict__.has_key('transition') :
      #for $transition in $otherstate.transition :
         #if ($transition.target.next == $state.id) :
rti1516::ParameterHandleValueMap ${transition.event}MessageMap;
rti1516::InteractionClassHandle ${transition.event}MessageH;
            #for $parametr in $transition.parametr :
rti1516::ParameterHandle ${parametr.name}H;
            #end for
            #for $parametr in $otherstate.parametr :
${parametr.type} ${parametr.name};
            #end for
         #end if
      #end for
   #end if
#end for
```

Рисунок 62. Шаблон кода получения сообщения от федератов

3.3.2 Результат

Средство для генерации кода HLA-совместимых моделей по шаблонам было доработано и на данный момент позволяет создавать по шаблону и scxml представлению федерации полностью функциональный шаблон федерата. Экспериментальные исследования показали работоспособность генерируемого кода в разрабатываемой системе имитационного моделирования. Реализованное средство позволяет сократить время разработки имитационной модели. Единственное, что остается пользователю – описать внутреннюю логику функционирования федерата в цикле mainLoop().

Вопрос генерации кода внутренней логики федерата на основании описывающих её конечных автоматов будет рассмотрен на следующем этапе проекта. Предполагается создание дополнительных шаблонов для генерации кода контроллера, исполняющего описанный автомат, и оформление состояний автомата в виде отдельных C++ классов. Так же необходимо предусмотреть интерфейс подключения готовых модельных компонент: возможность использовать реализованные C++ классы и функции в разрабатываемой имитационной модели.

```
#if $state.__dict__.has_key('attribute') :
   #if $state.attribute.__dict__.has_key('regulating'):
enableRegulationModeWrapper();
   #end if
   #if $state.attribute.__dict__.has_key('constrained'):
enableConstrainedModeWrapper();
   #end if
#end if
#if $state.__dict__.has_key('transition') :
   #for $transition in $state.transition
publishWrapper( ${transition.event}MessageH );
   #end for
#end if
#for $otherstate in $stateChartXML.state:
   #if $otherstate.__dict__.has_key('transition') :
      #for $transition in $otherstate.transition :
         #if ($transition.target.next == $state.id) :
subscibeWrapper( ${transition.event}MessageH );
         #end if
      #end for
   #end if
#end for
#if $state.__dict__.has_key('parametr') :
   #for $parametr in $state.parametr :
${parametr.name}X = ${parametr.initval};
   #end for
#end if
```

Рисунок 63. Шаблон кода обработки атрибутов федерата.

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

На предыдущем этапе проекта[2] 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 системе верификации UPPAAL для спецификации сетей временных автоматов используется некоторое подмножество формул темпоральной логики деревьев вычислений (Computational Tree Logics, CTL). В целом ряде статей (см., например, [25]) показано, что некоторые разновидности отношений бисимуляции между системами переходов сохраняют выполнимость формул CTL. Таким образом, для доказательства корректности предложенного нами алгоритма трансляции иерархических временных автоматов в сети плоских временных автоматов достаточно убедиться в том, что система переходов произвольного иерархического временного автомата находится в отношении бисимуляции с системой переходов соответствующей сети плоских временных автоматов.

Доказательство этого утверждения представлено в настоящем разделе. В качестве подходящего отношения бисимуляции нами было выбрано отношение расходящейся затемненной прореженной бисимуляции (divergence blind stuttering bisimulation), исследованной в статье [26]. Далее для краткости указанное отношение бисимуляции называется dbs-эквивалентностью. Вначале приведено краткое описание семантики иерархических временных автоматов и сетей плоских временных автоматов. В этих описаниях определяются правила построения систем переходов (моделей) для временных автоматов указанных двух видов. Затем дано краткое описание алгоритма трансляции иерархических временных автоматов в сети плоских временных автоматов. Алгоритм трансляции работает в два этапа:

1. построение иерархического временного автомата по UML-диаграмме,
2. построение сети плоских временных автоматов по иерархическому временному автомату.

Первый этап работы алгоритма не нуждается в обосновании, т. к. на этом этапе происходит переписывание графических обозначений удобной для визуального восприятия

UML-диаграммы в более удобное для формального анализа математическое описание структуры иерархического автомата с той же семантикой. Поэтому обоснование корректности алгоритма трансляции проводится для второго этапа работы алгоритма, и далее под «алгоритмом трансляции» подразумевается только второй его этап. С подробными описаниями семантик автоматов указанных видов и полным описанием алгоритма трансляции можно ознакомится в материалах отчета по предыдущему этапу данному проекту.

Далее в данном разделе приведены строгие математические определения структур Крипке, соответствующих моделям иерархических и плоских автоматов, даны определения dbs-эквивалентности структур Крипке и отношения выполнимости формул того подмножества CTL, которое используется в системе верификации UPPAAL. В заключение этого раздела доказано утверждение о dbs-эквивалентности модели произвольного иерархического временного автомата и модели сети плоских временных автоматов, в которую транслируется этот иерархический автомат.

3.4.1 Модель иерархических временных автоматов

Иерархический временной автомат (далее – просто автомат) — это система

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

состоящая из следующих компонент:

1. S — множество состояний,
2. $S_0 \subseteq S$ — множество инициальных состояний,
3. $\eta : S \to 2^S$ — функция вложенности состояний,
4. Type : S → {AND, XOR, BASIC, ENTRY, EXIT} – функция типов состояний,
5. Var – множество переменных,
6. Clocks – множество таймеров,
7. Chan – множество каналов,
8. Inv : S → Invariant – функция инвариантов состояний,
9. $T \subseteq S \times (Guard \times Sync \times Reset \times \{true, false\}) \times S$ — множество переходов,
10. Chantype :Chan → {h, b} — функция типов каналов.

Каждый элемент множества Invariant — это либо одна из констант true или false, либо множество выражений вида «x op n» или «(x-y) op n», где x, y — таймеры из множества Clocks, n – целочисленная константа, op — один из знаков <, ≤, =.

Каждый элемент множества Guard — это либо одна из констант true или false, либо множество предикатов. При этом каждый предикат имеет вид либо «E' op E''», где E', E'' – арифметические выражения над переменными из множества Var, op – один из знаков сравнения (<, =, >, ≤, ≥), либо «x op n», где x – таймер из множества Clocks, n — целочисленная константа, op — также один из знаков сравнения.

Каждый элемент множества синхронизаций Sync имеет вид либо «c!» (отправление сообщения по каналу c), либо «c?» (прием сообщения по каналу c), либо «none» (отсутствие отправления/приема сообщения). При этом функция Chantype для каждого канала определяет, является ли этот канал широковещательным (broadcast) или каналом взаимодействия типа «точка-точка» (handshake).

Каждый элемент множества Reset является множеством присваиваний вида «x ← 0», где x – таймер из множества Clocks, и «v ← E», где v – переменная из множества Var, а E – арифметическое выражение над переменными из множества Var.

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

Если не учитывать функции вложенности состояний, иерархический автомат может рассматриваться как помеченный ориентированный мультиграф. В связи с этим к автомату будет также применяться терминология, используемая в теории графов. В частности, переходы при таком рассмотрении оказываются дугами мультиграфа, несущими четыре пометки. По порядку, обозначенному при определении модели, будем называть эти пометки, соответственно, предусловием, синхронизацией, присваиванием и флагом срочности. Если флаг срочности имеет значение true, соответствующий переход будем называть срочным, иначе — несрочным. Кроме того, каждая вершина рассматриваемого мультиграфа помечена ровно одним из пяти типов, определяемых функцией Type, инвариантом, определяемым функцией Inv, и может быть помечена как инициальная.

Ограничения на систему, обеспечивающие корректность автомата, описываются следующим образом:

- ограничения на структуру состояний:
  - ❖ функция η задает ориентированное корневое дерево с множеством Sв качестве вершин и дугами, идущими от корня к листьям дерева (в дальнейшем будем называть его деревом состояний);
  - ❖ все метасостоянияи только они имеют потомков в дереве состояний;

- ❖ потомками метасостояния типа AND в дереве состояний могут быть только входы, выходы и метасостояния;
- ограничения на множество инициальных состояний:
  - ❖ корень дерева состояний является инициальным состоянием;
  - ❖ инициальными могут быть только простые состояния и метасостояния;
  - ❖ если некоторое состояние является инициальным, то и его предок в дереве состояний является инициальным состоянием;
  - ❖ среди потомков состояния типа XOR в дереве состояний ровно один является инициальным;
  - ❖ все потомки состояния типа AND в дереве состояний являются инициальными;
- ограничения на встречающиеся выражения:
  - ❖ любые две дуги, исходящие из входа, вложенного в состояние типа AND, несут присваивания, использующие непересекающиеся множества переменных;
  - ❖ инварианты входов и выходов суть константы true;
- ограничения на переходы:
  - ❖ из входа, вложенного в метасостояние типа XOR, исходит ровно одна дуга;
  - ❖ из входа, вложенного в метасостояние типа AND, исходит ровно по одной дуге, ведущей в каждого потомка, вложенного в данноеметасостояние и также являющегося метасостоянием, и больше не исходит никаких дуг;
  - ❖ все ограничения на соотношение инцидентных переходам состояний приведены на рисунке 64; на этом рисунке кругами обозначены простые состояния, тонкими линиями — входы, толстыми линиями — выходы, квадратами — метасостояния, стрелками — переходы, вложенность состояний соответствует геометрической вложенности на рисунке;
  - ❖ все псевдопереходы (дуги, исходящие из входа или ведущие в выход) являются несрочными и не несут синхронизации (т.е. несут синхронизацию «none»);
  - ❖ если дуга, являющаяся псевдопереходом, исходит из входа, то она не несет предусловия (т.е. несет предусловие true);
  - ❖ если дуга, являющаяся псевдопереходом, ведет в выход, то она не несет присваиваний (т.е. несет присваивание, являющееся пустым множеством);
  - ❖ если дуга, являющаяся псевдопереходом, исходит из выхода и ведет в выход, то она не несет ни предусловия, ни присваиваний.

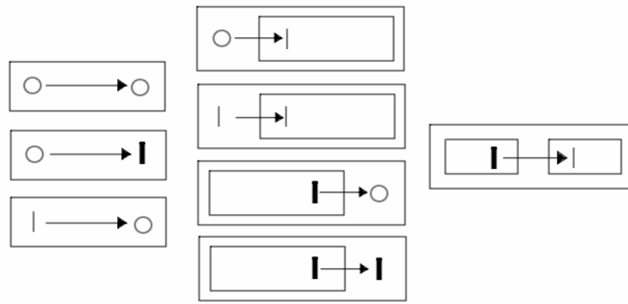

Рисунок 64. Ограничения на соотношение инцидентных переходам состояний.

3.4.2 Модель сети плоских временных автоматов

Сеть плоских временных автоматов (далее – просто сеть) — это система

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

состоящая из следующих компонентов:

- A – множество процессов (определение процесса приведено далее),
- Vars – множество переменных, каждая из которых принимает значения из заранее ограниченного с обеих сторон целочисленного отрезка,
- Clocks – множество таймеров,
- Chan – множество каналов синхронизации,
- Chantype : Chan → {h, b} – функция типов каналов,
- Chanurg : Chan → {c, u} – функция срочности каналов.

Каналы, как и в модели автоматов, могут быть широковещательными и каналами типа «точка-точка». Кроме того, каждый канал может быть помечен функцией Chanurg как срочный (urgent).

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.

3.4.3 Описание алгоритма трансляции

Алгоритм трансляции принимает на вход корректный иерархический автомат

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

В результате работы алгоритма трансляции конструируется сеть плоских временных автоматов

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

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 вершиной.

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, вложенному во вложенную компоненту, ведется дуга в соответствующую вложенную компоненту, отвечающая активации данной компоненты с задействованием данного 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 подпараграфе «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 Mперед началом основной обработки добавляется процесс, состоящий из одного состояния и одной дуги, несущей синхронизацию «Hurry?» по срочному каналутипа «точка-точка». При

обработке срочного перехода, не несущего синхронизации, соответствующей дуге процесса приписывается синхронизация «Hurry!».

Каждый канал «с» в множестве каналов HTAпорождает в результирущей модели четыре канала для моделирования взаимодействия двух несрочных, несрочного со срочным, срочного с несрочным и двух срочных переходов с синхронизацией по каналу «с» в HTA. 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 XORкомпоненты 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 втем, 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 специального вида. Дуги этого дерева направлены к корню, в качестве которого выступает рассматриваемый выход.. Листьями дерева являются простые состояния. Остальные вершины дерева - это все те выходы, из которых переходом по дугам через другие выходы достижим рассматриваемый выход.

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 заводятся 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.

3.4.4 Корректность алгоритма трансляции

3.4.4.1 Структуры Крипке и отношение бисимуляции

Введем определения, с помощью которых будут описаны поведение автоматов и поведение сетей и обоснована корректность алгоритма трансляции.

Пусть задано непустое множество предложений A. Записью 2^A будем обозначать множество всех подмножеств A.

Определение. *Структурой Крипке над множеством A* называется система $(S, \mathcal{L}, \rightarrow)$, где

- S – произвольное множество состояний,
- $\mathcal{L} : S \rightarrow 2^A$ – разметка состояний,
- $\rightarrow \subseteq S \times S$ – множество переходов.

Определение. *Трассой* структуры Крипке $K = (S, \mathcal{L}, \rightarrow)$, начинающейся в состоянии s, где $s \in S$, будем называть максимальный маршрут в графе $(S, \rightarrow)$, начинающийся в вершине s.

Определение. Пусть $K_1 = (S_1, \mathcal{L}_1, \rightarrow_1)$, $K_2 = (S_2, \mathcal{L}_2, \rightarrow_2)$ – две произвольные структуры Крипке над общим множеством A. Отношение $R \subseteq S_1 \times S_2$ будем называть *dbs-эквивлентностью (divergence blind stuttering bisimulation),* если для любой пары (s_1, s_2), входящей в это отношение, выполнены следующие условия:

1. $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$;
2. если для состояния s_1' множества S_1 верно соотношение $s_1 \rightarrow_1 s_1'$, то существует такая последовательность $s^0, s^1, \ldots, s^n$, $n \geq 0$, состояний множества S_2, что
    - $s^0 = s_2$,
    - $s^0 \rightarrow_2 s^1 \rightarrow_2 \ldots \rightarrow_2 s^n$ и

- для некоторого i верны соотношения s_1 R s^0, …, s_1 R s^i, s_1' R s^{i+1}, …, s_1' R s^n;

3. если для состояния s_2' множества S_2 верно соотношение $s_2 \to_1 s_2'$, то существует такая последовательность $s^0, s^1, \ldots, s^n$, $n \geq 0$, состояний множества S_1, что
   - $s^0 = s_1$,
   - $s^0 \to_1 s^1 \to_1 \ldots \to_1 s^n$ и
   - для некоторого i верны соотношения s^0 R s_2, ..., s^i R s_2, s^{i+1} R s_2', ..., s^n R s_2'.

Состояния $s_1 \in S_1$, $s_2 \in S_2$ будем называть dbs-эквивалентными, если существует отношение R, являющееся dbs и такое, что s_1 R s_2.

В статье [26] было показано, что если состояния s_1, s_2 структур Крипке над общим множеством являются dbs-эквивалентными, то множества формул логики CTL*-X, истинных в состояниях s_1 и s_2, совпадают. Этот факт будет существенно использоваться при обосновании корректности алгоритма трансляции.

3.4.4.2 Поведение иерархического временного автомата

Рассмотрим автомат H = (S, S_0, η, Type, Vars, Clocks, Chan, Inv, T, Chantype). Записью $root_H$ будем обозначать метасостояние автомата H, не вложенное ни в какое другое метасостояние.

Считаем для простоты, что в автомате H все переменные булевы. Такое допущение можно сделать, т.к. переменная ограниченного целого типа моделируется конечным множеством булевых переменных.

Определение. *Конфигурацией* автомата H назовем тройку (ρ, μ, ν), где

- $\rho : S \to 2^S$ – функция, определяющая дерево активных состояний HTA,
- μ : Vars → { true, false } – значение переменных HTA,
- ν : Clocks → R_+ – значение таймеров HTA.

Определение. *Начальной конфигурацией* HTA H будем называть конфигурацию (ρ, μ, ν), где

- $\rho^*(root_H) = S_0$,
- μ(x) = false для всех переменных x из множества Vars и
- ν(y) = 0 для всех таймеров y из множества Clocks.

Обозначим записью Expr(Vars, Clocks) множество, содержащее выражения вида ($x \bowtie c$), ($x - y \bowtie c$) и ($a \bowtie b$), где $x, y \in$ Clocks, $a, b \in$ Vars $\cup$ Z, $c \in$ Z, $\bowtie \in \{ <, \leq, =, \geq, > \}$.

Поведение автомата H описывает структура Крипке K = (S, $\mathcal{L}$, $\rightarrow$) над множеством Expr(Vars, Clocks), где

- S – множество всех конфигураций H,
- $\mathcal{L}$ (s) есть множество выражений из Expr, истинных в конфигурации s,
- s' $\rightarrow$ s'' тогда и только тогда, когда в автомате H в конфигурации s'
    - возможен переход (или группа переходов соответственно синхронизации), по выполнении которого (которых) автомат H из конфигурации s' переходит в конфигурацию s'' или
    - возможно продвижение времени, приводящее конфигурацию s' к конфигурации s''.

3.4.4.3 Поведение сети плоских автоматов

Рассмотрим сеть P = (A, Vars, Clocks, Chan, Chantype, Chanurg).

Каждый процесс p в множестве A имеет вид p = (L_p, T_p, $Type_p$, Inv_p, l^0_p).

Для простоты считаем, что в сети P все переменные булевы. Такое допущение можно сделать, т.к. каждая булева переменная автомата транслируется в булеву переменную сети и все остальные переменные, порождаемые алгоритмом трансляции, моделируются конечным числом булевых переменных.

Определение. *Конфигурацией сети P* будем называть тройку (l, μ, ν), где

- $l : A \rightarrow \cup\{p \in A\}\ L_p$ – активные вершины сети,
- верно соотношение $l(p) \in L_p$,
- μ : Vars $\rightarrow$ { true, false } – значение переменных сети,
- ν : Clocks $\rightarrow R_+$ – значение таймеров сети.

Определение. *Начальной конфигурацией сети P* будем называть конфигурацию (l, μ, ν), где

- l(p) = idle для всех процессов p, $p \in A$,
- $\mu(x)$ = false для всех переменных x из множества Vars и
- $\nu(y) = 0$ для всех таймеров y из множества Clocks.

Конфигурацию (l, μ, ν) сети P, содержащую активные committed-вершины (т.е. для некоторых p и v верно $l(p) = v$ и $Type_p(v) = committed$), будем называть срочной.

Поведение сети P описывает структура Крипке K = (S, $\mathcal{L}$, →) над множеством Expr(Vars, Clocks), где

- S – множество всех конфигураций P,
- $\mathcal{L}$ (s) есть множество выражений из Expr, истинных в конфигурации s,
- s' → s'' тогда и только тогда, когда в сети P в конфигурации s'
  - возможен переход (или группа переходов соответственно синхронизации), по выполнении которого (которых) сеть P переходит из конфигурации s' в конфигурацию s'' или
  - возможно продвижение времени, приводящее конфигурацию s' к конфигурации s''.

3.4.4.4 Допустимые формулы

Определим синтаксис и сематнику формул, проверка которых на сети поддерживается в средстве верификации UPPAAL.

Определение. *Допустимая формула ψ* сети P = (A, Vars, Clocks, Chan, Chantype, Chanurg) описывается грамматикой

ψ ::= AF f | AG f | EF f | EG f | f --> f ,

где f – локальное свойство, описываемое грамматикой

f ::= (deadlock) | (a.v) | (e) | (not f) | (f or f) | (f and f) | (f imply f) ,

где $a \in A$, $l \in L_a$, $e \in$ Expr(Vars, Clocks).

Определим семантику допустимых формул на сети P. Пусть K = (S, $\mathcal{L}$, →) – структура Крипке, описывающая поведение сети P.

Определение. Истинность локального свойства f в конфигурации s = (l, μ, ν) сети P ($s \models_P f$) определяется индуктивно по построению свойства:

- $s \models_P$ (deadlock) ⇔ ни для какой s' не выполнено s → s',
- $s \models_P$ (a.v) ⇔ l(a) = v,
- $s \models_P$ (e) ⇔ $e \in \mathcal{L}(s)$,
- $s \models_P$ (not f) ⇔ $\neg(s \models_P f)$,
- $s \models_P$ (f_1 or f_2) ⇔ $s \models_P f_1$ или $s \models_P f_2$,

- $s \models_P (f_1 \text{ and } f_2) \Leftrightarrow s \models_P f_1$ и $s \models_P f_2$,
- $s \models_P (f_1 \text{ imply } f_2) \Leftrightarrow \neg(s \models_P f_1)$ или $s \models_P f_2$.

Определение. Истинность допустимой формулы ψ в конфигурации s сети TA ($s \models_P \psi$) определяется следующим образом:

- $s \models_P$ AF f $\Leftrightarrow$ все трассы структуры K, начинающиеся в конфигурации s, содержат конфигурацию, в которой выполнено свойство f,
- $s \models_P$ AG f $\Leftrightarrow$ во всех конфигурациях всех трасс структуры K, начинающихся в s, выполнено свойство f,
- $s \models_P$ EF f $\Leftrightarrow$ хотя бы одна трасса структуры K, начинающаяся в конфигурации s, содержит конфигурацию, в которой выполнено свойство f,
- $s \models_P$ EG f $\Leftrightarrow$ во всех конфигурациях хотя бы одной трассы структуры K, начинающейся в конфигурации s, выполнено свойство f,
- $s \models_P f_1 \text{ --> } f_2 \Leftrightarrow s \models_P \text{ AG } (f_1 \Rightarrow \text{ AF } f_2)$.

Синтаксис и семантика допустимых формул для автомата дословно повторяют синтаксис и семантику допустимых формул для сети с заменой l на ρ, (a.v) на (s), где $s \in S$, и $l(a) = v$ на $s \in \rho^*(root_H)$.

3.4.4.5 Модификация алгоритма трансляции

Для того чтобы упростить доказательство корректности алгоритма, внесем в описание его работы небольшое изменение. Рассмотрим процесс в сети, порождаемый метасостоянием S типа XOR при трансляции. Рассмотрим в нем вершину с именем S'_active. Рассмотрим цепочку переходов, начинающуюся в этой вершине и порождаемую дугой с меткой (g, s, r, u), соединяющей состояние S' с произвольным состоянием S'', вложенным в S.

Если $r = \emptyset$, добавим вершине, в которую ведет первая дуга цепочки переходов, инвариант, равный инварианту I состояния S''. Если же $r \neq \emptyset$, добавим к предохранителю, которым помечена первая дуга цепочки переходов, конъюнкт I[r], то есть инвариант I, к переменным которого применены присваивания из r.

3.4.4.6 Дополнительные ограничения

Алгоритм трансляции работает корректно только при дополнительно наложенных ограничениях на структуру транслируемого автомата и на множество рассматриваемых формул:

- все переходы, начинающиеся в вершине типа ENTRY, помечены четверками (true, none, $\emptyset$, false),

- если переход помечен четверкой (g, c?, r, u), то r = $\emptyset$,
- допустимые формулы для автомата не содержат подформул вида (s), s ∈ S и
- допустимые формулы для сети не содержат подформул вида (a.v).

Последние два ограничения приводят к тому, что грамматики, описывающие допустимые формулы над автоматом и сетью, становятся одинаковыми, если у них совпадают множества переменных Vars и совпадают множества таймеров Clocks. Множество таких формул обозначим записью Ψ(Vars, Clocks).

3.4.4.7 Модификация иерархического автомата

Рассмотрим иерархический автомат H = (S, S_0, η, Type, Vars, Clocks, Chan, Inv, T, Chantype). Автомат Mod(H), называемый в дальнейшем модификацией автомата H, образуется из H следующим образом.

Добавим в автомат H служебную булеву переменную b. Каждый фрагмент автомата H вида $S' \xrightarrow{(g,s,r,u)} S''$, где r ≠ $\emptyset$, заменим на фрагмент $S' \xrightarrow{(g',s,r',u)} S \xrightarrow{(true,none,r'',true)} S''$, где

- BASIC(S),
- g' = g ∧ I[r] ∧ (b = false), где I – инвариант вершины S'' и запись I[r] означает замену переменных в I в соответствии с присваиваниями r,
- r' = {b := true},
- r'' = r ∪ {b := false} и

Состояния, добавляемые при описанной замене фрагментов, будем называть добавочными. Конфигурацию (ρ, μ, ν) автомата Mod(H), содержащую активные добавочные состояния (т.е. для некоторого добавочного состояния s верно s ∈ $\rho^*(root_H)$), будем называть добавочной. Кроме того, к предохранителям всех остальных дуг, не являющихся псевдопереходами, добавим конъюнкт (b = false).

Для конфигураций, не являющихся добавочными, верно соотношение (b = false). Значит, в этом случае переменная b не влияет на выполнимость дуг по сравнению с исходным автоматом H. Если же мы переходим в добавочную конфигурацию, то все дуги, кроме исходящих из добавочных вершин, блокируются.

Выполнение перехода в автомате H, несущего непустое присваивание, заменяется на последовательное выполнение двух переходов в автомате Mod(H). Условие возможности совершения такого перехода в H (т.е. истинность предохранителя и истинность инварианта после применения присваиваний) совпадает с возможностью совершить первый переход в заменяющем фрагменте в Mod(H). Если рассматриваемый переход несет какую-либо

синхронизацию и выполняется одновременно с другими переходами в Mod(H), то эти переходы, по ограничениям модели, не несут никаких присваиваний. Значит, они не ведут в добавочные вершины. При этом рассматриваемый переход попадает в добавочную вершину и блокирует выполнение всех остальных переходов Mod(H) до выполнения второго перехода заменяющего фрагмента. Второй переход является срочным, следовательно, при возможности его совершения он немедленно совершается. Этот переход несет предохранитель true, а значит, единственное, что его может блокировать, – это ложность инварианта целевой вершины перехода после применения присваиваний. Но совершение первого перехода заменяющего фрагмента гарантирует истинность этого инварианта. Следовательно, переход, исходящий из добавочной вершины, немедленно выполняется.

Значения переменных множества Vars и таймеров множества Clocks совпадают до и после выполнения первого перехода заменяющего фрагмента в Mod(H). После выполнения второго перехода заменяющего фрагмента в Mod(H) их значения совпадают с таковыми после выполнения заменяемого перехода в H.

Проведенные рассуждения позволяют рассматривать при доказательстве корректности алгоритма автомат Mod(H) вместо автомата H.

3.4.4.8 Соответствие конфигураций

Пусть P = (A, $Vars_P$, $Clocks_P$, $Chan_P$, $Chantype_P$, $Chanurg_P$) – сеть, получаемая из автомата H = (S, S_0, η, Type, Vars, Clocks, Chan, Inv, T, Chantype) в результате трансляции.

Для простоты формулировок в дальнейшем полагаем, что автомат H не содержит ни одного срочного перехода, то есть если переход несет метку (g, s, r, u), то u = false. Четвертую компоненту меток переходов будем опускать за ненадобностью.

Тогда $Chan_P = Chan \cup Chan_{add}$, где $Chan_{add}$ – служебные каналы, добавляемые в процессе трансляции. При этом $Chantype_P(c) = Chantype(c)$ для $c \in Chan$, $Chantype_P(c)$ = handshake иначе, и для всех каналов c из $Chan_P$ верно $Chanurg_P(c)$ = common. При наличии срочных переходов проводятся те же рассуждения, но с учетом того, что каждый канал транслируется в четыре, добавляется срочный канал Hurry и срочность переходов моделируется синхронизацией по этим каналам.

Алгоритм трансляции ставит в соответствие метасостоянию в H уникальный процесс в P. Сеть P состоит из уникальных процессов, соответствующих всем метасостояниям H, и двух служебных процессов – Global_kickoff и Hurry. Таким образом, можно считать, что алгоритм задает биективное отображение

$$tr : S_{meta} \rightarrow A_{rel},$$

где $S_{meta} = \{ s \in S \mid AND(S) \vee XOR(S) \}$, $A_{rel} = A \setminus \{ Global_kickoff, Hurry \}$.

В каждом процессе p из множества A_{rel} выделяются вершины, соответствующие активности каждого из подсостояний S', вложенных в прообраз S процесса p. В образе метасостояния типа AND это вершина с именем «active», в образе метасостояния типа XOR – вершина с именем «S'_active». Таким образом, можно считать, что алгоритм задает функцию соответствия подсостояний

$$ac : S_{nr} \rightarrow \cup\{p \in A\}\ L_p,$$

где $S_{nr} = \{ s \in S \mid AND(s) \vee XOR(s) \vee BASIC(s) \} \setminus \{ root_H \}$, причем для всех s из S_{nr} верно соотношение $ac(s) \in L_{tr(\eta\text{-}1\ (s))}$.

Переменные и таймеры автомата H в результате трансляции переписываются в переменные и таймеры сети P. Кроме того, в сеть добавляются служебные переменные $Vars_{add}$. Иначе говоря, верны равенства $Clocks_P = Clocks$, $Vars_P = Vars \cup Vars_{add}$.

Значение служебных переменных сети P однозначно определяется множеством активных вершин. Таким образом, можно считать, что алгоритм задает функцию значений служебных переменных

$$\mu_{add} : L \times Vars_{add} \rightarrow \{ true, false \},$$

где L – семейство всех возможных множеств активных вершин сети.

Поставим в соответствие конфигурации $c = (\rho, \mu, \nu)$ автомата H конфигурацию $\tau(c) = (l, \mu_P, \nu_P)$ сети P, где

- $l(tr(s)) = ac(\rho(s))$ для $s \in \rho^*(root_H)$,
- $l(tr(s)) = idle$ для $s \notin \rho^*(root_H)$,
- $\mu_P(x) = \mu(x)$ для $x \in Vars$,
- $\mu_P(x) = \mu_{add}(l, x)$ для $x \in Vars_{add}$ и
- $\nu_P = \nu$.

Также поставим в соответствие конфигурации $c = (\rho, \mu, \nu)$ автомата H конфигурацию $mod(c) = (\rho, \mu', \nu)$ его модификации Mod(H), где функция μ' выдает значение false на служебной переменной b и совпадает с функцией μ на остальных переменных.

3.4.4.9 Корректность алгоритма

Пусть $H = (S, S_0, \eta, Type, Vars, Clocks, Chan, Inv, T, Chantype)$ – автомат, Mod(H) – его модификация, $P = (A, Vars_P, Clocks_P, Chan_P, Chantype_P, Chanurg_P)$ – сеть, получающаяся из H в результате трансляции.

Обоснование корректности алгоритма состоит в доказательстве справедливости формулы:

$$\psi \in \Psi(Vars, Clocks)\ .\ (\ (c_0^H \models_{Mod(H)} \psi) \Leftrightarrow (c_0^P \models_P \psi)\ ), \quad \mathbf{(1)}$$

где c_0^H – начальная конфигурация автомата Mod(H), c_0^P – начальная конфигурация сети P. Пусть $K_1 = (S_1, \mathcal{L}_1, \rightarrow_1)$, $K_2 = (S_2, \mathcal{L}_2, \rightarrow_2)$ – структуры Крипке, описывающие поведения Mod(H) и P соответственно. Уберем из области значений функций $\mathcal{L}_i$ все выражения, содержащие переменные, не содержащиеся в Vars, и таймеры, не содержащиеся в Clocks.

Добавим к области значений функций $\mathcal{L}_i$ выражение deadlock следующим образом:

- deadlock $\in \mathcal{L}_1(c)$ тогда и только тогда, когда ни для какого c' не выполнено $c \rightarrow_1 c'$,
- deadlock $\in \mathcal{L}_2(c)$ тогда и только тогда, когда каждый маршрут в структуре Крипке K_2, начинающийся в c,
    - конечен,
    - имеет общее значение функции $\mathcal{L}_2$ на всех своих конфигурациях и
    - содержит только срочные конфигурации, если не считать последней.

Тогда K_1 и K_2 становятся структурами Крипке над общим множеством Expr(Vars, Clocks) $\cup$ { deadlock }. Рассмотрение измененных таким образом структур Крипке равносильно рассмотрению структур Крипке, описывающих поведения автомата Mod(H) и сети P, при проверке истинности формул из множества Ψ(Vars, Clocks).

Для доказательства формулы (1) достаточно показать, что конфигурации s_0^H и s_0^P dbs-эквивалентны. Из этого будет следовать истинность формулы (1) для всех формул логики CTL*-X над множеством Expr(Vars, Clocks) $\cup$ { deadlock } и, как следствие, для всех формул из множества Ψ(Vars, Clocks).

Для доказательства dbs-эквивалентности конфигураций s_0^H, s_0^P опишем отношение $\sim \subseteq S_1 \times S_2$, являющееся dbs. Для этого выпишем все пары, входящие в это отношение.

Рассмотрим произвольную конфигурацию τ(c). Для описания всех пар, входящих в отношение $\sim$, введем два множества:

$next(c) = \{\ s \in S_2 \mid \exists\, s_1, s_2, ..., s_k : (\ \tau(c) \rightarrow_2 s_1 \rightarrow_2 s_2 \rightarrow_2 ... \rightarrow_2 s_k \wedge \mathcal{L}_2(\tau(c)) = \mathcal{L}_2(s_1) = ... = \mathcal{L}_2(s_k) \wedge s = s_k \wedge$ конфигурации $s_1, ..., s_k$ являются срочными},

$prev(c) = \{\ s \in S_2 \mid \exists\, s_1, s_2, ..., s_k : (\ s_1 \rightarrow_2 s_2 \rightarrow_2 ... \rightarrow_2 s_k \wedge \mathcal{L}_2(s_1) = ... = \mathcal{L}_2(s_k) \wedge s = s_1 \wedge \tau(c) = s_k \wedge$ конфигурации $s_1, ..., s_{k-1}$ являются срочными}.

Отношение $\sim$ описывается следующим образом:

$\sim = \{\ \langle mod(c), s_2\rangle \mid$ c – конфигурация автомата H, $s_2 \in prev(c)\ \} \cup \{\ \langle s_1, s_2\rangle \mid mod(c) \rightarrow_1 s_1$, s_1 – добавочная конфигурация, $s_2 \in next(\tau(c))$, c – конфигурация автомата H }.

Проведем обоснование того, что ~ является dbs, не углубляясь в технические детали, следующим образом.

Истинность пункта 1 определения dbs для отношения ~ следует непосредственно из описания отношения ~ и множеств prev(c), next(c).

Рассмотрим фрагмент структуры K_1 вида mod(c) $\rightarrow_1$ mod(c'). Выполнимость последнего соотношения означает, что c' получается из c за счет

- продвижения времени

или

- выполнения перехода (или группы переходов согласно синхронизации), не несущего присваиваний.

В первом случае имеем $\tau(c) \rightarrow_2 \tau(c')$. Во втором случае рассматриваемому фрагменту соответствует конечное число маршрутов в структуре K_2, ведущих из конфигурации $\tau(c)$ в конфигурацию $\tau(c')$, и каждый такой маршрут проходит только через конфигурации множества prev(c'), если не считать первой вершины маршрута.

Рассмотрим теперь фрагмент структуры K_1 вида mod(c) $\rightarrow_1 c_{add} \rightarrow_1$ mod(c'). Этому фрагменту соответствует конечное число маршрутов в структуре K_2, ведущих из конфигурации $\tau(c)$ в конфигурацию $\tau(c')$, и каждый такой маршрут проходит сначала через конфигурации множества next(c), затем через конфигурации множества prev(c'), если не считать первой вершины маршрута.

Таким образом, пункт 2 определения dbs для отношения ~ выполнен.

В структуре K_2 из вершины $\tau(c)$ исходят только маршруты, описанные при обосновании пункта 2 определения dbs для отношения ~ . Следовательно, пункт 3 определения dbs для отношения ~ также выполнен.

Остается заметить, что выполнено соотношение $c_0^P \in prev(\tau(c_0^H))$, а значит, конфигурации c_0^H и c_0^P dbs-эквивалентны.

Проведенные рассуждения доказывают утверждение, обосновывающее корректность алгоритма трансляции:

Теорема. Пусть H – произвольный автомат, Mod(H) – его модификация, P – сеть, получающаяся в результате трансляции из автомата H. Тогда для любой формулы из множества Ψ(Vars, Clocks) верно соотношение

$$(s_0^H \models_{Mod(H)} \psi) \Leftrightarrow (s_0^P \models_P \psi).$$

Эта теорема означает, что для для проверки ее выполнимости любой спецификации, выраженной формулой CTL из класса Ψ(Vars, Clocks), в модели иерархического временного автомата H необходимо и достаточно проверить выполнимость той же спецификации в модели сети плоских временных автоматов P. Решение последней задачи можно осуществить автоматически при помощи системы верификации UPPAAL. Одно из достоинств этой системы состоит в том, что в случае невыполнимости формул, начинающихся операторами AG или AF, она строит контрпример – сценарий вычисления сети автоматов, в котором нарушается проверяемая спецификация. Этот контрпример может быть очень полезен для проектировщиков проверяемой системы, поскольку с его помощью гораздо легче установить источник обнаруженной ошибки. Однако, поскольку проверке подвергается не сама проектируемая система (UML-диаграмма), а ее плоская трансляция, этот контрпример описывает поведение сети плоских автоматов. Ценность контрпримера значительно возрастет, если удастся построить соответствующий ему сценарий поведения иерархического автомата (UML-диаграммы). Исследование этой задачи планируется провести на следующем этапе проекта.

3.5 Оптимизация алгоритма трансляции UML во временные автоматы

Оптимизация алгоритма трансляции UML в сеть временных автоматов происходит непосредственно после преобразования UML в иерархический временной автомат. Оптимизация проходит в два этапа.

Первый этап оптимизации может быть осуществлен независимо от второго и сам по себе рассматриваться как оптимизация. Однако второй этап оптимизации не может быть осуществлен к произвольному автомату без осуществления перед ним первого этапа.

В дальнейшем считаем, что HTA = (S, S_0, η, Type, Vars, Clocks, Chan, Inv, T, Chantype).

3.5.1 Первый этап оптимизации

На первом этапе структура входного HTA приводится к виду, в котором для любых двух метасостояний S_1, S_2 одного типа (типа AND или типа XOR) верно соотношение $S_1 \notin \eta(S_2)$. Проще говоря, в котором ни одно метасостояние не вложено в метасостояние того же типа.

Преобразование входного НТА производится следующим образом. Пока в текущем НТА существует хоть одна пара S_1, S_2 состояний одинакового типа таких, что S_1 вложено в S_2, производятся следующие действия:

- удаляется метасостояние S_1 и вложенные в него входы и выходы,
- все компоненты типов AND, XOR, BASIC, вложенные в S_1 до его удаления, вкладываются непосредственно в S_2,
- переходы между этими компонентами сохраняются,
- просматривается каждый выход EX состояния S_1; для этого выхода просматривается каждая пара переходов $S_1 \rightarrow EX \rightarrow S_2$; для каждой такой пары создается переход $S_1 \rightarrow S_2$, несущий все метки заменяемых переходов,
- то же самое делается для каждого входа состояния S_1.

Преобразование первого этапа уменьшает как время работы средства верификации, так и число состояний, при этом не усложняя существенно их структуру.

Псевдокод, описывающий преобразование метасостояний на первом этапе, приведён в подразделе «Псевдокод первого этапа оптимизации».

3.5.2 Второй этап оптимизации

После первого этапа становится верным следующее утверждение о стуктуре текущего НТА: все метасостояния, вложенные в любое наперед взятое метасостояние, имеют одинаковый тип, отличный от типа рассматриваемого метасостояния.

Рассмотрим метасостояние типа AND. В него могут быть вложены только входы, выходы и метасостояния типа XOR.

Преобразование второго этапа состоит из многократно повторяющихся однотипных шагов. На каждом шаге среди рассматриваемых метасостояний типа XOR находится пара, работающая «как можно более непараллельно». В лучшем случае можно разбить компоненты, вложенные в эту пару метасостояний, на участки, работающие последовательно один за другим. В менее «хороших» случаях строится параллельная композиция некоторых пар таких участков, и различные параллельные композиции последовательно комбинируются. После этого совершаются следующие действия:

- множество компонент, вложенных в рассматриваемые метасостояния, особым образом разбиваются на подмножества,
- составляется список пар подмножеств разбиения, одновременно достижимых в процессе работы НТА,

- создаётся новое метасостояние S типа XOR,
- в метасостояние S добавляются параллельные композиции одновременно достижимых подмножеств,
- разные пары подмножеств также соединяются дугами согласно дугам в исходной паре метасостояний и
- рассматривается пара метасостояний заменяется на метасостояние S.

Преобразование второго этапа в сочетании с первым этапом может уменьшить число состояний до любого наперед заданного числа. Но при этом усложняется структура состояний, и время работы средства верификации может как увеличиться, так и уменьшиться (в зависимости от вида участков разбиения метасостояний).

В зависимости от важности трех критериев оптимизации – времени работы, числа состояний и сложности стурктуры состояний – следует выбирать пары объединяемых метасостояний и момент прекращения второго этапа.

Псевдокод, описывающий преобразование метасостояний на втором этапе, приведён в подразделе «Псевдокод второго этапа оптимизации».

3.5.3 Псевдокод первого этапа оптимизации

```
procedure move(B, B_old, B_new)
begin
  η(B_old) := η(B_old) \ {B};
  η(B_new) := η(B_new) ∪ {B};
end;

procedure add_transition_to_hta(B_1, g, s, r, u, B_2)
begin
  T := T ∪ {(B_1, (g,s,r,u), B_2)};
end;

procedure remove(B_rem)
begin
  forall B ∈ η^*(B_rem)
  do
    forall t = (B, l, B') ∈ T
    do
      T := T \ {t};
    od
    forall t = (B', l, B) ∈ T
    do
      T := T \ { t };
    od
```

```
    η.domain := η.domain \ {B};
    η.value_area := η.value_area \ {B};
    S0 := S0 \ {B};
    S := S \ {B};
  od
end;

begin
  while ∃ S_1, S_2 ∈ S such that
      ((AND(S_1) and AND(S_2)) or (XOR(S_1) and XOR(S_2))) and S_1 ∈ η(S_2)
  do
    forall S ∈ η(S_1) such that AND(S) or XOR(S) or BASIC(S)
    do
      move(S, S_1, S_2);
    od
    forall EX ∈ η(S_1) such that EXIT(EX)
    do
      forall (t_1, t_2) such that
          t_1 = (S', (g_1, none, ∅, false), EX) and
          t_2 = (EX, (g_2, s, r, u), S'')
      do
        add_transition_to_hta(S', g_1 and g_2, s, r, u, S'');
      od
    od
    forall E ∈ η(S_1) such that ENTER(E)
    do
      forall (t_1, t_2) such that
          t_1 = (S', (g, s, r_1, u), E) and
          t_2 = (E, (true, none, r_2, false), S'')
      do
        add_transition_to_hta(S', g, s, r_1 ∪ r_2, u, S'');
      od
    od

    remove(S_1);
  od
end.
```

3.5.4 Псевдокод второго этапа оптимизации

Для удобства записи обрабатываемые метасостояния типа XOR рассматриваются отдельно от HTA. Кроме того, считаем, что метасостояние S типа XOR имеет вид

S = <states, type, inv, transitions, in_transitions, out_transitions>,

где:

1. states — множество вложенных в состояние S компонент,
2. type : states → {BASIC, AND, ENTRY, EXIT} — разметка типов вершин,
3. для всех компонент s из множества states, имеющих тип AND, имеется множество вложенных входных (s.entries) и выходных (s.exits) вершин.

4. inv : states → invariant — разметка компонент инвариантами,
5. transitions — множество переходов HTA, соединяющих компоненты множества states (напрямую или через вложенные в них входные-выходные вершины),
6. in_transitions — множество переходов HTA, начинающихся вне рассматриваемого состояния и
7. out_transitions — множество переходов HTA, оканчивающихся вне рассматриваемого состояния.

Также будем считать, если явно не обозначено обратное, что во всех переходах вместо компоненты s_e, где s_e ∈ s.entries ∪ s.exits, s ∈ states, используется компонента s.

Псевдокод шага преобразования второго этапа имеет следующий вид.

```
function move_dis(c_1, c_2, ss_1, ss_2, var reachable_states) : boolean
begin
  let in_transitions_1 = {(s', l, s'') ∈ ss_1.transitions |
                          s' ∈ c_1 and s'' ∈ c_1};
  let in_inv_1 = ∧{s ∈ c_1.states} inv(s);
  let init_states_2 = {s'' ∈ c_2 | (s', l, s'') ∈ ss_2.transitions and
                                   s' ∉ c_2};
  let blocked_vertices = {s ∈ c_2 |
                          s is not reachable in c_2 from init_states_2};

  // we work with copy of ss_2,
  // states and transitions are not deleted in real
  ss_2.states := ss_2.states \ blocked_vertices;
  ss_2.transitions := ss_2.transitions \
                      {(s', l, s'') | s' ∈ blocked_vertices or
                                      s'' ∈ blocked_vertices};
  c_2 := c_2 \ blocked_vertices;

  let out_transitions_2 = {(s', l, s'') ∈ ss_2.transitions |
                           s' ∈ c_2 and s'' ∉ c_2 and
                           (AND(s'') or BASIC(s''))};
  let out_states_2 = {<r, s''> |
                      (s', (g,s,r,u), s'') ∈ out_transitions_2};
```

```
  let blocked_out_states_2 = {s | <r, s> ∈ out_states_2 and
                              in_inv_1[r] ∧ inv(s) ≡ false};

  ss_2.states := ss_2.states \ blocked_out_states_2;
  out_transitions_2 := out_transitions_2 \
                     {(s', l, s'') | s'' ∈ blocked_out_states_2};

  let out_g_2_set = {<g, s''> | (s', (g,s,r,u), s'') ∈ out_transitions_2};
  reachable_states := {s'' | <g, s''> ∈ out_g_2_set and
                             not(g ∧ in_inv_1 ≡ false)};

  if reachable_states = ∅ then
    return true;
  else
    return false;
  fi
end;

function expand(c_1, c_2, ss_1, ss_2) : state_set_pair
begin
  c_1_new := c_1;
  c_2_new := c_2;

  \\expanding c_2, goal - it is blocked
  while not move_dis(c_1, c_2_new, ss_1, ss_2, reachable_states)
  do
    c_2_new := c_2_new ∪ reachable_states;
  od

  \\expanding c_1, goal - it is blocked
  while not move_dis(c_2, c_1_new, ss_2, ss_1, reachable_states)
  do
    c_1_new := c_1_new ∪ reachable_states;
  od

  if min(|c_1|, |c_2_new|) < min(|c_1_new|, |c_2|) then
    return <c_1, c_2_new>;
```

```
  else
    return <c_1_new, c_2>;
  fi
end;

procedure remove(c, var com, var processed)
begin
  com := com \ c;
  processed := processed \ ({<c, c'> | ∀ c'} ∪ {<c', c> | ∀ c'});
end;

function next(c, ss) : state_set
begin
  return {s | s ∉ c and (BASIC(s) or AND(s)) and
              ∃ s' ∈ c such that
                 (s', l, s) ∈ ss.transitions};
end;

procedure divide(ss_1, ss_2, s^1, s^2, var com_1, var com_2, var processed)
begin
  let l, s_N^1, s_i^1, c_i^1, 1 ≤ i ≤ l such that
      s^1 = ∪{i=1..l} s_i^1 ∪ s_N^1 and
      s_i^1 ⊆ c_i^1 ∈ com_1 and
      (i ≠ j) ⇒ (c_i^1 ≠ c_j^1) and
      s_N^1 ∩ ∪{c ∈ com_1} c = ∅;

  let k, s_N^2, s_i^2, c_i^2, 1 ≤ i ≤ k such that
      s^2 = ∪{i=1..k} s_i^2 ∪ s_N^2 and
      s_i^2 ⊆ c_i^2 ∈ com_2 and
      (i ≠ j) ⇒ (c_i^2 ≠ c_j^2) and
      s_N^2 ∩ ∪{c ∈ com_2} c = ∅;

  assume (k = 1 and s_N^2 = ∅) or (k = 0); //w.l.o.g.

  if k = 1 then
    c^2 := c_1^2;
```

```
else
  c^2 := s_N^2;
  com_2 := com_2 ∪ {c^2};
fi

let to_process = {<c_i^1, c^2> | 1 ≤ i ≤ l};

to_process := to_process \ processed;

let s_N^1 = {st_j | 1 ≤ j ≤ p};

forall j such that 1 ≤ j ≤ p
do
  com_1 := com_1 ∪ {st_j};
  to_process := to_process ∪ {<{st_j}, c^2>};
od

forall c^1 such that <c^1, c^2> ∈ to_process
do
  <c^1_new, c^2_new> := expand(c^1, c^2, ss_1, ss_2);

  if c^1_new ≠ c^1 then
    forall c ∈ com_1 such that c ⊊ c^1_new
    do
      remove(c, com_1, processed);
    od
    com_1 := com_1 ∪ c^1_new;
  fi
  if c^2_new ≠ c^2 then
    forall c ∈ com_2 such that c ⊊ c^2_new
    do
      remove(c, com_2, processed);
    od
    com_2 := com_2 ∪ c^2_new;
  fi

  processed := processed ∪ {<c^1_new, c^2_new>};
```

```
    md_1 := move_dis(c^1_new, c^2_new, ss_1, ss_2, unused_param);
    md_2 := move_dis(c^2_new, c^1_new, ss_2, ss_1, unused_param);

    if md_1 and not md_2 then
      divide(ss_1, ss_2, next(c^1_new, ss_1), c^2_new, com_1, com_2, processed);
    fi
    if md_2 and not md_1 then
      divide(ss_1, ss_2, c^1_new, next(c^2_new, ss_2), com_1, com_2, processed);
    fi
    // if (md_1 and md_2) then do not perform any recursion
    // (not md_1 and not md_2) cannot be satisfied
  od
end;

procedure add_state(var A, TYPE, state);
begin
  A.states := A.states ∪ {state};
  assume type(name) = TYPE;
end;

procedure add_transition(var A, tr)
begin
  A.transitions := A.transitions ∪ {tr};
end;

procedure add_in_transition(var A, tr)
begin
  A.in_transitions := A.in_transitions ∪ {tr};
end;

procedure add_out_transition(var A, tr)
begin
  A.out_transitions := A.out_transitions ∪ {tr};
end;

procedure add_parallel_composition(var A, curr_1, curr_2, ss^1, ss^2, c^1, c^2)
begin
  if ∃ s ∈ c^1 such that AND(s) then
```

```
  assume c^2 = {st};
  forall s ∈ c^1
  do
    add_state(A, type(s), <s, st>);
    assume A.inv(<s, st>) = (ss^1.inv(s) and ss^2.inv(st));
  od
  forall (s', l, s'') ∈ ss^1.transitions such that
      s', s'' ∈ c^1
  do
    add_transition(A, (<s', st>, l, <s'', st>));
  od
else
  if ∃ s ∈ c^2 such that AND(s) then
    assume c^1 = {st};
    forall s ∈ c^2
    do
      add_state(A, type(s), <st, s>);
      assume A.inv(<st, s>) = (ss^1.inv(st) and ss^2.inv(s));
    od
    forall (s', l, s'') ∈ ss^2.transitions such that
        s', s'' ∈ c^2
    do
      add_transition(A, (<st, s'>, l, <st, s''>));
    od
  else
    forall s_1 ∈ c^1, s_2 ∈ c^2
    do
      add_state(A, BASIC, <s_1, s_2>);
      assume A.inv(<s_1, s_2>) = (ss^1.inv(s_1) and ss^2.inv(s_2));
    od
    forall s_2 ∈ c^2, (s', l, s'') ∈ ss^1.transitions such that
        s', s'' ∈ c^1 and
        (l.sync = 'c?' and broadcast(c)) ⇒
          not ∃ (v', l', v'') ∈ ss^2.transitions such that
            l'.sync = 'c?';
    do
      add_transition(A, (<s', s_2>, l, <s'', s_2>));
```

```
od
forall s_1 ∈ c^1, (s', l, s'') ∈ ss^2.transitions such that
    s', s'' ∈ c^2 and
    (l.sync = 'c?' and broadcast(c)) ⇒
      not ∃ (v', l', v'') ∈ ss^1.transitions such that
        l'.sync = 'c?';
do
  add_transition(A, (<s_1, s'>, l, <s_1, s''>));
od
forall (s'_1, (g_1, c!, r_1, u_1), s''_1) ∈ ss^1.transitions,
       (s'_2, (g_2, c?, r_2, u_2), s''_2) ∈ ss^2.transitions
such that
    s'_1, s''_1 ∈ c^1 and s'_2, s''_2 ∈ c^2 and handshake(c)
do
  add_transition(A,
                 (<s'_1, s'_2>,
                  (g_1 and g_2, none, r_1 ∪ r_2, u_1 or u_2),
                  <s''_1, s''_2>));
od
forall (s'_1, (g_1, c!, r_1, u_1), s''_1) ∈ ss^1.transitions,
       (s'_2, (g_2, c?, r_2, u_2), s''_2) ∈ ss^2.transitions
such that
    s'_1, s''_1 ∈ c^1 and s'_2, s''_2 ∈ c^2 and broadcast(c)
do
  add_transition(A,
                 (<s'_1, s'_2>,
                  (g_1 and g_2,
                   c!,
                   r_1 ∪ r_2,
                   u_1 or u_2),
                  <s''_1, s''_2>));
  add_transition(A,
                 (<s'_1, s'_2>,
                  (g_1 and not g_2, c!, r_1, u_1),
                  <s''_1, s'_2>));
od
forall (s'_1, (g_1, c?, r_1, u_1), s''_1) ∈ ss^1.transitions,
```

```
          (s'_2, (g_2, c?, r_2, u_2), s''_2) ∈ ss^2.transitions
      such that
          s'_1, s''_1 ∈ c^1 and s'_2, s''_2 ∈ c^2 and broadcast(c)
      do
        add_transition(A,
                       (<s'_1, s'_2>,
                        (g_1 and g_2,
                         c?,
                         r_1 ∪ r_2,
                         u_1 or u_2),
                        <s''_1, s''_2>));
        add_transition(A,
                       (<s'_1, s'_2>,
                        (g_1 and not g_2, c?, r_1, u_1),
                        <s''_1, s'_2>));
        add_transition(A,
                       (<s'_1, s'_2>,
                        (not g_1 and g_2, c?, r_2, u_2),
                        <s'_1, s''_2>));
      od
    fi
  fi
end;
procedure connect_parallel_compositions(var A, ss_1, ss_2, to_process);
begin
  forall <c_1, c_2> ∈ to_process
  do
    md_1 = move_dis(c_1, c_2, ss_1, ss_2, unused_param);
    md_2 = move_dis(c_2, c_1, ss_2, ss_1, unused_param);
    if md_1 and not md_2 then
      forall s ∈ c_2, (s', l, s'') ∈ ss_1 such that
          s' ∈ c_1 and s'' ∉ c_1 and (BASIC(s'') or AND(s''))
      do
        add_transition(A, (<s', s>, l, <s'', s>));
      od
    fi
    if md_2 and not md_1 then
      forall s ∈ c_1, (s', l, s'') ∈ ss_2 such that
```

```
        s' ∈ c_2 and s'' ∉ c_2 and (BASIC(s'') or AND(s''))
      do
        add_transition(A, (<s, s'>, l, <s, s''>));
      od
    fi

    forall (s_1, (g_1, none, ∅, false), E_1) ∈ ss_1.transitions,
           (s_2, (g_2, none, ∅, false), E_2) ∈ ss_2.transitions
    such that
        s_1 ∈ c_1 and s_2 ∈ c_2 and EXIT(E_1) and EXIT(E_2) and
        ∃ E such that (
          (E_1, l_1, E) ∈ ss_1.out_transitions and
          (E_2, l_2, E) ∈ ss_2.out_transitions
        )
    do
      add_transition(A, (<s_1, s_2>,
                         (g_1 and g_2, none, ∅, false),
                         <E_1, E_2>));
    od
    forall (E_1, (true, none, r_1, false), s_1) ∈ ss_1.transitions,
           (E_2, (true, none, r_2, false), s_2) ∈ ss_2.transitions
    such that
        s_1 ∈ c_1 and s_2 ∈ c_2 and ENTRY(E_1) and ENTRY(E_2) and
        ∃ E such that (
          (E, l_3, E_1) ∈ ss_1.in_transitions and
          (E, l_4, E_2) ∈ ss_2.out_transitions
        )
    do
      add_transition(A,
                     (<s_1, s_2>,
                      (g_1 and g_2, none, ∅, false),
                      <E_1, E_2>));
    od
  od
end;
```

```
function init_states(ss) : state_set
begin
  return { s | (E, (g, s, r, u), s) ∈ ss and ENTRY(E) };
end;

procedure join_step(curr_1, curr_2, ss_1, ss_2, com_1, com_2, var A, to_process)
begin
  md_1 := move_dis(curr_1, curr_2, ss_1, ss_2, unused_param);
  md_2 := move_dis(curr_2, curr_1, ss_2, ss_1, unused_param);
  if md_1 and md_2 then
    return;
  fi
  if md_1 then
    next_components :=
      {<c, curr_2> |
       c ∈ com_1 and c ∩ next(curr_1, ss_1) ≠ ∅};
  fi
  if md_2 then
    next_components :=
      {<curr_1, c> |
       c ∈ com_2 and c ∩ next(curr_2, ss_2) ≠ ∅};
  fi

  forall <c^1, c^2> ∈ next_components such that
      <c^1, c^2> ∈ to_process
  do
    add_parallel_composition(A, curr_1, curr_2, ss_1, ss_2, c^1, c^2);
  od

  to_process := to_process \ {<curr_1, curr_2>};

  forall <c^1, c^2> ∈ next_components such that
      <c^1, c^2> ∈ to_process
  do
    join_step(c^1, c^2, ss_1, ss_2, A, to_process);
  od
end;
```

```
function join(ss_1, ss_2, com_1, com_2, to_process) : boolean
begin
  HTA_superstate A = empty;

  forall E_1 ∈ ss_1.states, E_2 ∈ ss_2.states, E such that
      ENTRY(E_1) and ENTRY(E_2) and
      (E, (true, none, r_1, false), E_1) ∈ in_transitions and
      (E, (true, none, r_2, false), E_2) ∈ in_transitions
  do
    add_state(A, ENTRY, <E_1, E_2>);
    add_in_transition(A, (E, (true, none, r_1 ∪ r_2, false), <E_1, E_2>));
  od
  forall E_1 ∈ ss_1.states, E_2 ∈ ss_2.states, E such that
      EXIT(E_1) and EXIT(E_2) and
      (E_1, (true, none, ∅, false), E) ∈ out_transitions and
      (E_2, (true, none, ∅, false), E) ∈ out_transitions
  do
    add_state(A, EXIT, <E_1, E_2>);
    add_out_transition(A, (<E_1, E_2>, (true, none, ∅, false), E));
  od

  let curr_1 ∈ com_1 such that init_states(ss_1) ⊆ curr_1;
  let curr_2 ∈ com_1 such that init_states(ss_2) ⊆ curr_2;

  add_parallel_composition(A, curr_1, curr_2, ss_1, ss_2);
  join_step(curr_1, curr_2, ss_1, ss_2, com_1, com_2,
            A, to_process \ {<curr_1, curr_2>});
  connect_parallel_compositions(A, ss_1, ss_2, to_process);
  return A;
end;

procedure step(superstate_1, superstate_2)
begin
  state_set_set components_1, components_2;
  state_set_pair_set processed;
  divide(superstate_1, superstate_2,
         init_states(superstate_1), init_states(superstate_2),
```

```
        components_1, components_2, processed);
  if ∃ <c^1, c^2> ∈ processed, s_1 ∈ c^1, s_2 ∈ c^2 such that
      AND(s_1) and AND(s_2)
  then
    return FAIL;
  fi
  if ∃ <c^1, c^2_1>, <c^1, c^2_2> ∈ processed, s ∈ c^1 such that
      AND(s) and c^2_1 ≠ c^2_2
  then
    return FAIL;
  fi
  if ∃ <c^1_1, c^2>, <c^1_2, c^2> ∈ processed, s ∈ c^2 such that
      AND(s) and c^1_1 ≠ c^1_2
  then
    return FAIL;
  fi
  if ∃ <c^1, c^2> ∈ processed, s ∈ c^1 such that
      AND(s) and |c^2| ≥ 2
  then
    return FAIL;
  fi
  if ∃ <c^1, c^2> ∈ processed, s ∈ c^2 such that
      AND(s) and |c^1| ≥ 2
  then
    return FAIL;
  fi
  return join(superstate_1,superstate_2,components_1,components_2,processed);
end;
```

3.6 Оптимизация средств трансляции UML во временные автоматы

Для решения новых задач средства трансляции UML во временные автоматы необходимо было подвергнуть доработке, расширив их функциональность.

3.6.1 Расширение возможностей вставки ссылок на другие автоматы

Добавлена поддержка stub states: можно при вставке ссылки на другой автомат ссылаться на конкретное входное состояние вставляемого автомата.

Добавлена поддержка вставки нескольких экземпляров одного и того же автомата, при этом осуществляется полное копирование с правильным переименованием состояний.

Пример для этих двух усовершенствований приведен на рисунках 65-66.

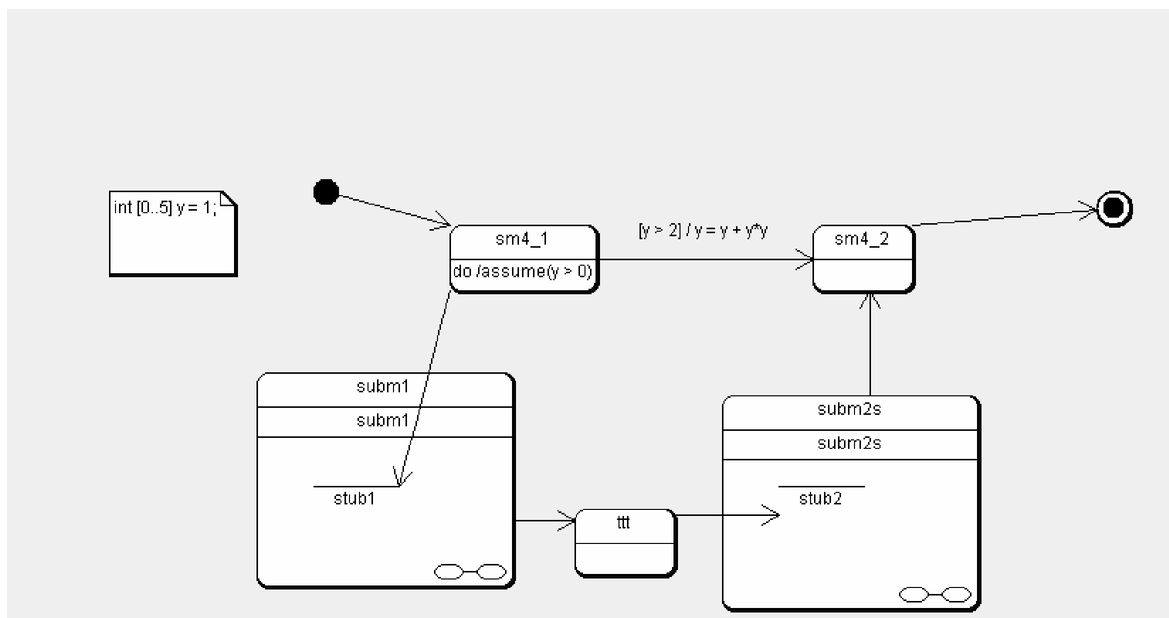

Рисунок 65. UML-диаграмма, иллюстрирующая stub states.

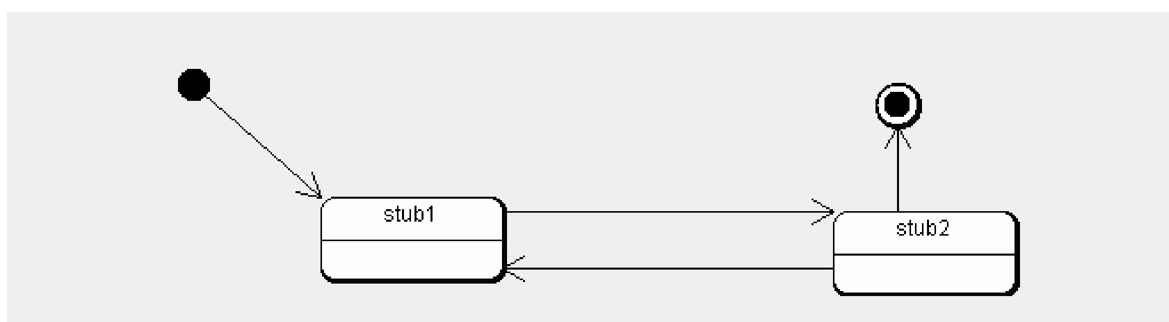

Рисунок 66. UML-диаграмма, иллюстрирующая stub states (2).

На первом рисунке два состояния-ссылки на один и тот же автомат sm5, при этом при помощи stub states (на диаграмме изображаются линиями) в sumb1 вход осуществляется через состояние stub1, а в subm2 – через состояние stub2.

При вставке происходит переименование во избежание коллизии имен, и в результате получается следующая диаграмма, проиллюстрированная в формате SCXML (рисунок 67):

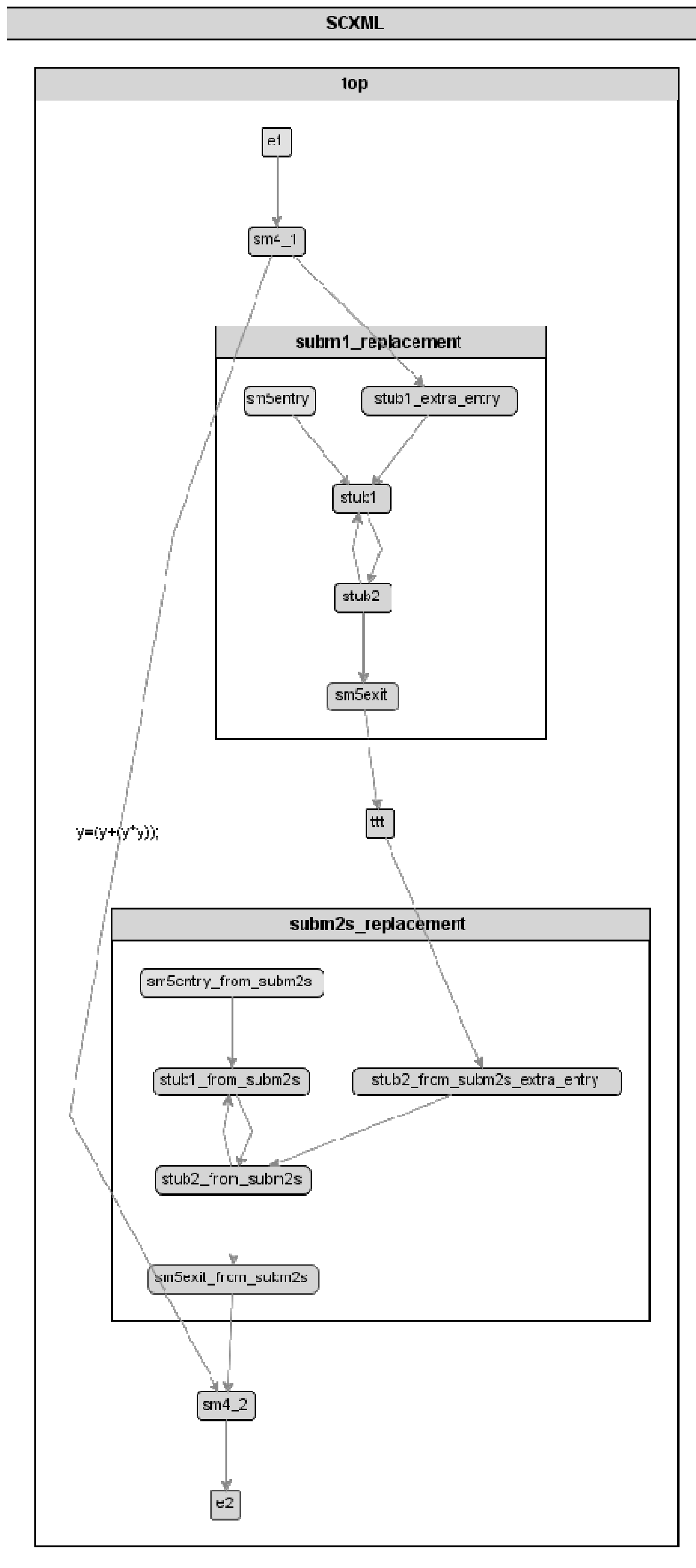

Рисунок 67. SCXML диаграмма, соответствующая диаграммам с рисунков 65-66

3.6.2 Расширенный синтаксис предусловий и действий

Также был расширен допустимый синтаксис предусловий и действий.

В предусловиях теперь допускается оператор логического ИЛИ || и отрицание выражений (не только переменных).

В действиях допускается использование операций сравнения и тернарного оператора x ? y : z.

Новый синтаксис предусловий:

```
Atom  ::= Disj | Disj '||' Disj
Disj  ::= Expr < Expr | Expr <= Expr | Expr > Expr | Expr >= Expr
        | Expr == Expr | Expr != Expr
        | in(S)
        | BoolExpr == BoolExpr | BoolExpr != BoolExpr
        | BoolExpr ? Expr : Expr
Expr  ::= ClockVar | ClockVar – ClockVar
        | IntExpr | IntExpr + IntExpr | IntExpr – IntExpr |
        IntConst | (Expr)
ClockVar ::= <идентификатор>
IntExpr ::= IntVar | IntConst
IntVar   ::= <идентификатор>
IntConst ::= <целочисленная константа>
BoolExpr ::= BoolVar | false | true | ! BoolExpr
BoolVar  ::= <идентификатор>
```

3.7 Экспериментальное исследование модели PBC PB Dr Tesy

После трансляции с оптимизацией XOR-состояний получилась система из 23 автоматов. На рисунках 68-74 приведены некоторые из них. Остальные опущены из-за слишком большого размера.

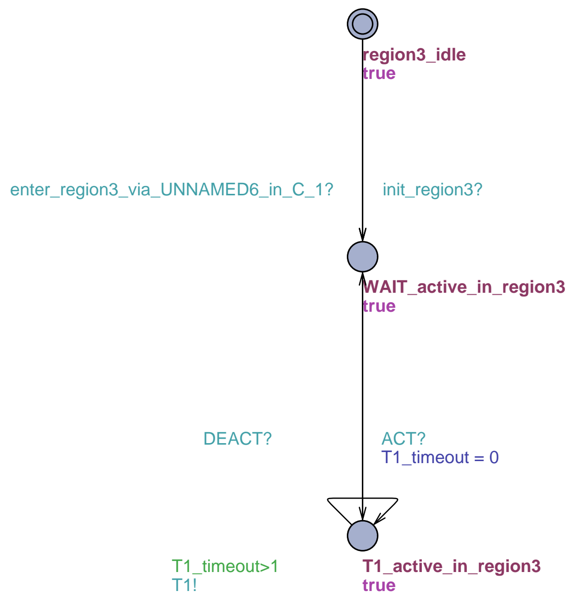

Рисунок 68. Диаграмма UPPAAL для C_1.region3.

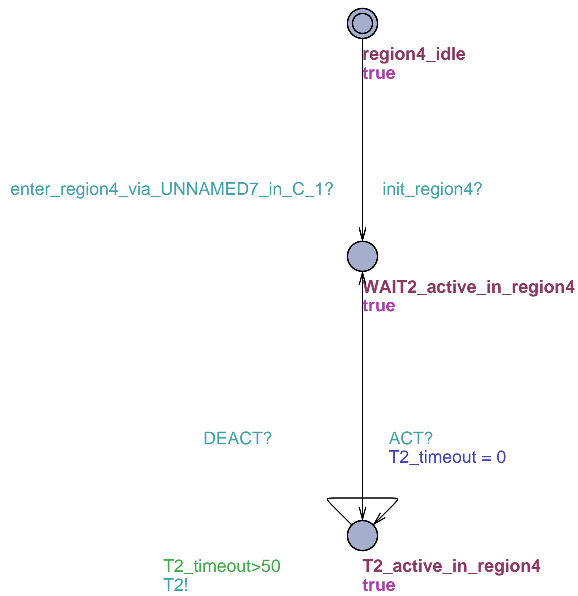

Рисунок 69. Диаграмма UPPAAL для C_1.region4.

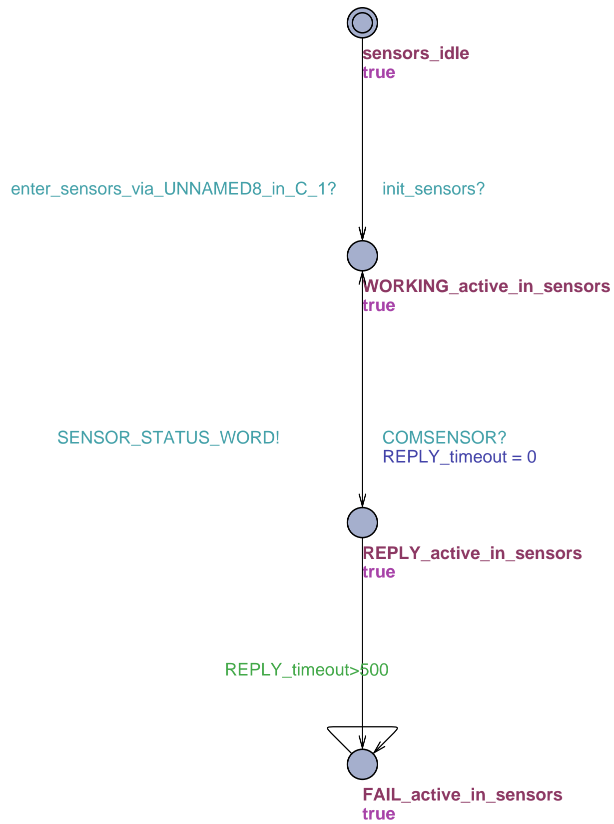

Рисунок 70. Диаграмма UPPAAL для C_1.sensors.

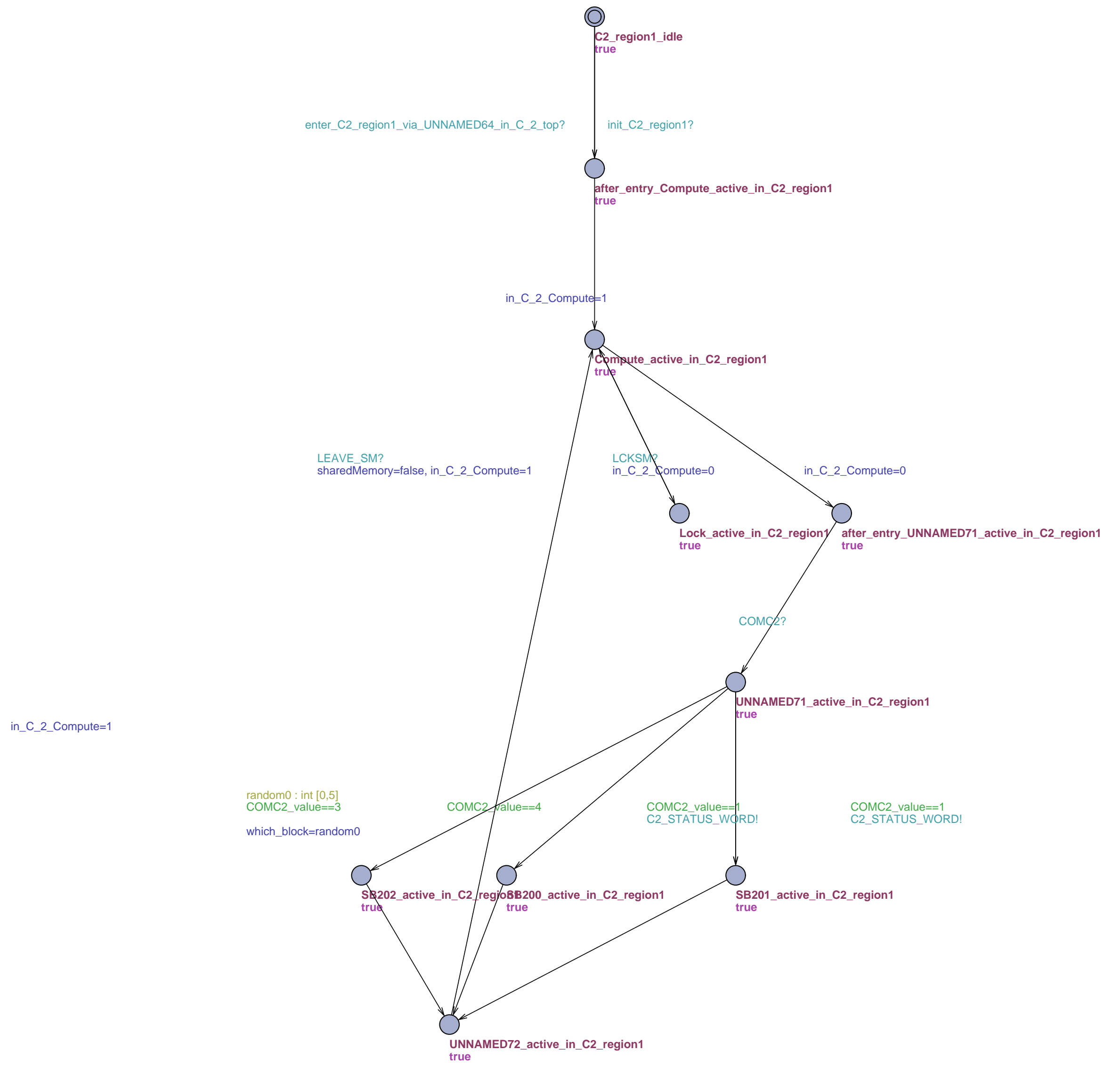

Рисунок 71. Диаграмма UPPAAL для C_2_region1.

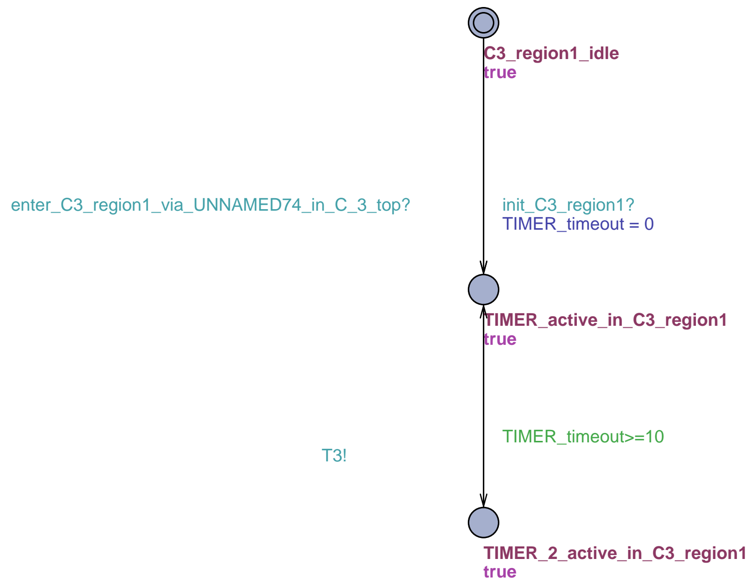

Рисунок 72. Диаграмма UPPAAL для C_3_region1.

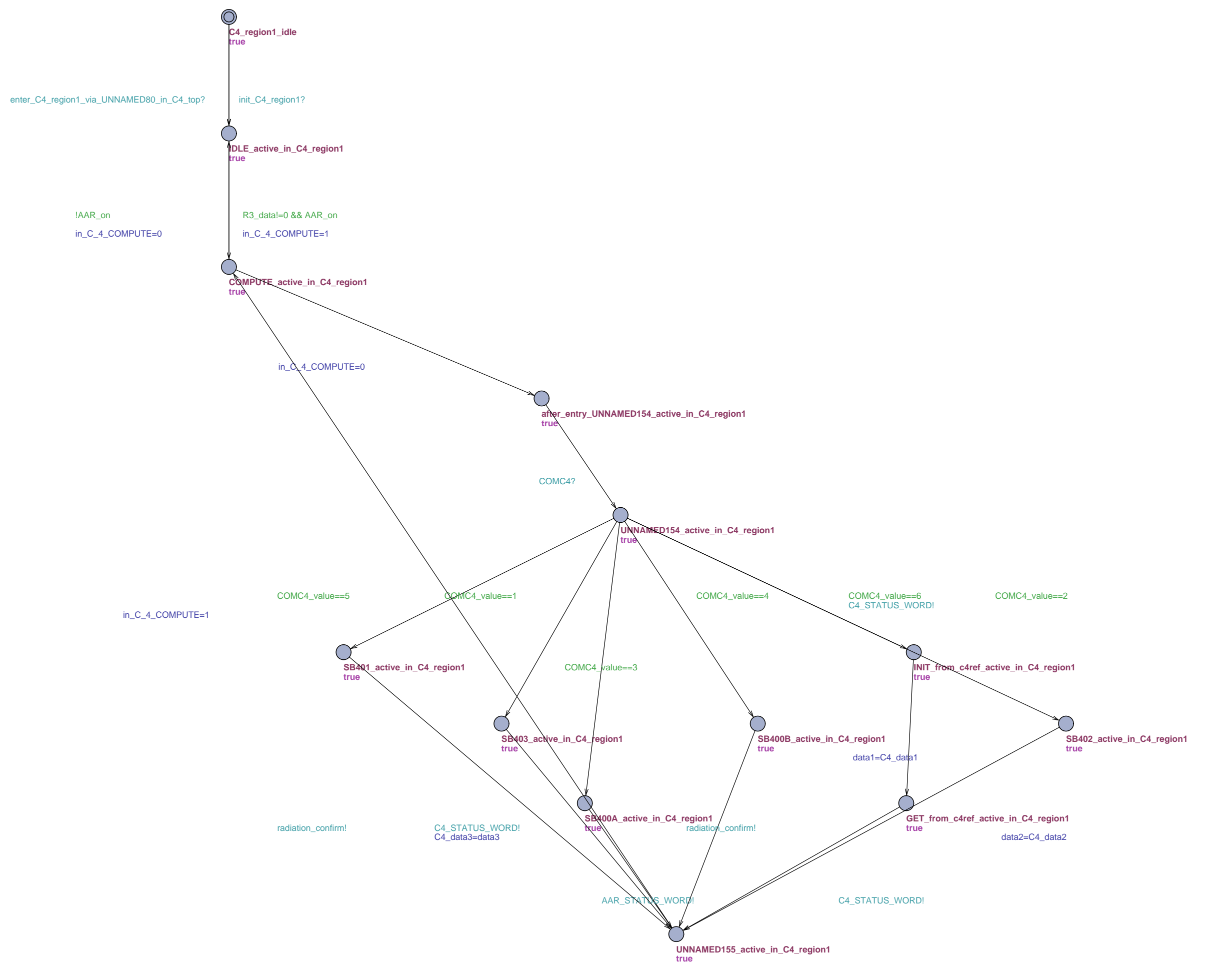

Рисунок 73. Диаграмма UPPAAL для C_4_region1.

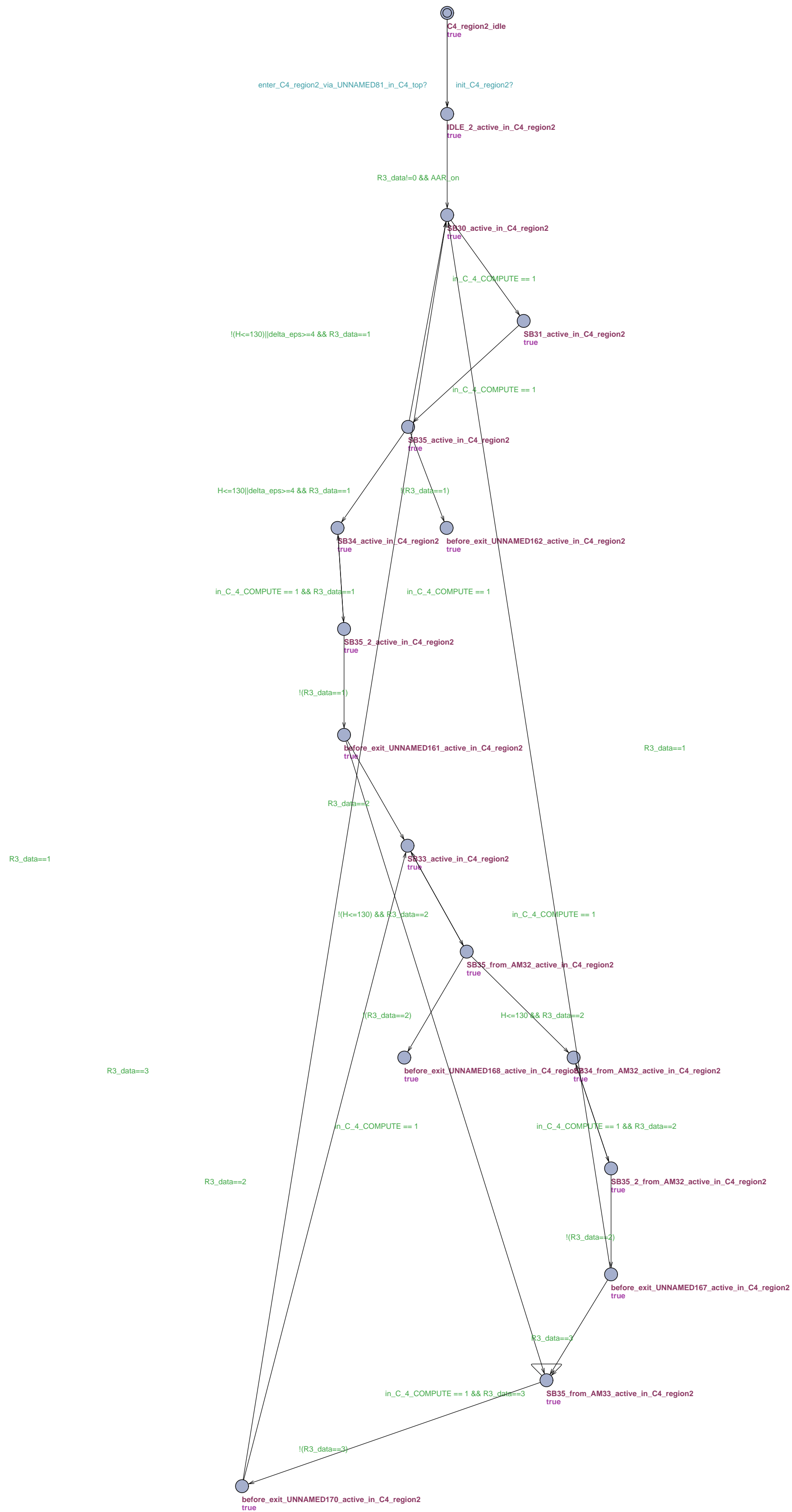

Рисунок 74. Диаграмма UPPAAL для C_4_region2.

На этой системе были верифицированы различные свойства.

E<> region2_process_proc.SB13_active_in_region2

E<> region2_process_proc.S2_active_in_region2

E<> region2_process_proc.SB27_active_in_region2

E<> region2_process_proc.ENDE_from_sb109ref_active_in_region2

E<> region2_process_proc.SB26_active_in_region2

E<> region2_process_proc.SB111_active_in_region2

Достижимость различных состояний. Время верификации этих свойств варьируется от нескольких секунд до порядка 10 минут в зависимости от того, насколько близко данное состояние к начальному. Объем расходуемой памяти варьируется от 5 Мб до более 100 Мб.

region2_process_proc.WAIT_4_active_in_region2 --> C2_region1_process_proc.SB201_active_in_C2_region1

region2_process_proc.S2_active_in_region2 --> C2_region1_process_proc.SB201_active_in_C2_region1

region2_process_proc.WAIT_from_SB106A_active_in_region2 --> C4_region1_process_proc.SB403_active_in_C4_region1

region2_process_proc.WAIT_2_1_active_in_region2 --> C4_region1_process_proc.SB402_active_in_C4_region1

Свойства, гарантирующие, что на различные управляющие сообщения по шине компьютер C_2 реагирует переходом в соответствующее состояние с обработкой события.

(R3_data == 1) --> C4_region2_process_proc.SB30_active_in_C4_region2

(R3_data == 2) --> C4_region2_process_proc.SB35_3_active_in_C4_region2

(R3_data == 3) --> C4_region2_process_proc.SB33_active_in_C4_region2

Проверка правильности переключения состояний процессора C_4. При получении сигнала с кодом R3_data процессор переходит к соответствующему состоянию.

C3_region2_process_proc.AM13_STS6_active_in_C3_region2--> C3_region3_process_proc.SB7_from_AM23_active_in_C3_region3

C3_region2_process_proc.SB1_active_in_C3_region2--> C3_region3_process_proc.SB8_from_c3ref_active_in_C3_region3

C3_region2_process_proc.SB15_active_in_C3_region2 --> C3_region3_process_proc.SB7_from_AM22_active_in_C3_region3

Проверка правильности переключения режимов в двух параллельно выполняющихся циклах процессора C_3.

(C2_fault && C3_fault) -->
region2_process_proc.WAIT_from_SECTION3_active_in_region2

(C2_fault && !C3_fault) -->
region2_process_proc.WAIT_from_SECTION3_active_in_region2

(!C2_fault && !C3_fault) -->
region2_process_proc.WAIT_from_SECTION3_active_in_region2

Проверка правильности работы главного цикла C_1. В зависимости от того, каким процессоры отказываются, осуществляется переход в ту или иную секцию.

Верификация всех этих свойств, связанных с синхронизацией, требует нахождения только одного пути в графе, поэтому делается сравнительно быстро, расходует порядка 1 Гб памяти и около часа времени.

A[] critical_C2 == false || critical_C3 == false

Свойство безопасности: невозможен одновременный доступ к разделяемой памяти. Переменные critcal_C2 и critical_C3 были добавлены на диаграмму специально для верификации и означают, что вычислители C_2 и C_3 соответственно находятся внутри критической секции.

Это свойство было проверено также следующим образом: в диаграмме RW2 было убрано изменено предусловие для входа в критическую секцию, и было проверено, что указанное выше свойство перестало выполняться.

A[] !deadlock

Отсутствие в системе тупиков

Верификация последних двух свойств требует перебора всех возможны состояний, поэтому для нее необходимы значительные ресурсы: она требует более 4 Гб памяти и длится несколько часов. Контрпример для заведомо некорректной системы был найден менее чем за 5 секунд.

4 Подготовка научно-методических материалов для учебных материалов по тематике проекта объёмом 64 академических часов

По курсам «Технологии разработки встроенных систем», «Математические методы спецификации и верификации ПО», «Математическая логика и логическое программирование», «Прикладные логики» и «Распределенные алгоритмы» были подготовлены учебные материалы, представленные в приложении А.

По курсу «Технологии разработки встроенных систем» были разработаны материалы в виде слайдов для презентации объёмом 54 академических часа. Подготовлены материалы для следующих лекций:

- обзор курса. Архитектура бортовых встроенных систем;
- архитектуры процессоров MIPS, ARM для встроенных систем;
- жизненный цикл разработки встроенной системы;
- динамические методы обеспечения качества во встроенных системах;
- статические методы обеспечения качества во встроенных системах;
- обзор моделей надёжности ПО встроенных систем;
- расчёт надёжности встроенной системы;
- методы повышения надёжности встроенных систем;
- технология SRE: определение необходимого уровня надёжности;
- технология SRE: функциональные срезы;
- технология SRE: подготовка к тестированию;
- технология SRE: запуск тестов;
- средства анализа надёжности;
- технология SRE: анализ данных об отказах для принятия решений;
- разработка надёжного ПО встроенных систем на разных этапах ЖЦ;
- методика Cleanroom как пример методики разработки надёжного ПО встроенных систем;
- стандарт взаимодействия моделей HLA;
- системы и языки моделирования;
- графические языки описания имитационных моделей;

- разработка встроенных систем через моделирование. Среды моделирования Диана, Стенд ПНМ;
- модельное время и работа с ним;
- алгоритмы синхронизации времени в системах имитационного моделирования;
- средства визуализации и форматы хранения результатов имитационного моделирования встроенных систем;
- использование средств верификации в системах моделирования;
- средства разработки: средства версионного контроля;
- средства разработки: средства поддержки ЖЦ ПО встроенных систем;
- средства разработки: средства отладки программ.

По курсу «Математические методы спецификации и верификации ПО» были разработаны материалы в виде текста лекции объёмом 4 академических часа. Подготовлены материалы для следующих лекций:

- алгоритмы проверки отношений подобия на моделях распределенных систем;
- методы верификации параметризованных моделей распределенных программ.

По курсу «Математическая логика и логическое программирование» были разработаны материалы в виде текста лекции “Темпоральные логики и их применение в проектировании распределенных вычислительных систем” объёмом 2 академических часа.

По курсу «Прикладные логики» были разработаны материалы в виде слайдов для презентации лекции “Неклассические логики и их применение в информатике” объёмом 2 академических часа.

По курсу «Распределенные алгоритмы» были разработаны материалы в виде текста лекции “Методы верификации распределенных алгоритмов ” объёмом 2 академических часа.

Заключение

В результате выполнения работ по третьему этапу НИР были получены следующие результаты:

1. Выделены основные проблемы построения среде выполнения моделей компонентов РВС РВ (см. раздел 1.2).
2. Проведён обзор и сравнительная оценка инструментов с открытым исходным кодом для UML-моделирования (см. раздел 1.3).
3. Выполнен ряд работ по оптимизации и доработки средства трансляции UML в исполняемые модели, совместимые со стандартом HLA (см. разделы 1.5 и 3.3).
4. Разработано средство анализа и визуализации трасс Vis4 с поддержкой формата OTF (см. раздел 1.5).
5. Реализована модель РВС РВ и проведено её экспериментальное исследование (см. разделы 2 и 3.7).
6. Проведено экспериментальное сравнение систем CERTI и Стенд ПНМ для моделирования РВС РВ (см. раздел 3.1).
7. Выполнен ряд работ по оптимизации системы CERTI (см. раздел 3.2).
8. Доказана корректность алгоритма трансляции иерархических временных автоматов в сеть плоских временных автоматов (см. раздел 3.4).
9. Разработан и реализован ряд оптимизации алгоритма трансляции UML во временные автоматы (см. разделы 3.3 и 3.6).

Ниже детализированы полученные результаты.

В рамках данной работы были сформулированы основные проблемы построения среды выполнения моделей компонентов РВС РВ. Главным выводом из этого раздела является тот факт, что разрабатываемая среда выполнения имитационных моделей должна быть сформирована вокруг концепций заложенных в стандарт распределённого моделирования HLA. Из-за требований предполагаемых задач, таких как поддержка разнообразных QoS и не описанных спецификациями стандарта HLA, в новую среду выполнения необходимо добавить дополнительный уровень для передачи данных и расширить функциональность, предоставляемую сервисами RTI с помощью системы, основанной на стандарте DDS.

В результате обзора и сравнительной оценки инструментов с открытым исходным кодом для UML-моделирования был выбран UML редактор ArgoUML. Определяющими факторами выбора стали удобство интерфейса и поддержка экспорта в XMI.

В рамках оптимизации и доработки средства трансляции UML в исполняемые модели, совместимые со стандартом HLA был разработан транслятор из XMI в SCXML и обратно. Также был добавлен код получения сообщения от федератов, для которых обозначено взаимодействие в файле управления федерации, и код обработки атрибутов федерата, и определение типов передаваемых параметров при обмене между федератами, подписанными на параметры друг друга.

При разработке средства анализа и визуализации трасс Vis4 с поддержкой формата OTF была проанализирована архитектура программного средства Vis3 и её возможности по встраиванию поддержки новых форматов трасс, проведен обзор систем управления сборкой проектов и в качестве единой для всех программных средств НИР (в том числе и для Vis4). В качестве наиболее подходящего средства была выбрана система CMake. Программное средство Vis4, поддерживающее формат OTF было разработано и протестировано на имеющихся трассах моделей.

Разработанная модель РВС РВ состоит из четырех вычислителей, подключенных к общей шине. Два процессора имеют общую память. Каждый из вычислителей выполняет свой круг задач. На этой моделе были верифицированы различные свойства. Для некоторых из них, таких как достижимость различных состояний, время верификации варьировалось от нескольких секунд до порядка 10 минут в зависимости от того, насколько близко данное состояние к начальному. Объем расходуемой памяти варьировался от 5 Мб до более 100 Мб. Для верификации свойств, связанных с синхронизацией, таких, как например, проверка правильности переключения состояний процессора, где требуется нахождение только одного пути в графе, требовалось порядка 1 Гб памяти и около часа времени. Верификация свойства отсутствия в системе тупиков требовало более 4 Гб памяти и длилось несколько часов. Контрпример для заведомо некорректной системы находится менее чем за 5 секунд.

Было проведено экспериментальное сравнение систем CERTI и Стенд ПНМ для моделирования РВС РВ. Результаты, показанные реализацией CERTI RTI на простейших тестовых задачах, в несколько раз уступают результатам системы «Стенд ПНМ» и не позволяют использовать её для решения того же диапазона задач без предварительных модификаций. Таким образом, вновь полученные экспериментальные данные подтверждают выводы предыдущего исследования. Система CERTI проигрывает в производительности из-

за своей неэффективной архитектуры, показатели которой, однако, могут быть значительно улучшены. Следовательно, на основе системы CERTI может быть построено средство для эффективного решения задач моделирования РВС РВ в реальном времени.

По результатам экспериментального исследования был выполнен ряд работ по оптимизации системы CERTI. Основным направлением данных разработок было осуществление внедрения выбранных на предыдущем этапе моделей потоков управления и стратегии их выполнения в существующую архитектуру инфраструктуры CERTI RTI. При этом была сохранена возможность сборки традиционной для данного средства централизованной модели, не использующей многопоточные процессы.

На предыдущем этапе проекта [2] 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. 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, была доказана корректность предложенного алгоритма трансляции.

Оптимизация алгоритма трансляции UML-диаграммы в сеть временных автоматов происходит непосредственно после преобразования UML-диаграмм в иерархический автомат. Оптимизация проходит в два этапа. При этом усложняется структура состояний, и время работы средства верификации может как увеличиться, так и уменьшиться (в зависимости от вида участков разбиения метасостояний). В зависимости от важности трех критериев оптимизации – времени работы, числа состояний и сложности стурктуры состояний – следует выбирать пары объединяемых метасостояний и момент прекращения второго этапа. Эти оптимизации были реализованы в существующих средствах.

В рамках работы данного этапа также были сформулированы задачи, требующие дальнейших экспериментов и исследования:

- Создание федерата, обеспечивающего трассировку событий и запись трассы в формате OTF, а также интеграция его с RTI CERTI.
- Разработка и реализация алгоритма смешанной схемы продвижения модельного времени и интеграция его в RTI CERTI.

- Исследование эффективности применения для этой задачи различных структур хранения данных, минимизировав накладные расходы, связанные с излишним копированием, выделением и освобождением динамической памяти в RTI CERTI.
- Генерация кода внутренней логики федерата исполняемой модели, совместимые со стандартом HLA на основании описывающих её конечных автоматов.
- Разработка и реализация алгоритма построения сценария поведения иерархического автомата (UML-диаграммы) по контрпримеру поведения сети плоских автоматов.
- Интеграция разработанных на этом этапе средств в единую среду моделирования РВС РВ.

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. Nance, R. E. A History of Discrete Event Simulation Programming Languages // Technical Report TR-93-21, Computer Science, Virginia Polytechnic Institute and State University, 1993.
4. Object Management Group; Object Interface Systems, Inc; Real-Time Innovations, Inc; THALES, Data Distribution Service for Real-time Systems, version 1.2. 2007.
5. 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.
6. Казаков Ю.П., Смелянский Р.Л. Об организации распределенного имитационного моделирования // Программирование, 1994, No. 2, стр. 45-64
7. Fujimoto R.D. Parallel and Distributed Simulation Systems. Wiley Interscience. 2000.
8. Weingartner, E., vom Lehn, H., Wehrle, K. A performance comparison of recent network simulators // In Proceedings of IEEE International Conference on Communications (ICC '09), Dresden, Germany, 14-18 June 2009, pp. 1-5.
9. Boichat. R., Dutta P., Frølund S., Guerraoui R. Deconstructing paxos // ACM SIGACT News, V. 34, Issue: 1, Publisher: ACM, Pages: 47-67, 2003.
10. Fujimoto R. M., Perumalla K., Park A., Wu H. Ammar M. H., Riley G.F., Large-scale network simulation How big? How fast? // In Proceedings of the 11th IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS’03), Orlando, USA. 2003.
11. Александров А. А. Единая облачная имитационная среда GPSS Cloud // Труды пятой всероссийской научно-практической конференции по имитационному моделированию и его применнеию в науке и промышленности ИММОД-2011, Санкт-Петербург, 2011, С. 315-319.

12. D'Ausbourg B., Noulard E., Siron P. Running real time distributed simulations under Linux and CERTI // In Proceedings of European Simulation Interoperability Workshop - EURO SIW 2008, 16-19 June 2008, Edimburgh, Scotland.
13. GNU ‘make’, http://www.gnu.org/software/make/manual/make.html
14. Boost.Build V2 User Manual, http://www.boost.org/boost-build2/doc/html/index.html
15. Balashov V.V. et al., A hardware-in-the-loop simulation environment for real-time systems development and architecture evaluation // in International Conference on Dependability of Computer Systems, 2008, pp. 80-86.
16. Балашов В., Бахмуров А., Волканов Д.,Смелянский Р.,Чистолинов М., Ющенко Н. Стенд полунатурного моделирования для разработки встроенных вычислительных систем реального времени // Имитационное Моделирование. Теория и Практика: Четвёртая Всероссийская научно-практическая конференция по имитационному моделированию и его применению в науке и промышлености. Сборник докладов. - Санкт-Петербург, 2009. - С.215-219.
17. Noulard E., Rousselot J.-Y., CERTI, an Open Source RTI, why and how // Spring Simulation Interoperability Workshop. San Diego, USA, 2009.
18. Chaudron J.-B., Noulard E., Siron P. Design and model-checking techniques applied to real-time RTI time management // In Proceedings of 2011 Spring Simulation Multiconference - SpringSim'11, Boston, USA, 2011.
19. 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.
20. Stokes J. Introduction to Multithreading, Superthreading and Hyperthreading [ARS] (http://arstechnica.com/old/content/2002/10/hyperthreading.ars).
21. Karlsson M., Karlsson P. An In-Depth Look at RTI Process Models // In Proceedings of 2003 Spring Simulation Interoperability Workshop, Stockholm, Sweden, 2003.
22. Knight P., Corder A., Liedel R., Giddens J., Drake R., Jenkins C., Agarwal P. Evaluation of Run Time Infrastructure (RTI) Implementations, 2002.
23. Möller B., Karlsson M. Making RTI Tuning Easy with Performance Profiles // In Proceedings of 2005 Spring Simulation Interoperability Workshop, Toulouse, France, 2005.
24. Williams A. The Boost Thread Library // [HTM] (http://www.boost.org/doc/libs/1_47_1/libs/boost_thread/boost_thread.htm), 2011.

25. Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. Lecture Notes in Computer Science, 1990, Volume 443/1990, p. 626-638
26. Rocco De Nikola, Frits Vaandrager. Three Logics for Branching Bisimulation. // Journal of the ACM, Vol.42, Issue 2. – 1995. – p. 458-487.