Министерство образования и науки Украины
Харьковский национальный университет радиоэлектроники

На правах рукописи

СОЛОДОВНИКОВ АНДРЕЙ СЕРГЕЕВИЧ

УДК 004.2:004.4’2

ИНФОРМАЦИОННАЯ ТЕХНОЛОГИЯ СИНТЕЗА ПРОГРАММНОЙ АРХИТЕКТУРЫ НА ОСНОВЕ ГРАФОВОЙ МОДЕЛИ

05.13.06 – информационные технологии

Диссертация
на соискание ученой степени
кандидата технических наук

Научный руководитель
Чайников Сергей Иванович,
кантдидат технических наук,
старший научный сотрудник

Харьков – 2016

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ, ЕДИНИЦ, СОКРАЩЕНИЙ И ТЕРМИНОВ

АОГ – Ациклический Ориентированный Граф

БД – База Данных

ВП – Вычислительный Процесс

ВС – Вычислительная Система

ВУ – Вычислительный Узел

ЖЦ – Жизненный Цикл

ИС – Информационная Система

КТ – Контрольная Точка

МПО – Модель Предметной Области

ОС – Операционная Система

ОУ – Объект Управления

ПО – Программное Обеспечение

ПрО – Предметная Область

ПП – Параллельная Программа

ПС – Программное Средство

РКТ – Распределенная Контрольная Точка

РП – Распределённая Программа

САА – Система Алгоритмических Алгебр

САПР – Система Автоматизированного Проектирования

ССКТ – Средства Создания Контрольных Точек

УВВ – Узлы Ввода-Вывода

УО – Управляющий Объект

ЭВМ – Электронная Вычислительная Машина

ЯПФ – Ярусно-Параллельная Форма

ADL– Architectural Description Languages

ANSI –American National Standards Institute

ARIS –Architecture of Integrated Information Systems

CASE – Computer-Aided Software Engineering

CMMI– Capability Maturity Model Integration

CTL – Computational Tree Logic

DFD – Data Flow Diagrams

DSPL – Dynamic Software Product Line

ERD – Entity-Relationship Diagram

FDD – Feature-Driven Development

GML – Graph Modelling Language

GXL – Graph eXchange Language

IDEF – Integrated DEFinition

IEEE – Institute of Electrical and Electronics Engineers

IEC – International Electrotechnical Commission

ISO – International Organization for Standardization

LTL – Linear Temporal Logic

MPI – Message Passing Interface

MSF – Microsoft Solutions Framework

NUMA – Non-Uniform Memory Architecture

RUP – Rational Unified Process

SADT – Structured Analysis and Design Technique

SMP – Symmetric Multiprocessing

SSADM – Structured systems analysis and design method

UFO – Unit, Function, Object

UML – Unified Modeling Language

WF – WorkFlow

XML – eXtensible Markup Language

XP – eXtreme Programming

ВВЕДЕНИЕ

Актуальность темы.

Существующие методы, технологии и инструментальные средства создания программного обеспечения информационных систем такие как Agile, Service Oriented Architecture (SOA), Structured Systems Analysis And Design Method (SSADM), Meris, Test-Driven Development (TDD), Behavior-Driven Development (BDD) , Dynamic Software Product Line (DSPL) позволяют выполнять разработку как при определенных требованиях на основе водопадной стратегии, так и при эволюционно изменяющихся требованиях конечного пользователя на основе стратегии гибкой разработки.

В то же время в современных рыночных условиях высокой конкуренции нужна кастомизация программного обеспечения (ПО), которая заключается в его адаптации к изменяющимся во времени потребностям конечного пользователя, с учетом специфики конкретного предприятия и конкретного рабочего места. Необходимость в кастомизации может возникать как в процессе разработки, так и сопровождения ПО. Методы и технологии, основанные на водопадной стратегии, нецелесообразно применять для решения задачи кастомизации, поскольку они используют только априорно известные требования конечного пользователя. Технологии гибкой разработки основаны на использовании эвристик при построении программной архитектуры и поэтому их применение в задаче кастомизации приводит к чрезмерным материальным затратам.

Исходя из этого, несоответствие между возможностями существующих технологий проектирования и практической необходимостью адаптации ПО к изменяющимся во времени функциональным требованиям конечного пользователя приводит к возникновению проблемы разработки эффективных формальных подходов к кастомизации ПО информационной системы (ИС). Решение указанной проблемы требует разработки новых формальных графовых

моделей программной архитектуры ИС и совершенствования информационной технологии (ИТ) синтеза программной архитектуры на основе этих моделей.

Значительный вклад в развитие теории проектирования и разработки ПО на основе графовых моделей программной архитектуры внесли В. В. Воеводин, А. Л. Перевозчикова, А. А. Шалыто, Е. Л. Ющенко, а в создание и исследование методов структурного синтеза, адаптации и кастомизации ПО – И. Д. Зайцев, Е. М. Лаврищева, Петров Е. Г., Karl Lieberherr, Pekka Kalevi Abrahamsson, Harald F. O. von Korflesch, Matthias Bertram и другие специалисты.

Однако следует отметить, что при разработке формальных графовых моделей программной архитектуры с использованием существующих методов эволюционные изменения требований конечного пользователя обычно не рассматриваются, что приводит к трудностям при решении указанной задачи кастомизации.

В связи с этим, разработка методов структурного синтеза программной архитектуры ИС с учетом требований конечного пользователя, которые изменяются во время ее разработки и внедрения, является актуальной научно-практической задачей.

Связь работы с научными программами, планами, темами. Диссертационная работа выполнена в рамках плана научно-исследовательских работ кафедры системотехники Харьковского национального университета радиоэлектроники по теме №293 «Розробка методології математичних моделей соціально-економічних систем при реалізації концепції їх стійкого розвитку» за разделом № 293-1 «Розробка математичних моделей реалізації концепції стійкого розвитку соціально-економічних систем» (№ ГР 0115U002429).

Цель и задачи исследования. Целью исследования является разработка модели, методов и информационной технологии синтеза программной архитектуры информационной системы для повышения эффективности конфигурирования программного обеспечения в условиях изменяющихся требований конечного пользователя для формализуемых задач.

Для достижения цели необходимо было решить такие задачи:

– анализ методов и инструментальных средств разработки и конфигурирования программной архитектуры ИС;

– разработка ярусно-параллельной графовой модели программной архитектуры ИС;

– разработка метода объединения вершин графовой ярусно-параллельной модели программной архитектуры ИС на основе оценки показателей сложности и связности программных модулей;

– разработка графового метода оценки функциональной сложности ПО;

– разработка автоматного метода проверки выполнения ограничений формирующегося ПО;

– разработка ИТ структурного синтеза программной архитектуры ИС с возможностью конфигурирования ПО в условиях изменяющихся требований конечного пользователя;

– программная реализация ИТ структурного синтеза программной архитектуры ИС с возможностью конфигурирования ПО в условиях изменяющихся требований;

– внедрение результатов исследования для решения практических задач конфигурирования ПО.

Объектом исследования являются процессы синтеза программной архитектуры информационной системы в условиях изменяющихся требований конечного пользователя.

Предметом исследования являются графовые модели и методы структурного синтеза программной архитектуры информационной системы в условиях эволюционно изменяющихся требований конечного пользователя на основе графовых моделей.

Методы исследования. В процессе диссертационного исследования использованы теория графов при построении графовой модели программной архитектуры информационной системы и разработки методов упрощения структурной и функциональной сложности графовой модели, а также теория

автоматов и неклассические логики при разработке автоматного метода проверки выполнения ограничений на формирующееся программное обеспечение.

Научная новизна полученных результатов заключается в разработке модели и методов структурного синтеза программной архитектуры информационной системы на основе графовых моделей, для чего:

– *впервые* предложен метод объединения вершин ярусно-параллельной графовой модели программной архитектуры информационной системы, отображающие программные модули, который отличается от существующих последовательным расчетом показателей сцепления и связности модулей и позволяющий уменьшить структурную сложность модели, тем самым снижая временные затраты на адаптацию программной архитектуры;

– *впервые* предложен графовый метод оценки функциональной сложности программного обеспечения, который отличается от существующих использованием графовой ярусно-параллельной модели программной архитектуры информационной системы и включающий в себя этапы определения функциональных характеристик программных модулей и расчета критериев функциональной сложности на основе выделения подграфов функциональных задач. Метод позволяет оценить соответствие формирующегося программного обеспечения функциональным требованиям конечного пользователя при конфигурировании программной архитектуры;

– *усовершенствован* автоматный метод проверки выполнения ограничений к формирующемуся программному обеспечению, выраженных в форме нефункциональных требований. Метод отличается от существующих этапами синтеза автоматной модели взаимодействия программных модулей, проверки выполнимости функций программного обеспечения, отказоустойчивости в условиях аппаратных ограничений и сравнения с требованиями к сценариям взаимодействия модулей. Метод позволяет повысить эффективность конфигурирования программной архитектуры с учетом требований к развертыванию программного обеспечения;

усовершенствована графовая ярусно-параллельная модель программной архитектуры информационной системы, которая отличается от существующих учетом взаимосвязей по данным между модулями, которые представлены в модели в виде вершин графа. Модель позволяет на основе оценки структурной сложности повысить эффективность адаптации программной архитектуры информационной системы к эволюционно изменяющимся требованиям конечного пользователя.

Практическая значимость полученных результатов. На основе предложенных модели и методов усовершенствована информационная технология структурного синтеза программной архитектуры информационной системы, которая, в отличие от существующих технологий, содержит этапы синтеза графовой ярусно-параллельной модели программной архитектуры, конфигурирования архитектуры с учетом текущих функциональных требований и проверки выполнения нефункциональных требований на основе соответствующего автоматного метода, что позволяет снизить функциональную и структурную сложность программного обеспечения и, тем самым, снизить затраты на реализацию функциональных задач информационной системы с учетом изменяющихся во времени требований конечного пользователя. Информационная технология реализована в виде программного продукта, работа которого основывается на предложенной графовой ярусно-параллельной модели программной архитектуры информационной системы, методе объединения вершин ярусно-параллельной графовой модели программной архитектуры информационной системы, методе оценки функциональной сложности программного обеспечения, методе проверки выполнения ограничений к формирующемуся программному обеспечению. Информационная технология позволяет формировать программное обеспечение на основе изменяющихся требований конечного пользователя.

Внедрена информационная технология в виде программного обеспечения в научную и проектную деятельность Института физики высоких энергий и ядерной физики Национального научного центра «Харьковский физико-

технический институт» (ИФВЕЯФ ННЦ ХФТИ, акт внедрения от 18.04.2016 г.) и методы упрощения структурной и функциональной сложности графовой ярусно-параллельной модели программной архитектуры в проектную деятельность Харьковского научно-исследовательского института технологии машиностроения (акт внедрения от 9.12.2014 г.).

Личный вклад соискателя. Основные научные положения и результаты, изложенные в диссертации, получены автором лично. В работах, опубликованных в соавторстве, соискателю принадлежат: [91] – определение основных принципов организации вычислительных процессов и их взаимодействия, которые основанны на отражении состояний вычислений на графовой модели программной архитектуры ИС; [93] – формализованное описание графовой модели программной архитектуры ИС; [85] – алгоритм восстановления результатов работы вычислительных процессов на основе контрольных точек восстановления, который применяется в ИТ структурного синтеза программной архитектуры; [89] – модель конечных автоматов, обеспечивающая взаимодействие программных модулей, которая используется в автоматном методе проверки выполнения ограничений формирующегося ПО; [118] – ИТ, обеспечивающая структурный синтез программной архитектуры ИС на основе заданной модели предметной области, а также отражены доработанные методы объединения вершин ярусно-параллельной графовой модели и автоматный метод проверки выполнения ограничений на формирующееся ПО, которые выражены в форме нефункциональных требований.

Апробация результатов диссертации. Основные результаты исследования, научные выводы и рекомендации докладывались и обсуждались на 15-ом юбилейном молодежном форуме «Радиоэлектроника и молодежь в XXI веке» (г. Харьков, 18-20 апреля 2011 г.) [87], Международной научно-технической конференции «ИСТ 2012» (А.Р. Крым, пгт. Морское, 22-29 сентября 2012 г.) [84], 15 –ой Международной конференции «Системный анализ и информационные технологии (SAIT'2013)» (г. Киев, 27- 31 мая 2013) [90], Международной конференции «Информационные технологии в управлении (ИВС 2014)»

(г. Санкт-Петербург, 21-28 мая 2014 г.) [75], Международной научно-технической конференции «ИСТ-2014» (г. Харьков, 21-27 сентября 2014 г.) [88], Международной научно-технической конференции «ИСТ-2015» (г. Харьков, 21-27 сентября 2015 г.) [76].

Публикации. По результатам диссертационного исследования опубликовано 13 научных работ: 7 статей (из них 2 единолично), в том числе 5 статей в научных изданиях Украины по техническим наукам; 2 статьи в зарубежных изданиях (в том числе 3 статьи опубликованы в изданиях, включеных в международные наукометрические базы данных Research Bible, Open Academic Journals Index, Index Copernicus, Eurasian Scientific Journal Index, IndianScience, The Journals Impact Factor, Directory Indexing of International Research Journals); 6 публикаций в сборниках материалов и тезисов докладов научно-технических конференций.

Структура и объем работы. Диссертация состоит из введения, 4 глав, выводов, списка использованных источников и приложений. Полный объем диссертации составляет 191 страницу (из них 139 страниц – основного текста) и содержит: 38 рисунков и 14 таблиц по тексту, список использованных источников из 159 наименований (из них – 97 кириллицей, 62 – латиницей) на 18 страницах; 4 приложения на 34 страницах.

РАЗДЕЛ 1
АНАЛИЗ МЕТОДОВ И ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ РАЗРАБОТКИ И КОНФИГУРИРОВАНИЯ ПРОГРАММНОЙ АРХИТЕКТУРЫ ИНФОРМАЦИОННОЙ СИСТЕМЫ, ПОСТАНОВКА ЦЕЛИ И ЗАДАЧ ИССЛЕДОВАНИЯ

1.1 Обзор методов структурного синтеза и автоматизированного конфигурирования программной архитектуры информационной системы

Многие области человеческой деятельности в связи с тенденцией к усложнению за последнее время требуют поддержки ИТ с целью оптимизации и автоматизации труда. В рамках конкурентной структуры рынка ПО немаловажную роль в процессах проектирования и разработки играет не только качество, надежность, информационная безопасность, но и скорость формирования готового программного продукта. Так же заказчики предъявляют высокие требования к скорости выполнения функций самим ПО. Особенно это характерно для ИС, характеризующихся структурной, функциональной, информационной сложностью, сложной динамикой поведения. Проектирование и разработка подобного рода ИС требует значительных трудовых, временных затрат. В связи с увеличением сложности ИС, согласно стандарту ISO/IEC 12207-2008, усложняются процессы проектирования программной архитектуры, менеджмента конфигурации, менеджмента повторного применения программ и сопровождения ПО ИС. Поэтому на современном этапе активно развиваются автоматизированные методы синтеза программмной архитектуры, её компоновки и конфигурирования. Возникает задача кастомизации ПО, выражающаяся в адаптации программной архитектуры и функционала программного продукта к требованиям конечного пользователя.

Один из распространенных подходов к формированию архитектуры ПО – применение сборочного подхода или использование технологии композитных приложений, которая показывает свою эффективность при использовании

готовых программных компонентов сторонних разработчиков [100]. Тем не менее, в некоторых случаях существуют проблемы, связанные с принципами стыковки программных компонентов, обеспечением совместимости их функций. Решение этих проблем происходит: а) в рамках использования ограниченного числа компонентов, известных пользователю; б) в случае принадлежности компонентов одному разработчику, когда их технологическая и методологическая совместимость изначально обеспечена [38]. В этих случаях логично использование проблемной ориентации целевого ПО [107], применяемого для задач моделирования, контроля, анализа, автоматизированного управления, которое ограничено узкой предметной областью (ПрО). Синтез таких программных систем или комплексов программных средств (ПС) возможен на базе существующих программных компонент, сервисов и программных модулей.

Сборка ПО осуществляется в ручном, автоматическом или полуавтоматическом (автоматизированном) режимах [48].

Автоматический режим, хотя и позволяет снизить время разработки программы, все же обладает рядом недостатков в сравнении с автоматизированными методами, а именно – сложность генерации ПО для распределенных или параллельных вычислительных систем (ВС) и при наличии недетерминированных, трудно формализуемых ПрО.

В общем смысле выделяют два подхода к синтезу программной архитектуры ИС, основываясь на степени формализации исходной модели ПрО: 1) логический и 2) структурный синтез [34, 48, 80]. Логический синтез программной архитектуры базируется на математическом исчислении, представляющем закономерности функционирования объектов и их взаимосвязи, и трудно применим для нетривиальных ПрО. Структурный синтез обладает бóльшими преимуществами в таких случаях и позволяет использовать наглядное представление структуры.

Логичным совмещением формальных и неформальных средств описания архитектуры ИС является архитектурный фреймворк [44, 144], вмещающий в себя конвенции, принципы и методы описания архитектуры.

Для выявления особенностей и подходов к автоматизации процесса синтеза программ рассмотрим существующие методы.

Для процесса проектирования программной архитектуры ИС за основу может быть взята одна из существующих технологий проектирования (SADT, IDEF, SSADM, Meris) [60], основываясь на критерии схожести интерпретации этапов ЖЦ ИС. Однако для обеспечения автоматизации процесса синтеза архитектуры в качестве исходной информации используется формализованное описание ПрО, формализованное представление программной архитектуры, а также требования конечного пользователя к ПО [92].

Такой формальный подход реализован на базе совокупности формальных документов, адекватно отражающих ПрО, в виде инструментария – генератора проектов, который позволяет на конечных этапах генерировать программный код системы и выполнять технологическую сборку [18, 19].

Разработки в данном направлении велись с 80-х годов [28, 30]. Однако, в случае использования метода генерации проектов имееют место недостатки, связанные с употреблением генерируемых скриптов вместе с текстами программного кода для сборки программ проекта по исходным текстам в соответствии с выбранной платформой. Это является причиной разработки и поддержки дополнительного ПО, которое анализирует полученные скрипты.

В 80-х годах в качестве инструментария для генерации программной архитектуры была предложена диалоговая система, позволяющая осуществлять синтез ПО для промышленных объектов [22]. Ключевыми особенностями данной разработки является оптимизация процесса синтеза программной архитектуры путем: 1) запоминания состояния задачи на любом этапе с последующим восстановлением; 2) проверки исходных данных задачи до ее решения, в процессе решения и после него; 3) оперативного ввода исправлений в исходные данные; 4) предоставления пользователю возможности многосеансной работы. Такой функционал позволяет прорабатывать различные стратегии решения задачи, минимизировать время ожидания решения задачи за счет снижения количества ошибок, приводящих к сбою ВП и разбивать последовательность действий

пользователя на этапы (сеансы) с длительными временными перерывами между ними. Для осуществления синтеза программной архитектуры делается упор на развитие проблемно-ориентированного языка.

В 90-х годах было предложено использование диалоговых систем для автоматизированного формирования ВП на основе маршрутов, выделяемых на графовой модели вычислений [63]. Однако развитие диалоговых систем пошло в сторону проектирования диалога на базе естественного языка и в дальнейшем получило развитие в применении искусственного интеллекта и баз знаний [138].

В настоящее время также известны некоторые системы автоматизированного технологического проектирования, работающие в диалоговом режиме, например, «ТехноПро» [77]. Все же на современном этапе уделяется мало внимания вопросам оптимизации диалогового интерфейса пользователя в смысле разграничения функциональной нагрузки между пользователем и ПО ИС. В целях повышения эффективности работы пользователя применяются попытки оценивания физиологических показателей [125].

На современном этапе существует также другой подход к автоматизированной генерации ПО, который основывается на использовании алгебраических спецификаций – системах алгоритмических алгебр (САА) В. М. Глушкова [4, 20, 23] – и методе диалогового конструирования синтаксически правильных программ [5, 83]. Особенность заключается в совместном использовании трех представлений алгоритма при его конструировании: аналитического (САА-формула), естественно-лингвистического и графового (граф-схемы). Такое направление получило развитие в представлении алгоритмов в виде САА-М-схем со смещением акцента в сторону разработки инструментария автоматизированного преобразования параллельных алгоритмов [96, 97] с использованием технологий MPI (Message Passing Interface) и многопоточности в Java. Недостатком этого подхода является зависимость только от конкретного языка программирования.

Выделяют также технологию автоматического синтеза программной архитектуры с использованием онтологии прецедентов [39]. Онтологическое

описание позволяет накапливать опыт разработки, выполнять автоматическую классификацию программ на основе их спецификаций и выполнять построение программ путем адаптации известных решений [62]. Онтологическое моделирование ПрО получило развитие в области GRID-технологии, облачных вычислений, технологий e-Science [27].

Другое направление в автоматическом синтезе архитектупы ПО связано с развитием технологии генетических алгоритмов, позволяющих определять структуру управляющего автомата, который в свою очередь является системой вложенных и взаимовызываемых автоматов [1]. Такой подход реализует технологию автоматного программирования и обладает такими преимуществами как автоматизация процесса верификации, документируемости, упрощение процедуры внесения изменений. Автоматный подход находит широкое применение не только в синтезе программного кода [33], но и в управлении поведением самой системы, запущенной на выполнение [70].

Также интересен подход, реализованный для С-программ, а именно – синтез С-программы с использованием мажоритарного принципа в представлении структуры семантико-числовой спецификации и мажорированного С-графа [10]. Однако для данного метода присутствует недостаток, выраженный в ограничении применения к другим целевым языкам программирования.

Заслуживают внимания методы и средства автоматического построения параллельных программ с использованием технологии CUDA по непроцедурным спецификациям [8]. Такая технология базируется на методах декларативного программирования, что позволяет получать программу с высоким уровнем абстракции с отражением самого метода решения, а не его реализацию при конкретных условиях. Недостаток такого подхода – ограниченная применяемость в силу недостаточной универсальности методов, неприменимых для другого аппаратного обеспечения и ПрО.

Одним из эффективных подходов на современном этапе является сервис-ориентированный подход. Он требует применения архитектурных фреймворков и банков видов моделей, методов, знаний, правил и алгоритмов для

конструирования ИС в рамках выбранной методологии [44]. Для описания программной архитектуры ИС широко используются ADL-языки (Architectural description languages) [133, 159]. ADL-языки используются в качестве средств описания архитектурных спецификаций, их интеграции с целевыми моделями ИС, описанных аспект-ориентированными графами [121]. В случае параметрического синтеза ПО ИС применяются модели многослойного графа [3]. Кроме специализированых ADL-языков также применяются UML нотации для описания программной архитектуры [145].

Применение архитектурных шаблонов и проблемно-ориентированных языков описания архитектуры находит применение в развитии технологии построения композитных приложений [41, 42]. Примером служит инструментально-технологическая платформа CLAVIRE, использующая проблемно-ориентированный язык EasyFlow и позволяющая автоматизировать формирование архитектурных шаблонов, на основе которых выполняется генерация сценариев запуска приложения в распределенной среде [31].

С усложнением ПрО увеличивается сложность аппаратного и программного обеспечения. Повышаются требования к эффективности и производительности ПО (стандарт ISO/IEC 25041:2012). Это требует применения альтернативных технологий увеличения вычислительной мощности [81]. Решение проблем оптимизации ВП сводится к технологиям организации распределенных и параллельных вычислений. Однако в этой связи растет сложность проектирования и разработки качественного ПО [47]. В случае невозможности непосредственного использования конечным пользователем ИС с проблемной ориентацией на базе многопроцессорной или распределенной архитектуры прользователю предлагается использование GRID-технологии и предоставление вычислительных мощностей компьютерного кластера в качестве сервиса [48, 130, 141].

Сервисно-ориентированные технологии организации кластерных вычислений являются на данный момент наиболее перпективными. Они порождают новое ответвление – облачные технологии. Однако в области сервисно-ориентированных технологий присутствует существенная проблема –

обеспечение безопасности данных, находящихся во владении сторонних организаций. Среди известных типов угроз (сетевые атаки, вредоносное ПО, уязвимости в приложениях и ОС) при использовании облачных технологий добавляются сложности, связанные с контролем среды (гипервизора), трафика между гостевыми машинами и разграничением прав доступа [73].

В случае использования итерационных моделей ЖЦ ИС и динамически формируемых требований используются Agile-технологии, которые показывают свою эффективность для небольших компаний-разработчиков ПО [105]. В случаях средних и крупных компаний, которые ведут разработку сложных программных систем с заданными высокими требованиями к надежности, точности и эффективности, применяются технологии, учитывающие функциональные и нефункциональные требования конечного пользователя. Программные спецификации, составляемые на основе требований, используются в качестве основы для разработки ПО через тестирование TTD [14] и с учетом поведенческих свойств ПО – BDD [106, 154]. Однако постепенный рост дополнительной функциональности готовых программных продуктов в процессе его технической поддержки приводит к неконтролируемому разрастанию программной архитектуры, увеличению сложности программ, что влечет за собой увеличение стоимости или прекращение сопровождения ПО. Это становится проблемой для технологий гибкой и интенсивной технологий разработки в современных условиях рынка ПО [137]. В качестве решения проблемы неконтролируемого разрастания функциональности программ предлагается использовать мониторинг актуального состояния ПО на предмет идентификации устаревшей функциональности и удалении ненужных функций из программной архитектуры. Автоматический процес сокращения функциональной сложности базируется на отслеживании изменения значимости функций приложения во времени от версии к версии в компании разработчика путем фиксации частоты использования функций ПО конечным пользователем [137].

Рост функциональной сложности, являющийся одной из причин повышения стоимости сопровождения программных продуктов, обусловливает

необходимость применения также и другого подхода в решении этой проблемы. Кстомизация ПО для бизнес-процессов и ИТ-инфраструктуры является выходом из сложившейся ситуации, например, для больших программных комплексов такого класса как ERP-системы [123]. Поскольку такие системы требуют нескольких месяцев или лет для развертывания, внедрения и начала успешной эксплуатации [143]. Кастомизация подразумевает адаптацию ПО к организационной структуре ИС, основываясь на использовании сервис-ориентированной архитектуры и сервис-доминантной логики (Service Dominant Logic) [123]. Для автоматизации процессов кастомизации и снижения функциональной сложности ПО требуется применение формального аппарата – графовых модели программной архитектуры разрабатываемых ИС.

На современном этапе для решения проблем кастомизации и динамического конфигурирования компонентов ПО применяется развитая технология динамических линеек программных продуктов DSPL [131, 134]. Такие системы адаптируются не только к изменяемым требованиям пользователя, но также и к среде функционирования, позволяя изменять свою функциональность без перезагрузки системы. DSPL технология применяется для разработки саморегулирующихся систем [152]. Технология DSPL может базироваться на использовании сервис-ориентированной архитектуры программного продукта [111]. Недостаток этих технологий – недостаточное обеспечение безопасности данных.

По данным отчетов за 2015 и 2016 годы компании Panorama Consulting Solutions о применяемых технологиях в проектировании и разработке программных систем ERP класса наблюдаются тенденции значительного снижения процента использования сервис-ориентированой технологии в рамках модели «приложение как сервис» SaaS (с 33% до 17%); увеличение процента использования ERP-платформы на базе технологии ERP-облако (с 11% до 27%). При этом остается неизменным процент использования локального развертывания и использования ПО (56%) [98, 99]. Согласно данным этого же отчета предприятия, преобретающие программный продукт, отказываются от

применения модели SaaS и технологии облачных вычислений по причине недостаточного уровня защиты данных: процент неудовлетворенности возрос с 20% до 29% на фоне снижения других причин (рис. 1.1 а).

а) б)

Рисунок 1.1 – Данные отчетов: а) причины отказа от использования облачных технологий; б) требуемая кастомизация приобретенного ПО

Для предприятий, которые внедряют ERP-системы, важным показателем является процент кастомизации приобретенного ПО, то есть процент доработок, осуществляемых при адаптации программного функционала к условиям бизнес-процессов предприятий и требованиям пользователей. В соответствии с отчетами 2015-2016 гг. наибольшее количество предприятий (41%) требуют 11-25% доработок [98, 99]. Однако на 2016 год их количество снизилось до 31% на фоне медленного увеличения количества предприятий (с 22% до 23%), которые требуют 26-50% кастомизации своего программного продукта (рис.1.1, б). Одновременно с этим, благодаря высокой стандартизации и унификации бизнес-процессов произошло значительное увеличение числа предприятий, требующих небольшие изменения в ПО после преобретения (уровень кастомизации 1-10%). Тем не менее, приведенные данные говорят о сохраняющейся актуальности использования своих технологических решений при проектировании и разработке ПО, которое развертывается локально на предприятиях, но, подобно DSPL или сервис-ориентированным технологиям, позволяет обеспечить высокую гибкость, динамическую конфигурируемость, адаптируемость программной архитектуры и функционала, предлагаемого на рынке ПО.

На основе проведенного анализа, можно выделить следующие методы синтеза архитектуры ПО ИС: 1) ручной; 2) автоматизированный (в т. ч. с использованием метода диалогового конструирования); 3) автоматический.

В рамках автоматизированного и автоматического методов наиболее широко используются следующие средства формализации: 1) логико-алгебраические спецификации; 2) автоматные модели; 3) графовые модели; 3) ADL-языки; 4) средства онтологического инжиниринга.

Перечисленные средства формализации могут применяться в рамках *синтезирующего* (на базе модели вычислений, отображающей понятия и отношения ПрО и программной спецификации), *композиционного* (на базе функций и операций композиции в логико-математической системе) и *сборочного программирования* (на базе модели сборки в виде ориентированного нагруженного графа) [41, 42].

Что касается методов проектирования и разработки ПО ИС с использованием наиболее распространенного сборочного подхода, на данный момент существует следующая обобщающая классификация этих методов [42]: 1) модульно-ориентированный; 2) объектно-ориентированный; 3) компонентно-ориентированный; 4) метод генерации; 5) сервисно-ориентированный.

Каждый последующий в этом списке метод является развитием предыдущего. Для реализации данных методов необходимо использовать хранилища готовых решений, компонент повторного использования. При этом указывается на наличие существенных проблем – обеспечение межмодульного интерфейса при сборке ПО ИС [42] и присутствие конфликта между нефункциональными требованиями к компонентам [107].

Актуальность использования методов ситеза программной архитектуры ИС обусловливается сложными ПрО, которые характеризуются [15]: 1) структурной сложностью и территориальной распределенностью; 2) функциональной сложностью; 3) информационной сложностью; 4) сложностью динамики поведения при высокой изменчивости внешней среды.

Для проектирования современных технически сложных систем широко применяются системы автоматизированного проектирования (САПР). Основными функциями этих систем являются [61]: автоматизация выполнения различных проектных процедур с целью нахождения оптимальных вариантов проектируемого объекта, автоматизация выбора схемы или конструкции, автоматизация составления проектной и технической документации. САПР, ориентированные на конкретную ПрО, используют специальные методы, алгоритмы и программы, оригинальные математические модели, учитывающие специфические качества объектов проектирования. В целях решения проблем повышения эффективности работы ПО ИС и возможности координации действий программ указывается на необходимость разделения программной системы на управляющий объект (УО) и объект управления (ОУ) [71]. УО может служить некая исполнительная система, определяемая как компоновщик, который формирует общий программный код на основе атомарных фрагментов (существующих программных модулей) и алгоритма сборки. Компоновщик дополняет программную архитектуру диспетчером, который контролирует исполнение функций ПО ИС (рис. 1.2) [48].

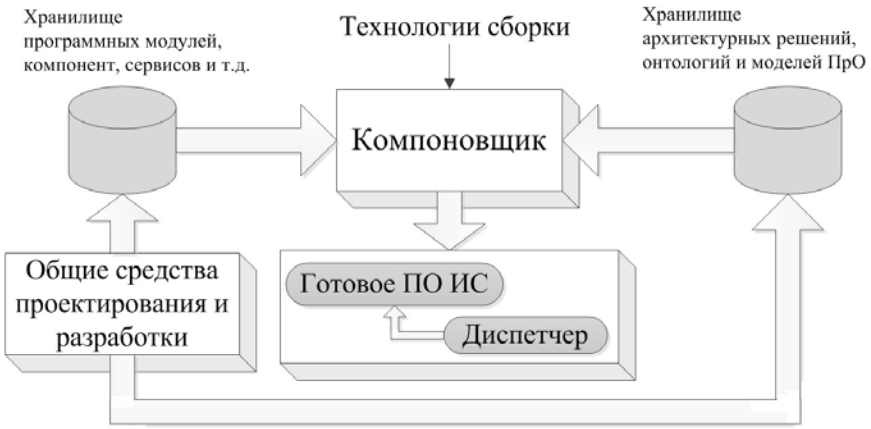


Рисунок 1.2 – Концептуальная структура инструментальных средств, реализующих сборку ПО ИС

При указанной архитектуре компоновщик может быть сориентирован на повторное использование программных модулей, компонент, инструментального

и прикладного ПО. Он может учитывать версионность сборок, а архитектура ИС при этом может быть расширена до включения в её состав нескольких компаний-разработчиков ПО (рис. 1.3), что влечет за собой необходимость разработки процессов повторного применения программ [136].

Рисунок 1.3 – Менеджмент повторного применения ПО

Компания-разработчик может использовать версии ПО, изменять исходный код и, при необходимости, возвращать ПО данной версии с изменениями, указывая, что было модифицировано. В репозитории эти изменения интегрируются с базовой версией ПО. Интегратором является организация, разрабатывающая новые версии ПО. Подобная архитектура хорошо подходит для спиральных и итеративных моделей ЖЦ ПО. В случае спиральной модели осуществимо накопление и повторное использование программных компонентов, моделей и прототипов [32]. Также возможна ориентация на развитие и модификацию ПО в процессе его проектирования.

Основные стадии ЖЦ ПО могут варьироваться в зависимости от выбранной модели ЖЦ (спиральная модель, RUP, MSF), однако процессы ЖЦ программных средств регламентируются соответствующими стандартами.

К итеративным моделям относятся наиболее распространенные MSF (Microsoft Solutions Framework) и RUP (Rational Unified Process), которые используют стандарты ISO/IEC [13].

К подвиду итеративных моделей относят модели, применяемые в рамках гибкой методологии разработки ПО (Agile software development): XP (eXtreme

Programming), Crystal, FDD (Feature-Driven Development), Scrum, которые могут быть сориентированы как на стандарты CMMI v.1.2 – 1.3 так и на стандарт SPICE (ISO/IEC 15504) [14, 40, 77]. Последние модели (относящиеся к Agile методологии) ориентированы на небольшие компании и штат разработчиков.

Учитывая вариативность представлений стадий ЖЦ в зависимости от моделей, следует ориентироваться на существующие стандарты в этой области для выявления этапов и процессов ЖЦ, которые затрагиваются при применении подходов к оптимизациции ПС.

Согласно выполненному анализу методов синтеза в соответствии со стандартом ISO/IEC 12207:2008 затрагиваются следующие процессы ЖЦ: 1) процесс анализа требований к программным средствам; 2) процесс проектирования архитектуры программных средств; 3) процесс детального проектирования программных средств; 4) процесс конструирования программных средств; 5) процесс комплексирования программных средств. Теория графов нашла широкое применение в рамках указанных процессов. Графовые модели, обладая математической простотой, позволяют описать программную архитектуру и формализуемые задачи ИС. Такие модели являются наглядными и хорошо согласовываются с парадигмами объектно-ориентированного, функционально-ориентированного и компонентно-ориентированного программирования. В связи с тем, что способы представления информации о программной архитектуре ИС, необходимой для генерации программного обеспечения, играют важную роль, подробно рассмотрены графовые модели программной архитектуры.

1.2 Анализ формальных графовых моделей программной архитектуры информационной системы

Теория графов получила широкое распространение и применяется во многих областях, в частности – для описания программной архитектуры ИС. Графовые модели используются для различных целей в рамках процессов

проектирования и разработки ПО: отображения информационных зависимостей между программными компонентами, последовательности выполнения функциональных задач системы, описания версии конфигурации ПО, схемы связей по управлению между элементами программной системы.

Для получения графовой модели программной архитектуры ИС обычно используется информация о заданной ПрО в виде таких моделей ПрО как [45, 86]: 1) информационные модели; 2) модели потоков данных; 3) функциональные модели. Данные модели ПрО берутся за основу при проектировании ПО в рамках существующих технологий проектирования ARIS, IDEF, DFD или UML и являются исходной информацией для формального представления программной архитектуры. [34]. Преимуществом графовых моделей архитектуры ПО, базирующихся на формализме теории графов, является способность к визуализации и возможность к автоматизации как процессов проектирования так и разработки программных систем. Теория графов используется для описания архитектуры ПО в граф-ориентированных программных моделях для ВС с параллельной и распределенной архитектурой, а также в целях конфигурирования программных систем [15, 115, 116]. При этом в структуру программмной платформы, реализующей подход к разработке приложений с динамически конфигурируемой параллельной и распределённой архитектурой, включен программный модуль менеджера (диспетчера). Данный модуль осуществляет изменение конфигурации системы во время работы и система не нуждается в перезагрузке. На графовую модель такой программной системы ложится задача описания программных примитивов (программных модулей), реализующих функциональные задачи. В процессе динамического конфигурирования формируется последовательность программных модулей, которые запускаются на выполнение. Эта последовательность исполнения называется конфигурационным планом. Графовая модель представляет собой ориентированный граф, для которого конечному множеству вершин сопоставляются программные модули, а направленным дугам – показатели стоимости и временной задержки передачи данных от одной к другой вершине. Подход к разработке ПО, в основу которого

положены граф-ориентированные программные модели, применяется для кластерных вычислений, web-сервисов, компонентно-ориентированных вычислений.

На современном этапе динамическое конфигурирование осуществляется на основе технологии функционального программирования, с использованием функционально-ориентированных языков программирования, таких как Scala [117]. Недостатком указанного граф-ориентированного подхода является то, что при его использовании не учитывается возможность объединения вершин графовой модели для получения супервершин с целью уменьшения связности и сцепления программных модулей, а также упрощения графового представления программной архитектуры ИС. Подобные графовые модели [135] не содержат дополнительной информации, требуемой для их обработки, используются для описания словарей программных классов и представляются кортежем:

$$G = \langle VC,\ VA,\ VR,\ A, EC, ECO, EA, ER, ERO \rangle,$$

где VC – конструктивные вершины;

VA – заменяемые или изменяемые вершины;

VR – вершины повторного использования;

A – метки;

EC – конструктивные дуги;

ECO – необязательные конструктивные дуги;

EA – изменяемые дуги;

ER – дуги повторного использования;

ERO – необязательные дуги повторного использования.

Каждой вершине данной графовой модели сопоставляются классы программных модулей. Выделеные на множестве вершин модели подмножество классов повторного использования и подмножество изменяемых классов позволяют описать динамическую часть программной архитектуры. Подобные направленные ациклические графовые модели применяются также в

функциональном программировании для автоматизации типизации, поиска соответствующих функций по их заданным аргументам [119].

Кроме того, на теоретико-множественном уровне представления ИС в качестве основы для процесса синтеза также выступают графовые модели. Примером служит подход, основанный на описании ВП в форме потоков заданий (workflow, WF). В этом случае WF – это ориентированный граф, вершинами которого являются запускаемые задачи, а ребрами – зависимости между задачами по данным и по управлению [36]. Такой подход находит применение для ИС, ориентированных на распределенную вычислительную среду.

Кроме того, среди графовых моделей внимания заслуживают вероятностные модели систем зависимостей. Данные модели на основе графов возникли на стыке многомерного статистического анализа, теории вероятностей, теории графов, теории информации и искусственного интеллекта. Данный класс моделей играет роль строгого языка представления знаний в условиях неопределенности (в частности, в экспертных системах нового поколения) и эффективного аппарата решения разнообразных аналитических задач.

Наиболее привлекательны модели на базе ациклических ориентированных графов (АОГ-модели). Выделяют такие достоинства АОГ-моделей [122, 146, 147, 158]: наглядность, способность отображать причинно-следственные связи и прогнозировать последствия действий (решений), компактное представление систем зависимостей, вычислительная эффективность вероятностного вывода.

Эти свойства обеспечивают эффективное применение таких моделей в медицинской и технической диагностике, социометрическом, эконометрическом и эпидемиологическом анализе, моделировании генетических механизмов, распознавании речи в виде комплекса дисциплин e-Science [6].

Также лапласианы и их спектры играют важную роль в теории и приложениях графов и орграфов [69, 112, 124.]. Так, в задачах согласования характеристик при децентрализованном управлении многоагентными системами [2] исследуется применение спектров лапласианов взвешенных орграфов. В

основу положена базовая модель распределенного согласования характеристик агентов, непосредственно учитывающая расхождения в значениях характеристик пары агентов, что позволяет поставить ей в соответствие взвешенный орграф коммуникаций [14].

Для задач детального описания архитектуры ПО ИС могут применяться четыре основные графовые модели: граф управления, информационный граф, операционно-логическая история и история реализации [17]. Первые две модели не зависят от входных данных и строятся непосредственно по тексту программы. Две последние модели для своего построения формально требуют слежения за выполнением всех операндов. Сложность построения модели возрастает в порядке указанного перечисления. Все указанные модели существуют для всех программ. Если задать вектор исходных данных $Data\_Inp = \{d_p^{INP}\}, \forall p = \overline{1,P}$, где d_p^{INP} – переменная (или класс) заданного типа; P – количество всех входных элементов данных; и проследить за выполнением программы, фиксируя каждое срабатывание операторов отдельной вершиной, получаеся ориентированный граф, называемый историей реализации программы. Объединение таких графов на различных векторах входных данных $Data\_Inp$ называется информационным графом, который описывает последовательность выполнения операций и взаимную зависимость между различными макрооперациями или блоками операций, где однонаправленными дугами являются каналы обмена данными [56].

В связи с масштабностью применения графовых методов, важной задачей является использование специальных языков, работающих с графами как со структурами. Примеры таких языков – ГРАФСЕТ [26, 94] и язык ЯРД **[**82**]**. Также на современном этапе выделяют следующие средства представления графов [35]: *.dl файлы UCINET и *.net файлы Паёка (Pajek); также языки GXL и DyNetML, основанные на XML формате, язык GML (Graph Modelling Language), и один из более современных языков – GraphML, также основанный на стандарте XML. Применение XML-стандарта является средством решения проблемы совместимости форматов предлагаемых средств описания графовых моделей

программной архитектуры ПО со сторонним ПО и существующих стандартов обработки информации.

Существует множество сложных научных, инженерных и экономических задач, для решения которых даже на современных ВС требуется несколько недель. При этом количество таких задач постоянно растёт. Учитывая приведённую выше статистику отказов в крупномасштабных ВС, вероятность потери результатов вычислений очень высока [66]. Исходя из этого, существует серьёзная необходимость обеспечения отказоустойчивого выполнения программ на ВС и в тоже время оптимизации времени работы ВС.

В литературе известен подход к решению этой задачи под названием «backtracking» [9, 48]. Смысл этого метода заключается в использовании «истории» взаимодействий для анализа программы. Вводятся вспомогательные переменные, которые хранят истории взаимодействия по каждому каналу программы. Для хранения историй вводится специальная *историческая переменная* – массив значений, последовательно переданных по соответствующему каналу. А далее при необходимости восстановления этой истории программа обращается к массивам. Кроме принципа backtracking в литературе известен и другой подход, основанный на создании контрольных точек (КТ) вычислений (checkpoint) [102]. Подходы к созданию КТ разделяются на реактивный (reactive) и проактивный (proactive) [149]. Наибольшее распространение получил реактивный подход, также называемый *Checkpoint/Restart* или *Rollback/Recovery* [110]. Он предусматривает периодическое создание КТ восстановления, хранящих состояние выполняющейся программы. В случае отказа одного или нескольких вычислительных узлов (ВУ) любая КТ может быть использована для повторного запуска программы на исправной подсистеме. При этом работа продолжится с момента времени, соответствующего созданию этой КТ. Реактивный подход используется также для балансировки нагрузки ВС [150] и в воспроизводящих отладчиках (Playback Debuggers) [120].

Применение механизма КТ связано с накладными расходами при

выполнении параллельных программ (ПП). Данная операция характерна интенсивным использованием узлов ввода-вывода (УВВ), поэтому среднее время создания КТ может быть весьма значительным.

Существует достаточно много средств создания контрольных точек (ССКТ) [108, 109, 157], каждое из которых имеет свои преимущества и недостатки.

Различают две основные схемы взаимодействия ССКТ с защищаемой программой: явная и прозрачная (неявная). Средства создания КТ, построенные на основе явной схемы, позволяют задать ограниченный набор информации, которую необходимо сохранить в КТ. Недостатком явной схемы является необходимость модификации исходного кода, что не позволяет применять её к программам, доступным только в бинарном виде.

Средства создания КТ, построенные на основе прозрачной схемы, осуществляют сохранение КТ незаметно для программы. Недостатком данного подхода является больший объём дискового ввода-вывода информации, так как сохраняется всё пространство памяти.

По классам поддерживаемых программ ССКТ подразделяют на сосредоточенные и распределённые. Сосредоточенные ССКТ обеспечивают отказоустойчивость выполнения одного или нескольких процессов в рамках вычислительного узла ВС. Распределённые ССКТ обычно строятся на базе сосредоточенных и применимы для распределённых и параллельных программ, что делает их важным инструментом организации функционирования ВС.

Для распределённых ССКТ различают координированный и некоординированный подходы. При создании РКТ каждый процесс РП сохраняет свое состояние в КТ. Целостной РКТ называется набор из N локальных КТ, формирующих допустимое состояние программы [110]. Такая РКТ может быть использована для восстановления программы после сбоя. При координированном подходе создание КТ происходит синхронно, что гарантирует целостность РКТ. При некоординированном подходе каждый процесс создает КТ независимо от других. Следовательно, при восстановлении необходимо выполнять поиск

целостного состояния программы на основе набора независимых КТ. Последнее вносит дополнительные накладные расходы.

Распределённые ССКТ также можно разделить на универсальные и MPI-ориентированные. Первые ССКТ позволяют создавать РКТ для любых распределённых и параллельных программ, в том числе для различных реализаций модели передачи сообщений (PVM, MPI). Что касается вторых, то существует несколько ССКТ, построенных на базе конкретных реализаций MPI (например, OpenMPI, MVAPICH2 [109]). Все они используют пакет BLCR [129] для создания сосредоточенных КТ и реализуют собственные механизмы сохранения графа связей и транзитных сообщений.

Для организации резервирования данных существует возможность применения методики двоичного сжатия [114] с помощью дельта-сжатия. При таком подходе выбирается базовая КТ, которая сохраняется полностью, а для всех остальных КТ сохраняются лишь модифицированные фрагменты. Существует два варианта дельта-сжатия: инкрементное и дифференциальное. Первый подход предусматривает сохранение изменений в текущей КТ относительно предыдущей. Второй – сохранение всех изменений относительно базовой КТ [97, 103]. Недостатком данного подхода является отсутствие гибкости при выборе минимальной единицы модификации, которая имеет фиксированный объём, равный размеру страницы памяти. Изменение одного байта приводит к сохранению страницы целиком.

Технология автоматного программирования применяется при проектировании такого ПО как системы автоматизации ответственных объектов управления. Стандарт IEC 61499, унифицирующий правила создания распределенных управляющих систем, рекомендует описывать базовые функциональные блоки с помощью конечных автоматов (КА). Выбор в пользу автоматных моделей обусловливается требованиями к организации бесперебойного сохранения и восстановления данных ВП, устранения блокировок и минимизации ошибок [68, 95], что позволяет обеспечивать высокую надежность работы ПО.

1.3 Анализ автоматных моделей функционирования программного обеспечения и выбор средств верификации

В стандарте ANSI/IEEE качество ПО понимается как набор характеристик продукта, определяющего степень, с которой ПО будет удовлетворять ожиданиям заказчика (ANSI/IEEE Standard 729-1983). Качество ПО характеризуется такими свойствами, как корректность, надежность, возможность сопровождения и переносимость. Качество ПО включает также его пригодность для целевого назначения, разумную стоимость, простоту использования и характеристики обновлений. Программным системам присущи некоторые нефункциональные требования к качеству сервиса, обеспечиваемого системой. Классификация нефункциональных требований осуществляется следующим образом: производительность, безопасность, надежность.

Надежность является важным качеством программных систем, так как сбой может привести к легко исправляемой ошибке или к катастрофическим потерям. В связи с этим ПО проектируется так, чтобы оно было устойчивым к ошибкам. Важными процессами для подтверждения желаемого качества системы являются верификация и валидация (V&V – verification and validation).

В настоящее время модели КА и их расширения используются преимущественно для спецификации и верификации поведения и взаимодействия подсистем и компонентов ПО [57, 101], так как язык «состояние – переход» является естественным средством описания поведения.

Верификация (проверка структуры на наличие ошибок) построенной модели осуществляется на базе электронных спецификаций, которые описывают стандарты сборки программных модулей [40, 58]. Такая информация как предусловия и постусловия классов программных модулей, описываемая графовой моделью, является исходной для верификации с помощью модели КА [148]. КА могут быть использованы для управления тем, какие наборы ресурсов следует удерживать в памяти в любой момент времени, а также какие элементы пользовательского интерфейса должны находиться на тех или иных участках

экрана пользовательского приложения [70]. Наиболее близкий аналог описываемого подхода, метод под названием Statemate, был предложен Д. Харелом и М. Полити [128]. Он базируется на процедурном стиле: использование подпрограмм (процедур, функций) в качестве модулей, составляющих архитектуру программной системы, и функциональной декомпозиции сверху вниз как основном методе проектирования [49]. В процессе проектирования автоматной модели каждому вложенному автомату может быть сопоставлена определенная непрерывающаяся последовательность действий или отделный объект управления [78]. В последнем случае система получается больше и сложнее, но положительный эффект заключается в раскрытии и использовании такой автоматной модели в объектно-ориентированном программировании с явным выделением состояний. При наличии идентичных автоматизированных объектов достаточно описать только один из них как класс, на основании которого создаются экземпляры класса.

Для обеспечения взаимодействия автоматов чаще всего используется механизм обмена сообщениями [6, 21].

Для обеспечения правильности функционирования автоматов используется процесс верификации. Для выбора верификаторов в качестве одного из критериев взята ориентация на использование линейной временной логики (LTL), служащей для описания требований к автоматной модели, поскольку с точки зрения верификации автоматных моделей – это наиболее удобный вариант, позволяющий при верификации и определении спецификации почти полностью ограничиться понятием КА.

С целью выбора необходимого ПО, ориентированного на применение LTL логики, находящегося в свободном распространении, выполнен обзор существующих верификаторов.

PROD [58**]** – инструмент для анализа достижимости высокоуровневых сетей Петри. Осуществляет верификацию LTL-формул оперативно, CTL-формулы верифицируются только после генерации пространства состояний. Поддерживается генерация контрпримеров.

The Kit (The Model-Checking Kit) [153] – совокупность программ, позволяющая моделировать системы КА и верифицировать их, используя в том числе проверку дедлоков, достижимости, проверку модели для темпоральных логик CTL и LTL. Применяются методы символической проверки модели (BDDs) и методы частичного порядка. Языки моделирования включают низкоуровневый язык Petri Nets. Поддерживается генерация контрпримеров.

Temporal Rover [127**]** – генератор кода для темпоральных логик (LTL/MTL/MTLS). Создает код на C, C++, Java, VHDL, Verilog, способен взаимодействовать с Matlab и Ada.

STeP (Stanford Temporal Prover) [113, 126] **–** инструмент для верификации реактивных систем и систем реального времени. Комбинирует алгоритмические и дедуктивные методы верификации. Существует интерактивный инструмент доказательства теорем для условий, которые не могут быть доказаны автоматически. В качестве языка моделирования используется SPL (Simple Programming Language – Pascal-подобный язык с поддержкой параллельности). Проверка модели осуществляется на основе линейной темпоральной логики LTL. Поддерживается генерация контрпримеров.

SPIN [40, 132] – инструмент формальной верификации распределенных систем. Может быть использован как симулятор для быстрого прототипирования модели, как верификатор для проверки истинности требований корректности модели. Для спецификации системы использует язык высокого уровня PROMELA (PROcess MEta-LAnguage). Применяется для трассировки логических ошибок проектирования распределенных систем, параллельных алгоритмов. Может быть использован в качестве полной системы проверки модели на основе линейной темпоральной логики (LTL). Поддерживается генерация контрпримеров.

Cadence SMV (Symbolic Model Verifier) [139] – инструмент снабжен двумя языками моделирования – расширенным SMV и синхронным Verilog. SMV допускает несколько форм спецификации, которые включают темпоральные логики CTL и LTL, КА, встроенные утверждения и спецификации уточнений. Поддерживает графический интерфейс пользователя и генерацию контрпримеров.

NuSMV (New Symbolic Model Verifier**) [**142**] –** символический верификатор моделей, который является расширением SMV. По сравнению с SMV обеспечивает такие возможности как взаимодействие КА, анализ инвариантов, реализация методов декомпозиции, проверка модели на основе LTL. Поддерживает генерацию и визуализацию контрпримеров.

Bogor [151**]** – работает в консольном режиме, а для использования графического интерфейса необходимо устанавливать специальный плагин для среды разработки Eclipse. Ядро верификатора Bogor не поддерживает верификацию свойств для темпоральной логики.

Приведенная информация о преимуществах и недостатках верификаторов обобщена соответствующим образом (табл. 1.1).

Таблица 1.1 – Сравнительные характеристики верификаторов [58]

Название	Цель		Другие особенности		Техническая поддержка, удобство использования	Бесплатность	GUI	Примеры ОС
	Проверка модели	Доказательство теорем	Реального времени	Гибридные				
Cadence SMV	+	+				+	+	Windows, Unix
PROD	+							Windows, Unix
The Kit	+				+			Unix
Temporal Rover	+		+					Unix
STeP	+	+	+	+	+		+	Unix, FreeRTOS
SPIN	+				+	+	+	Windows, Unix
NuSMV	+					+	+	Windows, Unix
Bogor	+							Windows, Unix

Анализ данных (табл. 1.1) с учетом универсальности ОС назначения, распространенности и бесплатности программного инструментария, технической поддержки и удобства использования позволяет сделать выбор в пользу верификатора SPIN, поскольку в целях дисертационного исследования

требовалось использовать только функцию проверки модели и быстро освоить язык программирования верификатора.

1.4 Постановка цели и задач исследования

Проведенный анализ позволяет утверждать, что при разработке формальных графовых моделей программной архитектуры с использованием существующих методов изменяющиеся во времени требования конечного пользователя либо не учитывается [42, 116, 159], либо не соответствуют заданному уровню безопасности данных [99, 104, 111, 134], либо не позволяют удовлетворить критериям сложности программной архитектуры и её гибкости [106, 137, 154]. В этой связи, разработка эффективных формальных подходов к конфигурированию ПО ИС путем использования графовых моделей программной архитектуры является актульной научно-практической задачей.

При этом область применения ИТ структурного синтеза программной архитектуры ограничивается классом производственных информационных систем со структурированными (формализуемыми) задачами. ПО такого класса ИС требует локального развертывания, имеет трехуровневую архитектуру и обладает возможностью конфигурирования в условиях эволюционно изменяющихся требований конечного пользователя.

В связи с чем, целью исследования является разработка модели, методов и информационной технологии синтеза программной архитектуры информационной системы для повышения эффективности конфигурирования ПО в условиях изменяющихся требований конечного пользователя для формализованных задач.

Для достижения цели исследования были поставлены следующие задачи:

– анализ методов и инструментальных средств разработки и конфигурирования программной архитектуры ИС;

– разработка ярусно-параллельной графовой модели программной архитектуры ИС;

– разработка метода объединения вершин графовой ярусно-параллельной модели программной архитектуры ИС на основе оценки показателей сложности и связности программных модулей;

– разработка графового метода оценки функциональной сложности ПО;

– разработка автоматного метода проверки выполнения ограничений формирующегося ПО;

– разработка ИТ структурного синтеза программной архитектуры ИС с возможностью конфигурирования ПО в условиях изменяющихся требований конечного пользователя;

– программная реализация ИТ структурного синтеза программной архитектуры ИС с возможностью конфигурирования ПО в условиях изменяющихся требований;

– внедрение результатов исследования для решения практических задач конфигурирования ПО.

1.5 Выводы по разделу 1

Анализ методов и инструментальных средств разработки и конфигурирования программной архитектуры ИС позволяет сделать следующие выводы.

1. Выполнен анализ существующих методов структурного синтеза программной архитектуры ИС, в том числе на основе графовых моделей. На основании проведенного анализа установлено, что решение задачи разработки методов и информационных технологий структурного синтеза программной архитектуры, адаптации ПО под изменяющиеся во времени требования конечного пользователя остается актуальным, поскольку на современном этапе достаточно большое число производственных предприятий и компаний (31% – 41%) не удовлетворены уровнем защиты данных для применяемых технологий, а также требуют большой объем доработок исходного ПО, приобретаемого заказчиком. В последнем случае процент кастомизации составляет примерно 11%-25%.

2. Задача разработки эффективных формальных подходов к кастомизации ПО, выраженной в адаптации и доработке программного кода с учетом требований конечного пользователя, в большинстве случаев решается путем использования графовых моделей прграммной архитектуры. Применение таких моделей обусловленно хорошо разработанным математическим аппаратом, наглядностью представления данных и простотой использования в рамках объектно-ориентированной, функционально-ориентированной и компонентно-ориентированной парадигм программирования.

3. Проведен анализ методов и инструментальных средств верификации ПО. Выбор сделан в пользу модели конечных автоматов, позволяющей обеспечить высокий уровень недежности функционирования ПО. В качестве верификатора выбран инструментарий SPIN, основываясь на легкости настройки среды, освоения языка программирования верификатора, распространенности документации и бесплатном свободном использовании. Верификатор SPIN обеспечивает высокую скороть прототипирования автоматной модели.

4. Сформулированы цель и задачи диссертационного исследования.

Разработка методов струткруного синтеза программной архитектуры ИС с учечем требований конечного пользователя, изменяющихся в течении её разработки и внедрения, позволит повысить эффективность конфигурирования ПО.

РАЗДЕЛ 2
ИССЛЕДОВАНИЕ И РАЗРАБОТКА ГРАФОВОЙ МОДЕЛИ ПРОГРАММНОЙ АРХИТЕКТУРЫ ИНФОРМАЦИОННОЙ СИСТЕМЫ

2.1 Исследование особенностей использования графовой модели программной архитектуры в процессах проектирования информационной системы

Анализ литературных источников показал необходимость использования графовых моделей, описывающих программную архитектуру ИС, для повышения эффективности конфигурирования ПО в условиях изменяющихся требований конечного пользователя для формализуемых задач ИС. Подобные графовые модели строятся на основе информации статических и динамических моделей ПрО. Если рассматривать, например, стандарт UML, то диаграмма объектов позволяет выявить подсистемы ПрО и пользователей системы, а диаграмма взаимодействия – описать функциональные отношения между подсистемами и пользователями. Диаграмма активности – выделить основные функции системы. Отталкиваясь от этой информации, разработчик формирует графовую модель и получает соответствующую ей программную архитектуру ИС. Статические и динамические диаграммы, в свою очередь, основываются на спецификациях требований к ПО, которые получают до того, как будет сформирована графовая модель, как это принято, например, для подходов BDD или TDD, в рамках которых свойства вершин графовой модели определяются на базе своих спецификаций программных модулей. ИТ, использующая графовую модель программной архитектуры, не является альтернативой существующим практикам передовых технологий разработки, а лишь может быть использована как дополнение к существующим методам проектирования, разработки и конфигурирования программных средств на соответствующих этапах используемых моделей ЖЦ ИС.

В соответствии со стандартом ISO/IEC 12207-2010 ИТ структурного синтеза программной архитектуры на основе графовой модели затрагивает такие этапы ЖЦ как: 1) процесс проектирования архитектуры программных средств; 2) процесс детального проектирования программных средств; 3) процесс конструирования программных средств; 4) процесс комплексирования программных средств. Указанная ИТ реализуется программным инструментарием, применяемым для формирования ПО ИС. Разработчик использует ИТ в рамках процессов проектирования и разработки ПО как дополнение к выбранной практике или технологии разработки в случае использования итерационных моделей ЖЦ. Результатом применения ИТ является сформированная программная архитектура.

Графовая модель программной архитектуры ИС позволяет описать на этапе проектирования отношения между елементами программной системы, называемыми условно программными модулями. При этом, для определения свойств и отношений применяются программные спецификации. На основании полученной модели разработчик определяет свойства и поведение будущей программы до полной разработки ПО и исправляет обнаруженные ошибки в программной архитектуре. В дальнейшем графовая модель служит основой для автоматизации компоновки программного продукта из готовых программных модулей, которые на завершающих стадиях или новых итерациях ЖЦ могут быть переконфигурированы в зависимости от изменяемых требований заказчика.

Ориентируясь на процессы ЖЦ, проведем исследование применимости графовых моделей программной архитектуры ИС.

Модели ЖЦ ПО стандартизируют процессы проектирования, программирования и развития программного продукта. Соответствующая модель, выбранная разработчиком, определяет степень эффективности, гибкости стратегии компании и качественно влияет на конечный продукт. Модели ЖЦ в широком смысле описывают стадии ПО, оставляя разработчику место для творчества и свободы выбора инструментальных средств и методов автоматизации своей работы. Формализуемые задачи, которые

идентифицируются на стадии анализа модели ПрО, позволяют перейти к формализованному представлению программной архитектуры на основе теории графов [86]. Являясь формальным описанием отношений и взаимосвязей между подсистемами и элементами исходной ПрО, графовая модель позволяет структурировать программную архитектуру получаемой системы и установить соответствие между функциональными задачами ИС и программными модулями, осуществляющих выполнение этих задач.

На первых стадиях модели ЖЦ ИС разработчик проводит логическое проектирование ИС, применяя структурно-аналитический подход. При этом используются спецификации процессов, словарь данных, диаграммы потоков данных, диаграммы состояний, диаграммы зависимостей объектов. На следующих стадиях, предшествующих процессам непосредственной разработки ПО, применяется объектно-ориентированный подход, который основывается на моделях зависимостей между объектами, поведения объектов и моделях взаимодействия объектов. На структурных и функциональных моделях ПрО выявляются те части, которые могут быть реализованы в виде элементов программной архитектры ИС. В случае динамически изменяемых требований конечного пользователя и необходимости адаптации программного продукта к условиям работы очень важным на данном этапе является обеспечение разработчиков инструментарием, позволяющим повысить эффективность процессов структурного синтеза и конфигурирования программной архитектуры ИС [51]. Наряду с этими задачами также важно применить подходы к оптимизации ВП, учитывая влияние человеческого фактора на корректность получаемого результата вычислений, а также снизить влияние ошибок в исходных данных решаемой задачи на общее время работы ПО и восстановление работы ВП путем резервирования данных процесса вычислений.

Другими словами, проблема сводится не только к обеспечению основных показателей качества ПО (например, стандарт ISO/IEC 25051:2014), но и к оптимизации или снижению влияния человеческого фактора на быстродействие программы. При этом для построения качественного ПО ИС важно делать упор

как на предварительную разработку системы, так и на анализ требований к ней. Такой анализ позволяет понять необходимость использования специальных технологий ведения истории вычислений, архивов данных, их восстановления, последовательного или параллельного исполнения ВП, оптимизации решения поставленных задач в конкретной ПрО.

Существующие программные инструментарии проектирования и разработки ПО основываются на CASE-технологиях. Применяются на стадиях модели ЖЦ, ускоряя процессы проектирования и разработки систем и повышая их качество. CASE-технологии обладают радом преимуществ [32], таких как, например, обеспечение высокого уровня технологической поддержки процессов разработки и сопровождения ПО, обеспечение возможности повторного использования программных компонент и архитектурных решений, осуществление адаптации и сопровождения ПО, снижение времени создания системы и получение на ранних стадиях проектирования прототипа формируемой системы вместе с оценкой его работоспособности. Данные достоинства характеризуют CASE-средства как высокоэффективный инструмент. Тем не менее, в зависимости от специфики и сложности конкретной ПрО, всегда требуется усовершенствование методов, заложенных в этот инструментарий. Такая задача также возлагается на предложенную ИТ структурного синтеза программной архитектуры ИС.

Рассмотрим применимость графовой модели программной архитектуры в рамках предложенной ИТ для моделей ЖЦ ИС.

При проектировании сложных систем (например, производственных информационных систем) всегда требуется уточнение требований к ПО, видоизменение функциональных особенностей с учетом ограничений на сроки получения готового продукта, что влечет за собой повышение требований к скорости разработки, поддержке версионности программного продукта, обеспечению возможности использования готовых архитектурных решений. Для решения этих задач применяются итерационные модели ЖЦ.

Согласно стандарту ISO/IEC 15288 выделяют 6 стадий ЖЦ: 1) формирование концепции; 2) разработка; 3) реализация; 4) эксплуатация; 5) поддержка; 6) снятие с эксплуатации.

Интерпретация этих стадий может отличаться в зависимости от используемых моделей ЖЦ. Например, спиральная модель состоит из 5 стадий (рис. 2.1, а) [32].

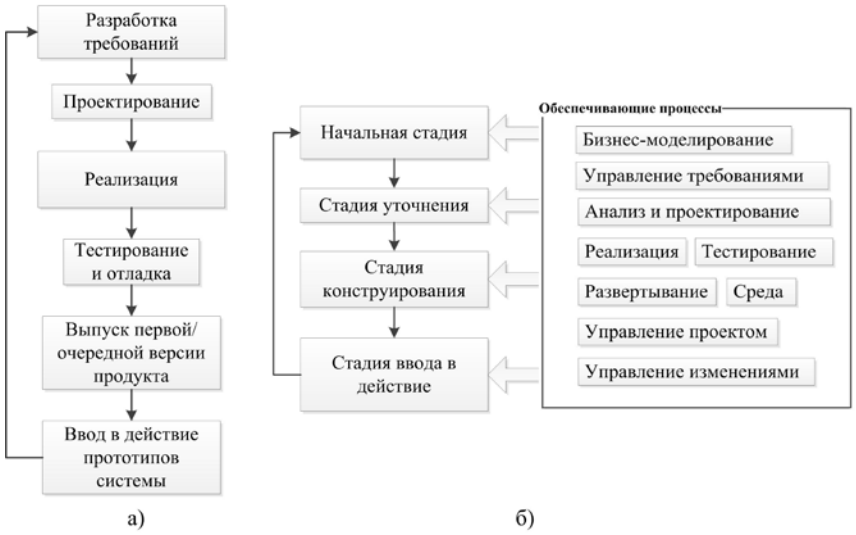


Рисунок 2.1 – Стадии ЖЦ для итерационных моделей:
а) спиральная; б) в рамках RUP-методологии

В случае применения методологии RUP ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта (как для спиральной модели) [13]. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии (рис. 2.1, б). Методология MSF ориентирована на 5 стадий (рис. 2.2, б).

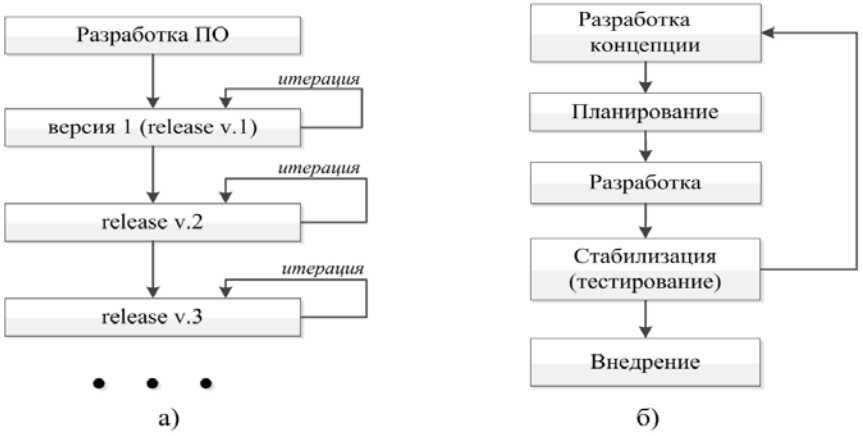


Рисунок 2.2 – Стадии ЖЦ для итерационных моделей:
а) в рамках XP-методологии; б) в рамках MSF-методологии

Для методологии XP (рис.2.2, а) ЖЦ состоит из последовательности версий программного продукта, каждая из которых выполняется в соответствии с выбранной итерационной моделью [14].

Также существует обобщенное представление алгоритма проектирования ПО [136], которое, в соответствии с итерационными моделями ЖЦ, можно представить схемой (рис. 2.3).

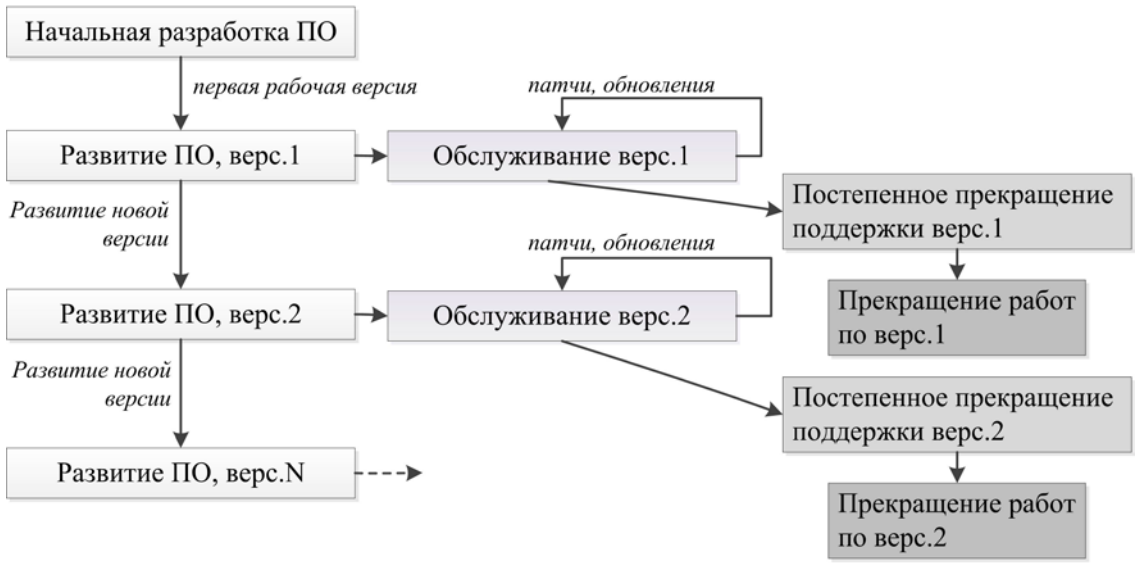


Рисунок 2.3 – Обобщенный алгоритм проектирования ПО ЖЦ

Наиболее распространенные методологии (MSF, SCRUM, RUP) объединяет итеративность и необходимость обеспечить процессы проектирования программной архитектуры, детального проектирования, конструирования и комплексирования ПО информацией о структуре ПрО, то есть данными в виде статических и динамических диаграмм моделирования ИС (стандарт ISO/IEC 12207-2008).

В случае использования предлагаемой ИТ структурного синтеза программной архитектуры ИС для спиральной модели задействываются стадии проектирования и реализации (рис. 2.1, а) или стадия конструирования, как в случае модели RUP (рис. 2.1, б). Спецификации программных модулей позволяют определить и изменить отношения между частями программы, обозначить связи между ними и выявить ошибки в стыковке программных модулей в случае очередной итерации, направленной на видоизменение архитектуры и

функциональных особенностей ПО (рис. 2.2, а) и развивать программный продукт, получая его новые версии (рис. 2.3).

Водопадная модель состоит из пяти основных стадий [59]:

1) анализ требований (получение спецификации ПО);

2) проектирование (составление программной архитектуры);

3) реализация (разработка программного кода);

4) интеграция (интеграция отдельных частей исходного кода);

5) тестирование (тестирование и устранение ошибок ПО).

В этом случае предложенная ИТ структурного синтеза программной архитектуры имеет место для стадий проектирования и интеграции, однако, учитывая однократный проход стадий, такая модель лишает стратегию разработки ПО нескольких преимуществ. Отсутствует возможность гибкой адаптации программной архитектуры к изменяющимся требованиям. Обнаруживается медленная реакция на запрос об изменении программного продукта. Становится трудно устранить ошибки, наследуемые от завершенных этапов.

В соответствии с тем, что предлагаемая ИТ подразумевает использование графовой модели программной архитектуры с целью в более сжатые сроки выполнить формирование измененной части программной архитектуры и более гибко изменить функционал программы, акцент делается на итеративные модели ЖЦ ПО. Такой выбор осуществляется также на основании изменяемых во времени требований, формируемых в процессе тестовой эксплуатации и внедрения программы.

В целях автоматизации процесса структурного синтеза программной архитектуры используются программные спецификации, что выявляет схожесть с процессами разработки ПО через тестирование TTD (Test Driven Development) [14] и BDD (Behavior Driven Development) – как развитие TTD с ориентацией на спецификации поведения и конкретную ПрО, для которой выполняется разработка программы [104]. Спецификации, используемые в этих технологиях, описывают то, как должна себя вести программа (в частности,

BDD). И тестирование, являющееся ключевым шагом в этих методологиях, основывается на информации о функциональности, предоставляемой этими документами. Причем для BDD свойственно отказываться от низкоуровневых модульных тестов и переходить на спецификации поведения более высокого уровня для компонент ПО [154]. Подход, основанный на TDD, применяется для небольших систем для проверки низкоуровневых компонент (существует одна корреляция между «поведением» и компонентой) и базируется на модульных тестах (или Unit-тестах). Подход, основанный на BDD, включает в себя идеи разработанных и существовавших ранее технологий в различных вариантах, например, разработки через тестирование TDD и таких подходов как «Спецификация по примеру» (Specification by Example), разработка на основе приемочных тестов (Acceptance-Test-Driven Development или Acceptance Test-Driven Planning). Такие технологии используют свою модель ЖЦ разработки ПО [154] (рис. 2.4).

Рисунок 2.4 – ЖЦ разработки ПО для TDD и BDD подходов

Такая подробная схема сворачивается в упрощенный трехфазный цикл, который позволяет использовать спецификацию поведения для технологии BDD:

1. Написание теста, вызывающего ошибку.
2. Изменения в основном коде, позволяющем пройти данный тест.
3. Рефакторинг основного кода с учетом обнаруженных ошибок.

BDD подход позволяет более наглядно с ориентацией на естественный язык представить требования к компонентам ПО, не теряя возможность

автоматизированного анализа содержания спецификаций. В связи с этим такая методика обладает большим преимуществом в быстрой и эффективной разработке надежного ПО. Отмечая схожесть с указанными подходами, следует указать на разницу между значением и форматом спецификаций, применяемых в BDD или TDD, и спецификациями, используемыми в предложенной ИТ. В последнем случае документ, специфицирующий программный модуль определяет только те данные, которые необходимы для вызова функций и передачи параметров на входе и получения их значений на выходе модуля. Тем самым устанавливается информационная зависимость меду частями программы, последовательность их выполнения и определяются функции программы, которые можно исполнять независимо друг от друга. Такие спецификации позволяют на основании графовой модели программной архитектуры ИС прийти к ЯПФ, что дает возможность хранения в формализованном виде свойств программных модулей по отношению к ярусам, а так же информации об адекватности графовой модели относительно модели ПрО [93]. Поэтому использование таких программных спецификаций в рамках предложенной ИТ не является альтернативой технологиям BDD или TDD, но может служить дополняющим их методом в том случае, если разработчик выбирает соответствующий ЖЦ и стратегию разработки ПО.

Исходную информацию для структурного синтеза можно получить из функциональных моделей ПрО (рис. 2.5), полученных на этапе структурно-системного проектирования и описывающих последовательность выполнения действий, определяющих внешние сущности, функции и потоки информации внутри и вне системы. Выделение функций и информационных потоков позволяет сформировать набор программных модулей, реализующих данные функции и сформировать спецификации требований к входной и выходной информации уже на этапе объектно-ориентированного проектирования. В зависимости от типа методологии могут быть применены UFO или UML модели, поскольку возможно не только непосредственное использование UFO представления, но и выполнение преобразования к стандарту UML (диаграммы классов) и обратно [16].

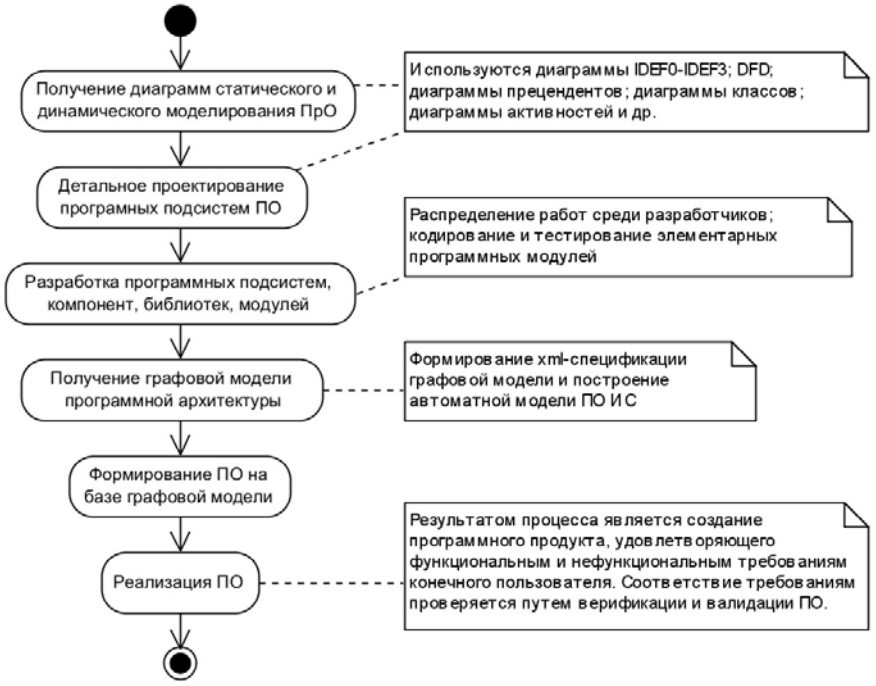


Рисунок 2.5 – Диаграмма активностей. Место графовой модели программной архитектуры ИС

Для моделирования статической структуры классов системы и связей между ними могут быть использованы такие UML-диаграммы как диаграммы классов, диаграммы пакетов и диаграммы компонентов. Для графовой модели программной архитектуры формируется файл спецификации формата xml.

Анализ существующих методов структурного синтеза программной архитектуры ИС показал возможность применения графовых моделей для задачи разработки и конфигурирования программной архитектуры на этапах структурно-системного проектирования и объектно-ориентированного проектирования.

Далее рассмотрены вопросы формализации базовой модели с помощью графо-аналитического метода.

2.2 Разработка ярусно-параллельной графовой модели программной архитектуры информационной системы

Графовая модель программной архитектуры являются удобным представлением, необходимым для формирования программного продукта с заданными свойствами. Представление является кроссплатформенным и

позволяет использовать объектно-ориентированные языки программирования для разработки ПО. В целях получения гибкой программной архитектуры ИС, необходимой для сокращения временных и трудовых затрат на разработку, поддержку и эксплуатацию программы, графовая модель формируется с избыточной для заданной ПрО функциональностью и обеспечивает гибкое конфигурирование системы в условиях эволюционно изменяющихся требований.

Для построения модели требуются данные спецификаций требований к конечному программному продукту, информация, получаемая на этапах предпроектных исследований, существующие архитектурные решения, паттерны. При этом подразумевается наличие существующих программных модулей, и модели ПрО.

Графовая модель представляет собой совокупность вершин, которым сопоставляются элементарные программные модули, выполняемые последовательно или независимо друг от друга. В этой связи граф, представляющий данную модель, является ориентированным и приводится к ЯПФ. Ориентированные дуги между вершинами графа определяют тип связей по данным и описываются парой множеств характеристик программных модулей.

Для формирования общего вида графовой модели сооставляющие её элементы определяются следующим образом. Пусть задан ориентированный граф в ярусно-параллельной форме вида:

$$G = F(G_{исх}), \tag{2.1}$$

где $F(G_{исх})$ – определяет множество операций над исходным графом, которые позволяют привести его к ЯПФ с супервершинами, а $G_{исх}$ является исходным графом вида:

$$G_{исх} = \langle V_{исх}, X_{исх} \rangle, \tag{2.2}$$

где $V_{исх}$ – множество вершин v, которым сопоставляются программные модули (или же, в общем понимании, ВП);

$X_{исх}$ – множество ориентированных дуг $x_{ij} = (v_i, v_j)$ графа $G_{исх}$,определяющих зависимость по данным между программными модулями.

Формирование множества вершин и дуг графа (2.2) происходит на основе выделения в ПрО соответствующих функциональных подсистем объекта исследования и установления потоков данных между ними. Связи между вершинами графовой модели характеризуются информационными потоками, данными, которыми обмениваются программные модули между собой. Часть этих данных является константами, служащими исходной информацией для начала работы модуля. Другая часть является изменяемыми массивами, структурами и переменными, которые претерпевают свои изменения в ходе работы программы. Поэтому для каждой i-ой вершины графа (2.2) выделены два вектора данных:

$$D_{in}{}^{i} = \left\langle d_1^{in}, d_2^{in}, ... d_k^{in} \right\rangle, \tag{2.3}$$

– вектор входных данных, неизменяемых в процессе работы и

$$D_{out}{}^{i} = \left\langle d_1^{out}, d_2^{out}, ... d_k^{out} \right\rangle, \tag{2.4}$$

– вектор выходных данных, которые были модифицированы в процессе работы программного модуля.

Сам граф (2.2) задается матрицей смежности:

$$A = \left\| a_{ij} \right\| = \begin{pmatrix} 0 & a_{12} & a_{13} & ... a_{1n} \\ a_{21} & 0 & a_{23} & ... a_{2n} \\ ... & ... & ... & ... \; ... \\ a_{n1} & a_{n2} & a_{n3} & ... a_{nn} \end{pmatrix}, \tag{2.5}$$

где элемент матрицы a_{ij} принимает два значения 1 и 0, в зависимости от наличия или отсутствия ориентированной дуги $x_{ij} \in X$ между вершинами $v_i, v_j \in V$. ВП для графовой модели программной архитектуры ИС являются порождением заданных программных модулей, которые выполняют функциональные задачи ИС [84].

В случае изменяемых требований, определяющих модификацию программной архитектуры и расширение функциональности программного продукта для графа G (2.1) формируется дополнительное подмножество новых программных модулей $V^H = \{ v_i^H \}$, включаемых в программную архитектуру, и подмножество модифицируемых модулей $V^M = \{ v_i^M \}$, чей функционал будет изменен каким-либо образом. Для получения новой версии программной архитектуры необходимо из графа G (2.1) исключить подмножество вершин $V^3 \subseteq V_{исх}$, которые требуется заместить на подмножество V^M, то есть необходимо убрать программные модули устаревшей версии. Такую же операцию проделать с дугами $X^3 \subseteq X_{исх}$. Это позволяет определить постоянную часть графа G , задаваемую как $G^П = \langle V^П, X^П \rangle, G^П \subseteq G$, где $V^П = V_{исх} \setminus V^3$, а $X^П = X_{исх} \setminus X^3$.

Определяемые связи между добавляемыми и изменяемыми программными модулями позволяют задать подмножества дуг $X^H = \{ x_p^H \}$ и $X^M = \{ x_q^M \}$ соответственно, что, в свою очередь, позволяет выделить подграфы $G_1^H = \langle V_1^H, X_1^H \rangle$, $G_2^H = \langle V_2^H, X_2^H \rangle$,…, $G_n^H = \langle V_n^H, X_n^H \rangle$ и $G_1^M = \langle V_1^M, X_1^M \rangle$, $G_2^M = \langle V_2^M, X_2^M \rangle$,…, $G_m^M = \langle V_m^M, X_m^M \rangle$. Объединяя указанные подграфы, получаем новую версию программной архитектуры ИС в виде графа:

$$G^B = \left(\bigcup_{i=1}^{n} G_i^M \right) \cup \left(\bigcup_{j=1}^{m} G_j^H \right) \cup G^П \qquad (2.6)$$

Для графа (2.1) задается множество характеристик вершин VC, которое определяется таким образом:

$$VC = \left\{ m_1^{VC}, m_2^{VC}, ..., m_n^{VC} \right\}, \tag{2.7}$$

где $m_i^{VC} = \langle isSubG, Comp \rangle$, $i = \overline{1,n}$, является подмножеством, состоящим из двух элементов: *isSubG* – логическая переменная, которая определяет наличие вложенного подграфа для супервершины; подмножество Comp – перечисляет номера вершин, принадлежащих компоненте сильной связности. Супервершины образовываются после применения метода объединения вершин ярусно-параллельной графовой модели и заменяют собой часть графа $G_{исх}$ (2.2), а компоненты сильной связности определяются на графе при помощи алгоритма Косарайю. Если задать $G_i^{SC} = \langle V^{SC}, X^{SC} \rangle$ (где V^{SC} – множество вершин, а X^{SC} – множество ориентированных дуг) как *i*-ую компоненту сильной связности $G_i^{SC} \subset G_{исх}$, обнаруживаемую при одноименном поиске, то тогда множество Comp будет совпадать с множеством V^{SC}, то есть Comp=V^{SC}, где $V^{SC} \subset V_{исх}$.

Множество функций PM, сопоставляемых с вершинами, определяется так:

$$PM = \left\{ p_1^{PM}, p_2^{PM}, ..., p_q^{PM} \right\}, \tag{2.8}$$

где

$$p_i^{PM} = \langle UID, PName, PMName, PUPath, PDes, SPath \rangle \tag{2.9}$$

– подмножество элементов, необходимых для описания программных компонент или модулей, которые эти функции выполняют. Для этого определяем *UID* как уникальный идентификатор вычислительного процесса; *PName* – имя процесса, используемое компоновщиком, при определении логической связи

между вершиной графовой модели и программным модулем; *PMName* – реальное имя программного модуля (или библиотеки); *PUPath* – физический путь к файлу программного модуля; *SPath* – физический путь к файлу спецификации, определяющей основные характеристики модуля, и является ссылкой на спецификацию модуля; *PDes* – дополнительные сведения об элементе. Атрибут *PName* является синтаксическим уникальным именем программного модуля, а *PMName* – реальным физическим именем (семантическим) процесса или модуля, на которое могут ссылаться несколько синтаксических имен.

На основании перечисленных элементов, графовая модель программной архитектуры ИС задается выражением вида:

$$M = \left\langle V_{исх}, X_{исх}, D_{in}, D_{out}, V^{H}, X^{H}, V^{M}, X^{M}, VC, PM \right\rangle, \qquad (2.10)$$

где $V_{исх}$ – множество вершин v, которым сопоставляются программные модули;

$X_{исх}$ – множество ориентированных дуг $x_{ij} = (v_i, v_j)$ графа $G_{исх}$ (2.2);

D_{in}, D_{out} – множества входных и выходных данных программных модулей (2.3-2.4);

$V^{H} = \{ v_i^{H} \}$ – подмножество вершин, сопоставляемых новым программным модулям;

X^{H} – подмножество новых связей по данным $x_{ij}^{н} = (v_i^{н}, v_j^{н})$;

$V^{M} = \{ v_i^{M} \}$ – подмножество модифицированных модулей $V^{M} = \{ v_i^{м} \}$, причем подмножества V^{H}, V^{M} используются в случае изменяемых требований конечного пользователя, и определяют модификацию архитектуры;

X^{M} – подмножество модифицированных связей по данным $x_{ij}^{м} = (v_i^{м}, v_j^{м})$;

VC – множество характеристик вершин $V_{исх}$ (2.7);

PM – множество функций, сопоставляемых с вершинами $V_{исх}$ (2.8).

Полученная графовая модель обладает следующими основными свойствами. На ней отсутствуют висячие вершины, для которых выполняется условие:

$$\forall v \in V, \{ V' \subseteq V : |V'| > 1 : \exists v_i \in V' : deg^-(v_i) = 0 \}. \quad (2.11)$$

Для модели отсутствуют изолированные вершины, для которых выполняется условие:

$$\forall v \in V, \{ \exists v_i \in V \,|\, deg^+(v_i) = deg^-(v_i) = 0 \} \quad (2.12)$$

Для нее отсутствуют парные дуги:

$$\forall i, j \in \mathrm{N}, \{ v_i, v_j \in V \mid \exists! x_{ij} \in X \} \quad (2.13)$$

И отсутствуют вершины с петлями:

$$\forall i \in \mathrm{N}, \{ v_i \in V \mid x_{ii} \notin X \} \quad (2.14)$$

Кроме этого, для ациклической графовой модели все вершины, для которых задан номер с учетом топологической сортировки и выполняется условие

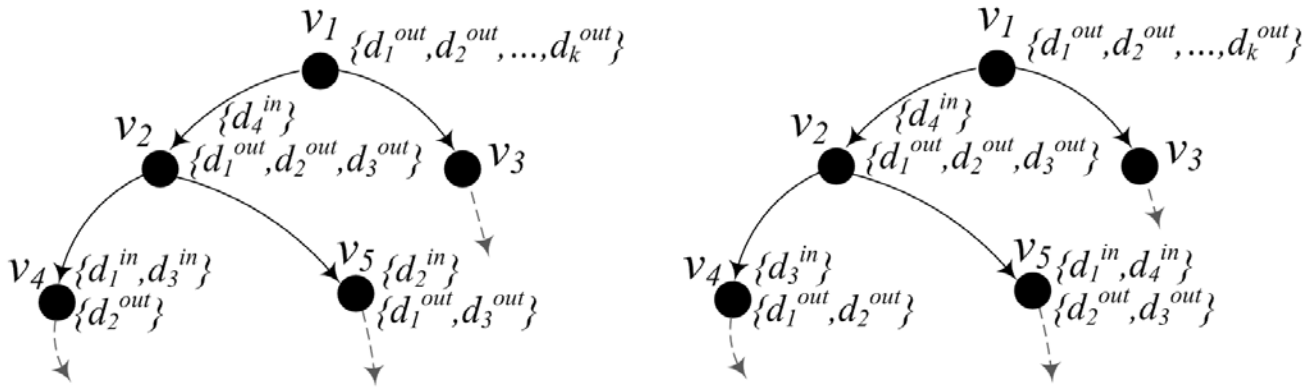
$$\begin{aligned} &\forall i, j \in \mathrm{N}, i < j, \{ \exists v_i, v_j \in V : \mu[v_i, v_j] \mid deg^-(v_i) = deg^+(v_i) = \\ &= deg^-(v_{i+1}) = deg^+(v_{i+1}) = \ldots = deg^-(v_j) = deg^+(v_j) = 1 \}, \end{aligned} \quad (2.15)$$

где $\mu[v_i, v_j]$ – маршрут от вершины v_i к вершине v_j, объединяются в одну супервершину.

Назначением предложенной графовой модели является описание программной архитектуры ИС, что позволяет оценить её до непосредственного построения ПО и обеспечить динамическое конфигурирование рабочих мест

пользователей. Это является преимуществом предложенной ярусно-параллельной графовой модели. Между множествами данных, которыми оперируют программные модули, могут быть разного рода зависимости. Их классификацию следует рассмотреть на примере.

Рассмотрим фрагмент графа вида (2.1), состоящего из пяти вершин (рис.2.6, а и б). Первая вершина графовой модели является фиктивной по своей сути и поэтому имеет только вектор выходных данных (2.4) (рис.2.6).



а) б)

Рисунок 2.6 – Пример маркировки вершин графовой модели векторами входных и выходных данных

При этом в соответствие с ЯПФ можно установить для первого уровня только первую начальную вершину, для второго – вершины v_2 и v_3, для третьего – вершины v_4 и v_5. Если рассматривать информационные зависимости вершин, то можно получить между ними различные отношения. Так, например, в одном случае (рис. 2.6, а) вершины v_4 и v_5 используют различные входные наборы переменных, и $D_4^{in} \subseteq D_2^{out}$, $D_4^{out} \subseteq D_2^{out}$, $D_5^{in} \subseteq D_2^{out}$, $D_5^{out} \subseteq D_2^{out}$. А в другом случае (рис. 2.6, б) также справедливо выражение $D_5^{in} \cup D_2^{in} \neq \varnothing$.

То есть для следующей после v_i вершины v_{i+1}, при условии существования ориентированной дуги $x_{i,i+1} = (v_i, v_{i+1})$, существует входной набор данных $D_{i+1}^{in} \subseteq D_i^{in} \cup D_i^{out}$. Но выходное множество данных четвертой вершины является входным множеством для пятой вершины. Если модуль, соответствующий вершине v_4 закончит свою работу до запуска модуля, соответствующего вершине

v_5, то значение на входе пятой вершины будет отличаться от того, какое могло бы быть, если бы на вход пришли данные из вершины v_2 непосредственно до окончания работы вершины v_4 и применения изменений выходных переменных. Для вершин, которые находятся на одном ярусе возможно одно из трех состояний: 1) $D_4^{in} \cup D_5^{out} \neq \varnothing$; 2) $D_4^{out} \cup D_5^{\text{in}} \neq \varnothing$ или 3) $D_4^{out} \cup D_5^{\text{int}} = \varnothing$ и $D_4^{in} \cup D_5^{out} = \varnothing$. Последнее состояние характерно для вершин, независящих друг от друга по данным. В более общем случае эти выражения будут записаны следующим образом. Пусть задан ациклический граф *G* вида (2.1), приведенный к ЯПФ. Соответственно для такого графа будет существовать *L* ярусов. Обозначим подмножество вершин, находящихся на *r*-ом ярусе как $V_r \subset V$, где $r = \overline{1, J}$ – номер текущего яруса ЯПФ графа. Тогда для всех $v_i \in V_r$ и $v_k \in V_r$ справедливо выражение:

$$\forall v_i, v_k \in V_r, x_{ik} = (v_i, v_k) \notin X \ . \tag{2.16}$$

И выполняется одно из следующих условий:

$$D_i^{in} \cup D_k^{out} \neq \varnothing, \tag{2.17}$$

$$D_i^{out} \cup D_k^{in} \neq \varnothing, \tag{2.18}$$

$$\begin{cases} D_i^{in} \cup D_k^{out} = \varnothing \\ D_i^{out} \cup D_k^{in} = \varnothing \end{cases}. \tag{2.19}$$

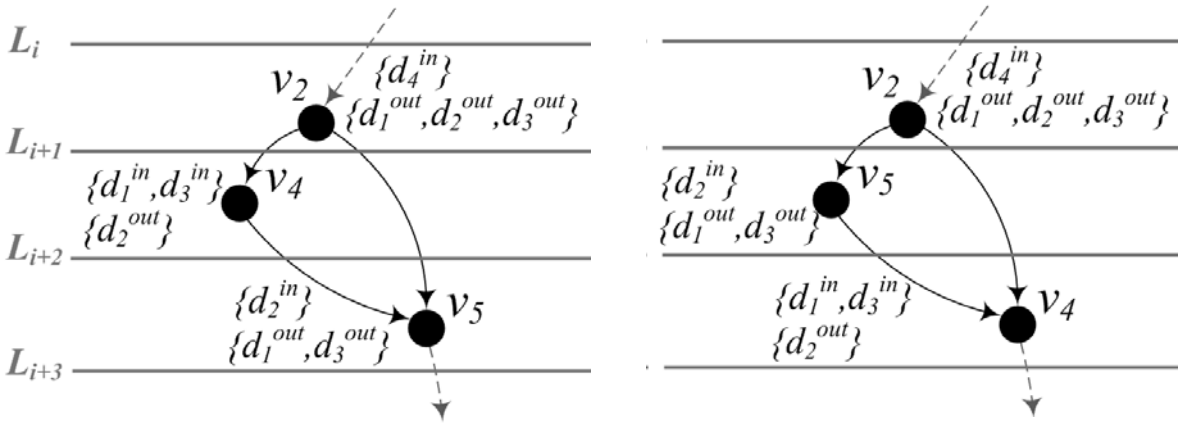
Отношения между входными и выходными данными вершин определяются согласно с принятой классификацией типов отношений между регионами параллельного или распределенного программного кода (табл.2.1) [11].

Если при анализе графа обнаруживаются свойства (2.17) или (2.18), то это говорит об упорядоченности или консервативности.

Таблица 2.1 – Типы отношений между параллельными фрагментами программы

Тип	Описание
Одновременность	Программный модуль v_i и v_k могут выполняться одновременно
Упорядоченность	Программный модуль v_i должен выбрать все, что ему требуется, прежде, чем модуль v_k запишет свои результаты
Консервативность	Программный модуль v_i должен записать свои результаты раньше, чем модуль v_k
Последовательность	Программный модуль v_i должен быть завершен до начала работы модуля v_k

Если же обнаруживается соблюдение свойства (2.19), то в этом случае вершины графовой модели действительно могут быть размещены на одном ярусе и назначаться исполнителем ВП к независимому и параллельному запуску. Для этого добавляется новый ярус, позволяющий размежевать эти вершины. Например, для графа G (рис. 2.6) вершины v_4 и v_5 распределяются по двум отдельным ярусам в случае упорядоченного исполнения вершины v_4 по отношению к вершине v_5 (рис.2.7, а) или в случае консервативного исполнения (рис. 2.7, б).



а) б)

Рисунок 2.7 – Пример размещения вершин графовой модели по ярусам ЯПФ модели в зависимости от набора входных и выходных данных

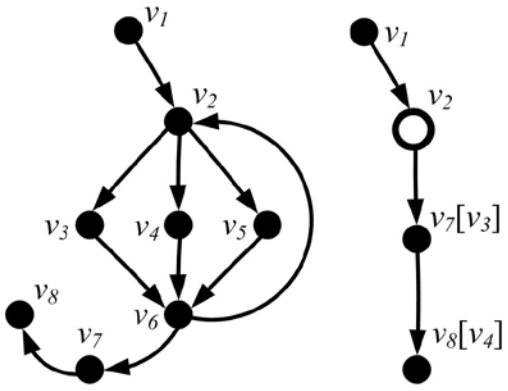
В дальнейшем работа осуществляется с графом, на котором проведена топологическая сортировка. Такая сортировка осуществима для графа (2.1) после того, как на нем убраны все циклы, и выполняется следующим образом [37]. На графе задается такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Для этого используется алгоритм поиска *в глубину* (англ. *Depth-first search, DFS*). В алгоритме поиска используется раскраска вершин тремя цветами. Изначально все вершины белые. Когда вершина обнаружена, она окрашивается в серый цвет. Когда просмотрен список всех смежных с ней вершин, вершина помечается черным цветом. Запускаем обход в глубину, и когда вершина обработана, заносим ее в стек. По окончании обхода вершины достаются из стека. Новые номера присваиваются в порядке вытаскивания из стека. Поиск компонент сильной связности на графе позволяет выделить подмножества вершин, которые на следующем шаге будут объединены в супервершину для приведения графовой модели к ЯПФ. Свертывание компоненты сильной связности в супервершину описывается алгоритмом конденсации графа [24]. На заданном графе выделяются области с циклами $\mu[v_b, v_e]$, где v_b – вершина начала цикла, v_e – его конец. Далее все вершины данного цикла исключаются из графа и заменяются одной вершиной v_p, для которой индекс p определяется как минимальный номер вершины, которая входит в цикл $\mu[v_b, v_e]$. У всех остальных вершин индекс заменяется по правилу: если $ind > p$, тогда $Nind = ind - p + 1$, где ind – текущий индекс вершины, $Nind$ – новый индекс вершины, p – номер вершины, соответствующей исключенному из графа циклу. По этому принципу исходный граф (2.2) преобразуется в конденсированный граф, называемый графом Герца [5].

Существуют различные способы поиска циклов, среди которых наиболее эффективными называются алгоритмы Косарайю, Габова и Тарьяна [72, 156]. Эти алгоритмы позволяют решить задачу поиска за линейное время и пространство памяти и могут быть использованы в одинаковых условиях. В данном случае взят алгоритм Косарайю.

ЯПФ наделяет графовую модель программной архитектуры ИС потенциалом организации параллельных вычислений [67]. Потенциал в дальнейшем может быть реализован в применении специальных технологий распределения ВП на архитектурах ВС, таких как компьютерные гетерогенные кластеры, многопроцессорные системы классов SMP или NUMA.

Покажем на примере, как происходит приведение графовой модели к ациклическому виду. Рассмотрим технологические процессы подготовки производства. Учитывая то, что технолог начинает работу с технологической спецификацией только после наличия конструкторской, получив извещение из конструкторского отдела, определим основные функции: 1) ввод конструкторской спецификации на изделие; 2) ввод технологической спецификации на изделие; 3) ввод технологического процесса; 4) ввод технологического маршрута изделия; 5) ввод норм для технологического процесса; 6) расчет норм для технологического процесса; 7) согласование технологической и конструкторской спецификации; 8) формирование технологического состава.

В этой последовательности присутствует циклический процесс: при неудовлетворительных значениях норм на изготовление изделий требуется внесение изменений в технологический маршрут (рис. 2.8, а).



а) б)

Рисунок 2.8 – Графовая модель технологического процесса подготовки производства: а) исходная модель; б) модель, приведенная к ациклической форме с использованием алгоритма конденсации

При конденсации вершины 2, 3, 4, 5 и 6 могут быть объединены в одну супервершину под названием «формирование технологической спецификации» (рис. 2.8, б). Данная супервершина представляет собой вложенный подграф с циклами, который может быть реализован отдельным программным модулем. Номера в квадратных скобках – это те номера вершин, которые присваиваются им после срабатывания алгоритма топологической сортировки. Наличие дополнительной нумерации и необходимость сохранять оба номера обусловливают использование семантического и синтаксического имени для программного модуля. Семантическое имя применяется для нужд самой системы, тогда как синтаксическое имя – это реальное, исходное имя.

Отметим преимущества ЯПФ графовой модели. Каноническая форма позволяет определить последовательность ВП во времени так, что можно выделить такой ВП, который может быть активным только, если все предшествующие процессы завершат свою работу. ЯПФ позволяет определить ВП, не имеющих информационных зависимостей друг от друга и потому выполняемых параллельно и размещаемых на одном ярусе. Такие свойства являются важными, так как при верификации возможно использование наименее трудозатратного метода. Поэтому при анализе средств верификации выбор делается в пользу темпоральной логики LTL (linear temporal logic) по сравнению с CTL (computational tree logic) [46, 50, 53].

Выполнение свойств (2.12) и (2.13) для графовой модели говорит о топологических некорректностях, приводящих к неправильной работе ВП и, иногда, принципиальной невозможности выполнения вычислений. Свойства (2.13) и (2.14) позволяют упростить программную архитектуру. Они выполнимы для графовой модели, поскольку в каждый момент времени рассматривается только один тип связей – по данным или по управлению. Необходимость объединения вершин в супервершину в случае выполнения условия (2.15) не так очевидна. Поэтому решение принимается на основе модифицированного алгоритма конденсации графовой модели [74], положенного в основу метода объединения вершин ярусно-параллельной графовой модели программной архитектуры ИС с

учетом специальных критериев, позволяющего снизить структурную сложность программной архитектуры. Рассмотрим подробнее методы снижения структруной и функциональной сложности ПО.

2.3 Разработка методов снижения структурной и функциональной сложности программной архитектуры информационной системы

2.3.1 Метод объединения вершин графовой ярусно-параллельной модели программной архитектуры информационной системы

Предложенный метод использует оценку сложности и связности программных модулей, и предназначен для приведения графовой модели к виду (2.10). Метод обеспечивает выполнимость свойств графовой модели (2.11)-(2.15) и позволяет сократить временные затраты на разработку системы путем объединения вершин в супер-вершины на базе модифицированного алгоритма Косарайю.

Данный алгоритм дополнительно обрабатывает графовую модель на основе комплексного критерия оценки эффективности программной архитектуры:

$$K_C = \frac{K_{общ}}{K'_{общ}} \cdot \frac{K'_{слож}}{K_{слож}}, \qquad (2.20)$$

где $K'_{слож}$ и $K'_{общ}$ – значения среднего коэффициента сложности и обобщенного критерия полученных после оптимизации графовой структуры;

$K_{слож}$ и $K_{общ}$ – значения критериев до оптимизации.

Чем ближе значение критерия к нулю, тем эффективнее программная архитектура. Если значение превысило единицу – улучшение не произошло. Средний коэффициент сложности для всей программной архитектуры имеет вид:

$$K_{сложн} = \frac{1}{n} \cdot \sum_{i=1}^{n} C(i), \qquad (2.21)$$

где $C(i)$ – это коэффициент системной сложности;

n – общее количество программных модулей.

Обобщенный критерий позволяет оценить программную архитектуру, основываясь на сравнении топологии графа с топологией дерева, так как такая топология является оптимальной по критерию минимизации количества связей между программными модулями [59], и задается выражением:

$$K_{общ} = \frac{n \cdot (n-1) - 2 \cdot e_r}{n^2 \cdot (n-1) \cdot (n-2)} \cdot \sum_{i=1}^{n} K_{CP}(i) \cdot \sum_{i=1}^{n} K_{CH}(i), \qquad (2.22)$$

где $K_{CH}(i) = \frac{1}{10} \cdot Ch(i)$ и $K_{CP}(i) = \frac{1}{8} \cdot (9 - Cp(i))$ – нормированные коэффициенты связности и сцепления модуля m_i соответственно.

Величина *Ch(i)* определяется в соответствии со значимостью связности (табл.2.2), а *Cp(i)* выбирается относительно значимости сцепления модулей (табл.2.3).

Таблица 2.2 – Тип связности модулей [59]

Значение *Ch*	Связность	Пояснение
0	по совпадению	В модуле отсутствуют явно выраженные связи
1	логическая	Части модуля объединены по функциональному подобию
3	временная	Части модуля не связаны, но необходимы в один и тот же период работы системы
5	процедурная	Части модуля связаны порядком выполняемых действий
7	коммуникативная	Части модуля связаны по данным (работают с одной и той же структурой данных)
9	информационная	Выходные данные одной части используются как входные данные в другой части модуля
10	функциональная	Части модуля вместе реализуют одну функцию

Таблица 2.3 – Тип сцепления модулей [59]

Значение *Cp*	Сцепление	Пояснение
1	по данным	Модуль А вызывает модуль В с использованием простых элементов данных
3	по образцу	В качестве параметров используются структуры данных
4	по управлению	Модуль А явно управляет функционированием модуля В
5	по внешним ссылкам	Модули А и В ссылаются на один и тот же глобальный элемент данных
7	по общей области	Модули разделяют одну и ту же глобальную структуру данных
9	по содержанию	Модуль А прямо ссылается на содержание модуля В (не через точку входа)

Коэффициент системной сложности $C(i)$ расчитывается по метрике Д. Карда и Р. Гласса:

$$C(i) = S(i) + D(i), \tag{2.23}$$

где $S(i)$ – это структурная сложность i-го модуля, которая вычисляется по формуле:

$$S(i) = (sfan_{OUT(i)} + ifan_{OUT(i)})^2, \tag{2.24}$$

а $D(i)$ – это сложность по данным i-го модуля:

$$D(i) = \frac{v(i)}{sfan_{OUT(i)} + ifan_{OUT(i)} + 1}, \tag{2.25}$$

где $v(i)$ – количество входных и выходных переменных i-го модуля;

$sfan_{OUT(i)}$ – количество модулей, которые вызываются i-м модулем;

$ifan_{OUT(i)}$ – это количество элементов и структур данных, которые обновляются i-м модулем.

Чем ближе значение $K_{общ}$ к единице, тем лучше структура программы.

Обозначим h_{ij} – как характеристику необходимости объединения вершин v_i и v_j в одну супервершину, если $h_{ij}=1$. И $h_{ij}=0$, если иначе:

$$h_{ij}=\begin{bmatrix}1, K_C<1\\0, K_C\geq 1\end{bmatrix} \tag{2.26}$$

На основании описанных выше критериев формируется алгоритм обработки графовой модели (рис. 2.9).

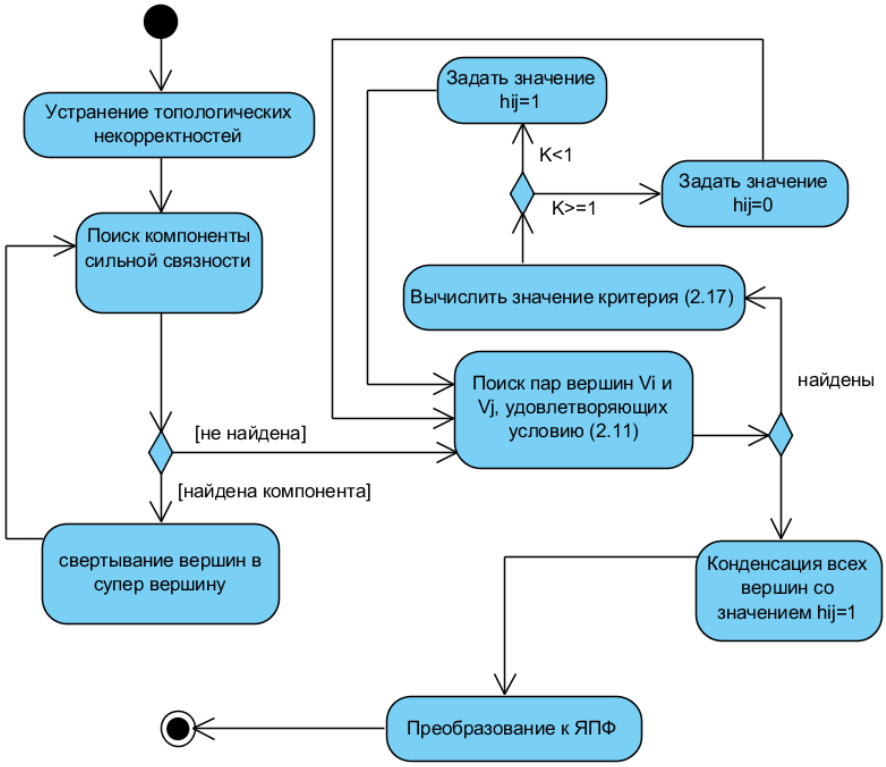


Рисунок 2.9 –Конденсация графовой модели программной архитектуры

Предложенный метод объединения вершин графовой ярусно-параллельной модели программной архитектуры ИС включает такие этапы.

Этап 1. Устранение топологических некорректностей графовой модели: не должно быть висячих (2.11) и изолированных (2.12) вершин.

Этап 2. Получение для каждого *i*-го программного модуля количества вызываемых модулей (2.24) (параметр $sfan_{OUT}$), количества элементов и структур

данных, обновляемых *i*-м модулем (2.24) **(**параметр $ifan_{OUT}$ **)** и значения коэффициента системной сложности $C(i)$ (2.23).

Этап 3. Расчет среднего коэффициета сложности $K_{сложн}$ (2.21).

Этап 4. Расчет значения критериев $K_{общ}$ (2.22) и $K_{сложн}$ (2.21), получение значения комплексного критерия оценки эффективности программной архитектуры K_C (2.20) для заданной версии требований к конфигурации системы.

Этап 5. Поиск компоненты сильной связности для заданного графа на основе алгоритма Косарайю и объединение вершин найденной компоненты в супервершину.

Этап 6. Поиск пары вершин по условию (2.15) после того, как на графе не осталось ни одной компоненты сильной связности (для любых двух вершин v_i и v_j невозможно найти двух одновременно существующих ориентированных путей $\mu[v_i, v_j]$ и $\mu[v_j, v_i]$).

Этап 7. Вычисление значения комплексного критерия оценки эффективности программной архитектуры (2.20) для найденной пары вершин.

Этап 8. Объединение пары вершин в одну супервершину, если значение комплексного критерия (2.20) строго меньше единицы, и повторение с этапа 3 до тех пор, пока значение комплексного критерия не будет удовлетворять конфигурационным требованиям. Завершение метода.

Оценка временной сложности предложенного метода опирается на алгоритм Косарайю**.** Так как число вершин графовой модели, которые объединяются в супервершины на основе показателей сцепления и связности программных модулей незначительно и им можно пренебречь, то оценка временной сложности метода соответствует оценке алгоритма Косарайю [72] и равна *O(n)*.

Используя приведенный пример ПрО (рис. 2.8), покажем применение критериев (2.20) – (2.25) на практике. Для программного модуля определяется набор входных и выходных параметров *v(i)*, количество элементов и структур данных, с которыми оперирует модуль и которые обновляет – $ifan_{OUT(i)}$, а также

количество модулей, вызываемые *i*-ым программным модулем – $sfan_{OUT(i)}$. Каждый программный модуль в качестве входного параметра получает логическую переменную, разрешающую запуск функций, и возвращает логическую переменную, соответствующую выполненной функции. Для модулей v_1, v_2 и v_3 передается идентификатор спецификации, для модуля v_4 – идентификатор технологического процесса. Для модулей v_5 и v_6 – идентификатор спецификации, а для модуля v_7 – два параметра, содержащие идентификаторы конструкторской и технологической спецификаций. Модуль v_8 работает с данными, содержащимися в таблицах базы данных. Структуры данных должны быть обновлены *i*-м программным модулем. Для модулей v_1 и v_2 – это реестры конструкторских и технологических спецификаций. Модуль v_3 работает только с реестром технологических спецификаций и справочником технологических процессов. Модуль v_4 обновляет данные технологических процессов. Модули v_5 и v_6 обрабатывают данные таблицы норм на изделие. Модуль v_7 обновляет данные двух реестров спецификаций и данные составов изделий. Модуль v_8 обновляет данные состава изделий. На основании такого описания сформируем значения параметров $v(i)$, $sfan_{OUT(i)}$ и $ifan_{OUT(i)}$ (табл. 2.4). Полученные данные используются для расчета коэффициента (2.24). Приведем полученные значения (табл. 2.5).

Таблица 2.4 – Числовые значения параметров коэффициента системной сложности

Программный модуль	$v(i)$	$sfan_{OUT(i)}$	$ifan_{OUT(i)}$
v_1	3	1	1
v_2	3	3	1
v_3	3	1	2
v_4	4	1	1
v_5	3	1	1
v_6	3	2	1
v_7	4	1	3
v_8	2	0	1

Таблица 2.5 – Значения коэффициента системной сложности

Программный модуль	*S(i)*	*D(i)*	*C(i)*
v_1	4	1	5
v_2	16	0,6	16,6
v_3	9	0,75	9,75
v_4	4	1,33	5,33
v_5	4	1	5
v_6	9	0,75	9,75
v_7	16	0,8	16,8
v_8	1	1	2

Исходя из расчетных данных для коэффициента системной сложности, получаем среднее значение сложности на основе коэффициента (2.21):

$$K_{сложн} = \frac{70,23}{8} \approx 8,78.$$

Теперь получим значение обобщенного коэффициента (2.22).

Для этого определим для модулей v_1 , v_2 и v_7 коммуникативную связность (величина *Ch*), так как эти программные модули взаимодействуют непосредственно с таблицами базы данных и их работа выполняется после обновления информации.

Для модулей v_3, v_4, v_5, v_6 и v_8 – информационную связность, поскольку выходные данные одних функций используются как входные данные для других в рамках одного и того же программного модуля.

Также тип сцепления модулей (величина *Cp*) для всех вершин за исключением v_6 определяется как сцепление по общей области – этому способствует архитектура ПС, включающего в себя базу данных.

Для модуля v_6 тип сцепления задается по управлению (табл. 2.6), так как наличие необходимой информации по технологическому процессу определяет возможность задания маршрута на основе технологии изготовления продукции.

Таблица 2.6 – Значения нормированных коэффициентов связности и сцепления программных модулей

Программный модуль	*Ch(i)*	*Cp(i)*	$K_{CH}(i)$	$K_{CP}(i)$
v_1	7	7	0,7	0,25
v_2	7	7	0,7	0,25
v_3	9	7	0,9	0,25
v_4	9	7	0,9	0,25
v_5	9	7	0,9	0,25
v_6	9	4	0,9	0,625
v_7	7	7	0,7	0,625
v_8	9	7	0,9	0,625

Пронормировав значения *Ch(i)* и *Cp(i),* и полагая $n = 8$, а $e_r = 10$ (рис. 2.8, а), рассчитаем обобщенный коэффициент (2.22):

$$K_{общ} = \frac{8 \cdot 7 - 2 \cdot 10}{64 \cdot 7 \cdot 6} \cdot 6{,}6 \cdot 3{,}125 = \frac{742{,}5}{2688} \approx 0{,}276.$$

Для новой графовой модели (рис. 2.8, б) проделываем те же самые операции. При этом учитываем, что для супервершины нет необходимости считать одинаковые параметры, которые являлись входными или выходными для отдельных модулей. Так для программного модуля v_2 получаем только логические переменные запуска и окончания ВП, идентификаторы редактируемых конструкторских и технологических спецификаций и идентификатор технологического процесса: 5 параметров. Аналогичным образом получаем количество элементов и структур данных, с которыми будет работать программный модуль, соответствующий супервершине. То есть получаем количество структур, убирая повторяющиеся для модулей, включенных в данную супервершину (табл. 2.7). На основе значений параметров *v(i), sfan*$_{OUT(i)}$ и *ifan*$_{OUT(i)}$ опять вычисляем значение коэффициента системной сложности (табл. 2.8). В таблицах (2.7) и (2.8) для колонки «программный модуль» в скобках

указывается номер согласно топологической сортировке, проводящейся после операции конденсации графовой модели.

Таблица 2.7 – Числовые значения параметров коэффициента системной сложности

Программный модуль	$v(i)$	$sfan_{OUT(i)}$	$ifan_{OUT(i)}$
v_1	3	1	1
v_2	5	1	4
$v_7(3)$	4	1	3
$v_8(4)$	2	0	1

Таблица 2.8 – Значения коэффициента системной сложности

Программный модуль	$S(i)$	$D(i)$	$C(i)$
v_1	4	1	5
v_2	25	0,83	25,83
$v_7(3)$	16	0,8	16,8
$v_8(4)$	1	1	2

Получаем среднее значение коэффициента системной сложности (2.21):

$$K'_{сложн} = \frac{49,63}{4} \approx 12,41.$$

Следующим шагом также рассчитываем новое значение обобщенного коэффициента (2.24). Для этого полагаем для супервершины v_2 функциональную связность и сцепление по управлению (табл. 2.9).

Таблица 2.9 – Значения нормированных коэффициентов связности и сцепления программных модулей

Программный модуль	$Ch(i)$	$Cp(i)$	$K_{CH}(i)$	$K_{CP}(i)$
v_1	7	7	0,7	0,25
v_2	10	4	1	0,625
$v_7(3)$	7	7	0,7	0,625
$v_8(4)$	9	7	0,9	0,625

Получаем новое значение обобщенного коэффициента (2.22):

$$K'_{общ} = \frac{4 \cdot 3 - 2 \cdot 3}{16 \cdot 3 \cdot 3} \cdot 3{,}3 \cdot 2{,}125 = \frac{42{,}075}{144} \approx 0{,}292.$$

На основании предыдущих и новых значений коэффициентов (2.21) и (2.22) рассчитываем значение комплексного критерия эффективности:

$$K_C = \frac{K_{общ}}{K'_{общ}} \cdot \frac{K'_{слож}}{K_{слож}} = \frac{0{,}276}{0{,}292} \cdot \frac{12{,}41}{8{,}78} \approx 1{,}336.$$

Расчет показывает, что в целом изменения произошли не в лучшую сторону, главным образом, из-за увеличения сложности программы. Однако с точки зрения топологии графовой модели и характеристик программных модулей новое значение обобщенного критерия лучше. На последнем шаге необходимо определиться с вершиной v_7 графовой модели программной архитектуры ИС, идущей сразу после супервершины – следует ли ее также объединить с вершиной v_2 или нет, поскольку вершина v_7 удовлетворяет условию (2.15). Для этого также следует рассчитать новые значения коэффициентов (2.21) и (2.22) и критерия оценки эффективности программной архитектуры (2.20). При объединении вершин v_2 и v_7 получаем значения коэффициентов для новой супервершины (табл. 2.10). Для других оставшихся вершин коэффициенты принимают те же значения.

Таблица 2.10 – Сводные значения коэффициентов при формировании новой супервершины

Программный модуль	$v(2)$	$sfan_{OUT(2)}$	$ifan_{OUT(2)}$	$S(2)$	$D(2)$	$C(2)$	$K_{CH}(2)$	$K_{CP}(2)$
v_2	5	1	3	16	1	17	1	0,625

На основании полученных данных (табл. 2.10) рассчитываем коэффициент сложности (2.21): $K''_{сложн} = \frac{24}{3} = 8$ и обобщенный коэффициент (2.22):

$$K''_{общ} = \frac{3 \cdot 2 - 2 \cdot 2}{9 \cdot 2 \cdot 1} \cdot 2{,}6 \cdot 1{,}5 = \frac{7{,}8}{18} \approx 0{,}43.$$

И, наконец, получаем значение комплексного критерия (2.20):

$$K_C = \frac{K'_{общ}}{K''_{общ}} \cdot \frac{K''_{слож}}{K'_{слож}} = \frac{0{,}276}{0{,}43} \cdot \frac{8}{8{,}78} \approx 0{,}585$$

Значение критерия, меньшее единицы, говорит о возможном объединении вершин в супервершину. В результате получаем упрощенную графовую модель, состоящую всего из трех вершин (рис. 2.10). Предложенный алгоритм реализуется в случае задания формальных спецификаций программных модулей, соотносящихся с вершинами модели.

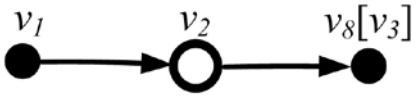


Рисунок 2.10 – Графовая модель после конденсации вершин

Причем такие, спецификации, которые участвуют в формировании программной архитектуры, должны отвечать следующим требованиям:

1) соответствие программной архитектуры аппаратной и операционной среде, их ресурсам и интерфейсам;

2) совместимость программных модулей ПО с другими системами по источникам и потребителям информации;

3) соответствие стандартам структурного построения и интерфейсов комплекса программ, функциональных компонентов и модулей;

4) состав, структура и способы обмена данными между функциональными компонентами и внешней средой ПО.

Для уточнения структуры программного продукта спецификации модулей задаются на формальном уровне [74]:

$$Sp_i = \left\langle D^{in}, D^{out}, T^{in}, T^{out}, R^{in}, R^{out}, F, C \right\rangle, \quad (2.27)$$

где D^{in} – множество входных массивов данных;

D^{out} – множество выходных массивов данных;

T^{in} – множество типов входных данных;

T^{out} – множество типов выходных данных;

R^{in} – множество правил ввода входных данных модуля;

R^{out} – множество правил вывода выходных данных модуля;

F – множество управляющих воздействий *i*-го модуля;

$C = \langle c_n \rangle$ – вектор характеристик модуля, где c_n – *n*-я характеристика, такая как трудоемкость модуля, максимальный объем памяти, выделяемый для работы модуля и друние характеристики.

К множеству *D* могут относиться переменные, структуры данных, ссылки на структуры данных, пакеты сообщений и так далее. Допустимые типы данных *T* определяются спецификацией языка программирования, на котором реализовано программное средство. К множеству *R* относятся модификаторы доступа переменных (например, private, global). К множеству *F* могут относиться флаги управления состоянием модуля. Для обеспечения хранения информации о выполненных вычислениях, текущем состоянии ИС и предоставления этой информации по запросам как пользователя, так и некоторого программного модуля, используются кортежи данных. Кортеж данных *C* для каждой вершины v_i, который в дальнейшем будет базой для организации хранения информации о работе ПО ИС в базе данных, определяется следующим образом:

$$C_i = \langle User\_ID, Time\_Op, Num\_V, Data\_Res, Data\_Inp, PCall\_Par \rangle, \quad (2.28)$$

где $User\_ID$ – идентификатор пользователя, активировавшего процедуру;

$Time\_Op$ – время активации процедуры;

Num_V – номер вершины, инициализирующей ВП;

$Data\_Res$ – ссылка на массив данных, полученных в результате вычислений *i*-ой процедуры;

$Data\_Inp$ – ссылка на входные массивы данных для *i*-ой процедуры;

$PCall\_Par$ – параметры вызова *i*-ой процедуры.

Входные и выходные данные процедуры записываются в виде векторов данных:

$$Data\_Res = \{ d_q^{RES} \}, \forall q = \overline{1,Q}, \quad (2.29)$$

$$Data\_Inp = \{ d_p^{INP} \}, \forall p = \overline{1,P}, \quad (2.30)$$

где Q, P – количество всех выходных и входных элементов данных.

Преимуществом предложенного метода объединения вершин ярусно-параллельной графовой модели программной архитектуры ИС является то, что метод позволяет уменьшить временные затраты на конфигурирование путем упрощения структуры графовой модели, опираясь на ее топологические особенности и конфигурационные требования к системе.

2.3.2 Графовый метод оценки функциональной сложности программного обеспечения информационной системы

Необходимость кастомизации ПО приводит к тому, что функциональные требования могут изменяться в процессе разработки и сопровождения ПО. При этом уложняется программная функциональность, что приводит к увеличению временных и трудовых затрат на внедрение и сопровождение программного продукта.

Пользователь, решая свою задачу, может столкнуться с необходимостью выбора другого метода решения, который отличается от существующего. Для этого необходимо переконфигурирование функциональности рабочего места и замещение существующего программного модуля новым, чей функционал отвечает очередной версии заявленных требований. В этом случае разработчик должен предусмотреть такую ситуацию путем включения избыточной функциональности в программную архитектуру ИС и разрешать пользователю самостоятельно формировать компонентную архитектуру для своей подсистемы.

Графовая модель программной архитектуры (2.10), являющаяся исходной информацией для графового метода оценки функциональной сложности ПО ИС, позволяет выделить подграфы пользовательских задач. А также сформировать граф всей программной архитектуры. Работая с полученным множеством вершин графовой модели, которым сопоставляются соответствующие программные модули, пользователь заменяет или добавляет новые программные модули в соответствии с подмножествами $V^N = \{ v_i^N \}$ или $V^M = \{ v_i^M \}$ и формирует новый подграф задач, связанный с критерием оценки функциональной сложности. Для этого выполняется расчет количества функциональных указателей (Functional Points) по формуле, предложенной А. Албрехтом [59]: $FP = TotalPoints \cdot \left(0,65 + 0,01 \cdot \sum_{i=1}^{14} F_i \right)$, где FP – общее количество функциональных указателей; $TotalPoints$ – общая оценка, получаемая при расчете FP-метрики на основе ранга и оценки сложности внешних вводов и выводов, запросов, внутренних логических и внешних интерфейсных файлов для текущего программного модуля; F_i – коэффициенты регулировки сложности, принимающие значения в зависимости от характеристики системных параметров приложения (программного модуля). Величины $TotalPoints$ и F_i вычисляются в соответствии со стандартным алгоритмом расчета FP-метрик [59] и позволяют оценить функциональную сложность программных модулей выбранного подграфа или графа в целом.

В случае использования графовой модели программной архитектуры ИС для каждой вершины этой модели необходимо хранить рассчитанную FP-метрику, значение которой используется при формировании подграфа или графа новой функциональной конфигурации. Общая оценка функциональной сложности ПО FS_{curr} основывается на сумме FP-метрик всех входящих в него компонентов:

$$FS_{curr} = \sum_{i=1}^{n} FP_i , \tag{2.31}$$

где n – общее количество вершин графа или подграфа;

FP_i – общее количество функциональных указателей (2.31) для i-ой вершины.

При учете нефункциональных требований к программным модулям на каждом ярусе ориентированного графа (2) решается задача подбора программных модулей с минимальным значением функциональной сложности, то есть:

$$FSL_k = \sum_{i=1}^{m} FP_i , \tag{2.32}$$

$$FSL_k \to min , \tag{2.33}$$

где FSL_k – значение функциональной сложности для k-ого яруса графовой модели;

m – общее количество вершин на k-ом ярусе для текущего яруса ярусно-параллельной формы графовой модели;

FP_i – общее количество функциональных указателей для i-ой вершины.

Исходя из этого, расчет общей функциональной сложности для всей графовой модели программной архитектуры ИС выглядит следующим образом:

$$FS_{curr} = \sum_{k=1}^{p} FSL_k \,, \tag{2.34}$$

$$FS_{curr} \to min, \tag{2.35}$$

где p – общее количество ярусов заданной ярусно-параллельной графовой модели; FSL_k – значение функциональной сложности для *k*-ого яруса графовой модели.

Для расчета функциональной сложности каждого программного модуля на основе FP-метрики требуется определить ранг и оценку пяти информационных характеристик. Для этого количеству внешних вводов ставится в соответствие величина полустепени захода соответствующей вершины графовой модели. Количеству внешних выводов соотносится величина полустепени исхода вершины графовой модели. Количество внутренних логических файлов, являющихся частью базы данных или отдельным файлом, рассчитывается на основе множества D_{in} (2.3). Для расчета количества внешних интерфейсных файлов используется множество D_{out} (2.4). При получении значений двух последних информационных характеристик также дополнительно используется элемент *PDes* подмножества p_i^{PM} элементов описания программных компонент, содержащий дополнительные сведения о программном модуле. Множества входных и выходных данных D_{in} (2.3) и D_{out} (2.4) используются для определения типов данных, с которыми работают программные модули. Исходя из этого, вся информация о программной архитектуре ИС, содержащаяся в ярусно-параллельной графовой модели программной архитектуры, используется для автоматического расчета информационных характеристик и FP-метрик, на базе которых выполняется оценка функциональной сложности конфигурируемого ПО. Предложенный графовый метод оценки функциональной сложности ПО ИС позволяет сформировать программную архитектуру с минимальным значением

функциональной сложности на базе графовой модели и включает в себя такие этапы (Приложение Е, рис.1).

Этап 1. Формирование подграфа пользовательской задачи для графа (2.1).

Этап 2. Формирование подмножеств вершин $V^M = \{ v_i^M \}$ и $V^H = \{ v_i^H \}$, которые будут замещать или добавляться к существующей графовой модели программной архитектуры ИС для соответствующего яруса этой модели.

Этап 3. Расчет FP-метрики (2.31) для множества вершин V, сформированного с учетом подмножеств V^M и V^H текущего яруса графовой модели (2.10)

Этап 4. Расчет общей функциональной сложности (2.32) для текущего яруса графовой модели (2.10).

Этап 5. Проверка соответствия значения показателя (2.33) функциональным требованиям пользователя. При отрицательном ответе повтор действий с этапа 2.

Этап 6. При удовлетворительном значении показателя (2.33) переход на новый ярус и повтор действий с этапа 2.

Этап 7. На завершающей стадии обработки всех ярусов графовой модели расчет показателя функциональной сложности всего подграфа пользовательской задачи (2.34).

Этап 8. При неудовлетворительном значении показателя (2.35) – возврат на исходный ярус графовой модели и повтор формирования подмножеств новых вершин с этапа 2. В случае невозможности осуществить дальнейшую минимизацию функциональной сложности ПО переход на этап 9.

Этап 9. Компиляция сформированной программной архитектуры ИС на основе обновленной графовой модели, завершение метода.

На базе предложенного метода для текущей версии требований конечного пользователя выполняется конфигурирование функциональности рабочего места с учетом оценки функциональной сложности отдельных программных модулей и ПО в целом. Преимуществом предложенного метода является то, что метод позволяет получить программную архитектуру ИС с наименьшей допустимой функциональной сложностью, позволяет оценить соответствие формируемого ПО

функциональным требованиям пользователя и обеспечить поддержку рабочих версий ПО. Недостатком метода оценки функциональной сложности ПО является субъективная оценка коэффициентов регулировки сложности и системные параметры программных модулей. Для определения временной сложности предложенного графового метода оценки функциональной сложности ПО ИС берется во внимание количество вершин графовой модели n, так как количество функциональных указателей FP расчитывается для каждой вершины. Учитывается количество ярусов для ЯПФ графовой модели. Поскольку, в крайнем случае, число таких ярусов также равно n, то оценка временной сложности предложенного метода соответствует $O(n^2)$. Графовый метод оценки функциональной сложности ПО ИС применим для динамического конфигурирования ПО, предназначенного для решения задач несколькими способами. Метод позволяет оценить функциональную сложность программных модулей на основе информации о связях по данным между программными модулями и типах этих данных. Эта информация берется из ярусно-параллельной графовой модели программной архитектуры ИС (2.10) и позволяет рассчитать FP-метрики. В дальнейшем система автоматически подбирает программные модули, соответствующие выбранному методу решения и отвечающие параметрам эффективности всей системы. Параметры эффективности системы задаются в требованиях пользователя. Дальнейшее исследование и улучшение предложенного метода направлено на отказ от субъективной составляющей при расчете функциональных указателей программных модулей.

2.4 Выводы по разделу 2

В разделе выполнены такие задачи.

1. Усовершенствована ярусно-параллельная графовая модель программной архитектруры ИС путем учета взаимосвязей по входным и выходным данным между программными модулями. Программные модули в модели представляются вершинами графа и реализуют функциональные задачи ИС. Направленные дуги

графа отображают связи по данным между программными модулями. Дополнение модели связями по данным, дополнительными функциональными характеристиками программных модулей, а также возможностью обезпечения избыточной функциональностью ПО обеспечивают возможность динамического конфигурирования программной архитектуры при изменении функциональных требований конечного пользователя, поддерживая тем самым процесс кастомизации ПО.

2. Предложен метод объединения вершин ярусно-параллельной графовой модели программной архитектуры на основе оценок сцепления и связности программных модулей, на основе значений которых метод позволяет объединить вершины графовой модели. Метод позволяет уменьшить временные затраты на кастомизацию ПО в условиях изменяющихся во времени требований и, тем самым, снизить затраты на конфигурирование. Для предложенного метода проведена оценка временной сложности, которая совпадает с алгоритмом Косарайю и соответствует величине $O(n)$.

3. Предложен графовый метод оценки функциональной сложности программного обеспечения путем использования ярусно-параллельной графовой модели программной архитектуры ИС. Метод включает в себя этапы определения функциональных характеристик программных модулей и расчета критериев функциональной сложности, что позволяет оценить соответствие формируемого ПО функциональным требованиям пользователя. Для предложенного метода проведена оценка временной сложности, соответствующая величине $O(n^2)$. Преиуществом предложенного метода является возможность получения программной архитектуры ИС с минимальной допустимой функциональной сложностью и для поддержки рабочих версий ПО.

РАЗДЕЛ 3
МЕТОД ПРОВЕРКИ ВЫПОЛНЕНИЯ ОГРАНИЧЕНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ ИНФОРМАЦИОННОЙ СИСТЕМЫ И ЕГО ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1 Разработка автоматного метода проверки выполнения ограничений к формируемому программному обеспечению

Автоматный метод проверки выполнения ограничений к формируемому ПО, выраженных в форме нефункциональных требований используется для проектирования исполнителя ВП, моделируя сценарии взаимодействия программных модулей и сравнивая результаты моделирования с нефункциональными требованиями к ПО. Входными данными метода является графовая ярусно-параллельная модель программной архитектуры ИС, для которой определены вершины, отвечающие за формирование пользовательского диалога и за формирование соответствующих им подграфов модулей, реализующих связанные функциональные задачи системы.

Нефункциональные требования, определяемые в спецификациях требований, разделяются согласно существующей классификации нефункциональных требований (модель FURPS+) к программному продукту: 1) требования к производительности; 2) требования к надежности; 3) требования к безопасности ПО. На основе спецификаций нефункциональных требований формализуются свойства ПО с помощью аппарата темпоральной логики. При этом на базе ярусно-параллельной графовой модели программной архитектуры формируется автоматная модель, позволяющая верифицировать указанные свойства. В целях построения автоматной модели, используемой в рамках предложенного метода, рассмотрим КА как один из средств структурирования приложения, организации взаимодействия программных модулей и проверки выполнения ограничений к формируемому ПО.

Состояния КА определяются как совокупность значений всех его переменных. Значения меняются под воздействием внешнего события. Для определения состояния КА возможно использовать переменную перечисления, которая сохраняется в памяти. Метод, подразумевающий использование технологий автоматного программирования, является относительно распространенным (известны методы верификации автоматов и их применение при проектировании ИС [52]).

Автоматы делятся на два типа: абстрактные и структурные. В абстрактных автоматах входные и выходные воздействия формируются последовательно, а в структурных – параллельно, поэтому структурные автоматы применяются в автоматном программировании.

В автоматном программировании управляющие состояния, отражающие качественные особенности, выделяются и перечисляются явно [29].

Для задания параллельных подсостояний возможно специфицировать два и более подавтомата, которые могут параллельно выполняться внутри составного события. Каждый из подавтоматов занимает некоторую область внутри составного состояния, которая отделяется от остальных горизонтальной линией. Если на диаграмме имеется составное состояние с вложенными параллельными подавтоматами, то экземпляр сущности может одновременно находиться в нескольких подсостояниях, но не более чем по одному из каждого подавтомата. Если какой-либо из подавтоматов пришел в свое конечное состояние раньше других, то он должен ожидать, пока другие подавтоматы не придут в свои конечные состояния.

В автоматном программировании применяются следующие обозначения [65]:

1) имя автомата начинается с символа *A*;

2) имя события с символа *e*;

3) имя входной переменной с символа *x*;

4) имя переменной состояния автомата с символа *y* или *s*;

5) имя выходного воздействия с символа *z*.

После каждого из указанных символов следует номер соответствующего автомата или воздействия.

Зададим КА формально (рис. 3.1):

$$A = \langle S, X, Y, s_0, \delta, \omega \rangle, \tag{3.1}$$

где S – конечное непустое множество состояний;

X – конечное непустое множество входных сигналов (входной алфавит);

Y – конечное непустое множество выходных сигналов (выходной алфавит);

$s_0 \in S$ – начальное состояние;

$\delta : S \times X \to S$ – функция переходов;

$\omega : S \times X \to Y$ – функция выходов.

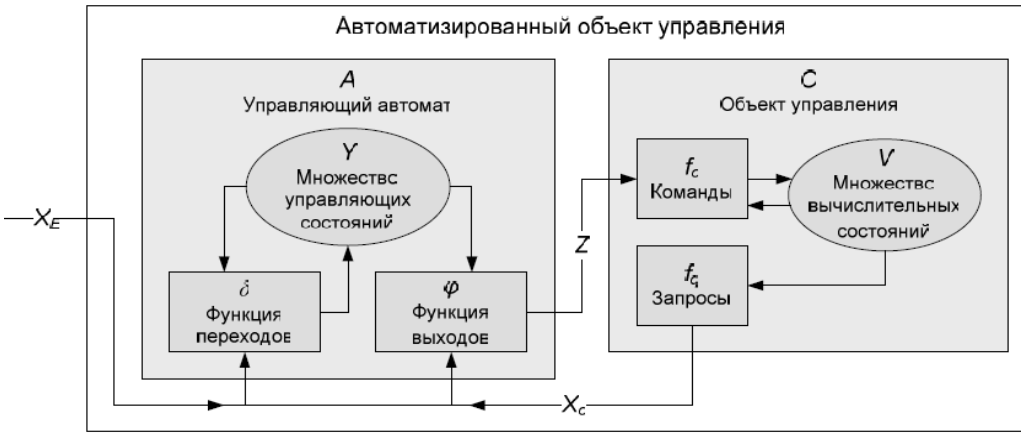


Рисунок 3.1 – Автоматизированный объект управления

Каждый конкретный набор данных всегда будет переводить КА в одно и то же множество состояний, которое называется предысторией КА. В случае эквивалентных предысторий, если они одинаковым образом влияют на дальнейшее поведение автомата, нет необходимости запоминать все входные истории, а хранить в памяти их класс эквивалентности, что обеспечивает оптимизацию памяти ВС. Кроме того, КА позволяет осуществлять централизованное явное управление ПС [50]. Для управляющего КА объектом управления будут служить программные сервисы, реализующие функции вложенных КА (рис. 3.1)

Программный сервис управляет взаимодействием программных модулей, описанных графовой ярусно-параллельной моделью программной архитеткруы через интерфейсы. Интерфейс класса содержит сигнатуры его компонентов (запросов и команд), а также семантические свойства: предусловия и постусловия запросов и команд, инварианты класса. Предусловия, постусловия и инварианты также называются утверждениями, в совокупности они образуют контракт класса или его компонента [64, 94]. Метод проектирования ПО, в котором утверждения являются неотъемлемой частью текста программной системы, называется проектированием по контракту [140]. В соответствии с этим методом система считается корректной только в том случае, если перед вызовом любого компонента выполняется его предусловие, а по завершении работы компонента – его постусловие. Инвариант класса должен выполняться во всех устойчивых состояниях любого объекта этого класса (в тех состояниях, которые могут наблюдать клиенты класса). Используя объектно-ориентированный подход к проектированию автоматной модели, можно задать шаблоны использования КА [65, 94] при разработке управляющего программного модуля (рис. 3.2) (в частности автомата Мили) [65].

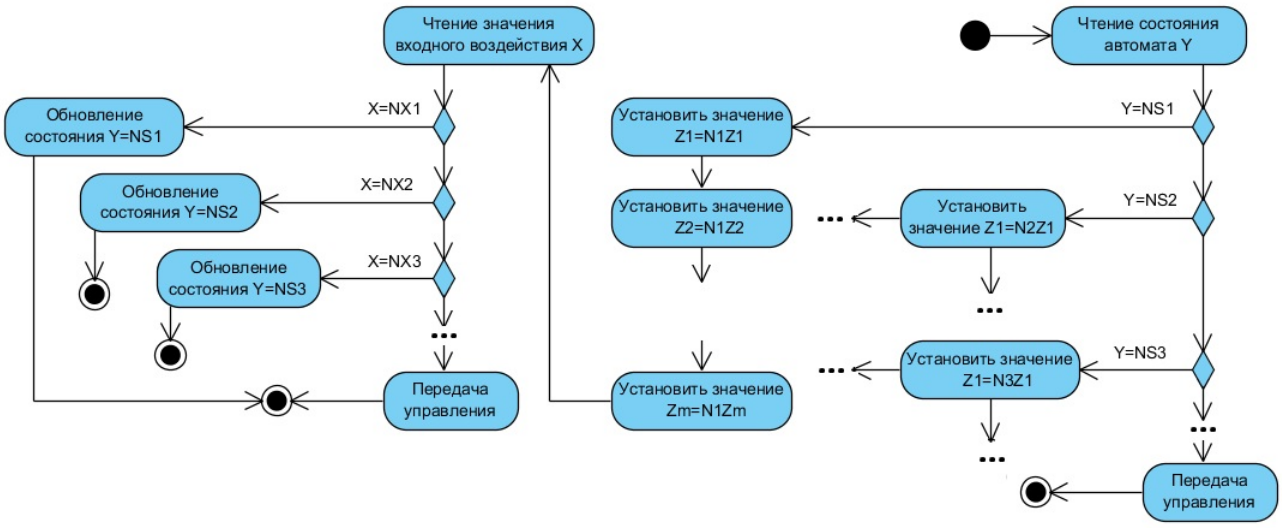


Рисунок 3.2 – Шаблон реализации одного такта автомата Мура

Учитывая, заявленные требования к исполнителю для объектно-ориентированных программ в качестве базового представления схемы связей автоматов выберем диаграмму классов. Также определим основные автоматы и их

взаимосвязи, участвующие в управлении взаимодействием программных модулей ПО ИС.

Основным автоматом (Приложение Г, рис. 9) является КА верхнего уровня AUserSM**.** Данный автомат управляет работой остальных вложенных подавтоматов. Основным объектом управления для него служит пользовательское приложение UserApp, реализующее диалог с пользователем и ведущее его через все этапы функций, исполнение которых приводит к решению поставленной задачи. Также на данной диаграмме заданы основные события, генерируемые объектом управления в ходе вычислений, котрые отображены на соответствующей диаграмме переходов (рис. 3.3).

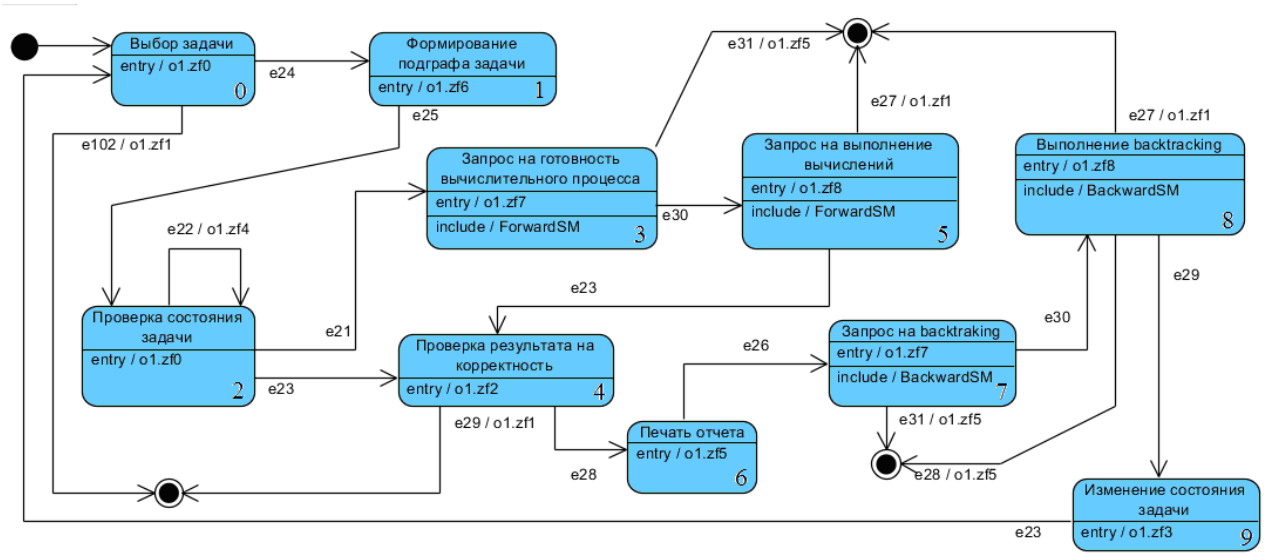


Рисунок 3.3 – Диаграмма переходов для управляющего КА AUserSM

Помимо этого необходимо задать два других КА, реализующих непосредственное управление и организацию ВП, а также восстановление данных для ВП. Для этого примем допущение, что V – является множеством вершин подграфа G, который определяется для текущей задачи, запускаемой пользователем. Обозначим текущий номер яруса буквой L; Imax(L) – количество вершин на ярусе L; Lmax(G) – количество ярусов для подграфа G. Обозначим ВП, соответствующий вершине v_i как P(Vi). Обозначим вершину $v_i \in V$, которая находится на ярусе L как Vi(V, L) . Также обозначим количество входных дуг (полустепень захода) для вершины $v_i \in V$ как DEG_IN(Vi). А вершину v_k , из

которой существует направленная дуга (v_k, v_i) как Vk. На этом основании зададим автоматы AForwardSM и ABackwardSM (Приложение Г, рис. 10).

Объектом управления для автомата AForwardSM является программный модуль ForwardUnit, контролирующий прохождение вычислительного процесса по подграфу задачи пользователя. Основные состояния КА AForwardSM определены соответствующей диаграммой переходов (рис. 3.4).

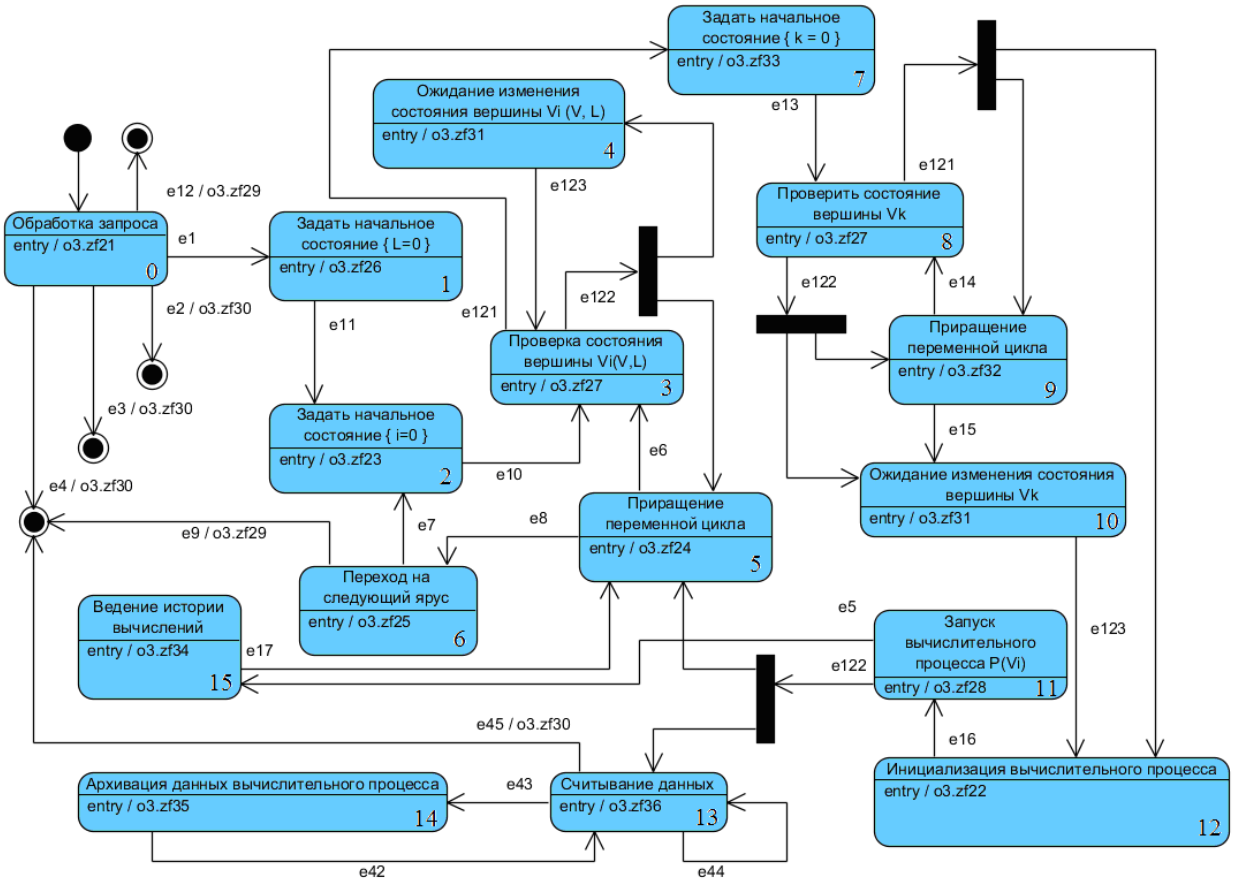


Рисунок 3.4 – Диаграмма переходов для вложенного КА AForwardSM

КА ABackwardSM управляет программой BackwardUnit, контролирующей восстановление данных ВП, которые необходимы для пересчета. Для данного автомата также задана соответствующая диаграмма переходов (рис. 3.5).

Состояние КА AUserSM «Запрос на готовность ВП» и состояние «Запрос на выполнение вычислений» являются составными (рис. 3.3) и описываются с помощью вложенного автомата AForwardSM (рис. 3.4). Также составные состояния «Запрос на backtracking» и «Выполнение backtracking» реализуются при помощи вложенного автомата ABackwardSM (рис. 3.5).

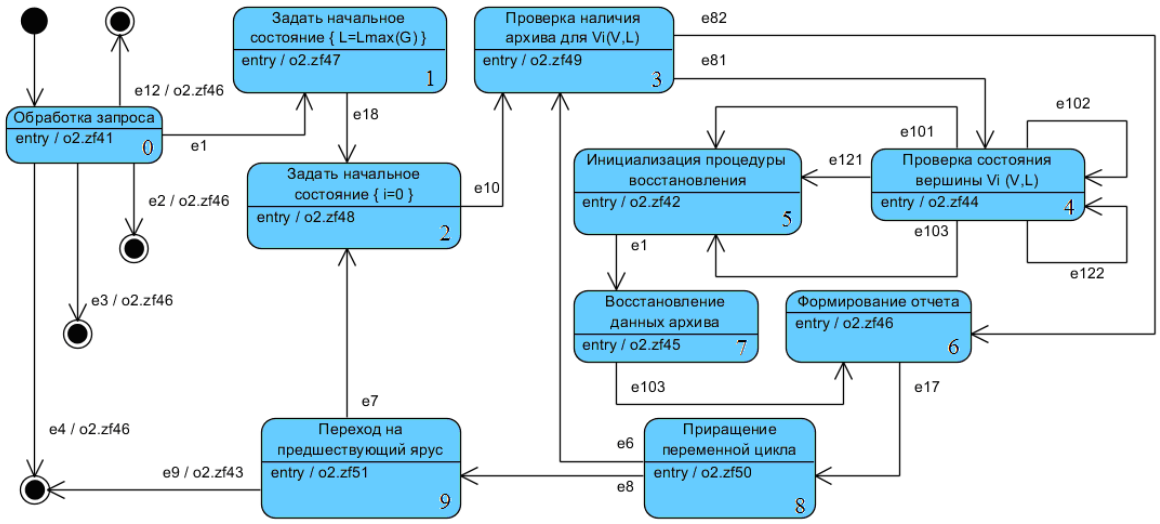


Рисунок 3.5 – Диаграмма переходов для вложенного КА ABackwardSM

Для указанных автоматов важно учитывать то, что ВП, которыми они управляют, могут выполняться параллельно и независимо друг от друга (для функциональных задач, располагающихся на одном ярусе), поэтому для диаграммы автомата AForwardSM (рис.3.4) заданы распараллеливающие и синхронизирующие переходы, обеспечивающие ввод автомата одновременно в несколько состояний.

Рассмотрим подробно основные задачи, реализуемые КА. Для управляющего КА AUserSM (рис. 3.3) это:

1) формирование подграфа задачи;

2) проверка состояния задачи;

3) проверка результата на корректность;

4) изменение состояния задачи.

Для автомата AForwardSM (рис. 3.4) это:

1) запуск вычислительного процесса;

2) архивация данных ВП.

Для КА ABackwardSM (рис. 3.5) это восстановление архивных данных каждой вершины соответствующих ярусов графовой модели программной архитектуры ИС. Основываясь на сформированных диаграммах переходов КА,

определим программно модель КА на языке PROMELA, с тем, чтобы потом можно было проверить заданные свойства верификатором JSPIN.

Определим следующие переменные**:** *A1States* – номер текущего состояния автомата, а *A1Events* – номер события, при котором КА переходит в это событие. Начальные значения этих переменных равны нулю.

int A1States;

int A1Events;

Затем определим процедуру *A1()*, описывающую правила перехода автомата из одного состояния в другое. Внутри этой процедуры определяется цикл *do,* в рамках которого процедура проверяет текущее значение переменной *A1States* и, в зависимости от значения, переходит в соответсвтующий вложенный цикл. Во вложенном цикле также перебираются все возможные переходы. При этом, переменной *A1Events* определяется номер события, которое послужило пречиной этого перехода (Приложение Д, «Код процедуры А1»):

Согласно с диаграммой переходов (рис. 3.3) начальное состояние *A1States=0* соответствует «выбору задачи», откуда автомат может перейти либо в состояние «Формирование пографа задачи» (*A1States=1*) либо завершить работу. В первом случае переход срабатывает при событии «Выбрана задача», и переменной *A1Events* присвоено значение *24* (Приложение Г, рис. 9). Во втором случае срабатывает переход *A1Events=102* «Отмена действий». В случае возниконовения непредвиденного события выход из данного вложенного цикла осуществляется с помощью оператора *break***.** Из состояния «Формирование подграфа задачи» (*A1States=1*) автомат может перейти только в состояние «Проверка состояния задачи» (*A1States=2*) при срабатывании собырия *A1Events=25* «Сформирован подграф задачи». Поэтому в этой ветке основного цикла нет необходимости во вложенном цикле. Далее, выйдя из состояния *A1States=2,* автомат либо находится в состоянии ожидания завершения задачи, что соответствует значениям переменных *A1Events=22* и *A1States=2*, либо переходит в состояние «Запрос на готовность ВП» для перехода «Задача не выполнена» (*A1States=3; A1Events=21*), либо в состояние «Проверка результата на

корректность» для перехода «Задача выполнена» *(A1States=4; A1Events=23).* Аналогичным образом описываются оставшиеся состояния, и формируется полный текст процедуры (Приложение В, «Код программы верификатора PROMELA – JSPIN»).

Сама процедура *proctype A1()* запускается таким образом:

```
init {
        A1Events=0;
                A1States=0;
 run A1();
   }
```

Для вложенного КА AForwardSM управления ходом ВП (рис. 3.4) задано тело основного цикла (Приложение Д, «Тело основного цикла автомата AForwardSM»).

В этом случае автомат начинает работу с обработки запроса – состояние A1States=0, A1Events=0. Начало работы ВП и управление им возможно только, если данный процесс не активен. Поэтому автомат переходит в состояние *A1States=1* «Задать начальное состояние L=0» (рис. 3.4) при возникновении события «Получено разрешение», соответствующее значению *A1Events=1* (Приложение Г, рис. 9). Для всех других событий автомат прекращает свою работу, переходя в состояние *A1States=16.* В состоянии *A1Events=1* КА определяет начальный ярус для ЯПФ подграфа, который выделяется при активации пользователем соответствующей задачи. По переходу *A1Events=11* «Задано начальное состояние L=0» КА попадает в состояние *A1Events=2* «Задать начальное состояние i=0», после которого обязан перебрать все вершины данного яруса. Для этих двух состояний 1 и 2 существует только по одному переходу, поэтому значения переменных *A1Events* и *A1States* определяются без вложенных циклов. Для состояния *A1States=3* характерны два параллельных перехода для одного и того же события *A1Events=122* «Выполняется ВП» в состояние «Ожидание изменения состояния вершины V_i (V, L)» (*A1States=4*) и «Приращение переменной цикла» (*A1States=5*). Смысл этих переходов обусловливается поиском незадействованной еще вершины подграфа на текущем ярусе. Как только

такая вершина будет найдена, соответствующий ей процесс будет запущен по переходу *A1Events=121* «Не запущен ВП».

Для вложенного КА ABackwardSM, реализующего backtracking вычислительных процессов (рис. 3.5) также задано тело основного цикла (Приложение Д, «Тело основного цикла автомата ABackwardSM»).

Работа КА, обеспечивающего восстановление данных для ВП, также начинается с обработки запроса и так же, как в предыдущем случае автомат запускает действия по восстановлению данных, если сработал переход по событию *A1Events=1* «Получено разрешение». Состояние *A1States=1* (рис. 3.5), в отличие от AForwardSM автомата формирует подграф задачи, начиная с нижнего уровня, так как восстанавливается обратный ход ВП. Поэтому переменной L присваивается номер последнего яруса. И далее, по переходу *A1Events=18* «Задано состояние L=Lmax(G)» (Приложение Г, рис. 9) КА переходит в состояние *A1States=2.* На следующем шаге (состояние *A1States=3*) проверяется наличие архива, требуемого для восстановления данных, и, в зависимости от его наличия, срабатывают два перехода *A1States=82* «Архив отсутствует» или *A1States=81* «архив существует». Подобным образом программируются оставшиеся состояния двух вложенных КА (Приложение В, «Код программы верификатора PROMELA – JSPIN»).

Зададим в текстовой форме свойства автоматных моделей, обязательные к выполнению. Для автомата AForwardSM это:

1) ВП P(Vi) будет запущен только по запросу пользователя;

2) гарантия того, что ВП P(Vi) обязательно когда-нибудь будет запушен, если не будет ошибок вычислений;

3) гарантия того, что ВП P(Vi) будет запущен только тогда, когда вершина Vi находится в состоянии «не запущен» или «выполнен»;

4) вершина Vi когда-нибудь обязательно будет находиться в состоянии «не запущен» или «выполнен»;

5) для каждой необходимой вершины графовой модели будет выполнен процесс архивации данных;

6) для последовательности ВП на основе существующей ЯПФ графовой модели можно выделить такой ВП, который становится активным только тогда, если все предшествующие процессы завершат свою работу;

7) для всего множества ВП программной системы можно выделить такое подмножество ВП, которые размещены на одном ярусе ЯПФ графовой модели и не имеют информационных зависимостей друг от друга, что позволяет запускать их одновременно.

Для КА ABackwardSM это:

1) восстановление данных ВП будет выполнено только по запросу пользователя;

2) восстановление данных ВП будет выполнено только при наличии архива данных для каждой вершины подграфа вычислительного процесса;

3) для каждой вершины выделенного подграфа восстановятся данные только в состоянии, если другой пользователь не задействовал эту вершину (статус «не запущен») и, если восстановление данных не происходило вовсе (статус «не восстановлен»), или данные уже были восстановлены ранее (статус «восстановлен»).

Формализуем приведенные свойства автоматных моделей на базе темпоральной логики LTL. Для автомата AForwardSM это:

1) $\neg(\neg e1 \mathbf{U} o3.zf\,28)$;

2) $\neg(e5 \mathbf{U} o3.zf\,28)$;

3) $\neg(\neg(e121 \vee e123)\mathbf{U} o3.zf\,28) = \neg((\neg e121 \wedge \neg e123)\mathbf{U} o3.zf\,28)$;

4) $\mathbf{F} o3.zf\,31$;

5) $e43\mathbf{F} o3.zf\,35$;

6) $F_{общ}^{AV} = \neg(F_1^{AV} \wedge F_2^{AV} \wedge ... \wedge F_n^{AV}) = \neg F_1^{AV} \vee \neg F_2^{AV} \vee ... \vee \neg F_n^{AV}$,

где $F^{AV} = \neg(\neg e122\mathbf{U} o3.zf\,31) \wedge \neg(e121\mathbf{U} o3.zf\,22)\mathbf{U}(\neg(\neg e16\mathbf{U} o3.zf\,28))$;

7) $F_{общ}^{PV} = \neg(F_1^{PV} \wedge F_2^{PV} \wedge ... \wedge F_n^{PV}) = \neg F_1^{PV} \vee \neg F_2^{PV} \vee ... \vee \neg F_n^{PV}$,

где $F^{PV} = \neg((\neg e6 \vee \neg e7)\mathbf{U} o3.zf\,28\mathbf{U}(\neg(\neg e122\mathbf{U}(o3.zf\,24 \wedge o3.zf\,36))))$.

Для автомата ABackwardSM это:

1) $\neg(\neg e1 \boldsymbol{U} o2.zf\ 45)$;

2) $\neg(\neg e82 \boldsymbol{U} o2.zf\ 45) \vee \neg(\neg e81 \boldsymbol{U} o2.f\ 45)$;

3) $\neg(\neg((e121 \wedge (e101 \vee e103) \boldsymbol{U} o2.zf\ 45) \vee \neg((e122 \wedge e102) \boldsymbol{U} o2.f\ 45) =$

$= \neg((\neg e121 \vee \neg e101 \wedge \neg e103) \boldsymbol{U} o2.zf\ 45) \vee \neg((e122 \wedge e102) \boldsymbol{U} o2.f\ 45)$.

Язык *LTL* состоит из множества атомарных высказываний *p1*, *p2*, …$\in$ *AP*, логических связей ¬, /\, \/, →, и темпоральных операторов. В приведенных свойствах используется оператор ***U*** (Until). Для данного темпорального оператора выражение $\psi \boldsymbol{U} \varphi$ означает, что $\boldsymbol{\psi}$ будет верно до тех пор, пока не станет верно $\boldsymbol{\varphi}$, где $\boldsymbol{\psi}$, $\boldsymbol{\varphi}$ – правильно построенные формулы. На следующем шаге данные свойства могут быть проверены на автоматной модели, преобразованной с помощью языка PROMELA для программы-верификатора SPIN.

Автоматный метод проверки выполнения ограничений к формируемому ПО включает в себя такие этапы.

Этап 1. Выделение базовых состояний компонентов программного обеспечения и условий переходов на основе входных и выходных данных вершин графовой модели.

Этап 2. Формирование структуры управляющего конечного автомата и его вложенных подавтоматов, реализующих поведение системы, с учетом данных спецификаций вершин графовой модели.

Этап 3. Определение принципов взаимодействия конечных автоматов с программными модулями на основе архитектурных шаблонов объектно-ориентированного программирования для организации распределенного или параллельного выполнения программы.

Этап 4. Формирование исходной информации, требуемой для организации работы конечных автоматов, такой как матрицы смежности графовой ярусно-параллельной модели программной архитеткуры и транспонированной матрицы смежности для управления прямым и обратным ходом вычислительных процессов.

Этап 5. Проверка выполнимости функций программного обеспечения, отказоустойчивости в условиях аппаратных ограничений и сравнение с

требованиями конечного пользователя моделируемых сценариев взаимодействия модулей.

Этап 6. Разработка программной архитектуры ИС с учетом взаимосвязи графовой модели и управляющих конечных автоматов.

Этап 7. Генерация программного обеспечения на основе проверенной программной архитектуры. Завершение метода.

Автоматный метод проверки выполнения ограничений к формируемому ПО, выраженных в форме нефункциональных требований, используется для обеспечения надежности программного продукта путем автоматического анализа покрытия модели КА при помощи LTL-формул, которые описывают заданные требования. Соответственно, временная сложность предложенного метода прямопропорциональна количеству тестовых покрытий всех состояний КА. Если обозначить через n количество состояний КА, а переменной k – количество переходов в эти состояния, то оценка временной сложности автоматного метода соответствует величине $O(n^3 \cdot \log k)$, расчитанной на основе временной сложности алгоритма поиска путей фиксированной длины на графе [72].

Предложенный автоматный метод проверки выполнения ограничений к формируемому ПО позволяет сформировать ограничения к программной архитектуре с учетом изменяющихся требований. Позволяет повысить надежность его функционирования, опираясь на данные графовой модели программной архитектуры ИС, позволяет оценить возможность развертывания ПО для существующей конфигурации аппаратных средств, что отличает метод от существующих аналогичных технологий (BDD, TDD).

3.2 Использование шаблонов проектирования программной архитектуры для определения взаимодействия управляющего конечного автомата и графовой ярусно-параллельной модели программной архитектуры

Разработанная модель управляющего КА и вложенных КА, реализующих ведение и восстановление данных ВП, берется в качестве программной

надстройки для формируемого ПО, которая позволяет управлять его работой. Функционирование ПО также контролируется и со стороны пользователя в интерактивном режиме. Программная система, работающая в режиме диалога, контролирует скорость взаимодействия с внешней средой, определяет режим активных действий, режим ожидания ответа пользователя или режим отложенного выполнения вычислений [65]. КА, реализованный в виде программного сервиса, управляет работой программных модулей (рис. 3.6), состовляющих программную архитектуру, считывает состояния на графовой моделе и, на основании полученной информации, принимает решение об активации соответствующего ВП, находящегося в области ответственности ПО. Состояние завершенности или активности ВП отображается на графовой моделе путем информирования управляющего автомата через передачу сообщений.

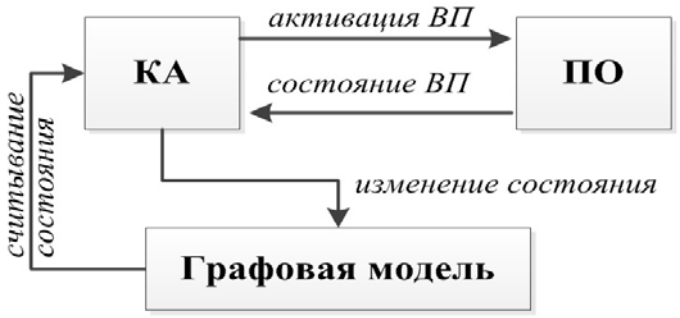


Рис. 3.6. Принцип взаимодействия КА и графовой модели программной архитектуры

Перед тем как запустить по запросу пользователя ВП на выполнение КА считывает состояние вершины, соответствующей этому процессу на графовой моделе. Состояния этой вершины определяют дальнейшие действия КА, которые могут отличаться в зависимости от того, находится ли ВП уже в акивном сотоянии, когда другой пользователь или другой ВП запустили функцию, или же ВП завершил свою работу, или данный ВП еще не был запущен вообще. Далее, после того, как информация считана с графовой модели, управляющий КА активизирует ВП и отмечает изменившееся состояние для текущей вершины. После получения от ПО ответа об изменениях, происшедших в результате

действий ВП, КА также отмечает эти данные на графовой модели и переходит к следующей вершине согласно подграфу задачи.

Аналогичным образом выполняется и backtracking задачи, для которого КА выполняет откат действий ВП по архивным данным. Архивы, формируемые для каждой узловой точки активного ВП, то есть вершины графовой модели, обусловливают необходимость структурированной информации о состоянии резервных данных, которая также ставится в соотвествии с вершинами сформированного подграфа задачи. При этом выделяются также три состояния: архивирование запущено, архивирование выполнено, данные архива отсутствуют. Поставленные условия взаимодействия КА, ПО и графовой модели, требуют специальных структур данных, обеспечивающих хранение и передачу информации между этими объектами.

Автоматизация процесса структурного синтеза базируется на упорядоченной и структурированной информации о необходимых программных библиотеках, участвующих в формировании ПО, об автоматных моделях, встраиваемых в программную архитектуру, о формализованном представлении программной архитектуры и спецификациях, регламентирующих правила стыковки программных модулей и функционирование компоновщика ПО. Все эти данные должны сохраняться на каждой стадии проектирования и разработки ПО, чем обосновано использование специальных форматов данных, обеспечивающих хранение и использование необходимой информации для разработчика в файле проекта. Такая информация необходима для того, чтобы обеспечить взаимосвязать компонентов ПО, управляющего КА и графовой модели.

Определим формально кортеж G_{SP} , который специфицирует данные графовой модели и структуру файла проекта, содержащего эту информацию:

$$G_{SP} = \langle N, S, VC, AM, CG, D \rangle, \tag{3.2}$$

где N – множество координат вершин, размещаемых на рабочей области экрана;

S – множество стадий проекта, фиксирующих текущее состояние процесса формирования ПО;

VC – множество характеристик вершин графовой модели, описывающих наличие компонент сильной связности;

AM – множество атрибутов, определяющих взаимосвязь вершин в соответствии с матрицей смежности графовой модели;

CG – множество атрибутов, определяющих взаимосвязь между вершинами в соответствии с матрицей смежности конденсированного графа;

D – множество атрибутов, определяющих характиристики функций, которые сопоставляются с вершинами графовой модели в соответствии с отображением «вершина-функция».

Для кортежа (3.2) множество координат вершин определяется следующим образом:

$$N = \left\{ m_1^N, m_2^N, ..., m_k^N \right\},\ m_i^N = \{vx, vy\}, \tag{3.3}$$

где *vx*, *vy* – координаты *i*-ой вершины.

Стадии проекта определяются как

$$S = \left\{ s_1, s_2, ..., s_r \right\},\ s_i = \{True, False\} \tag{3.4}$$

где s_i – логическая переменная, обозначающая текущую активную стадию проекта.

Это может быть, например, стадия, на которой осуществляется ввод матрицы смежности графовой модели; или стадия проверки корректности введенной информации.

Множество характеристик вершин графовой модели определяется так:

$$VC = \left\{ m_1^{VC}, m_2^{VC}, ..., m_n^{VC} \right\}, \tag{3.5}$$

где $m_i^{VC} = \{isSubG, Comp\}$ – является подмножеством, состоящим из двух элементов: *isSubG* – логическая переменная, которая определяет наличие вложенного подграфа для супервершины.

Множество атрибутов, позволяющих задать матрицу смежности графовой модели (2.10) в списочном виде определяется как

$$AM = \left\{ m_1^{AM}, m_2^{AM}, ..., m_k^{AM} \right\}, \tag{3.6}$$

где каждый атрибут $m_i^{AM} = \{V_{adj}\}$ представляет собой список вешин $v_j \in V_{adj}$, в которые идет дуга $x_{ij} \in X$ из вершины $v_i \in V$, при этом $V_{adj} \subset V$.

Матрица смежности конденсированного графа задается в списочном виде с помощью множества

$$CG = \left\{ m_1^{CG}, m_2^{CG}, ..., m_p^{CG} \right\}, \tag{3.7}$$

где каждый атрибут m_i^{CG}, по аналогии с (3.6) содержит список вершин $v_j \in V_{adj}^{CG}$, в которые идет дуга из вершины $v_i \in V$, но, в отличие от (3.6), дополняется подмножеством состояний ВП ST^{CG}, соответствующего *i*-й вершине, и подмножеством характеристик архивных данных AR^{CG}. Элемент множества *CG* определяется таким образом:

$$m_i^{CG} = \{V_{adj}^{CG}, ST^{CG}, AR^{CG}\}, \tag{3.8}$$

где $V_{adj}^{CG} \subset V$, $ST^{CG} = \{0,1,2\}$, $AR^{CG} = \{0,1,2\}$.

Другими словами, если задать пременную e^{st}, которая принимает значения из подмножества ST^{CG} и переменную e^{ar}, принимающую значения из подмножества AR^{CG}, то

$$e^{st} = \begin{cases} 0, \textit{если ВП не запущен}; \\ 1, \textit{если ВП запущен}; \\ 2, \textit{если ВП выполнен}. \end{cases}$$

И также

$$e^{ar} = \begin{cases} 0, \textit{если архив отсутствует}; \\ 1, \textit{если запущен процесс архивации}; \\ 2, \textit{если архив существует}. \end{cases}$$

Разделение информации об исходном графе и конденсированном необходимо для сохранения информации о слоях графовой модели и обеспечения быстрого доступа к нужному уровню.

Множество функций, сопоставляемых с вершинами, определяется таким образом:

$$D = \left\{ p_1^D, p_2^D, ..., p_q^D \right\}, \tag{3.9}$$

где p_i^D – подмножество элементов, необходимых для описания программных модулей, которые эти функции выполняют:

$$p_i^D = \{UID, PName, PMName, PUPath, PDes, SPath\}, \tag{3.10}$$

где *UID* – уникальный идентификатор ВП;

PName – имя процесса, используемое компоновщиком, при определении логической связи между вершиной графовой модели и программным модулем;

PMName – реальное имя программного модуля (или библиотеки);

PUPath – физический путь к файлу программного модуля;

SPath – физический путь к файлу спецификации, определяющей основные характеристики модуля, и является ссылкой на спецификацию модуля Sp_i вида (2.27);

PDes – дополнительные сведения об элементе.

Документ, предоставляющий проектировщику необходимую информацию для структурного синтеза ПО, может быть согласован по формату файла с форматом одного из XML-стандартов (например, языка GraphML).

Управляющий КА, синтез которого может быть выполнен на основании существующих методов [54], взаимодействует с информацией графовой модели и может быть реализован в виде программных библиотек. Эти библиотеки вместе с графовой моделью отчуждаются от основного инструментария разработчика, реализующего процессы сборки программных продуктов. В случае видоизменений требований заказчика к принципам работы ПО данные модели могут модифицироваться и вновь реализовываться в виде библиотек, поставляемых на сторону пользователей готовой системы и использоваться как обновления системы.

Для проектирования архитектуры исполнителя ВП в заданной ПрО, который опирается на управляющий КА, могут быть выбраны различные шаблоны, в зависимости от технологий последовательного, параллельного или распределенного программирования. Среди самых распространенных таких технологий можно выделить CORBA, DCOM, .Net Remoting.

DCOM, например, предоставляет инфраструктуру использования распределенных компонентов и определяет, каким образом компоненты взаимодействуют со своими клиентами. В случае территориального разделения клиента и компонента DCOM заменяет локальное соединение между процессами сетевым протоколом. При этом информация о способах организации сетевого соединения скрывается от объектов. Компонент, который разрабатывается как часть распределенного приложения, может быть повторно использован разработчиком, что дает преимущества в скорости разработки ПО.

Развитием технологий .Net Remoting и DCOM является модель WCF (Windows Communication Foundation), используемая для создания распределенных систем и осуществления интеграции изолированных систем, которые дают возможность программам, написанным на разных языках программирования, работающих в разных узлах сети, взаимодействовать друг с другом.

WCF используется для сервис-ориентированной архитектуры приложений, что позволяет абстрагироваться от технологий создания самого сервиса и использовать его из других приложений, разработанных с применением своих технологий.

В связи с явными преимуществами, стоит рассмотреть шаблон организации взаимодействия управляющего автомата и программных модулей с учетом использования технологии WCF (Приложение Г, рис. 4).

На диаграмме классов (Приложение Г, рис. 4) класс MCF службы ServiceMCF реализует интерфейс класса IServiceMCF с обязательными функциями GetData() и GetDataUsingDataContract(), организующими обмен информацией с клиентами. Последняя функция осуществляет прием и передачу параметров с использованием контракта данных, который используется для сериализации новых созданных сложных типов данных. Для класса ServiceMCF также задается процедура f_Call_SM() вызова управляющего автомата AUserSM с его методами.

Взаимодейсвие службы и программных модулей здесь также осуществляется посредством механизма классов-оберток. Но, в отличие от предыдущего шаблона, отображенного на диаграмме классов UML стандарта (Приложение Г, рис. 5), классы ClientMCF_Wrp1, ClientMCF_Wrp2, ClientMCF_WrpN самостоятельно информируют службу о состоянии ВП, после чего, служба передает необходимую информацию автомату AUserSM, который обновляет состояние графовой модели.

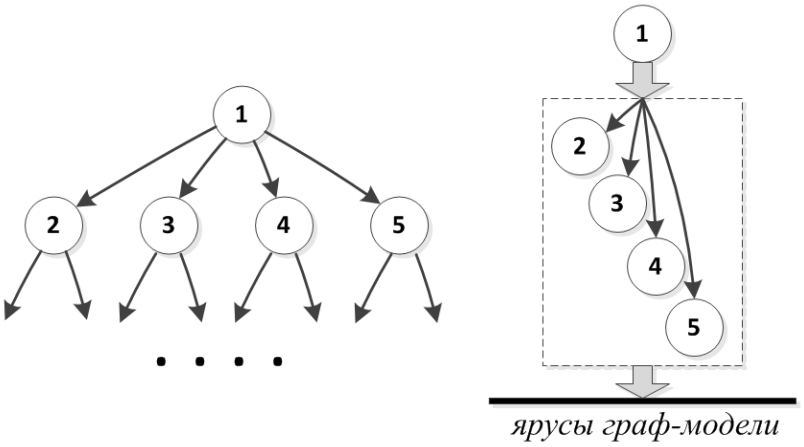
В свою очередь, технологии параллельных и распределенных вычислений применяются в зависимости от требований к задачам и условий их решения. Поскольку существуют различные архитектуры ВС: с распределенной или общей

памятью, гетерогенные или однородные ВС, – усовершенствование и применение этих технологий требует своего отдельного детального и глубокого изучения и соотносится с более низким уровнем проектирования и разработки ПО – например, задача динамического или статического отображения параллельной программы на ресурсы ВС [55]. Поэтому покажем в рамках каких шаблонов взаимодействия управляющего автомата AUserSM и программных модулей возможна реализация потенциала параллельного выполнения программного кода для предлагаемой ИТ.

Пусть задан класс AUserSM (Приложение Г, рис. 5), обеспечивающий работу КА в соответствии с логикой синхронного (процедура MethodNameSync()) или асинхронного (процедура MethodNameAsync()) способов исполнения программных модулей [48].

Класс AUserSM связывается с классами-обертками Wrp1, Wrp2,…, WrpN (wrapper), посредством которых осуществляются стыковка программных модулей ПО ИС и которые обеспечивают взаимодействие этих модулей и КА. PUnit1, PUnit2,…, PunitN являются классами программных модулей со своей собственной внутренней логикой. Предикат trg(cond) для класса-обертки задается условным выражением либо логической функцией cond. Основой работы этого предиката является подмножество m_i^{CG} (Приложение Г, рис. 4), которое фиксирует состояния самих ВП и архивов результатов выполнения этих процессов. На основании значений ST^{CG} и AR^{CG} формируется логическое высказывание cond. Если cond=true, тогда разрешается исполнение кода классов Wrp1, Wrp2, Wrp3. Call_PU(PU_InParam) – процедура, реализующая обращение к функциям библиотек программных модулей UsrFunc1(), UsrFunc2(), и др. Set_Mem() – процедура, изменяющая значение ячеек управляющей памяти для того, чтобы можно было запретить или разрешить исполнение какого-либо блока Wrp1, Wrp2,…, WrpN. Данная процедура также обращается к подмножеству m_i^{CG} (Приложение Г, рис. 4), чтобы присвоить соответствующие значения для атрибутов ST^{CG} и AR^{CG} в зависимости от результата выполнения.

В процедуре MethodNameAsync() закладываются средства управления классами-обертками и принципы синхронизации ВП, в соответствии с ярусно-параллельной графовой моделью программной архитектуры ИС. Асинхронная модель вычислений, закладываемая в этой процедуре, наиболее пригодна для описания вычислений на мультипроцессоре или мультикомпьютере. В такой модели определяется система независимо исполняющихся и взаимодействующих процессов (рис. 3.7, а).



а) б)

Рисунок 3.7 – Преобразование к синхронной модели исполнения программы:
а) каноническая ЯПФ; б) последовательное исполнение ВП, расположенных на одном ярусе

Процедура MethodNameSync() – реализует принцип синхронного исполнения ВП, когда следующий модуль начинает свою работу только по окончанию действий предыдущего модуля. Такой принцип может быть сориентирован на синхронное исполнение программ для каждого яруса ЯПФ графовой модели программной архитектуры (рис. 3.7, б).

В этом случае Wrp2 ожидает завершения выполнения Wrp1 и так далее. Такой шаблон может быть использован в рамках таких моделей параллельного программирования как MPI, OpenMP, HPF или POSIX Threads.

Определившись с моделями КА, используемых исполнителем ВП, опишем более детально процесс проектирования компоновщика ПО ИС

3.3 Проектирование программного инструментария «SWDesigner», реализующего задачи формирования программной архитектуры

Рассмотрим структуру разработанного компоновщика ПО, его основные задачи и средства встраивания инструментов управления ВП.

Архитектура программного инструментария, позволяющего выполнить сборку ПО ИС по предоставляемой информации о ПрО, разработана в соответствии со стандартом IEEE P1484.1/D11 (Приложение Г, рис. 6). На основании введенной формализованной графовой модели, которую разработчик может видеть в графическом редакторе инструментария (Приложение Г, рис. 6), осуществляется ввод программных спецификаций и программных модулей в систему до начала процесса комплексирования. Для каждого программного модуля оформляется класс-обертка (wrapper), обеспечивающий стыковку базовых программных модулей на основе выбранных стандартов взаимодействия программных библиотек. Стандарты взаимодействия задаются спецификациями. Также на основе информации ярусно-параллельной графовой модели программной архитектуры разрабатывается библиотека управляющего КА, реализуемого в виде сервиса. Данный сервис осуществляет управление и взаимодействие элементов формируемого ПО. На конечном этапе ПО ИС формируется последовательной компиляцией отдельных программных библиотек, а потом и всей системы.

Для выполнения более детального описания структуры инструментария «**SWDesigner**» в рамках объектно-ориентированного проектирования выделим его основные функции и зададим их при помощи диаграмм классов.

Функции работы с графовой моделью архитектуры ПО: 1) конденсация (объединение вершин) графовой модели (Приложение А); 2) приведение модели к канонической ЯПФ; 3) специфицирование графовой модели.

Основные функции компоновщика: 1) формирование класса-обертки для стороннего программного модуля, обеспечивающего регламентированный вызов его процедур и управление исполнителем; 2) специфицирование вызова

программного модуля; 3) компиляция отдельного программного модуля; 4) компиляция программной архитектуры ИС.

Для реализации функций обработки графовой модели необходимы следующие классы: *Condensation* (представляет набор функций по обработке графовой (2.10) модели), *AdjacencyMatrix* (регламентирует стандарт матрицы смежности), задаваемой в табличном формате), *ProjectStageConverter* (ведет стадийность проекта компоновки ПО в зависимости от результатов обработки графовой модели) (рис. 3.8).

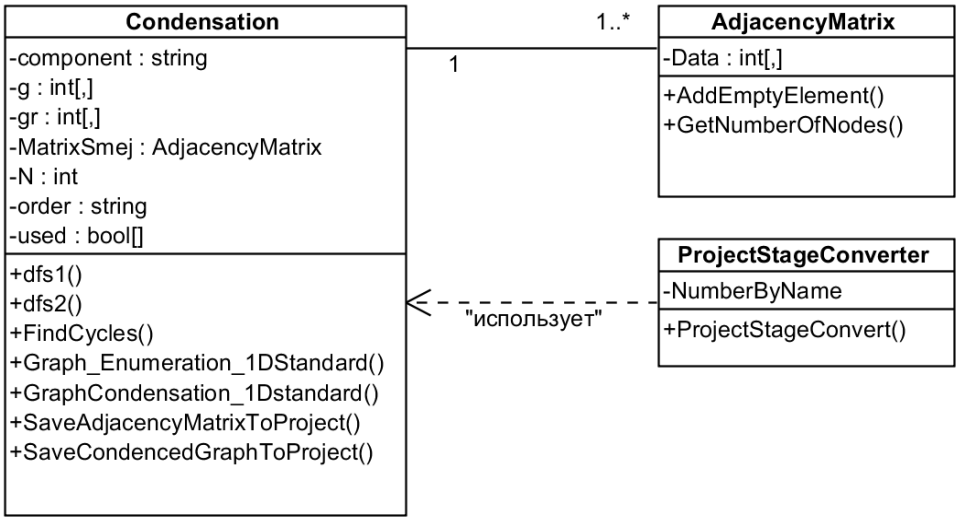


Рисунок 3.8 – Структура программного модуля конденсации графовой модели

Для класса *Condensation* задаются следующие атрибуты: *g* – массив содержащий копию исходной матрицы смежности графовой модели; *gr* – транспонированная матрица смежности; *N* – размерность матрицы; *order* – вектор, содержащий упорядоченную последовательность вершин согласно топологической сортировке; *used* – вектор,содержащий вершины, помеченные как пройденные при поиске компоненты сильной связности; component – вектор, содержащий вершины обнаруженной компоненты.

Для транспонированной матрицы смежности характерным является обратное отношение между вершинами графовой модели по отношению к исходной матрице. В отличие от обычной матрицы смежности в первом столбце стоит индекс той вершины, которая является управляемой относительно предшествующей вершины, указанной для текущей записи во втором столбце.

Другими словами, моменты времени активации и последовательности работы программных модулей меняются местами. Операции для класса следующие: *dfs1*() – осуществляет поиск компоненты на графе в ширину; *dfs2*() – осуществляет поиск в глубину (что это за функции и в рамках каких алгоритмов используются); *FindCycles*() – функция, реализующая функции *dfs1*() и *dfs2*(); *Graph_Enumeration_1DStandard*() – топологическая сортировка на графе, заданном в списочном виде; *GraphCondensation_1Dstandard*() – конденсация графа, заданного списком; *SaveAdjacencyMatrixToProject*() – сохранение исходного графа в файл проекта с установкой соответствующей стадии; *SaveCondencedGraphToProject*() – сохранение конденсированного графа в файл проекта с установкой соответствующей стадии. Топологическая сортировка, предпринимаемая для графовой модели программной архитектуры, подразумевает упорядочивание вершин ацикличного ориентированного графа согласно частичному порядку, заданному ребрами орграфа на множестве его вершин (Приложение Г, рис. 7). В качестве алгоритма берется алгоритм Тарьяна [43, 156]. Данный алгоритм выполняется за $O(n)$ времени и памяти, сортирующий вершины по времени выхода в поиске в глубину. То есть при помощи алгоритмов, реализованных функциями *dfs1*() и *dfs2*(). Для данного алгоритма используется принцип раскраски вершин. Изначально считается, что все вершины белые. Из каждой вершины проводится обход в глубину. При входе в вершину она становится серой, при выходе – чёрной. Черная вершина заносится в окончательный список отсортированных вершин. Если в процессе обхода алгоритм попадает в серую вершину – выдается сообщение о найденном цикле и невозможности дальнейшей сортировки. Для класса *ProjectStageConverter* (рис. 3.8) заданы такие атрибуты и операции: *NumberByName* – получает номер стадии по переданному имени для функции *ProjectStageConvert*(). Для визуализации (Приложение Г, рис. 1) графовой модели на экране инструментария создаются классы графических элементов (вершины и дуги) в соответствии со стандартом WPF языка программирования C#. Для базового класса *Vershina* (Приложение Г, рис. 1) задаются такие атрибуты и операции: *Canvas* – графический элемент,

объект для рисования; *Diam* – диаметр окружности вершины; *VNumber* – номер вершины, задаваемый пользователем на этапе ввода матрицы смежности; *Xpos* и *Ypos* – координаты размещения графических элементов на холсте *Canvas*; *Redraw*() – функция перерисовки вершины при изменении ее параметров. Класс *Vershina_wpf* определяется такими атрибутами и операциями: *ArrowRefObjectIn* – ссылка на графический элемент *Arrow_wpf*, входящий в данную вершину; *ArrowRefObjectOut* – ссылка на графический элемент *Arrow_wpf* , исходящий из данной вершины; *mAdjacencyMatrix* – матрица смежности. *SemanticName* – реальное физическое имя процесса или модуля, на который могут ссылаться несколько синтаксических имен; SyntacticalName – виртуальное имя процесса или модуля, которое должно быть уникальным; NYarus – номер яруса ЯПФ, на котором размещается данная вершина; ProjectPath – файл проекта или графовой модели, для которых определяется и в которых хранится графическая информация; *Vershina_wpf*() – конструктор; *element_MouseDown*() – захват курсора мыши; *element_MouseMove*() – обработка события перемещения курсора мыши; *GetIndexOfVertex*() – получение номера вершины; *menuItem1_Click*() – получаем подграф, состоящий из пограничных вершин; *menuItem2_Click*() – здесь получаем подграф зависимостей вплоть до корневой вершины. Класс SuperVershina определяется таким образом: component – компонента сильной связности, в переменной перечислены вершины, принадлежащие супервершине после конденсации; *subGraph* – матрица смежности, сохраняющая подграф; *SuperVershina*() – конструктор класса.

Класс *AdjacencyMatrix* задает стандарт хранения матрицы смежности (атрибут Data) и задает основные функции работы с массивом (*AddEmptyElement*() и *GetNumberOfNodes*()).

Класс Arrow – является базовым классом для графического элемента «дуга» и задается при помощи следующих атрибутов и операций: *Canvas* – объект для рисования, холст; *Xbeg, Ybeg* – координаты начала дуги; *Xend,Yend* – координаты конца дуги.

Класс *Arrow_wpf* – класс производный от базового, определяющийся такими атрибутами и операциями: *VershinaFromRef* – определяет ссылку на вершину, из которой исходит дуга; *VershinaToRefference* – определяет ссылку на вершину, в которую дуга входит; *DrawLineWithWings*() – определяет графическую форму стрелки; функции *OnMove_ArrowIn*() и *OnMove_ArrowOut*() определяют перерисовку дуги при перемещении вершины по рабочей области экрана; *Arrow_wpf*() – конструктор класса. Для определения принципов взаимодействия программных модулей с пользовательскими окнами и диалогами задается такой класс как *_wndOptions*, производный от системного класса окон Window (Приложение Г, рис. 2). Кроме этого для отображения информации графовой модели программной архитектуры ИС дополнительно определяется класс *UIProcess*. Для *_wndOptions* класса задаются такие атрибуты и операции: *_entity* – сущность, ВП, реализуемый соответствующим программным модулем; *_FileProjectPath* – атрибут класса, определяющий путь к файлу проекта, содержащего информацию о графовой модели; *tmp_FileOptions* – файл настроек пользовательского окна, графически отображающего диаграммы процессов; *btCancel_Click*() и *btSave_Click*() – функции стандартной обработки действий пользователя по отмене или сохранению информации в файл проекта по конкретной диаграмме; *btCompile_Click*() – функция, осуществляющая компиляцию ПО; *btSetSpecification_Click*() – функция, открывающая диалоговое окно для ввода программной спецификации; *btUnitOpen_Click*() – открытие редактора программного кода; *ThrowEvent_OnUpdateParam*() – функция генерирует событие об изменении параметров редактируемого процесса с тем, чтобы подписчик на это событие осуществил сохранение данных и перерисовку графических элементов рабочей области окна пользовалеля. *_wndOptions*() – конструктор класса, считывающего информацию из файла проектов по текущему процессу и отображающий данные в диалоге. Для класса *UIProcess* задаются следующие атрибуты и операции: *CorrespondingNode* – ссылка на вершину, которая соответствует данному вычислительному процессу; *CurrProjectFilePath* – путь к файлу проекта; Element_Name – ссылка на графический элемент,

прорисовывающий процессный блок; *MmatrixSmej* – матрица смежности, полученная из файла проекта; *ProcessDescription* – описание ВП, *ProcessName* – имя ВП (семантическое имя); *ProgrammModuleName* – имя программного модуля, физический файл, синтаксическое имя; *ProgramUnitPath* – путь к файлу программного модуля; SpecificationFilePath – путь к программной спецификации, описывающей данный программный модуль; wndOptions – конструктор окна; *DrawUIProcess*() – функция, прорисовывающая графический элемент; *MouseLoadOptions*() – вызов всплывающего меню; *ThrowEvent_OnUpdateFileOfProject*() – генерация сообщения о том, что произошло обновление файла проекта и подписчику надо обновить графические элементы процессов на рабочей области; UIProcess() – конструктор графического элемента. Таким же образом определяется взаимодействие между описанными классами и классами окон пользовательского приложения и интерфейсных функций (Приложение Г, рис. 3). Спецификация программного модуля имеет структуру (Приложение Д, «Спецификация программного модуля»), которая позволяет задать имена вызываемых функций, переменные и тип входных и выходных параметров функций, что позволяет определить для каждой вершины графовой модели векторы зависимостей по входным данным.

Для предложенной архитектуры исполнителя ВП задан алгоритм работы пользователя с программным иструментарием (Приложение Г, рис. 8). Шаги «Анализ ПрО», «Формализация ПрО» и «Ввод матрицы смежности» следует отнести к 1-ому этапу приведенной ИТ. Шаги «Проверка данных», «Исправление данных», «Сохранение модели» и «Анализ графовой модели на наличие циклов» относятся ко 2-ому этапу. Шаги «Конденсация графовой модели» и «Формирование ЯПФ» с дальнейшим сохранением информации относятся к 3-ему этапу. Шаги «Ввод спецификаций ПМ» или загрузка существующих данных, «Ввод характеристик ПМ» или загрузка уже существующих характеристик относятся к 4-ому этапу. К 5-ому этапу будут относиться шаги «Ввод условий комплексирования ПМ» и «Загрузка существующих условий комплексирования». Шаги «Формирование автоматной модели» и ее «Верификация» относятся

соответственным образом к 6-ому и 7-ому этапам. Последний 8-ой этап – это шаг «Компиляция ПС». При этом на предыдущем шаге анализа данных возможна итерация, возвращающая разработчика на шаги по видоизменению графовой модели с целью ее оптимизации.

3.4 Выводы по разделу 3

В разделе решены такие задачи:

1. Предложен автоматный метод проверки выполнения ограничений к формируемому ПО, выраженных в форме нефункциональных требований. Метод включает этапы формирования автоматной модели взаимодействия модулей и сравнения полученных в результате моделирования последовательностей взаимодействия модулей с нефункциональными требованиями к программному обеспечению. Сами требования конечного пользователя формализуются на базе темпоральной логики линейного времени LTL. Метод позволяет сформировать ограничения на структуру ПО, учитывающие нефункциональные требования, и, тем самым, повысить надежность его функционирования.

2. Показан процесс верификации автоматной модели при помощи ферификатора JSPIN на основе формул LTL и программным описанием автоматной можели на языке PROMELA.

3. Расчитана врменная сложность автоматного метода проверки выполнения ограничений к формируемому ПО, которая составила величину $O(n^3 \cdot \log k)$. Повышение надежности ПО, верифицируемого предложенным методом достигается засчет строго детерминированного количества состояний автоматной модели и покрытия тестами использующейся автоматной модели.

4. Разработана программная архитектура компоновщика ПО ИС «SWDesigner» и исполнителя ВП, осуществляющего контроль работы программных модулей на базе ярусно-параллельной графовой модели программной архитектуры.

РАЗДЕЛ 4
ИНФОРМАЦИОННАЯ ТЕХНОЛОГИЯ СТРУКТУРНОГО СИНТЕЗА ПРОГРАММНОЙ АРХИТЕКТУРЫ И ЕЕ ВНЕДРЕНИЕ В ПРОЦЕСС ПРОИЗВОДСТВА СЛОЖНЫХ ЭЛЕКТРОННЫХ УСТРОЙСТВ

4.1 Разработка информационной технологии структурного синтеза программной архитектуры информационной системы

Предложенная ИТ структурного синтеза программной архитектуры ИС в условиях изменяющихся требований позволяет отразить требования текущей версии на графовой модели и обеспечить переконфигурирование модулей ПО, а также снизить его функциональную и структурную сложность. Предложенные модель, методы и ИТ реализованы в виде программного инструментария «SWDesign», компоновщика программной архитектуры, концептуальная схема которого отображает принцип построения ПО на базе ярусно-параллельной графовой модели программной архитектуры ИС (рис. 4.1).

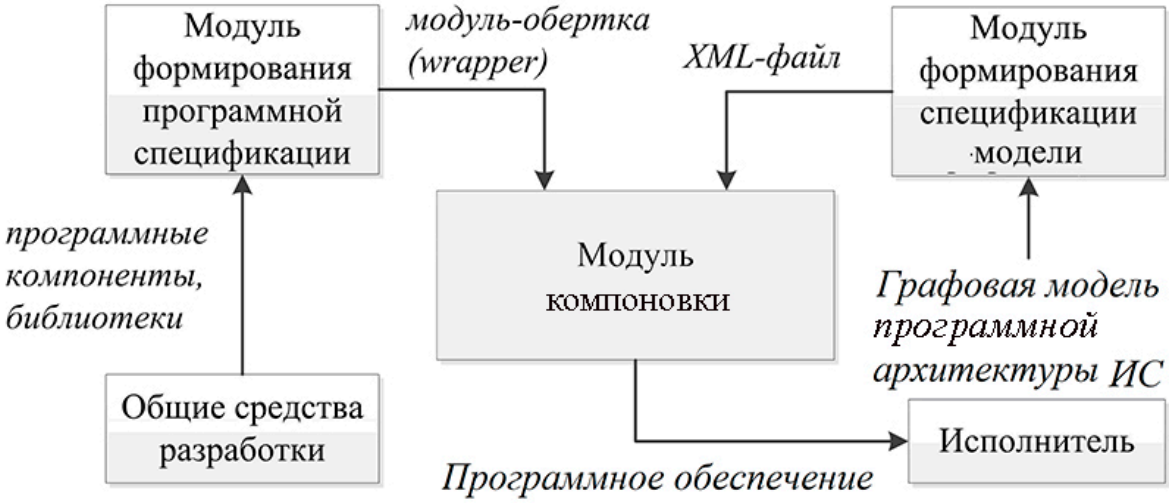


Рисунок 4.1 – Концептуальная схема компоновщика программной архитектуры ИС

устанавливает взаимодействие модуля, выполняющего компиляцию компонентов ПО, программной спецификации и графовой модели архитеткруы ПО ИС.

Модуль формирования программной спецификации позволяет разработчику получить класс-обертку для библиотек, из которых в дальнейшем будет

формироваться программный функционал, на основании спецификаций этих библотек. В свою очередь, такие классы (wrapper) стыкуются между собой на основании информации, получаемой из графовой модели. Модуль формирования спецификации на выходе получает XML-файл, являющийся основой для проекта комплексирования программного продукта.

Для формирования программной архитектуры ИС используется предложенная ИТ, которая включает в себя такие этапы (рис.4.2).

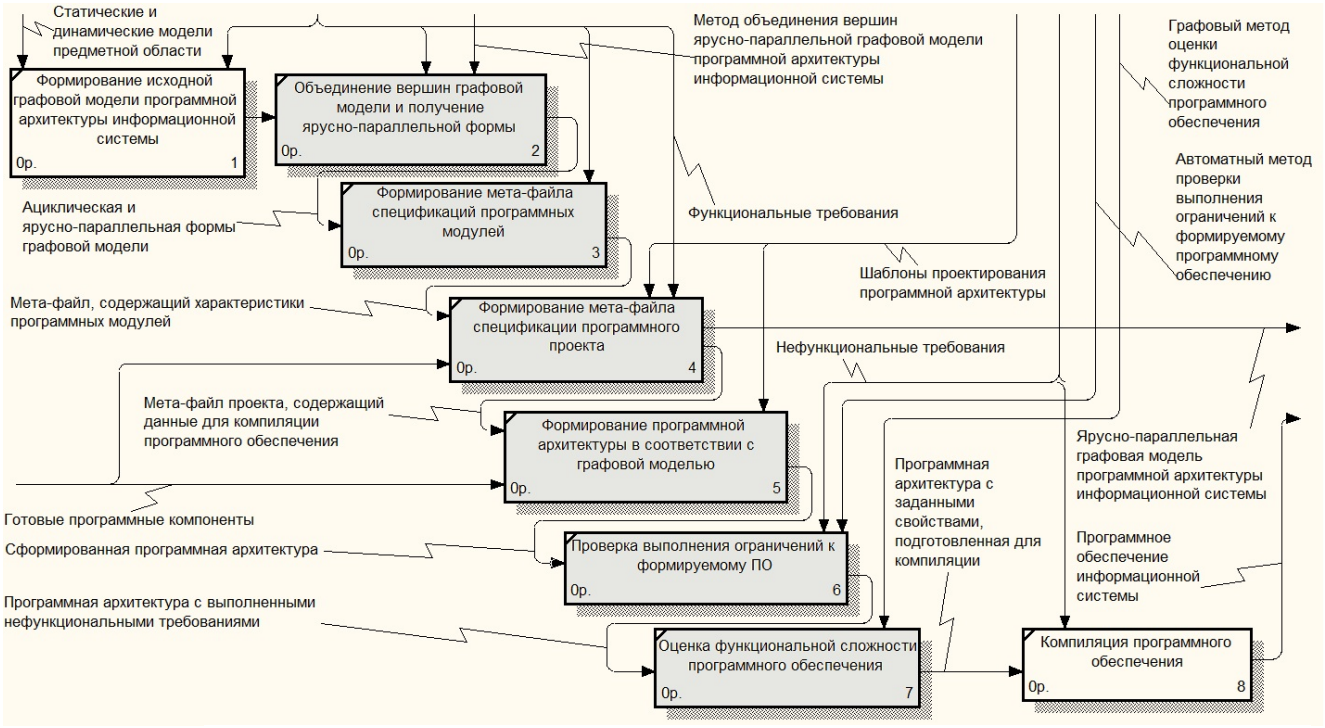


Рисунок 4.2 – Этапы ИТ структурного синтеза программной архитектуры ИС

Этап 1. Формирование исходной графовой модели программной архитектуры информационной системы.

Этап 2. Объединение вершин графовой модели и получение графовой ярусно-параллельной модели.

Этап 3. Формирование мета-файла спецификаций модулей.

Этап 4. Формирование мета-файла спецификации программного проекта.

Этап 5. Формирование программной архитектуры в соответствии с графовой моделью.

Этап 6. Проверка выполнения ограничений к формируемому программному обеспечению.

Этап 7. Оценка функциональной сложности программного обеспечения.

Этап 8. Компиляция программного обеспечения. Завершение метода.

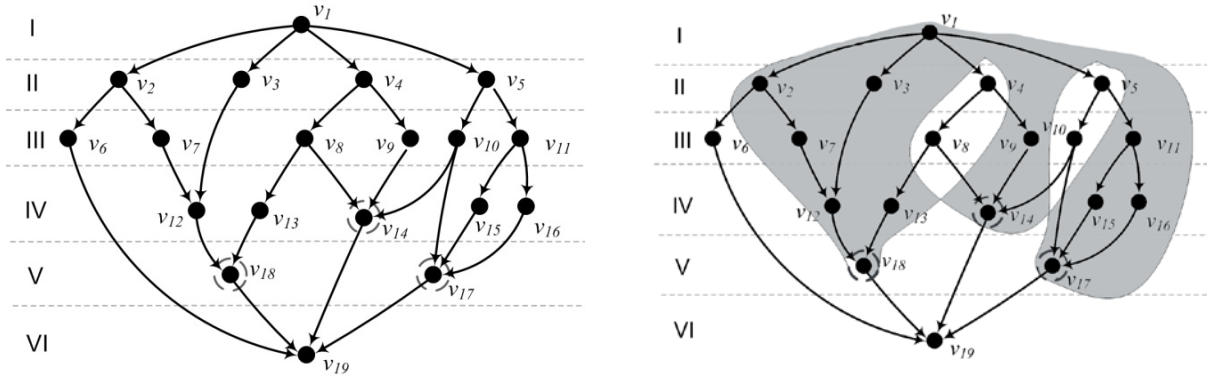
Первый этап ИТ обеспечивается информацией из доступных документов, описывающих требования заказчика: первичные спецификации, диаграммы статического и динамического моделирования ПрО. На основе функциональных и архитектурных требований к ПО для сущностей модели ПрО определяются функции объектов, формирующих графовую модель. Эта модель на втором этапе проверяется на топологическую корректность, а также упрощается структурная сложность графовой модели на основе требований заказчика с применением метода объединения вершин ярусно-параллельной графовой модели, формируется рациональная структура графа. Задачи, которые решаются на втором этапе – это устранение топологических некорректностей, выявление циклов (компонент сильной связности) заданной графовой модели. Если обнаруживаются изолированные или висячие вершины со свойствами (2.11, 2.12), то происходит анализ причин, по которым это произошло, и такие ошибки устраняются. Также проверяется наличие парных дуг и петель для вершин при проверке их на свойства (2.13, 2.14). На этом этапе для всех вершин графовой модели формируются множества кортежей данных вида C_i (2.28), которые включают в себя информацию, в том числе, и о входной и выходной информации программных модулей: массив выходных параметров $Data\_Res$ (2.29) и массив входных параметров $Data\_Inp$ (2.30). На третьем этапе на базе полученной структуры производится описание программных модулей при помощи мета-файлов спецификаций. Спецификации программных модулей формируются на основе данных, которыми штатные или сторонние разработчики снабжают свои программные модули, оформляя их в соответствующих стандартных документах в зависимости от выбранной технологии разработки ПО. При этом формируется файл-спецификация, формат которого сообразен кортежу p_i^D для каждого i-го программного модуля и вершины графа. Здесь значение переменной $PName$ (синтаксическое имя) может оставаться пустым или временно совпадать со

значением переменной *PMName* (семантическое имя) вплоть до 4-го этапа, где к синтаксическому имени предъявляется требование уникальности в рамках всей графовой модели. Также атрибут *SPath* кортежа p_i^D должен ссылаться на кортеж данных C_i для указания параметров исполяемой функции. На пятом этапе с использованием архитектурных шаблонов формируется необходимый для генерации ПО и динамического конфигурирования мета-файл программного проекта, который описывает принципы взаимодействия модулей. Данный файл на 6-м этапе является входным для процесса проверки выполнения ограничений к ПО на основе верификации последовательности взаимодействия модулей и соответствующего автоматного метода. Выходной информацией этого метода будет обобщенная спецификация проекта, объединяющая всю информацию, описанную конденсированной графовой моделью. Формат данной спецификации включает в себя информацию, главным образом, о стадиях проекта, по количеству и смысловому значению совпадающих с этапами предложенной ИТ, характеристики вершин графовой модели (2.10), описывающих наличие компонент сильной связности; матрицу смежности исходной не конденсированной графовой модели (2.2), матрицу смежности конденсированной графовой модели (2.1); характиристики функций, сопоставленных с вершинами графовой модели, и также техническую информацию, обеспечивающую интерфейс пользователя с компоновщиком ПО ИС. На седьмом этапе после применяется графововый метод оценки функциональной сложности ПО, после успешного выполнения которого на 8-м этапе выполняется генерация ПО и отчуждение графовой модели для конечного пользователя сформированной системы. В случае новой версии требований процесс повторяется с третьего этапа.

В процессе верификации программной архитектуры на 6-ом этапе разработчик берет за основу автоматную модель объединяющую в себя управляющий КА, КА, обеспечивающий исполнение ВП (*AForwardSM*) и КА, обеспечивающий откат ВП до заданной узловой точки процесса и восстановление исходных данных (*ABackwardSM*). Эту модель он адаптирует к формируемой программной архитектуре. На этой архитектуре в соответствии со

спецификациями программных модулей определяются узлы, обеспечивающие интерфейс пользователя – так называемые базовые вершины (*BaseNode*), начиная с которых формируется подграф текущего ВП. С этим подграфом уже работают КА *AForwardSM* и *ABackwardSM*. Другими словами, графовая модель, положенная в основу работы исполнителя ВП, дает возможность сформировать набор функций, которые предшествуют выполнению текущей задачи для активного пользователя и соответствующей ему базовой вершины путем выделения подграфа на графовой модели [82]. Программа, согласно хранящемуся в базе данных описанию, определяет подграф $G'=(V',X')$ на графовой модели программной архитектуры (2.10), для которого $V' \subseteq V$ и $X' \subseteq X$.

То есть, если для некоторого программного модуля существуют три пользователя, ответственных за выполнение своих задач, тогда зададим для графовой модели три базовые вершины v_{14}, v_{17} и v_{18} (рис. 4.3), которые реализуют диалог с пользователем.



а) б)

Рисунок 4.3 – Ориентированный граф, моделирующий структуру разрабатываемого программного средства: а) – ЯПФ графа с обозначенными вершинами, реализующими пользовательский диалог. б) – ЯПФ графа с выделенными подграфами

На основании этих вершин возможно выделение подграфов задач, решаемых пользователями системы независимо друг от друга или в зависимости. Поэтому можно выделить такие подграфы G_1, G_2 и G_3, для которых существуют следующие подмножества вершин:

$$G_1 : V_1 = \{ v_1, v_2, v_3, v_4, v_7, v_8, v_{12}, v_{13}, v_{18} \};$$
$$G_2 : V_2 = \{ v_1, v_4, v_5, v_8, v_9, v_{10}, v_{14} \};$$
$$G_3 : V_3 = \{ v_1, v_5, v_{10}, v_{11}, v_{15}, v_{16}, v_{17} \}.$$

Анализируя полученные подмножества, выявляем такие пересечения:

$$V_1 \bigcap V_2 = \{ v_1, v_4, v_8 \};$$
$$V_1 \bigcap V_3 = \varnothing;$$
$$V_2 \bigcap V_3 = \{ v_1, v_5, v_{10} \}.$$

Поскольку v_1 является фиктивной вершиной, интерес представляют вершины v_8, v_4 и v_5, v_{10} . Важно отметить, что для этих вершин возможны три состояния:

1) программные модули используют один и тот же выходной набор данных, который затем обрабатывается последующими программными модулями;

2) программные модули используют разный выходной набор данных для последующих модулей;

3) программные модули позволяют разделять часть своего набора выходных данных между последующими программными модулями.

Полученное формальное описание программной спецификации (2.28) позволяет выявить что:

$$\left[\begin{array}{l} \bigcap\limits_k Sp_k(D^{in}) = \varnothing, \forall Sp_k(D^{in}) \subset Sp_i(D^{out}) \\ \bigcap\limits_k Sp_k(D^{in}) \subseteq Sp_i(D^{out}) \end{array} \right. , \qquad (4.1)$$

где k – является индексом вершин, для которых существует дуга $x_{ik} = (v_i, v_k)$. Вершины графовой модели, отвечающие условию (4.1) теперь должны быть еще раз распределены по ярусам ЯПФ графовой модели программной архитектуры, но уже с учетом свойств (2.17 – 2.19) в целях правильной организации последовательности выполнения функций.

В конце 6-го этапа для каждой базовой вершины определяется алгоритм формирования подграфа задачи (рис. 4.4), на который опирается работа КА *AForwardSM* и *ABackwardSM*.

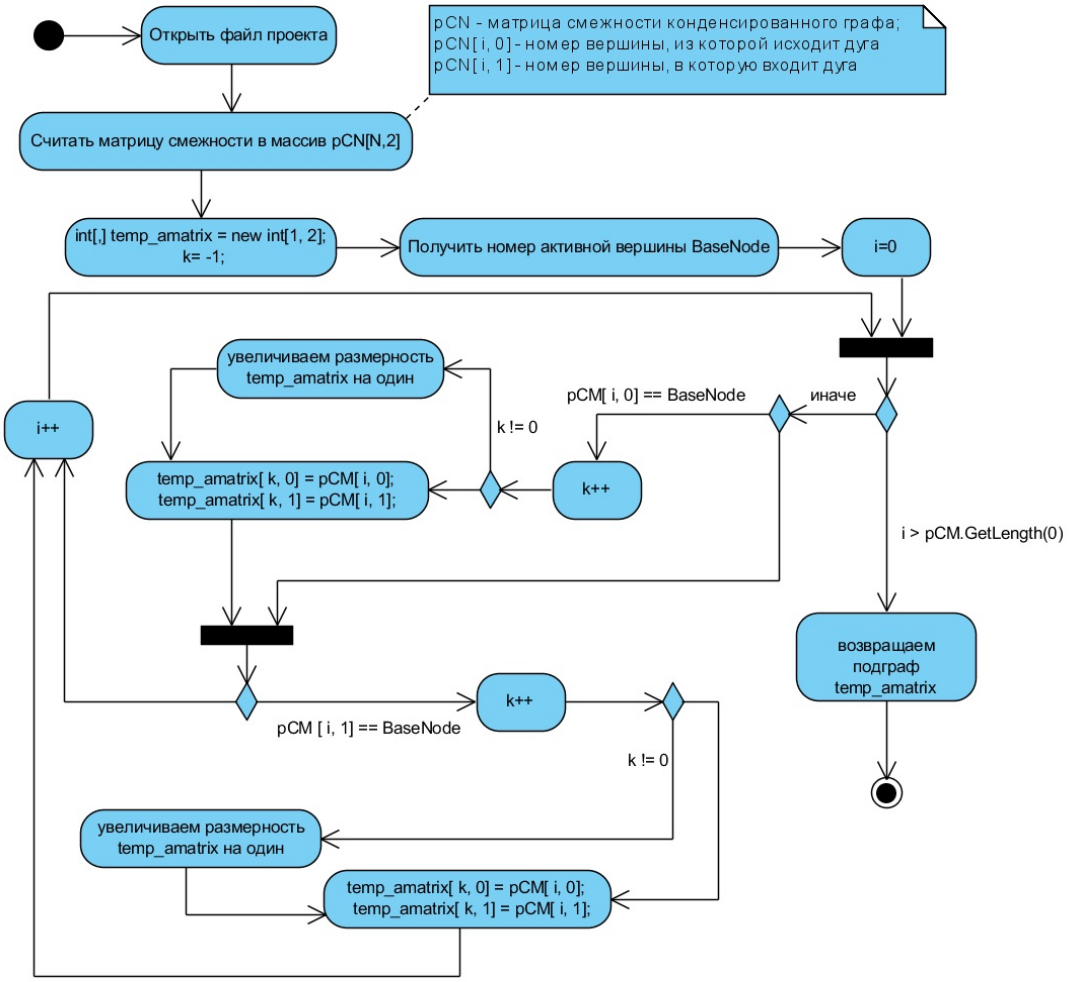


Рисунок 4.4 – Алгоритм формирования подграфа задачи

Алгоритм основывается на определении базовой вершины графовой модели, соответствующей активному процессу текущего пользователя, который запускает окно диалога для решения своих задач (Basenode). Далее, на основании матрицы смежности всей графовой модели, начиная с базовой вершины, программа формирует подграф до корневой вершины графовой модели.

Предложенная ИТ позволяет осуществить структурный синтез графовой модели программной архитектуры и динамически выполнить конфигурирование его архитектуры и функциональности рабочих мест в условиях изменяющихся

требований заказчика, а также снизить затраты на реализацию функциональных задач ИС путем снижения функциональной и структурной сложности.

4.2 Разработка архитектуры исполнителя вычислительных процессов в рамках заданной модели предметной области

Работа исполнителя ВП опирается на графовую ярусно-параллельную модель программной архитектуры (2.10), отображающую информацию о состоянии выполняемых задач. Формат, который использован для сохранения информации графовой модели, аналогичен языку GraphML. Файл с такой структурой служит основой для компоновщика программной архитектуры и является поставщиком информации далее для исполнителя, но уже в виде хранилищ данных, в которые эта информация переносится из файла графовой модели. Сама же архитектура исполнителя строится на основе модели КА. Рассмотрим архитектуру исполнителя и каким образом используются стандарты XML для обеспечения хранения информации проекта, как эта информация используется исполнителем ВП при выполнении функциональных задач ИС.

Так как язык GraphML [35] является одним из наиболее распространенных и удобных форматов описания графов, целесообразно им воспользоваться для описания надстроек и модификаций, позволяющих получить файл проекта. Такой формат позволяет описывать иерархические и вложенные структуры графов, задавать веса для дуг и направления. Документ формата GraphML содержит корневой элемент «*graph*», описывающий граф в целом, с вложенными элементами «*node*» – вершина и «*edge*» – ребро, с соответствующими атрибутами, которые также подразумевают хранение вложенных структур.

Основной документ, ведущий проектировщика через стадии проекта и предоставляющий необходимую информацию для структурного синтеза ПО ИС, согласовывается по формату файла с форматом языка GraphML с некоторыми модификациями (табл. 4.1).

Таблица 4.1 – Основные разделы файла проекта компоновщика

Раздел	Значение
<*Node*>	Хранение информации о расположении графических элементов на форме редактора (координаты)
<*Staging*>	Стадийность документа, определяющая основные стадии прохождения проекта от исходной графовой модели программной архитектуры до его формирования
<*VertexCharacteristics*>	Характеристики супервершин. Содержится информация в виде матрицы смежности о подграфе, свернутом в супервершину
<*AdjacencyMatrix*>	Матрица смежности исходного графа
<*CondencedGraph*>	Матрица смежности конденсированного графа
<*Diagram*>	Содержится техническая информация о программных модулях, оформления оберток модулей и правилах их стыковки

Элемент «*AdjacencyMatrix*» и «*CondencedGraph*» по своей структуре близки элементу «*graph*» языка GraphML, с той лишь разницей, что матрица смежности задается в списочном виде и подразумевает задание только ориентированного графа. Информации о графе достаточно чтобы выполнить операцию автоматического преобразования данных файла проекта к формату языка GraphML и наоборот. Реализация этой процедуры достигается путем использования программной библиотеки конвертера, которая разработана для компиляции программной архитектуры.

Элемент <*Node*> сопоставляется с подмножеством *N* (3.3), в котором каждый элемент определяет координаты *X* и *Y* соответствующей ему вершины:

```
<Node>
 <M1 X="304,859" Y="17,742" />
 <M2 X="167,859" Y="88,742" />
… …
</Node>
```

Количество атрибутов элемента <*Node*> совпадает с количеством строк матрицы смежности графовой модели.

Элемент <*VertexCharacteristics*> сопоставляется с подмножеством *VC* (3.5), для элементов которого задаются два атрибута, один из которых логического типа

«isHavingSubGrap»; и второй – «Component», который перечисляет номера вершин, входящих в компоненту сильной связности соответствующую текущей супервершине:

<*VertexCharacteristics*>
<M13 isHavingSubGrap="true" Component="16-14" />
<M8 isHavingSubGrap="true" Component="10-9-11" />
</*VertexCharacteristics*>

Здесь также количество атрибутов равно количеству строк матрицы смежности графа.

Элемент < *AdjacencyMatrix* > соответствует матрице смежности, заданной как *AM* (3.6). Атрибуты этого тэга позволяют задать матрицу в списочном виде:

<*AdjacencyMatrix*>
<M1 M3="true" M4="true" M6="true" />
<M2 M9="true" />
<M3 M9="true" M12="true" />
… …
</*AdjacencyMatrix*

В этом примере значение атрибутов M3="true", M4="true", M6="true" и т. д. означает наличие ориентированной дуги из вершины с индексом 1 (M1 – это имя атрибута элемента <*AdjacencyMatrix*>) в вершины с индексами 3, 4 и 6.

Элемент <*CondencedGraph*> позволяет аналогичным образом, как в случае с элементом <*AdjacencyMatrix*>, задать матрицу смежности для уже конденсированной графовой модели в соответствии с подмножеством *CG* (3.7):

<*CondencedGraph*>
<M1 M5="true" M9="true" VState="0" VArchive="0" />
<M2 M9="true" VState="0" VArchive="0" />
… …
</*CondencedGraph*>

Атрибуты «VState» и «VArchive» принимают свои значения в соответствии с определенными подмножествами состояний ВП ST^{CG} и характеристик архивов AR^{CG} (3.8). Значение «0» соответствует еще незапущенному ВП и невыполненной архивации результатов ВП; «1» означает активное состояние ВП и активный процесс архивации данных; значение «2» соответствует состоянию завершенности ВП и наличию данных архива для соответствующих подмножеств.

С помощью элемента <*Diagram*> устанавливается однозначное соотношение между вершиной и программным модулем вместе с множеством его характеристик, описанных атрибутами p_i^D в соответствии с кортежом D :

<*Diagram*>
<Process1 UID="0" PName="Процесс № 1" PMName="POlap" PUPath="C:\temp\f1" SPath=" C:\temp\sp1" PDes="" />
… …
</*Diagram*>

Значение атрибута *UID* является уникальным и содержит индекс той вершины, с которой соотносится данный программный модуль, а атрибут PName содержит уникальное имя ВП.

Помимо основных разделов, описывающих структуру графовой модели и вложенность подграфов на этой структуре, в файле проекта необходимо иметь сведения о стадиях проекта и программных модулях, участвующих в процессах формирования программной архитектуры ИС.

Для описания стадийности проекта согласно кортежу *S* принимаются к рассмотрению семь следующих атрибутов, которые соответствуют этапам применяемой ИТ: S1_notverified – начальная стадия, на которой осуществляется ввод матрицы смежности графовой модели; S2_verified – стадия, на которой выполняется проверка корректности введенной информации; S3_condensed – стадия конденсации графа и формирование вложенных структур графовой модели; S4_structured – стадия определения программных модулей и ввода для них спецификаций; S5_linked – стадия формирования структуры «модуль-вершина», на которой определяются связи между модулями на основе спецификаций; S6_modeled – стадия формирования LTL-спецификаций, тестового кода, автоматных моделей и процесса верификации формируемого ПО; S7_completed – стадия компиляции ПО. В этой связи определяется элемент <*Staging*>, атрибуты которого принимают одно из двух допустимых значений: «*» или «пусто», что соответствует значениям «true» – проект находится на этой стадии или «false» – проект находится на другой стадии:

<*Staging*>
<S1_notverified Current="" />

```
        <S2_verified Current="*" />
        … …
         </Staging>
```

Приведем пример содержания такого файла проекта:

```
<?xml version="1.0" ?>
<!—Файл проекта-->
 <Project>
 <Node>
         <M1 X="304,859" Y="17,742" />
          <M2 X="167,859" Y="88,742" />
          <M3 X="319,859" Y="80,742" />
        <!—координаты вершин графа на рабочей области формы-->
  </Node>
<AdjacencyMatrix>
          <M1 M3="true" M4="true" M6="true" />
          <M2 M9="true" />
          <M3 M9="true" M12="true" />
          <M4  M2="true"  M7="true"  M8="true"  M9="true"  M11="true"  M12="true"
M14="true" />
        …..
</AdjacencyMatrix>
 <Staging>
         <S1_notverified Current="" />
          <S2_verified Current="*" />
          <S3_condensed Current="" />
          <S4_structured Current="" />
          <S5_linked Current="" />
          <S6_modelled Current="" />
          <S7_completed Current="" />
  </Staging>
 <VertexCharacteristics>
          <M13 isHavingSubGrap="true" Component="16-14" />
          <M8 isHavingSubGrap="true" Component="10-9-11" />
  </VertexCharacteristics>
<CondencedGraph>
        <M1  M5="true"  M9="true"  M2="true"  M8="true"  M11="true"  M14="true"
VState="0" VArchive="0" />
        <M2 M9="true" VState="0" VArchive="0" />
        <M5  M2="true"  M9="true"  M10="true"  M11="true"  M15="true"  VState="0"
VArchive="0" />
        …….
  </CondencedGraph>
 <Diagram>
           <Process1          UID="0"          ProcessName="Процесс          №          1"
```

```
        ProgrammModuleName="POlap"  ProgramUnitPath="C:\temp\f1"
        SpecificationFilePath=" C:\temp\sp1" ProcessDescription="" />
         <Process2  UID="1"  ProcessName="Процесс  №  2"
        ProgrammModuleName=""  ProgramUnitPath=""  SpecificationFilePath=""
        ProcessDescription="" />
</Diagram>
</Project>
```

Данные примера визуализируются при помощи разработанного в процессе диссертационного исследования инструментария (рис. 4.5), а матрица смежности оформляется посредством соответствующей формы ввода.

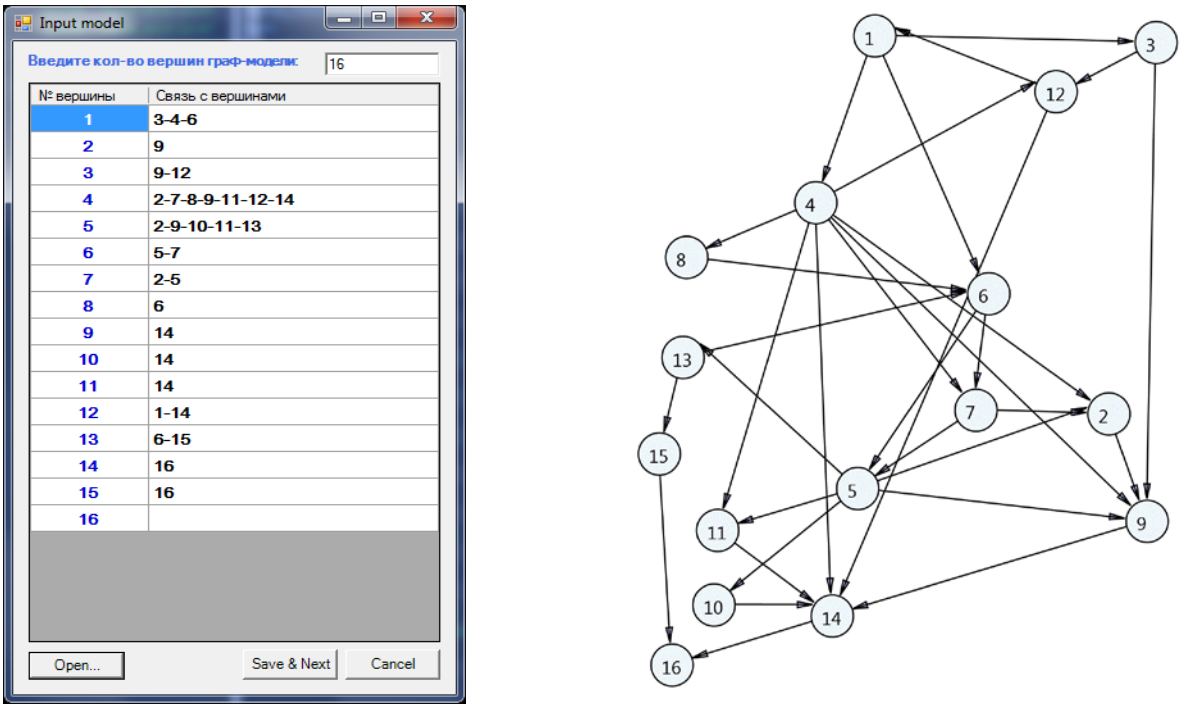


Рисунок 4.5 – Форма ввода графа, соответствующего графовой модели

Данные, хранящиеся в файле проекта, помещаются в соответствующие хранилища, чтобы быть использованными исполнителем ВП. Рассмотрим информационную модель на ERD-диаграмме, позволяющую оценить принципы взаимодействия исполнителя и массивов данных. Для начала формируем две сущности «Характеристики вершин» *VC* и «Компонента» *Comp* (рис. 4.6), отталкиваясь от множества характеристик (3.5) и устанавливаем между ними отношение «1:M» (один – ко многим), поскольку одной компоненте могут принадлежать несколько вершин графовой модели.

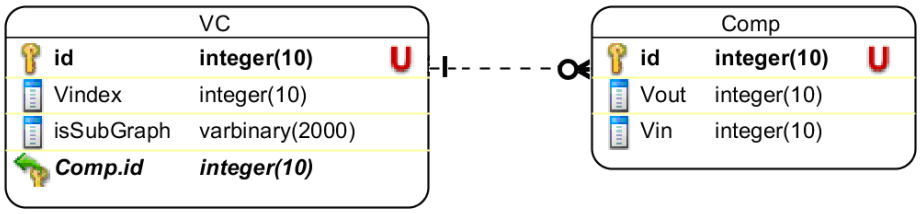


Рисунок 4.6 – ERD-диаграмма. Характеристики вершин

Затем устанавливаем отношения «1:M» между сущностями «Конденсированный граф» *CG*, «Состояние ВП» *ST* и «Состояние архива» *AR*, для которых полям *STValue* и *ARValue* присваиваются значения из соответствующих массивов (3.8). Далее выделяем сущность «Матрица смежности» *AM*, «Вычислительный процесс» *DP* и «Спецификация» *Spec*, которая соотносится с сущностью «Данные спецификации» *SpecData*, определяя атрибуты кортежей.

В свою очередь сущность «Состояние архива» *FileArch* так же сопоставляется с матрицей конденсированного графа *CG*, обеспечивая тем самым хранение физических характеристик файлов архивов.

Далее конденсированная графовая модель *CG* связывается с сущностью «Вычислительный процесс» так же как и сущность «Матрица смежности» отношениями «1:M» посредством третьей таблицы (рис. 4.7).

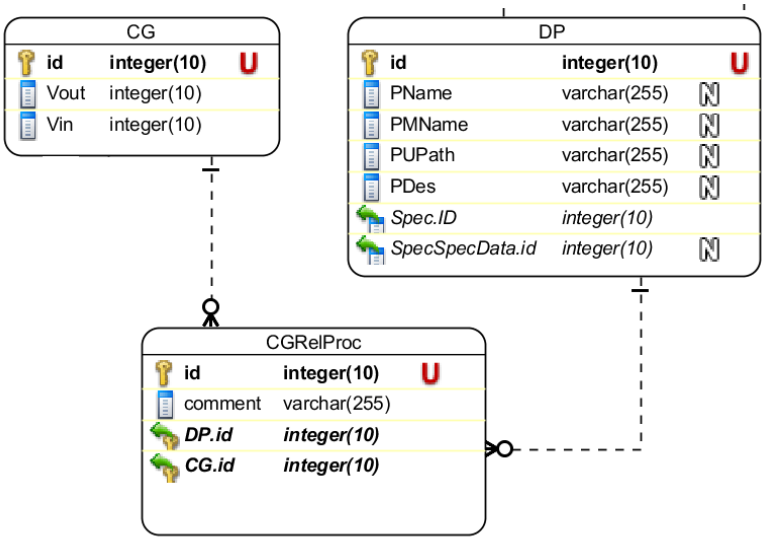


Рисунок 4.7 – ERD-диаграмма. Связь вершин конденсированной графовой модели и вычислительных процессов

Наличие таких связей позволяет отобразить и сохранить различный уровень детализации ВП по отношению друг к другу. Так пользователь и само ПО могут получить информацию о функциях, которые не свернуты в супервершину после конденсации и увидеть соотношение между укрупненной функцией и ее составляющими.

На следующем шаге определим сущности «Пользователи системы» *UserDict* и «Монитор состояний процессов» *ProcMonitor*, которые связываются с сущностью *DP* (рис. 4.8).

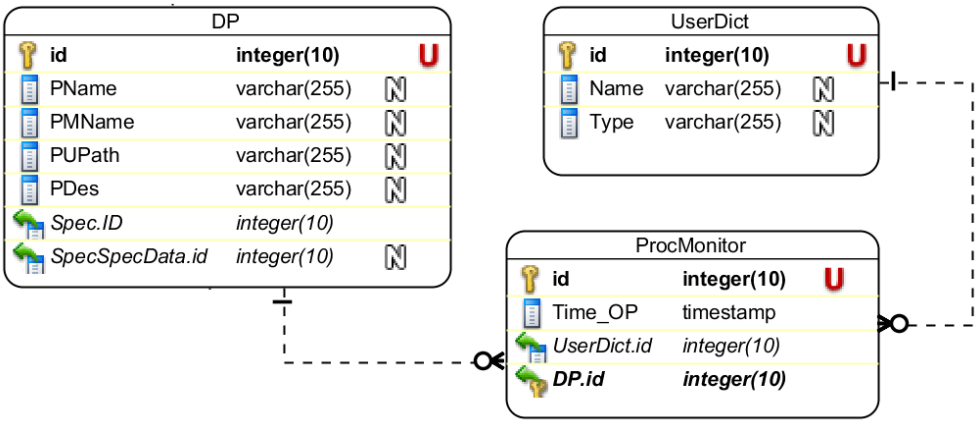


Рисунок 4.8 – ERD-диаграмма. Связь словаря пользователей системы и монитора ВП

Таблица *UserDict* представляет собой справочник системы, в котором хранятся все ее пользователи со своими характеристиками. Таблица *ProcMonitor* сохраняет текущие параметры ВП, работающего в данное время в системе, а связь между таблицами *UserDict* и *ProcMonitor* определяет ответственного пользователя за выполнение конкретного ВП.

Так же необходимо установить соотношение между пользователями системы и вершинами графовой модели, которые реализуют функцию пользовательского интерфейса – так называемыми базовыми вершинами *BaseNodes*, на основании которых строятся подграфы задач в результате действий алгоритма формирования подграфа (Приложение Г, рис. 7). Отношения между сущностями *CG* и *UserDict* устанавливаются как «1:M» посредством третьей сущности *CGRelUser* (рис. 4.9).

Для построения программной архитектуры ИС выбор сделан в пользу использования инструментария КА.

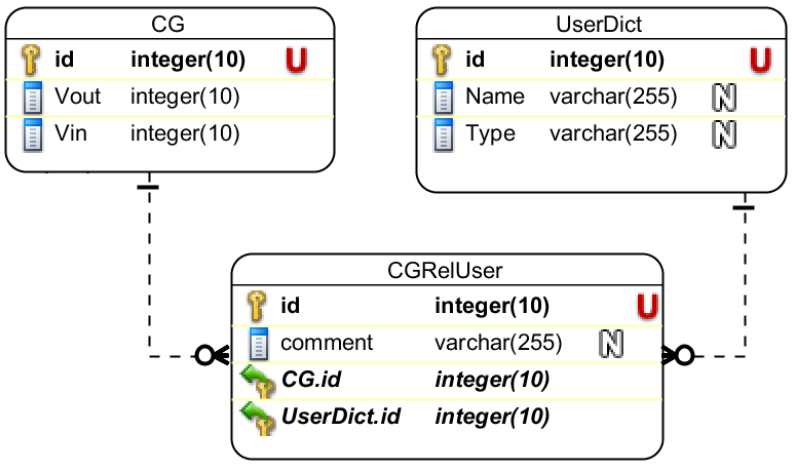


Рисунок 4.9 – ERD-диаграмма. Связь пользователя с супервершинами конденсированной графовой модели

КА (в частности, автомат Мили) используется в разработанном и предлагаемом инструментарии разработчика ПО как встраиваемое средство управления работой всего ПО, которое позволяет организовывать ВП с учетом условий функционирования, выполнять откат совершенных действий до заданного момента времени, и с этого момента заново запускать ход ВП, снижая количество сбоев в работе программного продукта. Снижение числа ошибок происходит за счет явного задания возможных состояний для управляющего КА и явного определения событий, возможных для данной программной системы.

Конечный автомат Мили был выбран исходя из таких соображений, что выходная последовательность этого КА зависит от состояния автомата и входных сигналов, явно переключающих КА из одного состояния в другое. Кроме этого использование КА при программировании позволяет упростить процесс верификации программных модулей и использовать формальный подход для обеспечения правильности работы системы на базе LTL-языка.

Однако применение КА связано с трудностью совмещения автоматного подхода с объектно-ориентированным. Наиболее простым решением является применение технологии WCF. В данном случае управляющий КА определяется как сервис, использующий контракты передаваемых данных и принимаемых

входных параметров, а также реализующий соответствующие интерфейсы, что позволяет унифицировать способы стыковки управляющего автомата и других вложенных автоматов. Вложенные КА, в свою очередь, организовывают управление ВП для конкретной вычислительной задачи, запускаемой на удаленном терминале или рабочей станции соответствующего пользователя.

4.3 Разработка программного обеспечения для решения задач производства сложных электронных устройств. Методика проведения эксперимента и оценка эффективности

ПрО в рамках поставленной задачи относится к классу сложных областей в связи с наличием большого числа подсистем, включенных в них элементов и большого числа связей между ними. Среди функциональных задач, которые реализуются элементами ПрО, можно выделить зависимые друг от друга функциональных задачи и с отсутствием такой зависимости. Также присутствуют составные функциональные задачи, которые возможно декомпозировать на более мелкие, и наличие итеративных подходов к их решению. Кроме этого имеет свойство менятся технологический процесс производства изделия, что влечет за собой необходимость видоизменять функциональность ПО. При этом изменение системы и ее адаптация к новым требованиям чувствительна ко времени выполнения этой процедуры. В этой связи становится актуальным использование предложенной ИТ струутурного синтеза программной архитектуры.

Для предприятия, занимающегося производством сложных изделий электронной техники, выделим подсистему, которая непосредственно занимается подготовкой производства. В такую подсистему входит конструкторский отдел (КО), технологическое управление (ТехУ), отдел нормировщика (ОН), отдел по подготовке производства (ОПП), отдел технологических маршрутов (ОТМ), управление материально-техническим обеспечением (УМТО) (рис. 4.10). Для указанных подразделений представим укрупненную схему основных функций подготовки производства и их взаимосвязь.

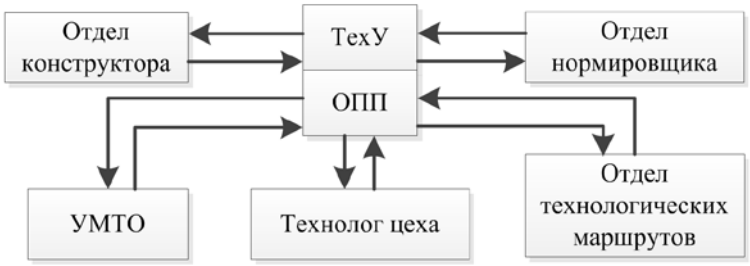


Рисунок 4.10 – Взаимодействие подразделений предприятия

Технолог цеха подчиняется подразделению ТехУ и учавствует в процессе утверждения технологической спецификации. Базовым процессом выделяется процесс подготовки производства. Исходной информацией, позволяющей понять взаимодействие подразделений предприятия, будет модель IDEF (рис. 4.11).

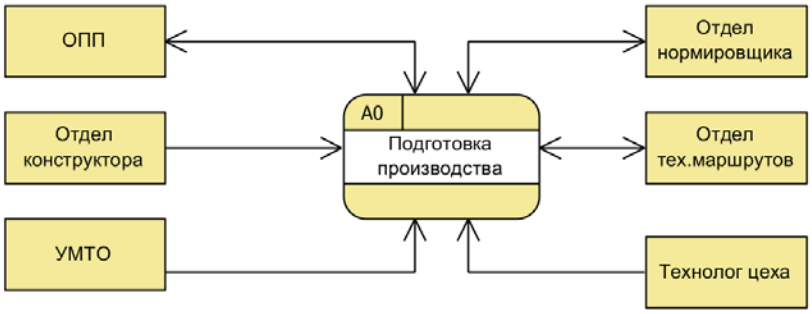


Рисунок 4.11 – Концептуальная схема подсистемы по подготовке производства

Основными этапами данного процесса являются: 1) сверка технологической и конструкторской спецификации изделий; 2) наполнение состава изделий на основании данных спецификаций; 3) формирование и утверждение состава для всех изделий; 4) формирование и утверждение плана производства – окончательный этап. Необходимо выделить основные процессы подготовки производства, поддерживаемые ПО, и задать информационные потоки для них (рис. 4.12). Процесс формирования технологического состава является итерационной задачей и требует согласования как с КО так и с технологами цехов, включенных в маршруты изделий. Кроме этого, получение производственного плана также происходит итерационно, поскольку требует учета состояния оборудования производственной линии, нагрузку на оборудование, время на пуско-наладочные работы, трудовые нормы, а также учета стоимости соответствующих операций и материалов.

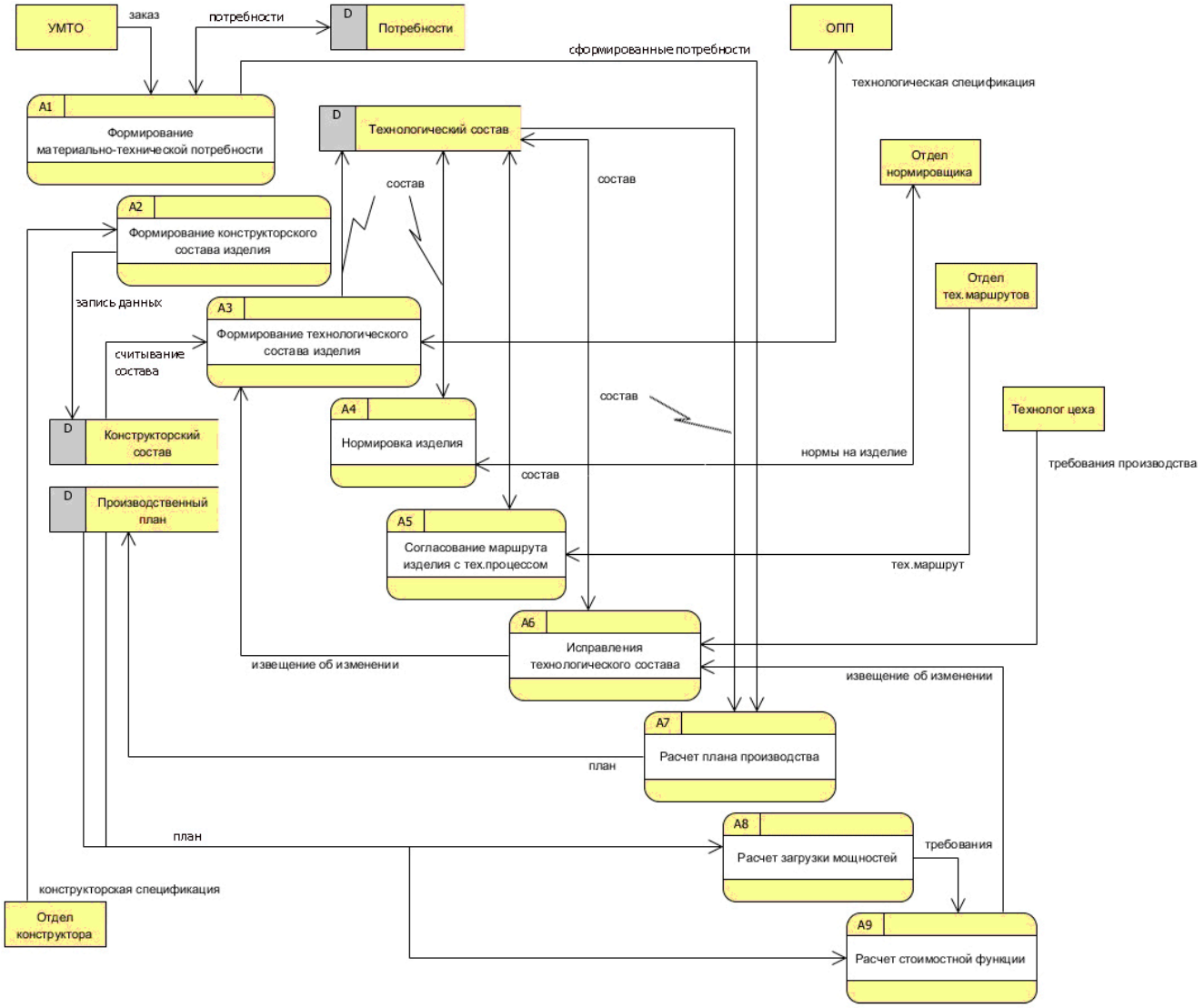


Рисунок 4.12 – Основные этапы процесса подготовки производства

Оптимальное решение возможно получить путем определения стоимостной функции, оптимизация которой происходит в несколько этапов. Для этого задаются параметры этой функции в различных вариациях, и итерация завершается при нахождении требуемых значений.

Выделим основные укрупненные функции, исполнение которых будет возложено на программные модули (рис. 4.13).

Функции обработки потребности, ввода конструкторской спецификации, норм и маршрутов являются независимыми и могут выполняться одновременно. Точно также и функции расчета конструкторского и технологического составов. Беря во внимание полученную схему, зададим для нее графовую модель

програмной архитеткруры, с учетом DFD-диаграммы, добавив фиктивную корневую вершину входа в вычислительный процесс под ***0***-ым номером.

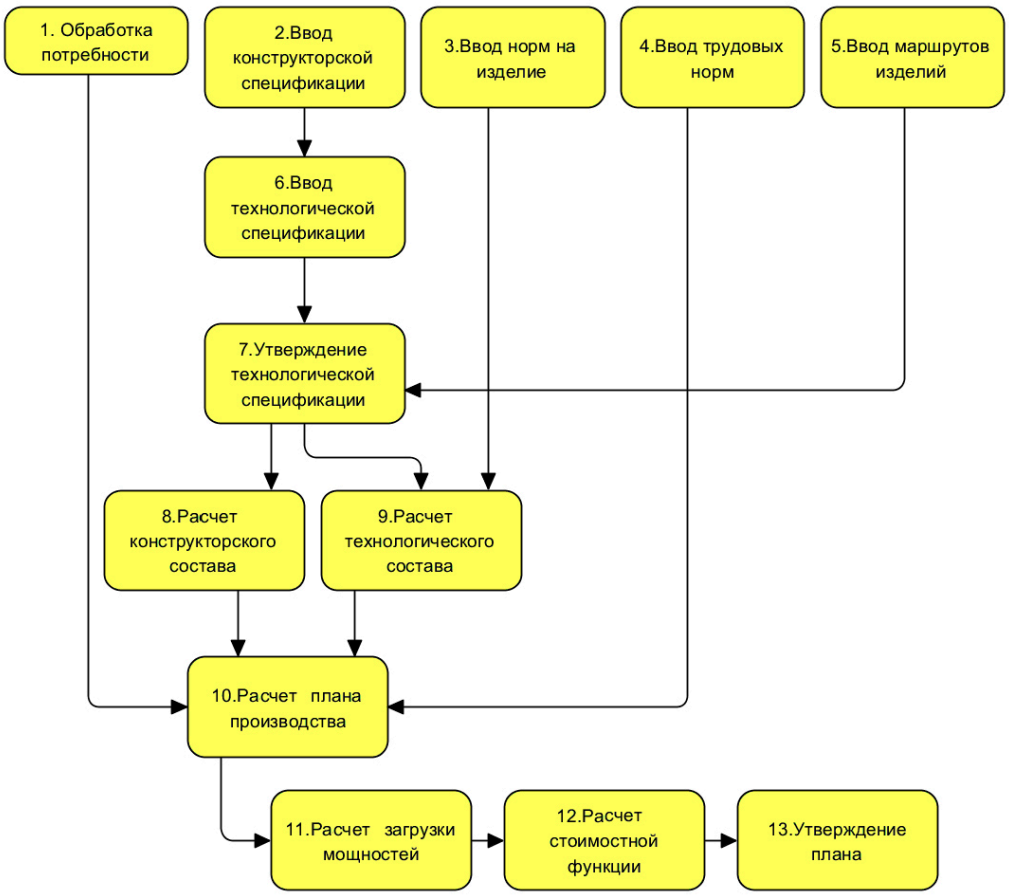


Рисунок 4.13 – Последовательность крупно-блочных функций процесса подготовки производства, реализуемых ПО

Учитывая указанные ранее итерационные процессы, выделим на графовой модели вершины, для которых требуется организовать резервирование данных. Базовая 8-ая вершина соответствует операции восстановления данных, которую пользователь выполняет для отдела конструктора. Для отдела ТехУ – это 9-ая вершина. Для процесса расчета стоимостной функции это 13-я вершина. Соответственно для всех вершин принадлежащих подграфам G_8, G_9 и G_{13} организуются хранилища данных (рис. 4.14).

Покажем на диаграммах классов структуру формируемого ПО, выполняющего функциональные задачи ИС (рис. 4.12, 4.13).

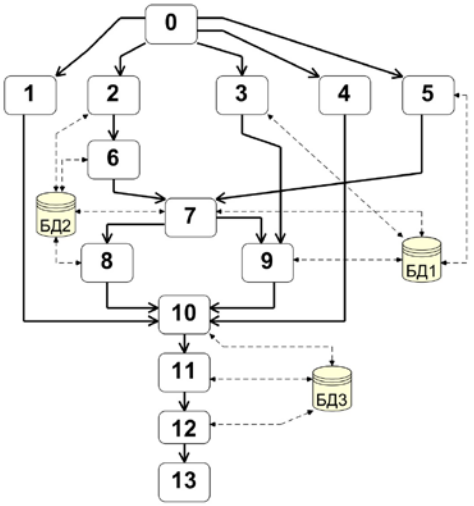


Рисунок 4.14 – Графовая модель программной архитектуры

Для этого зададим классы CDepartment и CComposition, которые описывают шаблоны подразделения и состава изделия соответственно. Затем определим наследование классов для сущностей КО (CConstrDep), ТехУ (CTechDep), ОН (CProductionRouteDep), ОПП (CPreproductionDep), для технологических (CTechComposition) и конструкторских (CConstrComposition) составов, а также для производственного плана. Взаимодействие КО и ТехУ строится вокруг двух основных сущностей: конструкторской (ConstrSpecification) и технологической (TechSpecification) спецификации (Приложение Г, рис. 12). В процессе совместной работы технолог, проводя изменения своих спецификаций в таблицах БД, согласует эти изменения путем формирования извещения конструктора (сущность CNotification «Извещение»). Конструктор (CConstructor), в свою очередь подтверждает изменения технолога (Technologist), что позволяет ТехУ на основе данных технологического процесса (Workflow) и данных маршрута изделия (CProductRoute) сформировать технологический состав (CTechComposition).

На следующем этапе (Приложение Г, рис. 13) формируется план производства на основе составов и данных норм на изделие (CNorm). Нормы определяет нормировщик (CRateSetter), находящийся в подчинении отдела ОН. Данные норм согласуются с документами, определяющими требования к изделию (CProductRequirements). Эти документы оформляются отделом УМТО на основании материально-технической потребности (CMaterialTechnicalNeeds),

которая, исходит от поступившего заказа (COrder) на производство изделия. Прежде, чем формирование плана производства будет завершено, отдел ОПП CPreproductionDep) сверяет данные плана и технологического состава на основании отчетной информации, передаваемой ему ПС (Приложение Г, рис. 13).

Теперь определим структуру исполнителя ВП, реализующего функции сформированной ПрО. Основываясь на полученной модели КА, выделим основные компоненты AUserSM, ABackwardSM, AForwardSM и DllWrapper (рис. 4.15), первые три из которых удобно реализовать как сервисы.

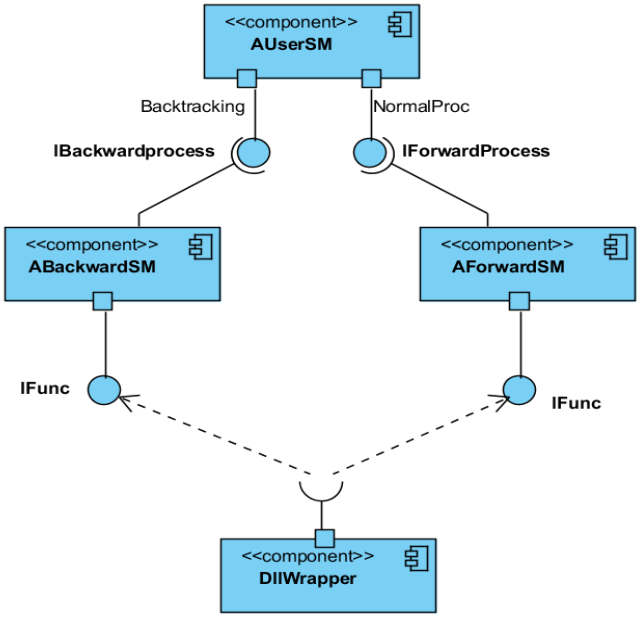


Рисунок 4.15 – Диаграмма компонентов для модели КА

Первый компонент AUserSM определяет два порта для нормального исполнения ВП (NormalProc) и отката действий ВП (Backtracking). Этим портам соответствуют интерфейсы IForwardprocess и IBackwardprocess, которые используются двумя другими компонентами, программирующими автоматы AForwardSM и ABackwardSM. Данные автоматы управляют работой программных модулей посредством компонента DllWrapper, позволяющего адаптировать стандарты внешних программных компонент, составляющих ПО ИС, со стандартами вызовов автоматов AUserSM, ABackwardSM и AForwardSM.

Программирование описанных компонентов в рамках объектно-ориентированного подхода выполняется с учетом предложенных шаблонов

использования службы MCF для распределенного исполнения ВП (рис. 3.9) и организации параллельного исполнения ВП (Приложение Г, рис. 5).

Проведем численный эксперимент в целях оценки качества сформированного ПО на основе примененной ИТ структурного синтеза программной архитектуры ИС. Входной информацией эксперимента является полученная ярусно-параллельная графовая модель программной архитектуры (рис. 4.13), состоящая из 14 базовых вершин. А также LOC-оценка программных модулей, соответствующих заданным вершинам, и классов-оберток, обеспечивающих взаимодействие модулей между собой и конфигурирование программной архитектуры. Оценка методов, используемых в предложенной ИТ, производилась в соответствии со стандартом ISO/IEC 9126-93, регламентирующим характеристики качества и руководства по их применению для оценки программной продукции. Методы были оценены согласно таким характеристикам: эффективность, сопровождаемость и надежность. Для определения уровня эффективности ПО был выбран показатель временной эффективности. Для определения уровня сопровождаемости – оценка структурной (топологической) и информационной сложности. В случае определения временной эффективности ПО ИС, сформированного на базе предложенной ИТ, время переконфигурирования одного программного модуля для заданной графовой модели программной архитектуры составило 120-200 мс, тогда как сравнение с существующими аналогами показывает лучшее время [159]: 20-50 мс для переконфигурирования одного программного компонента.

Для определения структурной сложности была выбрана метрика на основе критерия минимизации количества связей между программными модулями (2.24) без учета характеристик сцепления и связности, которая составляет следующую величину: [14(13)-2*18]/14*13*12=0,067. Величина меньше единицы, что говорит об успешном применении ИТ структкурного синтеза программной архитектуры. При увеличении количества вершин $\boldsymbol{r}$ полученной графовой модели (рис.4.24) также будет увеличиваться временная сложность согласно величине $O(r)$. В то время как анализ сущесвующих аналогов показывает увеличение временной

сложности в соответствии с величиной $O(2^r)$ для технологии динамических цепочек программных продуктов DSPL на базе сервис-ориентированной программной архитектуры ИС [104].

Для оценки информационной сложности полученного ПО ИС была выбрана метрика LOC-оценки. При этом для каждого программного модуля, отвечающего вершине графовой модели и соответствующего ему класса-обертки получены значения метрики (табл.4.2).

Таблица 4.2 – Значения LOC-метрики для сформированного ПО

№п/п	SLOC класса-обертки	SLOC программного модуля
1	120	800
2	50	900
3	95	950
4	100	755
5	115	710
6	78	600
7	90	1000
8	100	541
9	88	654
10	100	402
11	167	488
12	50	700
13	160	755
14	100	570
Всего	1413	9825

Приведенные данные (табл.4.2) были сравнены с данными отчетов по применению существующих аналогов технологий следующим образом. Процент кастомизации ПО, которая требуется при развертывании и внедрении ПО на предприятии класса ERP, взят на основе данных отчетов за 2015 и 2016 годы компании Panorama Consulting Solutions [98, 99] (рис. 4.16).

Процент ручной настройки системы, необходимой для динамического конфигурирования, для большинства предприятий (в среднем 44% предприятий всех отраслей промышленности и 38% предприятий сферы машиностроения)

варьируется от 26% до 50% изменения исходного кода в соответствии с настройками и требованиями заказчиков системы. В случае кастомизации ПО с помощью предложенной ИТ структурного синтеза программной архитектуры ИС процесс адаптации программы выполняется автоматически и требует долю ручной работы только при настройке ПО и написании класов-оберток.

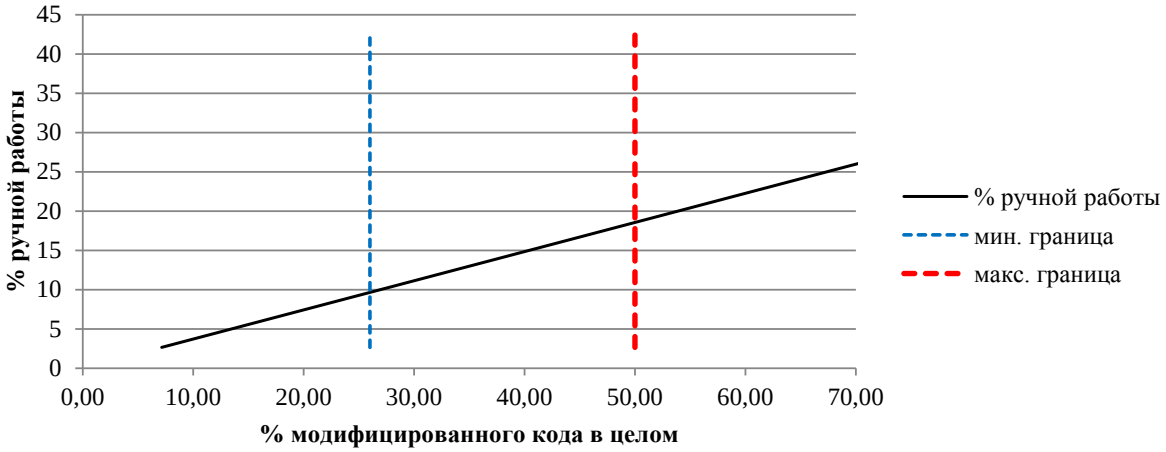


Рисунок 4.16 – Доля ручной настройки системы относительно процента изменения исходного программного кода

Поэтому, учитывая, что при пессимистической оценке требуется 50% кастомизации: $ML = \frac{SLOC_{cust}}{SLOC_{total}} \cdot 100\% = 50\%$, то для предложенной ИТ этот процент (по данным табл.4.1) составит 14,38%:

$$AL = \frac{SLOC_{wrap}}{SLOC_{total}} \cdot 100\% = \frac{1413}{9825} \cdot 100\% = 14{,}38\% .$$

Следовательно, благодаря автоматизации процесса конфигурирования программной архитектуры ИС, улучшение показателя составляет:

$$EN = ML\text{-}AL = 50\% - 14{,}38\% = 35{,}62\% \approx 36\% .$$

Дальнейшее тестирование сформированного ПО проводилось в целях определения временной эффктивности процесса восстановления результатов ВП,

необходимого при збоях в работе программного продукта. Для этого был определен алгоритм, добавляющий случайным образом по одной вершине в графовую модель для определения времени работы всего ПО. Также было задано условно, что каждая вершина графовой модели требует для выполнения 10 мс процессорного времени. Определив моменты резервирования данных и их восстановление, получена таблица сравнения времен работы ПО (табл. 4.3). Время при повторном запуске увеличивается в среднем в 1,3 раза. А при использовании информационной технологии сокращается в среднем от 5% до 10%.

Таблица 4.3 – Сравнение времени работы ПО при различных условиях и для различного количества вершин графовой модели

№ вершины	$L_{общ}$**, %**	$L_{руч}$**, %**	$T_{сред}^{тр}$**, мс**	$T_{сред}$**, мс**				
				P_{50}	P_{40}	P_{30}	P_{20}	P_{10}
1	7,1	2,7	1546	2752	2511	2413	2705	2766
2	14,3	5,3	2937	8543	8360	8543	8543	8543
3	21,4	7,9	4509	10252	9783	9838	9649	9498
4	28,6	10,6	6023	11918	11195	11221	10683	10434
5	35,7	13,3	7863	13773	12687	12481	11774	11320
6	42,9	15,9	9667	15446	14134	13670	12739	12327
7	50,0	18,6	11087	17285	15567	14916	13764	13281
8	57,1	21,2	12046	18960	17026	16172	14832	14157
9	64,3	23,9	14794	20703	18500	17384	15971	15137
10	71,4	26,5	15989	22430	19973	18642	16995	16418
11	78,6	29,2	17358	24352	21710	19858	18172	17420
12	85,7	31,8	19557	26112	23164	21039	19247	18366
13	92,7	34,5	20399	27946	24599	22298	20249	19301
14	100,0	37,1	21685	29796	25893	23605	21252	20218
15	107,1	39,8	22924	31511	27351	24801	22320	21166
16	114,3	42,4	24892	33390	28743	26044	23393	22047

Немаловажным фактором является процент ошибок в исходных данных, подлежащих корректировке. При необходимости 100% пересчета данных исходная информация полностью извлекается из архива, что существенно влияет на временные затраты. Чем меньше ошибок требуется исправить, тем меньше повторных расчетов требуется выполнить и, соответственно, тем быстрее

осуществляется работа с архивом. Исходя из этого, был проведен тест на время выполнения программы в зависимости от процента ошибок в исходных данных и в зависимости от количества вершин на графовой модели (рис. 4.17).

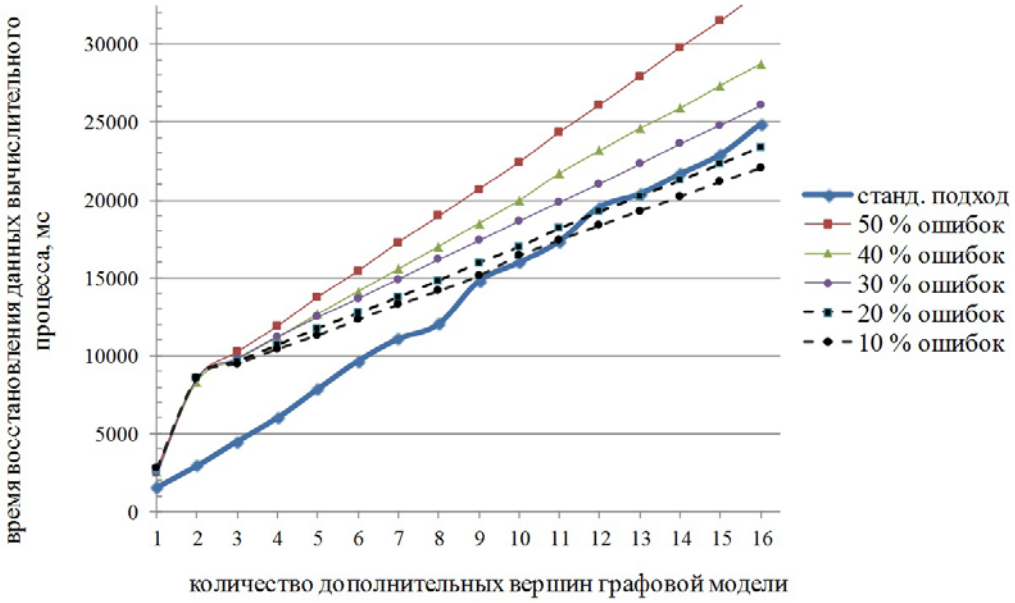


Рисунок 4.17 – График времени работы ПО

Анализ данных (рис. 4.17) показывает, что предложенная ИТ структурного синтеза программной архитектуры имеет преимущества при значительном усложнении существующей графовой модели, более 12-ти дополнительных функциональных задач, представленных вершинами графовой модели. При этом в исходных данных оказывается небольшой процент ошибок, менее 20%. Использование предложенной ИТ позволило улучшить быстродействие программной системы при изменении требований к конечному программного продукта в случае необходимости восстановления результатов работы ВП по сравнению со стандартными методами резервирования и восстановления данных в 1,12 раза для 10% ошибок, и в 1,1 раза для 20% ошибок на графовой модели, состоящей из исходного множества функциональных задач и дополненной 16-ю вершинами. Определение принципов управления, восстановление данных и синхронизации ВП в параллельных вычислительных средах, а также методов формирования гибкой программной архитектуры с ее адаптацией для сервис-

ориентированной и облачной архитектур является предметом дальнейших исследований.

4.4 Выводы по разделу 4

В разделе решены такие задачи.

1. Предложена ИТ структурного синтеза программной архитектуры ИС. Технология включает этапы синтеза графовой модели программной архитектуры ИС, конфигурирования архитектуры с учетом текущих функциональных требований на базе метода объединения вершин графовой ярусно-параллельной модели программной архитектуры, графового метода оценки функциональной сложности ПО и автоматного метода проверки выполнения ограничений к ПО. Предложенная ИТ позволяет снизить примерно на 36% трудоемкость переконфигурирования ПО.

2. Разработана архитектура исполнителя ВП для ПО ИС, осуществляющего контроль работы программных модулей на базе ярусно-параллельной графовой модели программной архитектуры в рамках заданной ПрО.

3. Разработанный программный инструментарий «SW Design» был использован для формирования ПО, автоматизирующего процессы подготовки производства сложных электронных устройств, что позволило выполнить анализ єффективности и применимости предложенной ИТ.

4. Практическая реализация ИТ структурного синтеза программной архитектуры показала следующие преимущества ИТ: 1) высокий уровень сопровождаемости программного продукта; 2) снижение временных затрат на переконфигурирование ПО для программных систем, требующих локального развертывания компонентов. Недостатками предложеной ИТ является низкая временная эффективность по сравнению с существующей технологией на базе сервис-ориентированной архитектуры ПО. В этой связи предложенную ИТ следует использовать в случае высоких требований к безопасности данных и необходимости локального развертывания.

ВЫВОДЫ

Основные научные и практические результаты работы заключаются в том, что решена актуальная научно-практическая задача разработки модели, методов и информационной технологии структурного синтеза программной архитектуры информационной системы в условиях изменяющихся требований конечного пользователя. При этом получены следующие результаты:

1. Анализ методов структурного синтеза и кастомизации ПО ИС показал, что современные методы не удовлетворяют требованиям к эффективной адаптации ПО под изменяющиеся во времени требования конечного пользователя, что подтверждает актуальность решения задачи разработки эффективных формальных подходов к кастомизации ПО ИС путем использования ярусно-параллельных графовых моделей программной архитектуры.

2. Усовершенствована ярусно-параллельная графовая модель программной архитектуры путем учета взаимосвязей по входным и выходным данным между программными модулями ИС. Программные модули в модели представляются вершинами графа. Направленные дуги графа отображают связи по данным между программными модулями. Дополнение модели связями по данным обеспечивает возможность динамического конфигурирования программной архитектуры при изменении функциональных требований к системе.

3. Впервые предложен метод объединения вершин графовой ярусно-параллельной модели программной архитектуры ИС на основе оценки взаимосвязей между этими вершинами по входным и выходным данным. Метод включает этапы оценивания показателей сцепления и связности, а также объединения вершин на основе полученных значений показателей. Метод позволяет уменьшить временные затраты на конфигурирование ПО при изменении требований и, тем самым, снизить затраты на конфигурирование. Проведенная оценка временной сложности метода составила величину $O(n)$.

4. Впервые предложен графовый метод оценки функциональной сложности ПО путем использования ярусно-параллельной графовой модели программной

архитектуры ИС. Метод включает в себя этапы определения функциональных характеристик программных модулей и расчета критериев функциональной сложности, что позволяет оценить соответствие формируемого ПО функциональным требованиям конечного пользователя. Проведенная оценка временной сложности метода составила величину $O(n^2)$.

5. Усовершенствован автоматный метод проверки выполнения ограничений к формируемому ПО, выраженных в виде нефункциональных требований. Метод включает этапы формирования автоматной модели взаимодействия модулей и сравнения полученных в результате моделирования последовательностей взаимодействия модулей с нефункциональными требованиями к ПО. Метод позволяет сформировать ограничения на структуру ПО, учитывающие нефункциональные требования, и повысить надежность его функционирования. Проведенная оценка временной сложности метода составила $O(n^3 \cdot \log k)$.

6. Предложена ИТ структурного синтеза программной архитектуры ИС. Технология включает этапы синтеза графовой модели программной архитектуры, конфигурирования архитектуры с учетом текущих функциональных требований на базе объединения вершин графа и проверки выполнения нефункциональных требований на основе соответствующего автоматного метода. Предложенная технология позволяет снизить стоимость изменения ПО.

7. Проведено внедрение модели, методов и ИТ с помощью разработанного программного инструментария «SWDesigner» при решении практических задач проектирования производственного процесса сложных изделий электронной техники на базе Харьковского научно-исследовательского института технологии машиностроения, а также в научную и проектную деятельность Института физики высоких энергий и ядерной физики Национального научного центра «Харьковский физико-технический институт», что подтверждается соответствующими актами внедрения. Предложенная ИТ позволяет снизить примерно на на 36 % временные затраты на конфигурирование программного обеспечения в условиях изменяющихся требований конечного пользователя.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Автоматический синтез системы управления мобильным роботом для решения задачи «Кегельринг» / С. А. Алексеев, А. И. Калиниченко, В. О. Клебан [и др.] // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. – 2011. – № 2 (72). – С.26–31.

2. Агаев Р. П. Согласование характеристик в многоагентных системах и спектры лапласовских матриц орграфов / Р. П. Агаев, П. Ю. Чеботарев // Автоматика и телемеханика. – 2009. – № 3. – 136–151.

3. Агеев Д. В. Параметрический синтез инфокоммуникационных систем с использованием модели многослойного графа / Д. В. Агеев, Фуад Вехбе // СВЧ-техника и телекоммуникационные технологии (КрыМиКо 2013): 23-я междунар. Крымская конф., 8-13 сентября, 2013 г.: тезисы докл. в 2 т. – Севастополь, 2013. – С. 507–508.

4. Акопян М. С. Использование многопоточных процессов в среде ParJava / М. С. Акопян // Труды ИСП РАН. – 2015. – Т. 27. – №. 2.– С.5–22.

5. Алгеброалгоритмические модели и методы параллельного программирования / [Андон Ф.И., Дорошенко Е.А., Цейтлин Г.Е., Яценко Е.А.] – Киев: Академпериодика, 2007. – 631 с.

6. Альшевский Ю. А. Механизм обмена сообщениями для параллельно работающих автоматов (на примере системы управления турникетом) / Альшевский Ю. А., Раер М. Г., Шалыто А. А. – Санкт-Петербург, 2003. – 49 с. – Режим доступа: http://is.ifmo.ru/download/turn.pdf.

7. Андон Ф. И. Структурные статистические модели: инструмент познания и моделирования / Ф. И. Андон, А. С. Балабанов // System Research & Information Technologies. – 2007. – №1. – С.79–98.

8. Андрианов А. Н. Автоматическая генерация программ для графических процессоров по непроцедурным спецификациям / А. Н. Андрианов, А. Б. Бугеря,

Е. Н. Гладкова // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2014. – № 1(3). – С.5–16.

9. Антонов В. В. Построение формальной модели предметной области с применением нечеткой кластеризации / В. В. Антонов, Г. Г. Куликов, Д. В. Антонов // Уфа: УГАТУ. – 2011 –Т. 15 –№ 5 (45). – С. 3–11.

10. Артюх А. А. Метод формального синтеза мажорированных Си-программ / А. А. Артюх, Г. А. Поляков, Е. Г. Толстолужская // Вісник Харківського національного університету. – 2011. – №977. – С.5–13.

11. Бакулев А. В. Модели и алгоритмы организации мобильных параллельных вычислений в среде многоядерных процессоров: автореф. дис. на соискание научн. степени канд. техн.: спец. 05.13.11 «математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей» / А. В. Бакулев. – Рязань, 2011. – 20 с.

12. Блюмин С. Л. Графоструктурное моделирование экосистем. Звездоцветы как графы и гиперграфы / С. Л. Блюмин // Экология ЦЧО РФ. – № 1-2 (30-31). – 2013. – С. 111–114.

13. Брагина Т. И. Сравнительный анализ итеративных моделей разработки программного обеспечения / Т. И. Брагина, Г. В. Табунщик // Радіоелектроніка, інформатика, управління. – 2010. – Вып. № 2 (23). – С.130–139.

14. Веденеев В. С. Применение экстремального программирования при разработке научных приложений / В. С. Веденеев, И. В. Бычков // Математические структуры и моделирование. – 2014. – №. 4 (32). – С.180–184.

15. Вендров А. М. Современные технологии создания программного обеспечения. Обзор / А. М. Вендров // Jet Info. –2004. –№4 (131). – С.3–32.

16. Взаимное преобразование UFO- и UML-моделей / М. Ф. Бондаренко, С. С. Пирог, Е. А. Соловьева [и др.] // Проблеми програмування. – 2004. – № 2, 3.– С. 150-154.

17. Воеводин В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – Санкт-Петербург.: БВХ-Петербург, 2002. – 608 с.

18. Генератор проектов – инструментальный комплекс для разработки «клиент-серверных» систем / Л. Л. Вышинский, И. Л. Гринев, Ю. А. Флеров [и др.] // Информационные технологии и вычислительные системы. – 2003. – №1(2). – С.6–25.

19. Генератор проектов – средство автоматизации проектирования прикладных информационно-вычислительных систем. / Флёров Ю. А., Вышинский Л. Л., Гринёв И. Л. [и др.] // Автоматизация проектирования инженерных и финансовых информационных систем средствами «Генератора проектов». – М.: ВЦ РАН. – 2010. – С. 3–15.

20. Городняя Л. В. О проблеме автоматизации параллельного программирования / Л. В. Городняя // Научный сервис в сети Интернет: многообразие суперкомпьютерных миров: междунар. суперкомпьютерная конф. 22-27 сентября 2014 г.: тезисы докл. – Новороссийск, 2014. – С.191–196.

21. Гуисов М. И. Интеграция механизма обмена сообщениями в Switch-технологию / М. И. Гуисов, А. Б. Кузнецов, А. А. Шалыто. – Санкт-Петербург: СПбГУ ИТМО, 2003. – Режим доступа: http://is.ifmo.ru/download/memech.pdf.

22. Диалоговая система синтеза многосвязных структур промышленных объектов / [Зайцев И.Д., Кисиль И.М., Вайнер В.Г., Губницкий С.Б.] – Киев: Институт кибернетики. – 1981. – 51 с.

23. Дорошенко Е. А. О синтезе программ на языке Java по алгеброалгоритмическим спецификациям / Е. А. Дорошенко, Е. А. Яценко // Проблемы программирования. – 2006. – № 4. – С. 58–70.

24. Евстигнеев В.А. Применение теории графов в программировании / В.А. Евстигнеев. – М.: Наука –1985. – 352 с.

25. Ерофеев Е. В. Инновационная мотивация в командообразующих группах для быстрой разработки программного обеспечения / Е. В. Ерофеев // Интернет-журнал «Науковедение». – 2014. – № 1(20). – С.1–11. – Режим доступа: http://cyberleninka.ru/ article/n/innovatsionnaya-motivatsiya-v-komandoobrazuyuschih-gruppah-dlya-bystroy-razrabotki-programmnogo-obespecheniya.

26. Затуливетер Ю. С. Проблемы программируемости, безопасности и надежности распределенных вычислений и сетецентрического управления. Ч. 2. Подход к общему решению / Ю. С. Затуливетер, Е. А. Фищенко // Проблемы управления. – 2016. – № 4. – С.58–69.

27. Зінькович В. М. Онтологічне моделювання предметної області з проблематикою e-Science / В. М. Зінькович // Проблеми програмування. – 2011. – № 3. – С. 30–37.

28. Инструментальная система ФАКИР / Л. Л. Вышинский, Ю. Д. Прибытков, В. И. Шиленко [и др.] // Известия АН СССР, техническая кибернетика. – 1986. – №3. – С. 6.

29. Инструментальное средство для поддержки автоматного программирования / В. С. Гуров, М. А. Мазин, А. С. Нарвский [и др.] // Программирование. – 2007. – №6.– С.65–80. Режим доступа: http://is.ifmo.ru/works/_2008_01_27_gurov.pdf.

30. Инструментальные средства САПР / Вышинский Л. Л., Гринёв И. Л., Шиленко В. И. [и др.] // Задачи и методы автоматизированного проектирования в авиастроении. – 1991. – С.52–70.

31. Интеллектуальные технологии распределенных вычислений для моделирования сложных систем / Марьин С. В., Ларченко А. В., Ковальчук С. В. [и др.] // Науч.- техн. вестн. СПбГУ ИТМО. – 2010. – № 70. – С. 123–124.

32. Калянов Г. Н. CASE-технологии: консалтинг в автоматизации бизнес-процессов / Калянов Г. Н. . – Москва: Горячая линия-Телеком, 2002. – 320 с.

33. Канжелев С. Ю. Автоматическая генерация кода программ с явным выделением состояний / С. Ю. Канжелев // Software Engineering Conference (Russia) – 2006 (SEC (R): матер. конф. 2006. – 2006. – С. 60–63.

34. Канович М. И. Логические методы синтеза программ / Канович М. И. – Калинин.: Калининский государственный университет, 1986. – 44 с.

35. Касьянов В. Н. Язык представления графов GRAPHML: базовые средства / В. Н. Касьянов // Информатика в науке и образовании: сб. науч. тр. – Новосибирск: Ин-т систем информатики им. А.П. Ершова СО РАН. – 2012. –

С.7–22. – Режим оступа: http://www.iis.nsk.su/files/articles/sbor_kas_21_kasyanov_part2.pdf.

36. Князьков К. В. Предметно-ориентированные технологии разработки приложений в распределенных средах / К. В. Князьков, А. В. Ларченко // Изв. вузов. «Приборостроение». – 2011. – Т. 54 – № 10 – С.36–43.

37. Кобозева А. А. Метод построения топологической сортировки объединения графов, используемый при распараллеливании последовательных алгоритмов / А. А. Кобозева, Е. В. Нариманова // Труды Одесского политехнического университета. – 2004. – № 2(22). – С.1–5.

38. Ковальчук С. В. Интеллектуальная поддержка процесса конструирования композитных приложений в распределенных проблемно-ориентированных средах / С. В. Ковальчук, В. Г. Маслов // Изв. вузов. Приборостроение. – 2011. –Т. 54, –№ 10 – С.29–36.

39. Корухова Ю. С. Автоматический синтез программ с использованием онтологии прецедентов / Ю. С. Корухова, Н. Н. Фастовец // Программные системы и инструменты: тематический сборник. – Т. 12. – 2011. – С.203–215.

40. Кузьмин Е. В. Моделирование, спецификация и построение программ логических контроллеров / Е. В. Кузьмин, В. А. Соколов // Моделирование и анализ информационных систем. – 2013. – №20(2). – С. 104–120.

41. Лаврищева Е. М. Сборочное программирование. Основы индустрии программных продуктов / Е. М. Лаврищева, В. Н. Грищенко: 2-изд. доп. и перераб.– Киев: Наук. думка, 2009. – 372 с.

42. Лаврищева Е. М. Software Engineering компьютерных систем. Парадигмы, технологии и CASE-средства программирования / Лаврищева Е. М.– К.: Наук. думка. – 2013. – 283 с.

43. Левитин А. В. Алгоритмы: введение в разработку и анализ / Левитин А. В.; [пер. с англ.] – Москва: Вильямс, 2006. – 576 с.

44. Левыкин В. М. Модель архитектурного фреймворка ускоренной разработки информационной системы / В. М. Левыкин, М. В. Евланов // Нові технології. – 2013. – № 1-2 (39-40). – С.51–57.

45. Лопатина Н. В. Управление информатизацией: теоретико–социологический подход: монография. – Москва: МГУКИ, 2006.– 236 с.

46. Лукин М. А. Разработка и автоматическая верификация параллельных автоматных программ / М. А. Лукин, А. А. Шалыто // Информационно-управляющие системы. – 2013. – № 5(66). – С.43–50.

47. Мазурчук В. С. Организация проблемно-ориентированных многопроцессорных систем со структурной интерпретацией итерационных вычислений: автореф. дис. на соискание учен. степени канд. техн. наук: спец. 05.13.13 «Телекоммуникационные системы и компьютерные сети» / В. С. Мазурчук. – Киев: Институт проблем моделирования в энергетике,1983. – 233 с.

48. Малышкин В.Э. Параллельное программирование мультикомпьютеров / В. Э. Малышкин, В. Д. Корнеев – Новосибирск: Новосибирский государственный технический университет, 2006. – 452 с.

49. Мейер Б. Объектно-ориентированное конструирование программных систем / Бертран Мейер. – М.: Русская Редакция. – 2005. – 1204 с.

50. Методы верификации программного обеспечения / Р. Е. Гурин, И. В. Рудаков, А. В. Ребриков // Наука и образование: научное издание МГТУ им. Н.Э. Баумана. – 2015. – № 10. – С.235–251

51. Михайлова Т. В. Анализ эффективности информационных систем / Т. В. Михайлова, С. В. Коваленко // Наукові праці Донецького національного технічного університету, серія «Інформатика, кібернетика та обчислювальна техніка». – Донецк: ДонГТУ. – 2008. – Вып. 9 –№ 132. – С. 277–280.

52. Михелев М. В. Формализация визуальных графоаналитических моделей процессов управления: автореф. дис. на соиск. степени канд. тех. наук: спец. 05.13.01 «Системный анализ, управление и обработка информации» / М. В. Михелев. – Белгород: Белгородский государственный национальный исследовательский университет. – 2011. – 21 с.

53. Мордвинов Д. А. Обзор применения формальных методов в робототехнике / Д. А. Мордвинов, Ю. В. Литвинов // Научно-технические

ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. – 2016. – №1(236). – С.84–107.

54. Мухопад Ю. Ф. Анализ и синтез управляющих автоматов сложных технических систем / Ю. Ф. Мухопад, А. Ю. Мухопад // Материалы XII-го Всероссийского совещания по проблемам управления, 16-19 июня 2014 г. – Москва, 2014. – Т. 16. – С. 7295–7306. Режим доступа: http://vspu2014.ipu.ru/proceedings/prcdngs/7295.pdf.

55. Мьё Т. Т. Разработка распределенных гетерогенных вычислительных систем и запуск приложений в распределенной вычислительной среде: автореф. дис. на соиск. уч. степени канд. тех. наук: спец. 05.13.18 «Математическое моделирование, численные методы и комплексы программ» / Тун Тун Мьё. – Санкт-Петербург: Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова, 2011. – 18 с.

56. Немнюгин С. А. Модели и средства программирования для многопроцессорных вычислительных систем / Немнюгин С. А. – Санкт-Петербург: С.-Петербургский ГУ, 2010. – 88 с.

57. Немченко В. П. Моделирование сетевых протоколов при построении тестовых последовательностей / В. П. Немченко, А. Н. Зиарманд, Ю. А. Чепелев // Інформаційно-керуючі системи на залізничному транспорті. – 2011. – №5. – С.18–21.

58. Обзор подходов к верификации распределенных систем / И. Б. Бурдонов, А. С. Косачев, В. Н. Пономаренко [и др.]. – 2006. – 61с. – (Препринт). – Режим доступа: http://www.ispras.ru/preprints/docs/prep_16_2006.pdf.

59. Орлов С. А. Технологии разработки программного обеспечения: [учеб. для вузов] / С. А. Орлов, Б. Я. Цилькер. – Санкт-Петербург: Питер, 2012. – 608 с.

60. Осипова Т. Ф. Моделирование процесса проектирования автоматизированной информационной системы структурным методом / Т. Ф. Осипова // Актуальные проблемы экономики и управления. – 2015. – № 2(6). – С. 89–96.

61. Пазухин А. В. Автоматизированное проектирование систем холодоснабжения: афтореф. дис. на соиск. уч. степени канд. тех. наук: спец. 05.13.12 «Системы автоматизации проектирования» / А. В. Пазухин – Санкт-Петербург: СПб ГУ ИТМО, 2008. – 25 с.

62. Палагин А. В. Методика проектирования онтологии предметной области / А. В. Палагин, Н. Г. Петренко, К. С. Малахов // Комп'ютерні засоби, мережі та системи. –2011. –№10. – С.5–12.

63. Перевозчикова О. Л. Диалоговые системы / О. Л. Перевозчикова, Е. Л Ющенко; [ин-т кибернетики им. В. М. Глушкова]. – Киев: Наук. Думка, 1990. – 184 с.

64. Подловченко Р. И. Конечные автоматы в теории алгебраических схем программ / Р. И. Подловченко // Труды ИСП РАН. – 2015. – Т. 27. – №. 2. – С.161–171.

65. Поликарпова Н. И. Автоматное программирование / Н. И. Поликарпова, А. А. Шалыто. – Санкт-Петербург: Питер, 2008. – 167 с.

66. Поляков А. Ю. Оптимизация времени создания и объема контрольных точек восстановления параллельных программ / А. Ю. Поляков, А. А. Данекина // Вестник СибГУТИ. –2010. – №2 –С.87–100.

67. Попова-Коварцева Д. А. Алгоритмы анализа и синтеза управляющих графов в задачах организации параллельных вычислений: афтореф. дис. на соиск. уч. степени канд. тех. наук: спец. 05.13.01 «Системный анализ, управление и обработка информации (технические системы и связь)» / Д. А. Попова-Коварцева. – Самара: Самарский государственный аэрокосмический университет им. академика С.П. Королева (Национальный исследовательский университет), 2013. – 20 с.

68. Построение автоматных программ по спецификации с помощью муравьиного алгоритма на основе графа мутаций / Чивилихин Д. С., Ульянцев В. И., Вяткин В. В. [и др.] // Научно-технический вестник информационных технологий, механики и оптики. – 2014. – № 6 (94). – С. 98–105.

69. Проектирование и разработка многоагентных систем / Е. А. Крушиневский , Д. А. Попов , А. Н. Матлаш [и др.] // Наука, образование и культура. – 2016. – №. 8. – С. 5–7.

70. Салмре И. Программирование мобильных устройств на платформе .NET Compact Framework / Иво Салмре; [пер. с англ.]. – Москва: Вильямс, 2006. – 736 с.

71. Самсонкин А. Н. Особенности организации управления эффективностью информационно-управляющих систем на этапе эксплуатации / Самсонкин А. Н., Чайников С. И. // Радиоэлектроника и информатика. – 1999. – №2(7) – С.57–59.

72. Седжвик Р. Фундаментальные алгоритмы на С ++. Часть 5. Алгоритмы на графах / Роберт Седжвик. – Москва: ДиаСофтЮП, 2002. – 496 с.

73. Сергеев Ю. Управление доступом к виртуальной инфраструктуре с помощью продукта HyTrust / Ю. Сергеев // Jet Info Информационный бюллетень. – 2012. – №3 (224). – 44 с.

74. Солодовников А. С. К вопросу оценивания эффективности и сложности структуры программного средства / А. С. Солодовников // Проблеми інформаційних технологій. – 2014. – №2 (16). – С.125–129.

75. Солодовников А. С. Об одном подходе к управлению вычислительными процессами в проблемно-ориентированных информационных системах / А. С. Солодовников // Информационные технологии в управлении (ИТУ-2014): междунар. конф. – Санкт-Петербург: ОАО «Концерн «ЦНИИ «Электроприбор», 2014. – С.245–247.

76. Солодовников А.С. Синтез проблемно-ориентированного программного комплекса на основе модели предметной области (по материалам диссертационного исследования) / А. С. Солодовников, С. И. Чайников // Информационные системы и технологи (ИСТ 2015): междунар. научн.-техн. конф., 21-27 сентября 2015г. – Харьков: НТМТ – 2015. С.108-109.

77. Суровцева О. А. Использование потенциала САПР ТП «ТехноПро» для формирования интегрированных комплексов на основе CALS технологий // Состояние и перспективы развития сельскохозяйственного машиностроения: 9-ая

междунар. научн.-практ. конф. в рамках 19-й междунар. агропромышленной выставки «Интерагромаш-2016», 2016. – Т. 9. – С. 330–332.

78. Туккель Н. И. SWITCH-технология – автоматный подход к созданию «реактивных» систем / Н. И. Туккель, А. А. Шалыто // Программирование. – 2001.– № 5. – С. 45–62.

79. Тушавин В. А. Кайдзен и Scrum проекты как инструмент организационного научения в ИТ-компании / В. А. Тушавин // Научный журнал НИУ ИТМО. Серия «Экономика и экологический менеджмент». – 2014. – № 2. – С. 80–91.

80. Тыугу Э. Х. Вычислительные фреймы и структурный синтез программ / Э. Х. Тыугу // Изв. АН СССР, Техническая кибернетика. – 1982. – № 6. – С. 176–182.

81. Фельдман Л. П. Эффективность реализации параллельных вычислений для кластерных систем на базе интерфейса MPI / Л. П. Фельдман, И. А. Назарова // Наукові праці Донецького національного технічного університету. Серія : Інформатика, кібернетика та обчислювальна техніка. – 2016. – № 1. – С. 136–141.

82. Царёв И. В. ЯРД – язык сетевого программирования в распределенных вычислительных системах с динамической архитектурой / И. В. Царёв. // Штучний інтелект. – 2008. – №3. – С.761–770.

83. Цейтлин Г. Е. Введение в алгоритмику / Г. Е. Цейтлин – Киев: Сфера – 1998. – 311 с.

84. Чайников С. И. Использование граф-модели предметной области при разработке программных средств / С. И. Чайников, А. С. Солодовников // Информационные системы и технологии: междунар. научн.-техн. конф., Морское-Харьков, 22-29 сентября 2012 г.: тезисы докл. – Харьков: НТМТ, 2012. – С.66.

85. Чайников С. И. К вопросу организации контрольных точек восстановления данных вычислительных процессов / С. И. Чайников, А. С. Солодовников // Бионика интеллекта: науч.-техн. журнал. – 2014. – №2(83).– С. 128-131.

86. Чайников С. И. Методы и алгоритмы априорной оценки параметров вычислительных процессов: автореф. дис. на соиск. уч. степени канд. техн. наук: спец. 05.13.01 «Техническая кибернетика и теория информации» / С. И. Чайников.– Харьков: ХИРЭ – 1983. – 16 с.

87. Чайников С. И. Об одном подходе к методике разработки сложных диалоговых систем / С. И. Чайников, А. С. Солодовников // Радиоэлектроника и молодежь в XXI веке: 15-ый юбилейный междунар. молодежный форум, 18-20 апреля 2011 г. – Харьков: ХНУРЭ, 2011. – Том 9. – С.160–161.

88. Чайников С. И. Оптимизация топологии модели предметной области и оценка эффективности структуры программного средства / С. И. Чайников, А. С. Солодовников // Информационные системы и технологи (ИСТ 2014): 3-я междунар. научн.-техн. конф., 15-21 сентября 2014 г. – Харьков: ХНУРЭ, 2014 – С.101–103.

89. Чайников С. И. Организация вычислительных процессов для заданной предметной области с использованием модели конечных автоматов / С. И. Чайников, А. С. Солодовников // Системи обробки інформації. – 2016. – №2 (139). – С.132–137.

90. Чайников С. И. Организация обмена данными между вычислительными процессами в диалоговой системе / С. И. Чайников, А. С. Солодовников // Системный анализ и информационные технологии (SAIT’2013): 15-ая междунар. конф., 27-31 мая 2013 г. – Киев.: «НТУ КПИ», 2013 – С. 493–494.

91. Чайников С. И. Принципы организации вычислений на базе граф-модели предметной области / С. И. Чайников, А. С. Солодовников // Бионика интеллекта: науч.-техн. журнал. – 2012. – №2 (79). – С. 72–75.

92. Чайніков С. І. Сучасні структурні методології проектування ІКС і особливості використання DFD та ERD / С. І. Чайніков // Автоматизированные системы управления и приборы автоматики. – 2002. – Выпуск 120. – С.63–72.

93. Чайников С. И. Формализованное описание граф-модели предметной области / С. И. Чайников, А. С. Солодовников // Бионика интеллекта: науч.-техн. журнал. – 2013. – №1 (80). – С.77–81.

94. Шалыто А. А. Switch-технология. Алгоритмизация и программирование задач логического управления / А. А. Шалыто.– Санкт-Петербург: Наука, 1998. – 628 с.

95. Шелехов В. И. Язык и технология автоматного программирования // Программная инженерия. – №4. – 2014. – С. 3–15.

96. Шкулипа И. Ю. Автоматизированный синтез программ на основе САА-М-схем / И. Ю. Шкулипа, С. Д. Погорелый // УСиМ. – 2010. – №4. – С. 58–63.

97. Шкуліпа І. Ю. Методи та засоби створення автоматизованої системи проектування паралельних алгоритмів: автореф. дис. на здобуття наукового ступеня канд. тех. наук 01.05.03 «Математичне та програмне забезпечення обчислювальних машин і систем» / І. Ю. Шкуліпа. – Київ: КНУ ім. Т. Шевченка. – 2011. – 23 с.

98. 2015 ERP report [Електронный ресурс] // Panorama Consulting Solutions, LLC. – Denver, Colorado, 2015. – Режим доступа: http://panorama-consulting.com/resource-center/2015-erp-report/

99. 2016 Report on ERP systems and enterprise software [Електронный ресурс]// Panorama Consulting Solutions, LLC. – Denver, Colorado, 2016. – 32 p. – Режим доступа: http://panorama-consulting.com/resource-center/2016-erp-report/

100. A Modular Reference Structure for Component-based Architecture Description Languages/Misha Strittmatter, Kiana Rostami, Robert Heinrich [et al.] // ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, September 28, 2015. – Ottawa, Canada, 2015. – P. 36–41.

101. Abdelmeged A. Navigating Object Graphs Using Incomplete Meta-Information / A. Abdelmeged, T. Skotiniotis, K. Lieberherr. – Technical Report NU-CCIS-10-2, CCIS/PRL, Northeastern University, Boston, 2010. – 13 p.

102. Adaptive incremental checkpointing for massively parallel systems / S. Agarwal, R. Garg, M. S. Gupta [at al.] // ICS`04: proc. of the 18th annual internat. conf. on supercomputing, June 26 – July 01, 2004. – Malo, France, 2004. – P. 277–286.

103. Adaptive page-level incremental checkpointing based on expected recovery time / Sangho Yi, Junyoung Heo, Yookun Cho [at al.] // Applied computing table of contents: proc. of the 2006 ACM symposium. – Dijon, France, 2006. – P. 1472–1476.

104. Adelsberger S. Towards Assessing the Complexity of Object Migration in Dynamic, Feature-Oriented Software Product Lines / S. Adelsberger, S. Sobernig, G. Neumann // Variability Modelling of Software-Intensive Systems: The Eight International Workshop, 2014. – P. 17–18.

105. Agile software development: Review and Analysis / [Abrahasson Pekka, Outi Salo, Jussi Ronkainen, Juhani Warsta]. – ESPOO 2002, VTT Publications 478, 2002. – 107 p.

106. Amodeo E. Learning Behavior-driven Development with javascript + Code / Enrique Amodeo. – Birmingham: Packt Publishing, 2015. – 392 p.

107. An evolutionary multiobjective optimization approach to component-based software architecture design / R. Li, R. Etemaadi, M.T.M. Emmerich, [at al.] // Congress on Evolutionary Computation (CEC), 5-8 June 2011. – New Orleans, LA.: IEEE, 2011. – P.432–439.

108. Ansel J. DMTCP: Transparent Checkpoint for Cluster Computations and the Desktop / J. Ansel, K. Arya, G. Cooperman // Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS`09). – IEEE Press, 2009. – P.1–12. – Режим доступа: http://dmtcp.sourceforge.net/papers/dmtcp.pdf

109. Application-transparent checkpoint/restart for MPI programs over InfiniBand / Q. Gao, W. Yu, W. Huang [at al.] // Parallel Processing. – 2006. – P.1–8. – Режим доступа http://mvapich.cse.ohio-state.edu:8080/static/media/publications/abstract/gaoq-icpp06.pdf.

110. A survey of rollback-recovery protocols in message-passing systems / Elnozahy E. N., Alvisi L., Wang Y. M. [at al.] // ACM Computing Surveys. – 2002. – Vol. 34 – №3. – P. 375–408.

111. Baresi L. Service-oriented dynamic software product lines / L. Baresi, S. Guinea, L. Pasquale // Computer. – 2012. – T. 45. – №. 10. – P. 42–48.

112. Biyikoglu T. Laplacian Eigenvectors of Graphs / T. Biyikoglu, J. Leydold, P. Stadler – Berlin: Springer, 2007. – 115 p.

113. Browne A. Linking STeP with SPIN / Browne Anca, Henny Sipma, Ting Zhang // Spin Model Checking And Software Verification: 7th International SPIN Workshop. – LNCS 1885, Springer-Verlag, 2000. – P.181–186.

114. Burns R. C. Efficient distributed backup with delta compression / R. C. Burns, D. D. E. Long // In Proceedings of the 1997 I/O in Parallel and Distributed Systems, (IOPADS'97), Nov. 1997. – San Jose, CA, USA, 1997. – P.1–11. – Режим доступа: http://hssl.cs.jhu.edu/papers/burns_iopads97.pdf.

115. Cao J. Dynamic configuration management in a graph-oriented Distributed Programming Environment / Jiannong Cao, Alvin Chan, Yudong Sun // Science of Computer Programming. – 2003. –№ 48. – P.43 – 65.

116. Cao J. GOP: a graph-oriented programming model for parallel and distributed systems / Jiannong Cao, Alvin Chan, Yudong Sun // New Horizons of Parallel and Distributed Computing. – Boston, MA: Springer. – 2005. – P.21–36.

117. Casadei R. Towards Aggregate Programming in Scala / R. Casadei, M.Viroli // First Workshop on Programming Models and Languages for Distributed Computing. – ACM, 2016. – P. 5.

118. Chainikov S. Information Technology of Software Architecture Structural Synthesis of Information System / S. Chainikov, A. Solodovnikov // EUREKA: Physical Science and Engineering. **–** 2016. **–**4(5). – P. 25–32.

119. Chadwick B. A functional approach to generic programming using adaptive traversals / B. Chadwick, K. Lieberherr // Higher-Order and Symbolic Computation. – 2010. – Vol. 23. – №. 4. – P. 433–463.

120. Checkpointing and Its Applications /Yi-Min Wang, Yennun Huang, Kiem-Phong Vo [at al.] // 25th Annual Int'l symposium on Fault-Tolerant Computing. –1995. – P. 22–30.

121. Coelho K. From Requirements to Architecture for Software Product Lines / K. Coelho, T. Batista // 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), 20-24 June 2011. – Boulder, CO: IEEE, 2011. – P. 282–289.

122. Computation, Causation and Discovery / Eds: C. Glymour and G.F. Cooper. – Menlo Park, CA, Cambridge, MA: AAAI Press, 1999. – 570 p.

123. Customer-Induced Interactions And Innovation In Professional Services: The Case Of Software Customisation / M. Schaarschmidt, W. Gianfranco, B. Matthias, V. K. Harald // International Journal Of Innovation Management. – 2015. – P.1–38. – Режим доступа: https://www.academia.edu/21938879/Customer-Induced_Interactions_and_Innovation_in_Professional_Services_The_Case_of_Software_Customization/

124. Cvetkovic D. Spectra of Graphs: Theory and Applications / D. Cvetkovic, M. Doob, H. Sachs. – Leipzig: Verlag, 1995. – 368 p.

125. Dan T. An Effort-Based Framework for Evaluating Software Usability Design / Dan Tamir, Carl J. Mueller, Oleg V. Komogortsev // ARPN Journal of Systems and Software. – 2013. –Vol. 3 – № 4. – P.65–77.

126. Deductive Verification of Real-time Systems using STeP / Nikolaj Bjorner, Zohar Manna, Henny Sipma, Tomas Uribe // Theoretical Computer Science. – 2001. – № 253. – P.27–60.

127. Drusinsky D. The Temporal Rover and the ATG Rover / Doron Drusinsky // International Workshop on Model Checking of Software – SPIN . – 2000. – P. 323–330.

128. Harel D. Modeling Reactive Systems with Statechart: The Statemate Approach / David Harel, Michal Politi. – NY: McGraw-Hill, 1998. – 258 p.

129. Hargrove P. H. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters / Paul H. Hargrove, Jason C. Duell // In Proceedings of SciDAC, June 2006. – 2006. – P.1–6. – Режим доступа: http://crd.lbl.gov/assets/pubs_presos/CDS/FTG/Papers/2006/LBNL-60520.pdf.

130. Hashemi S. M. Cloud Computing Vs. Grid Computing / Seyyed Mohsen Hashemi, Amid Khatibi Bardsiri // ARPN Journal of Systems and Software. – Vol. 2. – № 5. – 2012. P.188–194.

131. Hinchey M. Building dynamic software product lines / M. Hinchey, S. Park, K. Schmid // Computer. – 2012. – Vol. 45. – №. 10. – P. 22–26.

132. Holzmann G.J. The Model Checker SPIN / G.J. Holzmann // IEEE Transactions on Software Engineering, May 1997. – Vol. 23. – № 5. – P. 279–295.

133. HwaYoung J. A Practical Web Service Composition / J. HwaYoung, H. BongHwa // Parallel and Distributed Processing with Applications Workshops (ISPAW), 2011 9th IEEE International Symposium, – 2011, Busan.: IEEE. – P.367–370.

134. Learning and Evolution in Dynamic Software Product Lines / Amir Molzam Sharifloo, Andreas Metzger, Clement Quinton, [at al.] – 2016. – 8 p. – Режим доступа: https://hal.archives-ouvertes.fr/hal-01280837

135. Lieberherr K. Adaptive object-oriented software the demeter method // PWS Boston. – 1996. – 651 p.

136. Lifecycle Management of Open-Source Software in the Public Sector. A Model for Community-Based Application Evolution / Ju. Kääriäinen, P. Pussinen, T. Matinmikko, T. Oikarinen // ARPN Journal of Systems and Software. – 2012. – Vol. 2. – № 11. – P.279-288.

137. Marciuska S. Automated Feature Identification in Web Applications / S. Marciuska, C. Gencel, P. Abrahamsson // International Conference on Software Quality. – Springer International Publishing, 2014. – P. 100–114.

138. Masahiro Sh. Dialog System for Open-Ended Conversation Using Web Documents / Masahiro Shibata, Tomomi Nishiguchi, Yoichi Tomiura // Informatica. – 2009. – № 33. – P.277–284.

139. McMillan K.L Symbolic Model Checking / Kenneth L. McMillan. – Kluwer Academic Press, 1993. – 194 p.

140. Meyer B. Touch of Class: Learning to Program Well with Objects and Contracts / Bertrand Meyer. – [2nd printing edition]. – Springer, 2013. – 876 p.

141. Mohammadkhanli L. Ranking Approaches for Cloud Computing Services Based on Quality of Service: A Review / Leyli Mohammadkhanli, Arezoo Jahani //ARPN Journal of Systems and Software. – 2014. – Vol. 4. – № 2. – P.55–62.

142. NuSMV 2: An open source tool for symbolic model checking / Cimatti A., Clarke E., Giunchiglia E [et al.] // In Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02), Lecture Notes in Computer Science 2404. – 2002. – P.359–364.

143. Nwankpa J. K. Real Options and Subsequent Technology Adoption: An ERP System Perspective / J. K. Nwankpa, Y. Roumani // System Sciences (HICSS): 48th Hawaii International Conference. – IEEE, 2015. – P. 5020–5027.

144. On the Composition and Reuse of Viewpoints across Architecture Frameworks / R. Hilliard, I. Malavolta, H. Muccini [at al.] // Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 20-24 Aug. 2012. – Helsinki.: IEEE, 2012. – P.131–140.

145. Pandey R. K. Architectural description languages (ADLs) vs UML: a review / R. K. Pandey // ACM SIGSOFT Software Engineering Notes. – 2010. – Vol.35.– Issue 3. – P.1–5.

146. Pearl J. Probabilistic reasoning in intelligent systems: networks of plausible inference / Judea Pearl. – San Mateo: Morgan Kaufmann. –1988. – 480 p.

147. Pearl J. Causality: Models, Reasoning and Inference / Judea Pearl // Economic Theory. – 2003. – №19. – P.675–685.

148. Poskitt C. M. Verifying Monadic Second-Order Properties of Graph Programs / C. M. Poskitt, D. Plump // Lecture Notes in Computer Science. – 2014. – P.33–48.

149. Proactive fault tolerance for HPC with Xen virtualization / A.B. Nagarajan, F. Mueller, C. Engelmann, S. L. Scott // ICS 2007: proc. of the 21st Annual International Conference on Supercomputing. – ACM, New York, 2007. – P. 23–32.

150. Proactive process-level live migration in HPC environments / C. Wang, F. Mueller, C. Engelmann, S. L. Scott // In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC). – 2008. – P.1–12.

151. Robby A. Bogor: an extensible and highly-modular software model checking framework / A. Robby, M. Dwyer, J. Hatcliff // Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE September 1-5, 2003. – Helsinki, Finland, 2003. – P.1–10.

152. Rosenmüller M. Tailoring dynamic software product lines // ACM SIGPLAN Notices. – ACM, 2011. – Vol. 47. – №. 3. – P. 3–12.

153. Schröter C. The Model-Checking Kit / Claus Schröter, Stefan Schwoon, Javier Esparza // In Proceedings of the Tools Day Workshop, August 2002. – Brno, Czech Republic, 2002. – P.22–31.

154. Smart J. F. BDD in Action: Behavior-driven development for the whole software lifecycle / John Ferguson Smart / Publisher: Manning Publications, Shelter Island, NY, 2015. – 384 p.

155. Solodovnikov A. Developing Method for Assessing Functional Complexity of Software Information System. EUREKA: Physical Science and Engineering. – 2016.–*5*(6). – P. 3–9.

156. Tarjan R. E. Depth-first search and linear graph algorithms / R. E. Tarjan // SIAM Journal on Computing. – 1972. – Vol. 1. – № 2. – P. 146–160.

157. The design and implementation of checkpoint/restart process fault tolerance for Open MPI / Hursey J., Squyres J.M., Mattox T.I. [at al.] // In Proceedings of the 21st IEEE International Parallel and stributed Processing Symposium (IPDPS). – IEEE Computer Society. – 2007. – Vol. 3– №26 – P.1–8.

158. The TETRAD Project: Constraint Based Aids to Causal Model Specification / Richard Scheines, Peter Spirtes, Clark Glymour [et al.] // Multivariate Behavioral Research. – 1998. – Vol. 33 –№ 1. – P.65–118.

159. Woods E. Using an Architecture Description Language to Model a Large-Scale Information System – An Industrial Experience Report / E. Woods, R. Bashroush // Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 20-24 Aug. 2012 – Helsinki.: IEEE, 2012. – P.239–243.

ПРИЛОЖЕНИЯ

Приложение А

Код класса «Condensation», реализующего методы работы с графовой моделью архитектуры программного обеспечения информационной системы

```
public class Condensation
{
    private static int N = 0;
    private static int[,] g;
    private static int[,] gr;
    private static string order = "-1";
    private static string component = "-1";
    private static bool[] used;
    private int[,] MatrixSmej;
    public int[,] MatrixSmejnosti
    {
        get
        { return MatrixSmej; }
        set
        {
            MatrixSmej = value;
            N = MatrixSmej.GetLength(0);
            g = new int[N, N];
            gr = new int[N, N];
            used = new bool[N];
            order = "-1";
            component = "-1";
            ArraysInitialization();//ввод данных из матрицы смежности в рабочие массивы
        }
    }

    private void ArraysInitialization()
    {
      for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                g[i, j] = -1;
                gr[i, j] = -1;
            }
        }

        int[,] MatrTransp = new int[N, N];

        for (int i = 0; i < N; i++)
```

```
    {
        for (int j = 0; j < N; j++)
            MatrTransp[i, j] = MatrixSmej[j, i];
    }

    int k = 0;
    int l = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (MatrixSmej[i, j] == 1)
            {
                g[i, k] = j;
                k++;
            }
            if (MatrTransp[i, j] == 1)
            {
                gr[i, l] = j;
                l++;
            }
        }
        k = 0; l = 0;
    }

    for (int i = 0; i < N; i++)
    {
        used[i] = false;
    }

}

public void FindCycles(ref List<int[]> ncomponents) //ref int[,] CycleSequence
{
    ncomponents.Clear();
    string[] norder = new string[1];
    for (int i = 0; i < N; i++)
    {
        if (!used[i])
            dfs1(i);
    }

    string[] sbuf = order.Split('|');
    int[] order_int = new int[sbuf.GetLength(0) - 1];
    int k = 0;
    for (int i = 0; i < sbuf.GetLength(0); i++)
    {
        if (Convert.ToInt32(sbuf[i]) > -1)
```

```
        {
            order_int[k] = Convert.ToInt32(sbuf[i]);
            k++;
        }
    }

    for (int i = 0; i < N; i++)
    {
        used[i] = false;
    }

    for (int i = 0; i < N; i++)
    {
        int v = order_int[N - 1 - i];
        if (!used[v])
        {
            dfs2(v);
            norder = component.Split('|');
            //вывод компоненты
            int[] currcomp = new int[norder.GetLength(0) - 1];
            int xx = 0;
            for (int x = 0; x < norder.GetLength(0); x++)
            {
                if (norder[x] != "-1" && norder[x] != "")
                {
                    currcomp[xx] = Convert.ToInt32(norder[x]);
                    xx++;
                }
            }
            ncomponents.Add(currcomp);
            component = "-1";
        }
    }

}

private static void dfs1(int v)
{
    used[v] = true;
    for (int i = 0; i < N; i++)
    {
        if (g[v, i] != -1 && !used[g[v, i]])
        {
            dfs1(g[v, i]);
        }
    }
    order = order + "|" + v.ToString();
```

```
    }


    private static void dfs2(int v)
    {
       used[v] = true;
       component = component + "|" + v.ToString();
       for (int i = 0; i < N; i++)
       {
          if (gr[v, i] != -1 && !used[gr[v, i]])
          {
             dfs2(gr[v, i]);
          }
       }

    }

    private bool InArray(int val, int[] arr)
    {
       for (int i = 0; i < arr.GetLength(0); i++)
       {
          if (val == arr[i])
          {
             return true;
          }
       }
       return false;
    }
    private void RelabelingOfNodesInComponent(ref List<int[]> L_component, int
LastConstantNode, int delta, int current_comp_index)
    {
       //
       for (int i = 0; i < L_component.Count; i++)
       {
          if (current_comp_index != i)
          {
             int[] but = L_component[i];
             for (int j = 0; j < but.GetLength(0); j++)
             {
                if (but[j] > LastConstantNode)
                {
                   but[j] = but[j] - delta;
                }
             }
          }
       }
    }
```

```
public AdjacencyMatrix GraphCondensation_1DStandard(string fProjectName, string ModelName, string fReportNameLog, string SeparateFileWithCondensat, bool ShowRepMsg)
{
        string additional_errors = "";
        bool DoLog = false;
        bool DoSeparateFileWithCondensat = false;
        if (fReportNameLog != null && fReportNameLog != "")
        {
           DoLog = true;
        }

        if (SeparateFileWithCondensat != null && SeparateFileWithCondensat != "")
        {
           DoSeparateFileWithCondensat = true;
        }
        int[,] MatrixSmej1 = new int[1, 1];
        if (fProjectName == null || fProjectName == "")
        {
           OpenFileDialog fdOpen = new OpenFileDialog();
           fdOpen.InitialDirectory = System.Windows.Forms.Application.StartupPath;
           fdOpen.ShowDialog();
           if (fdOpen.FileName != "")
           {
              fProjectName = fdOpen.FileName;
              Functions.SaveFileHistory(fdOpen.FileName);
           }
           else
              return null;
        }

        Functions.GraphFillingFromFile(ModelName, ref MatrixSmej1);
        AdjacencyMatrix AdjMatr = new AdjacencyMatrix(1);
         AdjacencyMatrix resAdjMatr = new AdjacencyMatrix(1);
        Functions.Copy2D_to_AdjacencyMatrix_1DStandard(MatrixSmej1, ref AdjMatr);

        // сохраняем полную 1D матрицу смежности в файл проекта
        SaveAdjacencyMatrixToProject(fProjectName, AdjMatr);

      //запуск конденсации
        Condensation tst = new Condensation();
        tst.MatrixSmejnosti = MatrixSmej1;
        List<int[]> CompList = new List<int[]>();
        tst.FindCycles(ref CompList);

        int[,] rres = new int[1, 1];

        //1-сохраняем компоненты в соответствующий раздел файла проекта
```

```
        SaveVertexCharacteristicsToProject(fProjectName, CompList, AdjMatr);

        int[] ListOfSN = new int[1];

       AdjacencyMatrix _SuperGraph = tst.Graph_Enumeration_1DStandard(AdjMatr,
CompList);

        if (resAdjMatr == null)
        {
           additional_errors = "Конденсация графа закончилась неудачей. [Ошибка #0001]";
        }
        else
        {
           if (DoSeparateFileWithCondensat)
           {
            Functions.SaveMatrixAsXML_1DStandard(resAdjMatr,
SeparateFileWithCondensat, "CondencedGraph_1DStandard");

           }
           else
           {

              SaveCondencedGraphToProject(fProjectName, _SuperGraph);
              SaveAdjacencyMatrixToProject(fProjectName, AdjMatr);
           }
        }

         if (DoLog)
        {
           string[] msg_string = new string[1];
           msg_string[0] = "Конденсация графа завершена. На графе найдены
компоненты:";
           int[] compnum = new int[1];
           int k = 0;
           for (int i = 0; i < CompList.Count; i++)
           {
              if (CompList[i].Count() > 1)
              {
                 var arr = CompList[i];
                 UsrFunctions.Arrays.IncreaseLengthArray(ref msg_string, 1);
                 msg_string[k + 1] = "Компонента " + (k + 1).ToString() + ": ";
                 for (int h = 0; h < arr.GetLength(0); h++)
                 {
                    if (h == 0)
                    {
                       msg_string[k + 1] = msg_string[k + 1] + arr[h].ToString();
                    }
                    else
```

```
                {
                    msg_string[k + 1] = msg_string[k + 1] + "-" + arr[h].ToString();
                }
            }

            k++;
        }
    }


    if (DoSeparateFileWithCondensat)
    {
        UsrFunctions.Arrays.IncreaseLengthArray(ref msg_string, 1);
        msg_string[msg_string.GetLength(0) - 1] = "Конденсированный граф сохранен
в файл: " + SeparateFileWithCondensat;
    }
    else
    {
        UsrFunctions.Arrays.IncreaseLengthArray(ref msg_string, 1);
        msg_string[msg_string.GetLength(0) - 1] = "Конденсированный граф сохранен
в файл проекта: " + fProjectName;
    }

    if (additional_errors.Length > 2)
    {
        UsrFunctions.Arrays.IncreaseLengthArray(ref msg_string, 1);
        msg_string[msg_string.GetLength(0) - 1] = additional_errors;
    }
    Functions.SaveMsgToTxtFile(msg_string, fReportNameLog, false);
}

if (ShowRepMsg)
{
    string additional_msg = "";
    if (DoLog)
    {
        additional_msg = (char)13 + "Отчет сохранен в файл: " + fReportNameLog;
    }

    if (DoSeparateFileWithCondensat)
    {
        System.Windows.MessageBox.Show("Конденсация графа закончена!" +
(char)13 +
            "Конденсированный граф сохранен в файл: " + SeparateFileWithCondensat
+ additional_msg);

    }
    else
```

```
            {
                System.Windows.MessageBox.Show("Конденсация графа закончена!" + additional_msg);
            }
        }

        if (_SuperGraph != null && _SuperGraph.Data != null && _SuperGraph.Data.GetLength(0) > 1)
        {
            //все закончилось, выставляем стадийность проекта
            UsrFunctions.Staging.StagingOfProjectXML(fProjectName, ProjectStageConvert.ProjectStage.S3_condensed.ToString(), true);
        }
        return _SuperGraph;
    }


    public AdjacencyMatrix Graph_Enumeration_1DStandard(AdjacencyMatrix AdjMatrix, List<int[]> L_component)
    {
        AdjacencyMatrix ResulAdjacencyMatrix = new AdjacencyMatrix(1);

        ResulAdjacencyMatrix.Data = new int[AdjMatrix.Data.GetLength(0), AdjMatrix.Data.GetLength(1)];
        Array.Copy(AdjMatrix.Data, 0, ResulAdjacencyMatrix.Data, 0, AdjMatrix.Data.Length);

        List<int[]> Buf_L_component = new List<int[]>();
        Buf_L_component = L_component;
        int[] IsolatedV = new int[1];
        bool stop_func = true;
        for (int i = 0; i < Buf_L_component.Count; i++)
        {
            if (Buf_L_component[i].GetLength(0) > 1)
                stop_func = false;
        }
        if (stop_func)
        {
            ResulAdjacencyMatrix = AdjMatrix;
            return null;
        }

        //формируем список вершин, которые будут свернуты.  Одновременно с этим получаем список суппервершин.
        List<int[]> Buf_comp = new List<int[]>();
        for (int i = 0; i < L_component.Count; i++)
        {
            if (L_component[i].GetLength(0) > 1)
```

```
            {
               Buf_comp.Add(L_component[i]);
            }
         }
         for (int i = 0; i < Buf_comp.Count; i++)
         {
            int[] next_comp = Buf_comp[i];
            for (int j = 0; j < AdjMatrix.Data.GetLength(0); j++)
            {
               if (Functions.FindFirstElementInArray(next_comp, AdjMatrix.Data[j, 1]) >= 0
&& AdjMatrix.Data[j, 1] != next_comp.Min())
               {
                  ResulAdjacencyMatrix.Data[j, 1] = next_comp.Min();
               }

               if (Functions.FindFirstElementInArray(next_comp, AdjMatrix.Data[j, 0]) >= 0
&& AdjMatrix.Data[j, 0] != next_comp.Min())
               {
                  ResulAdjacencyMatrix.Data[j, 0] = next_comp.Min();
               }
            }
         }
         var                                          bb                                          =
UsrFunctions.Arrays.DeleteAAElementsFrom_Nx2_Array(ResulAdjacencyMatrix.Data);
         ResulAdjacencyMatrix.Data = bb;
         var                                          dd                                          =
UsrFunctions.Arrays.DeleteDoubleLinesFrom_Nx2_Array(ResulAdjacencyMatrix.Data);
         ResulAdjacencyMatrix.Data = dd;
         return ResulAdjacencyMatrix;

      }//component list
   } //
}
```

Приложение Б

Код функции формирования подграфа задачи на графовой модели архитектуры ПО ИС

```
 private void mnuGetGraph_Click(object sender, System.EventArgs e) //получение подграфа
     {
        var bb = this as Vershina_wpf;
        MultilevelGraph.SubGraph sbGraph = new MultilevelGraph.SubGraph(bb._ProjectFilePath, bb.VNumber - 1);
        DiagramControls.Controls.NodeMap NodeMapping = new DiagramControls.Controls.NodeMap();
        NodeMapping.SubGraph = sbGraph.SubGraphRes.Data;
        NodeMapping.BaseNode = (bb.VNumber - 1);
        NodeMapping.CurrFileOfProject = "";// this.CurrentModelProjectFilePath;
        NodeMapping.InitializeAllParam();
        //NodeMapping.cvLayout
        Graph.GraphFunctions.LoadModelToCanvas_BorderNodes(bb._ProjectFilePath, NodeMapping.cv_Layout, bb._mAdjMatr, bb.VNumber);
        NodeMapping.WindowStartupLocation= System.Windows.WindowStartupLocation.CenterScreen;
        NodeMapping.Show();
     }
```

Приложение В

Код программы верификатора PROMELA – JSPIN

```
#define p1 (A1Events=12)
#define p2 (events[0]=23)


/*AServerSM*/


int A1States;
int A1Events;


ltl { !(<>((A1Events==12)) V (A1States==16)) }


proctype A1()  {
A1States=0;
A1Events=0;
    do
    ::(A1States==0) ->
            do
                    ::A1States=1; A1Events=24;
                    ::A1States=10; A1Events=102;
                    ::break
            od;
    ::(A1States==1) ->
            A1Events=25;
            A1States=2;
    ::(A1States==2) ->
            do
                    ::A1Events=22;A1States=2;
                    ::A1Events=21;A1States=3;
                    ::A1Events=23;A1States=4;
                    ::break
            od;
    ::(A1States==3) ->
            do
                    ::A1Events=30;A1States=5;
                    ::A1Events=31;A1States=10;
```

```
                ::break
            od;
        ::(A1States==4) ->
            do
                ::A1Events=29;A1States=10;
                ::A1Events=28;A1States=6;
                ::break
            od;
        ::(A1States==5) ->
            do
                ::A1Events=27;A1States=10;
                ::A1Events=23;A1States=4;
                ::break
            od;
        ::(A1States==6) ->
                A1Events=26;A1States=7;
        ::(A1States==7) ->
            do
                ::A1Events=31;A1States=10;
                ::A1Events=30;A1States=8;
                ::break
            od;
        ::(A1States==8) ->
            do
                ::A1Events=27;A1States=10;
                ::A1Events=28;A1States=10;
                ::A1Events=29;A1States=9;
                ::break
            od;
        ::(A1States==9) ->
            A1Events=23;
            A1States=1;
        ::(A1States==10) ->
            do
                ::A1Events=102;break;
                ::A1Events=29;break;
                ::A1Events=31;break;
                ::A1Events=27;break;
                ::A1Events=28;break;
                ::break
            od;
    od;
}
```

```
init {
     A1Events=0;
     A1States=0;
    run A1();
    }


/*AForwardSM*/


#define p1 (A1Events=12)
#define p2 (events[0]=23)

int A1States;
int A1Events;

ltl { !(<>((A1Events==12)) V (A1States==16)) }

proctype A1()  {
A1States=0;
A1Events=0;
     do
     ::(A1States==0) ->
               do
                         ::A1States=1; A1Events=1;
                         ::A1States=16; A1Events=2;
                         ::A1States=16; A1Events=3;
                         ::A1States=16; A1Events=4;
                         ::A1States=16; A1Events=12;
                         ::break
               od;
     ::(A1States==1) ->
               A1Events=11;
               A1States=2;
     ::(A1States==2) ->
               A1Events=10;
               A1States=3;
     ::(A1States==3) ->
               do
                         ::A1Events=122;A1States=4;
                         ::A1Events=122;A1States=5;
                         ::A1Events=121;A1States=7;
                         ::break;
               od;
     ::(A1States==4) ->
```

```
        A1States=3;
        A1Events=123;
::(A1States==5) ->
        do
                ::A1Events=6;A1States=3;
                ::A1Events=8;A1States=6;
                ::break
        od;
::(A1States==6) ->
        do
                ::A1Events=7;A1States=2;
                ::A1Events=9;A1States=16;
                ::break
        od;
::(A1States==7) ->
        A1Events=13;
        A1States=8;
::(A1States==8) ->
        do
                ::A1Events=121;A1States=9;
                ::A1Events=121;A1States=12;
                ::A1Events=122;A1States=9;
                ::A1Events=122;A1States=10;
                ::break
        od;
::(A1States==9) ->
        do
                ::A1Events=14;A1States=8;
                ::A1Events=15;A1States=10;
                ::break
        od;
::(A1States==10) ->
        A1Events=123;
        A1States=12;
::(A1States==12) ->
        A1Events=16;
        A1States=11;
::(A1States==11) ->
        do
                ::A1Events=5;A1States=15;
                ::A1Events=122;A1States=5;
                ::A1Events=122;A1States=13;
                ::break
        od;
::(A1States==13) ->
```

```
            do
                ::A1Events=44;A1States=13;
                ::A1Events=45;A1States=16;
                ::A1Events=43;A1States=14;
                ::break
            od;
    ::(A1States==14) ->
            A1Events=42;A1States=13;
    ::(A1States==15) ->
            A1Events=17;A1States=5;
    ::(A1States==16) ->
            do
                ::A1Events=2;break;
                ::A1Events=3;break;
                ::A1Events=4;break;
                ::A1Events=12;break;
                ::break
            od;
    od;
}

init {
    A1Events=0;
    A1States=0;
   run A1();
}



/*ABackwardSM*/


#define p1 (A1Events=12)
#define p2 (events[0]=23)

int A1States;
int A1Events;

ltl { !(<>((A1Events==12)) V (A1States==16)) }

proctype A1()  {
A1States=0;
A1Events=0;
    do
    ::(A1States==0) ->
```

```
        do
                ::A1States=1; A1Events=1;
                ::A1States=10; A1Events=2;
                ::A1States=10; A1Events=3;
                ::A1States=10; A1Events=4;
                ::A1States=10; A1Events=12;
                ::break
        od;
::(A1States==1) ->
        A1Events=18;
        A1States=2;
::(A1States==2) ->
        A1Events=10;
        A1States=3;
::(A1States==3) ->
        do
                ::A1Events=82;A1States=6;
                ::A1Events=81;A1States=4;
                ::break
        od;
::(A1States==4) ->
        do
                ::A1Events=101;A1States=5;
                ::A1Events=102;A1States=4;
                ::A1Events=103;A1States=5;
                ::A1Events=121;A1States=5;
                ::A1Events=122;A1States=4;
                ::break
        od;
::(A1States==5) ->
                A1Events=1;
                A1States=7;
::(A1States==6) ->
                A1Events=17;
                A1States=8;
::(A1States==7) ->
        A1Events=103;
        A1States=6;
::(A1States==8) ->
        do
                ::A1Events=6;A1States=3;
                ::A1Events=8;A1States=9;
                ::break
        od;
::(A1States==9) ->
```

```
            do
                ::A1Events=7;A1States=2;
                ::A1Events=9;A1States=10;
                ::break
            od;
    ::(A1States==10) ->
            do
                ::A1Events=2;break;
                ::A1Events=3;break;
                ::A1Events=4;break;
                ::A1Events=12;break;
                ::break
            od;
    od;
}

init {
    A1Events=0;
    A1States=0;
    run A1();
}
```

Приложение Г

Вспомогательные диаграммы. Структурные схемы. Блок-схемы алгоритмов

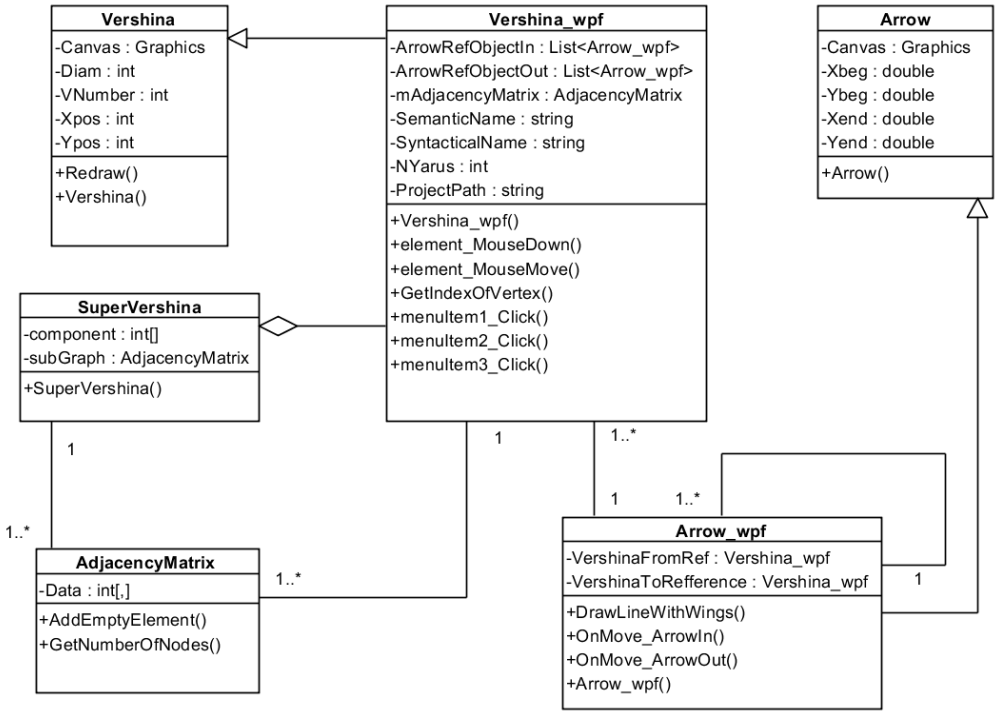


Рисунок 1 – Структура программного модуля, реализующего визуализацию графовой модели на экране инструментария

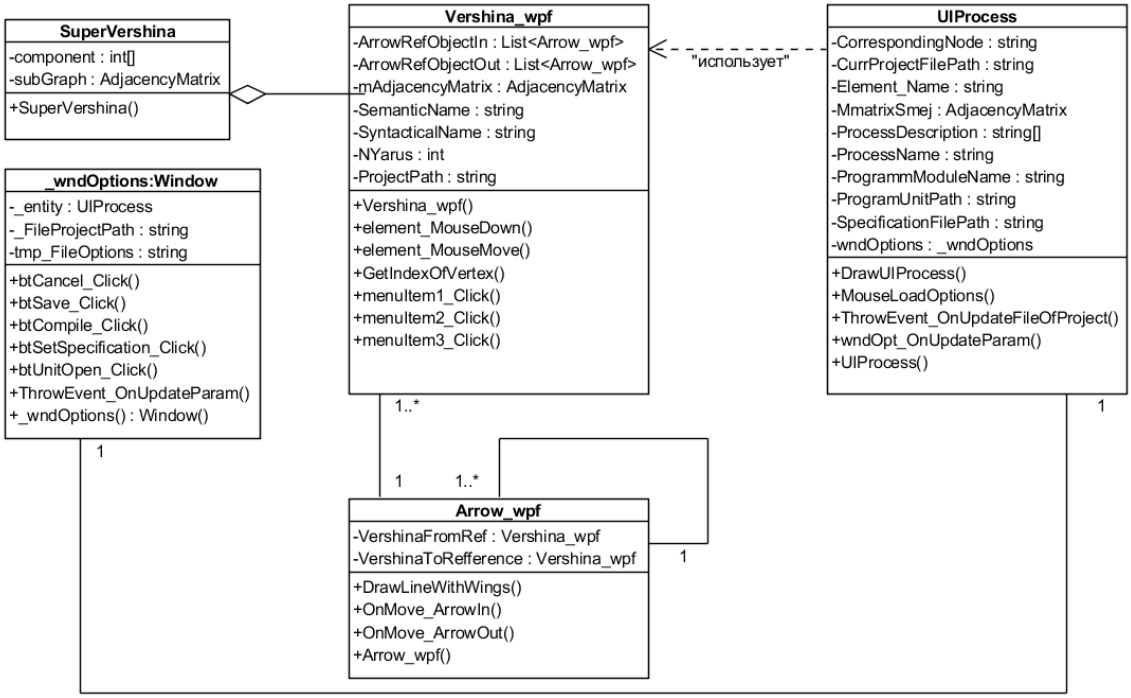


Рисунок 2 – Структура программного модуля, реализующего отображение элементов графовой модели на архитектуру ПО

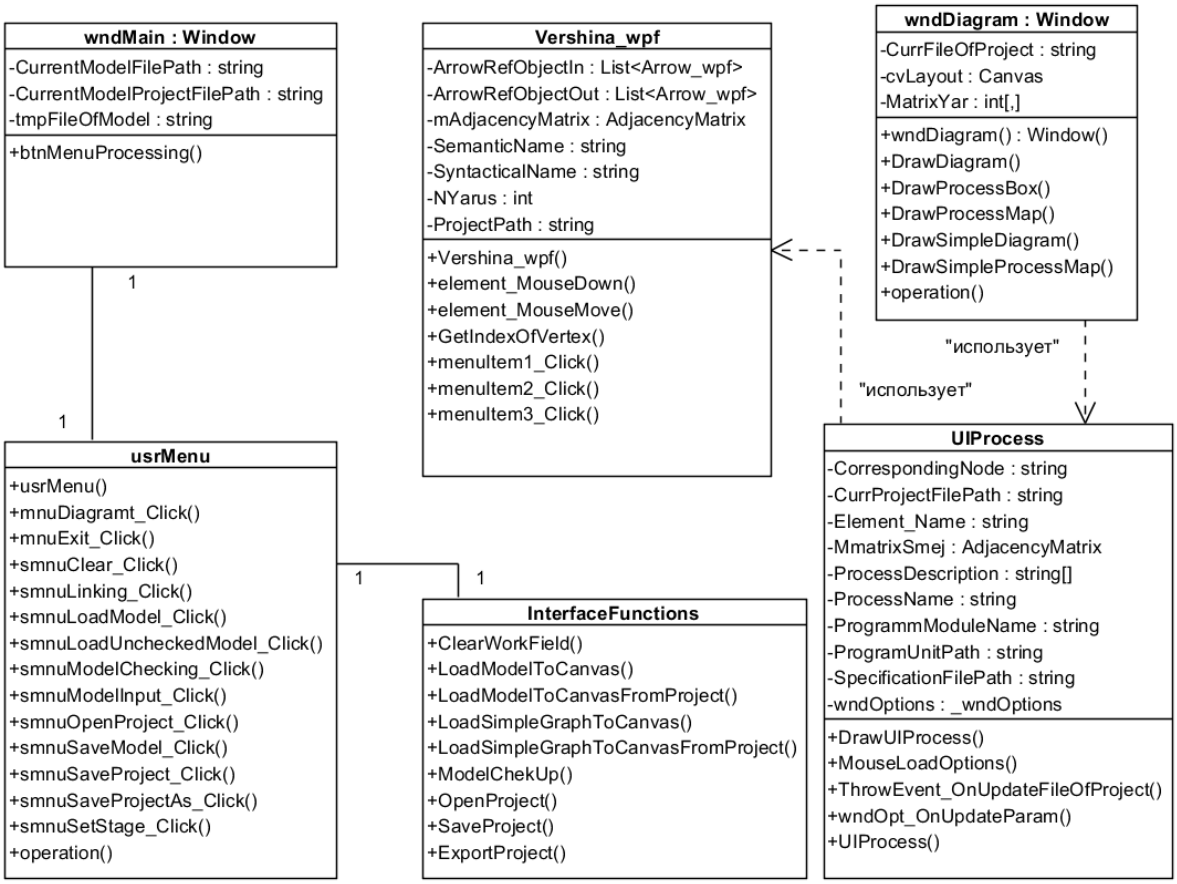


Рисунок 3 – Структура, описывающая взаимодействие пользовательских окон с метаинформацией графовой модели

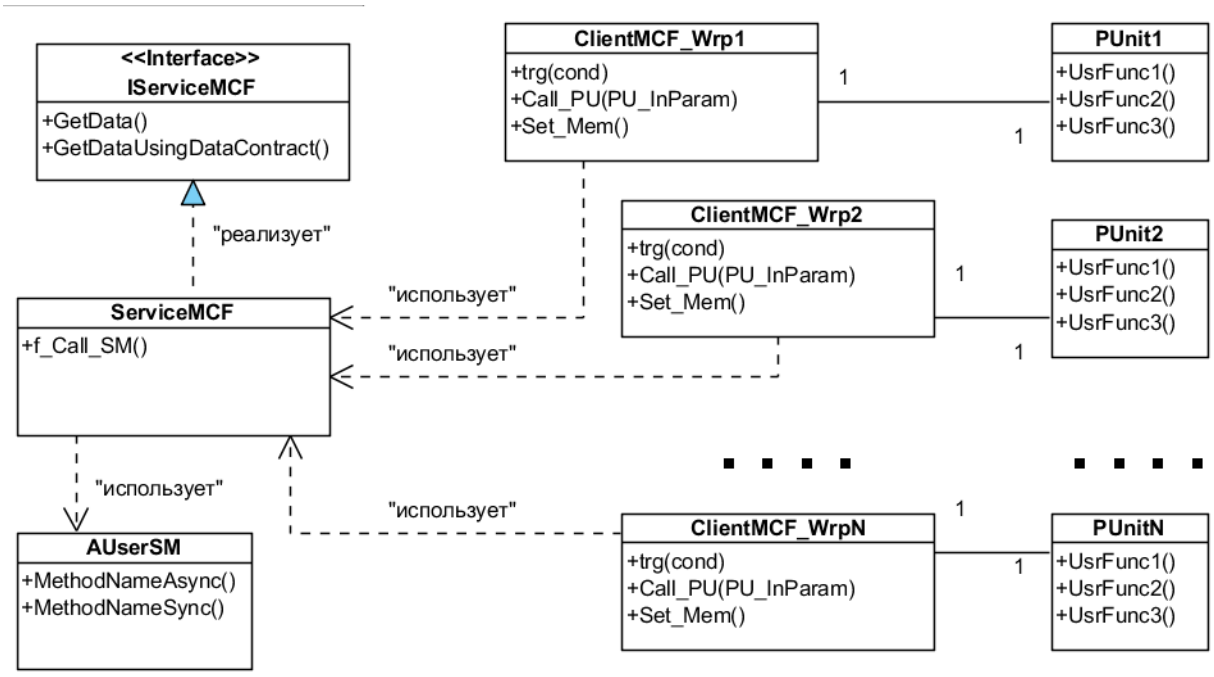


Рисунок 4 – Шаблон организации распределенного выполнения программы с использованием службы MCF

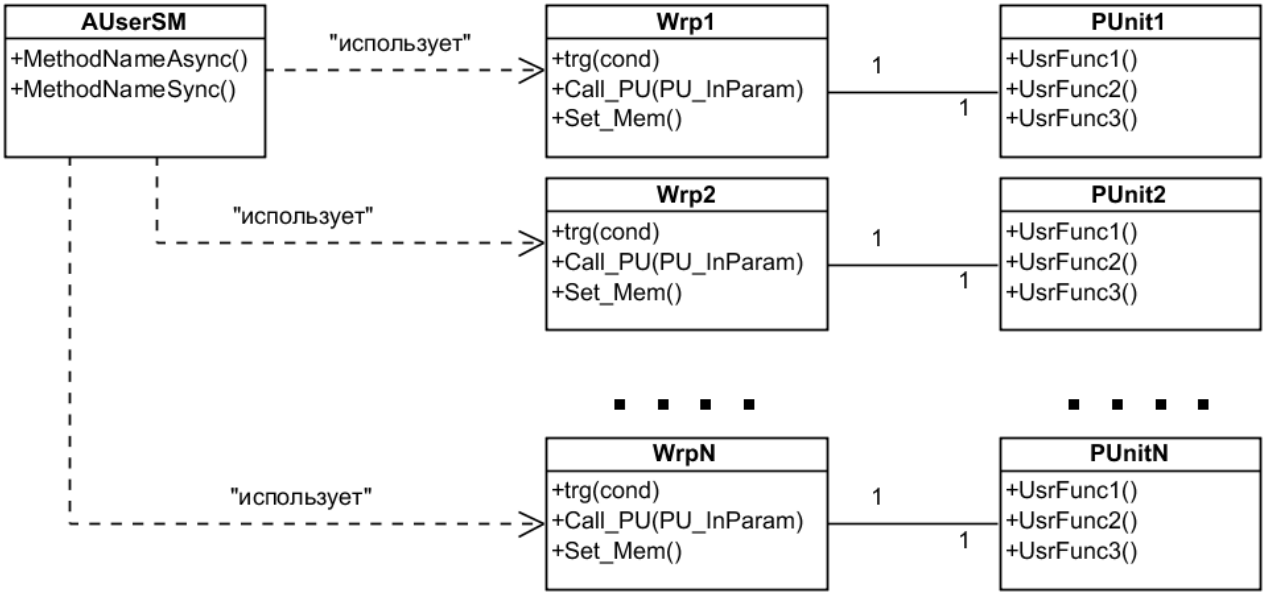


Рисунок 5 – Шаблон организации параллельного выполнения программы

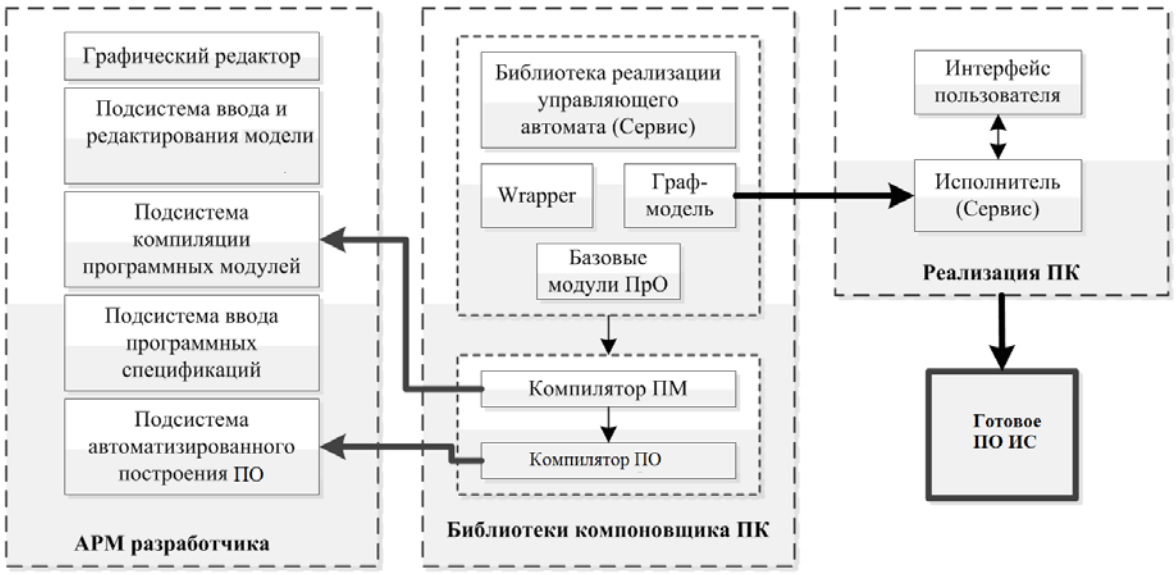


Рисунок 6 – Архитектура программного средства разработчика «SWDesigner»

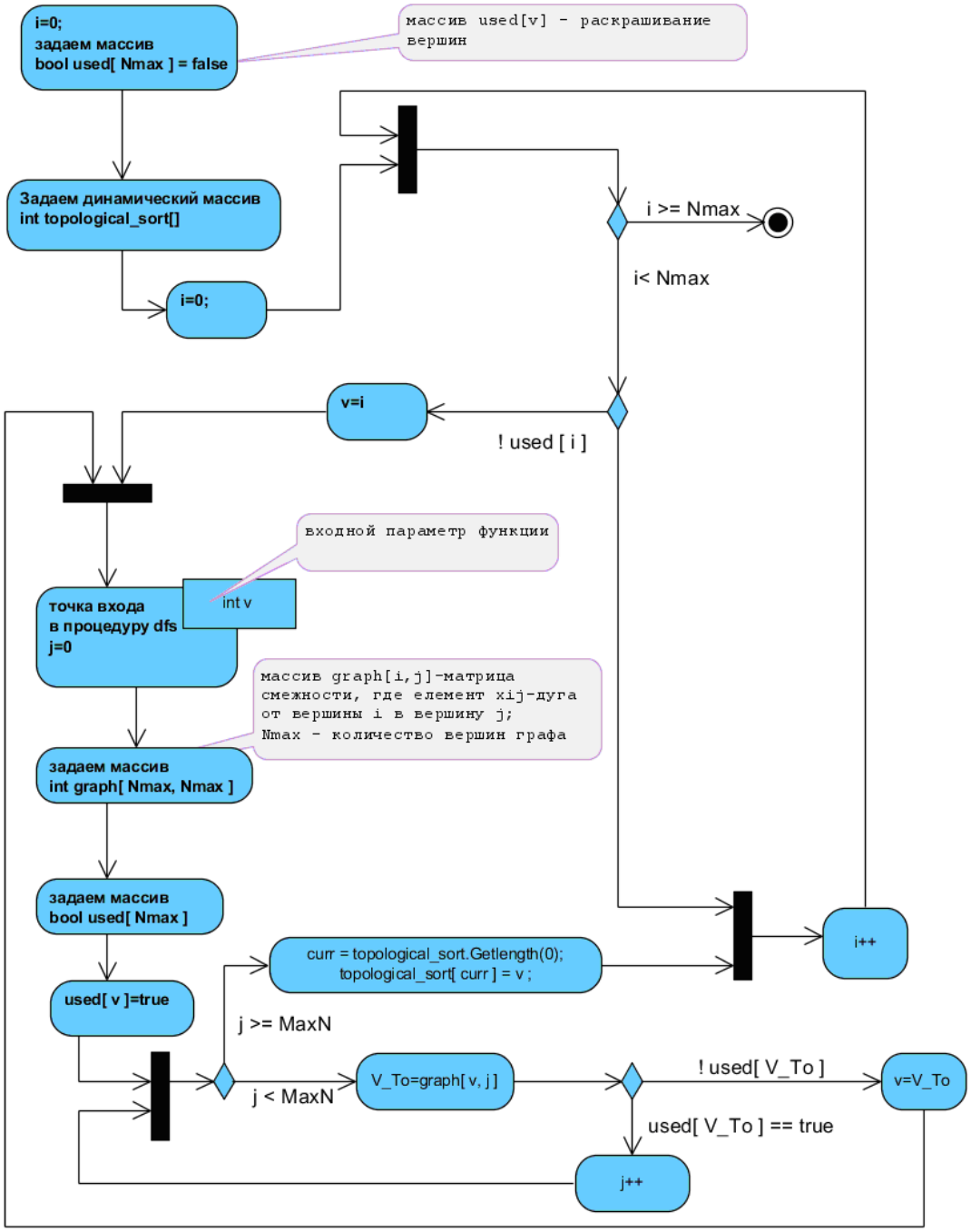


Рисунок 7 – Алгоритм топологической сортировки

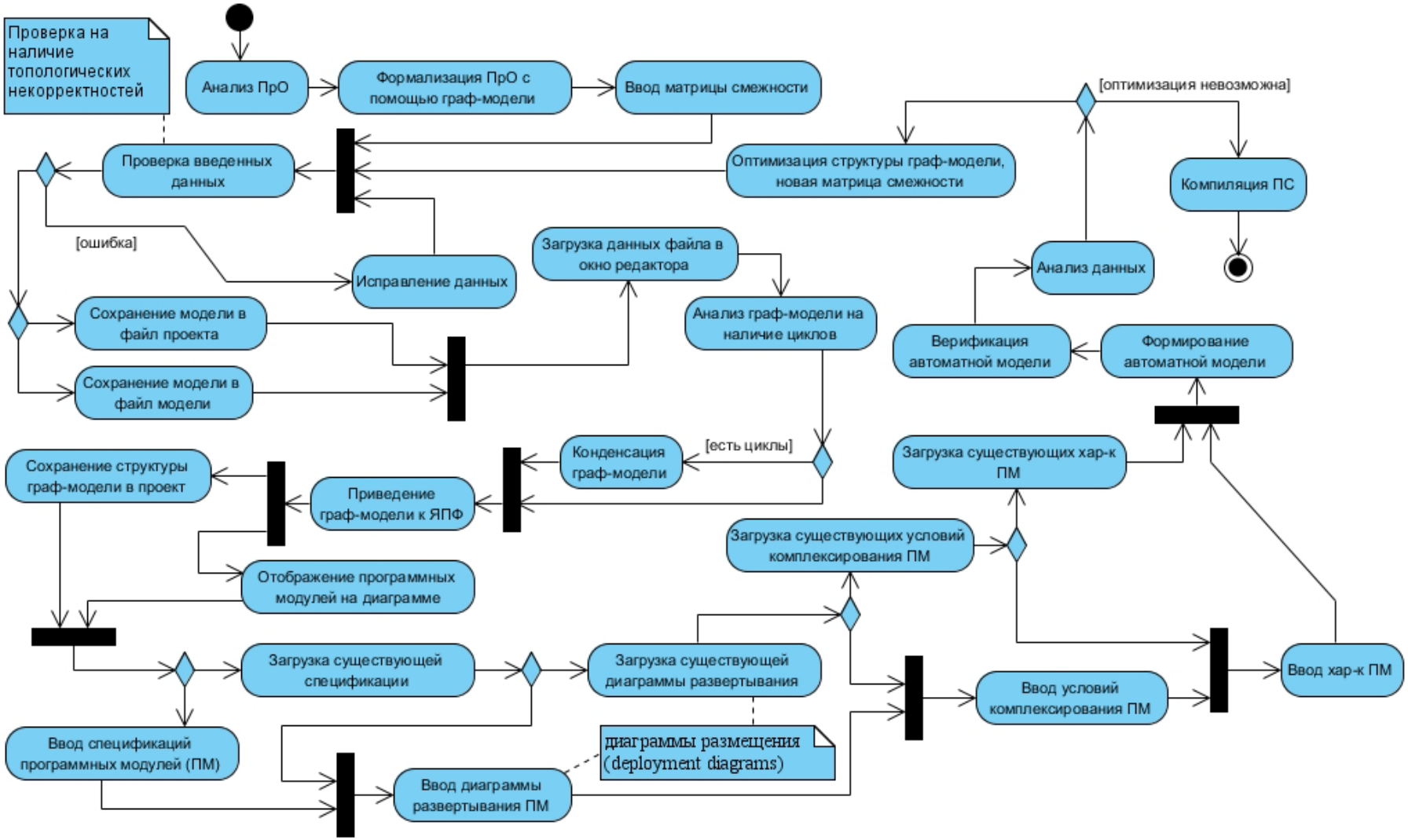


Рисунок 8 –Диаграмма активности. Обобщенный алгоритм работы разработчика с компоновщиком ПС

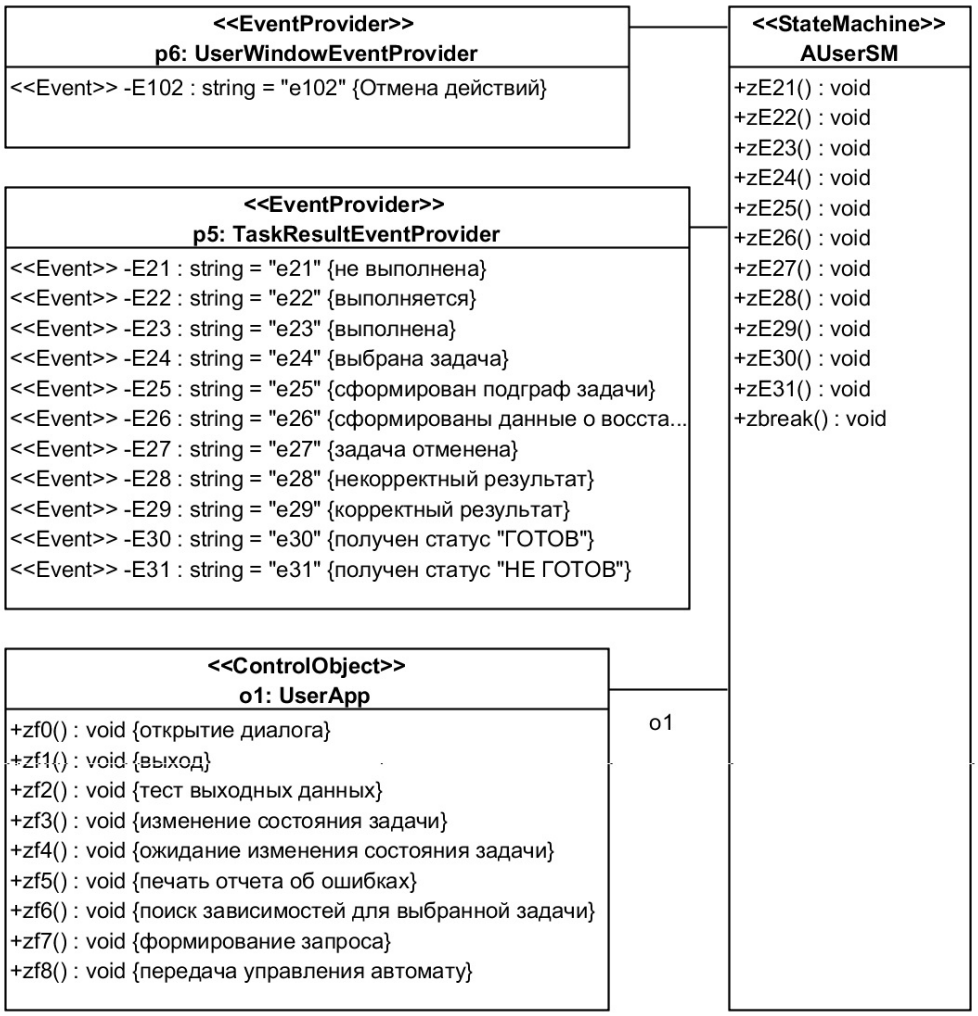


Рисунок 9 – Обобщенная структура исполнителя в виде управляющего КА

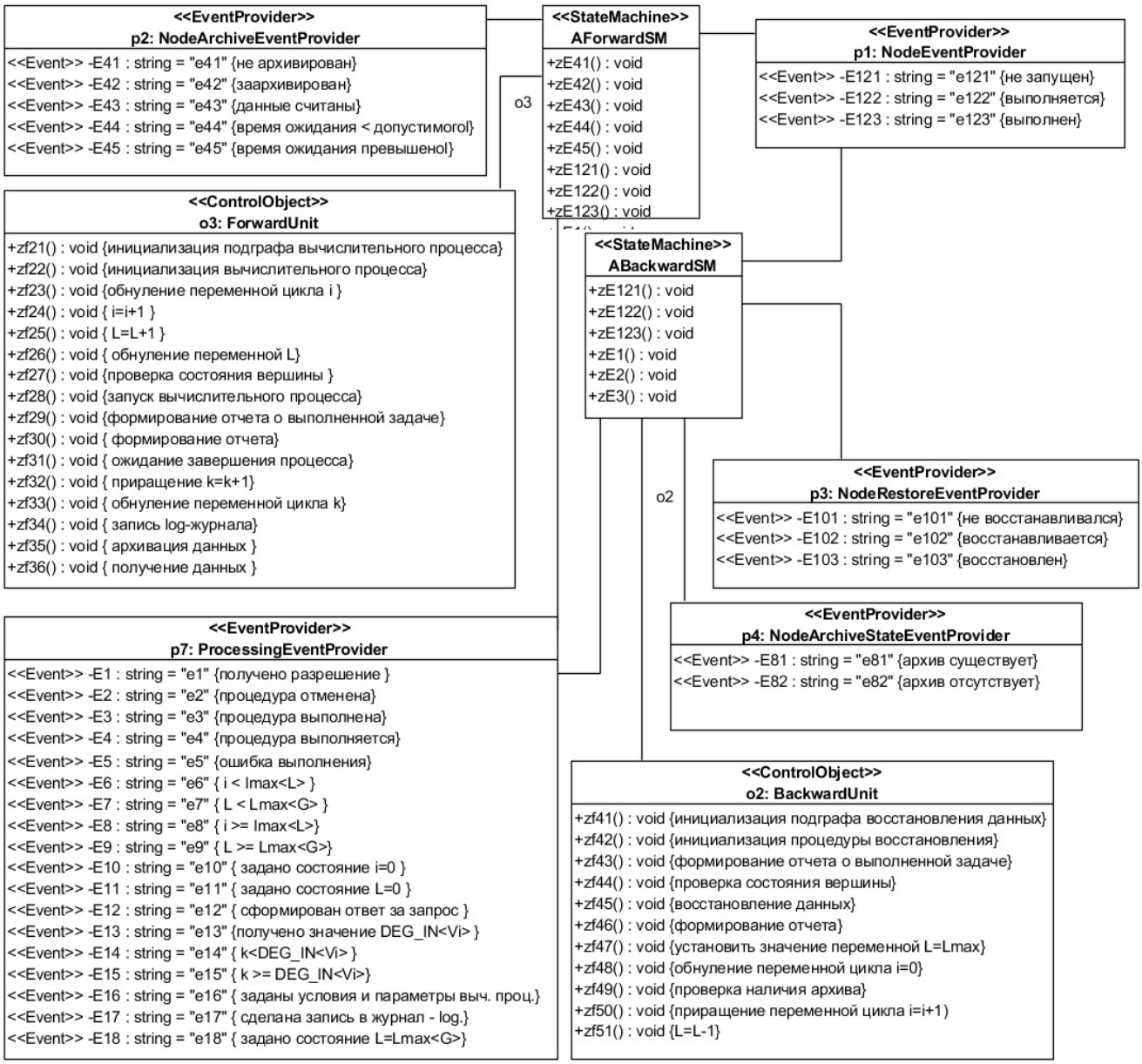


Рисунок 10 – Вложенные конечные автоматы

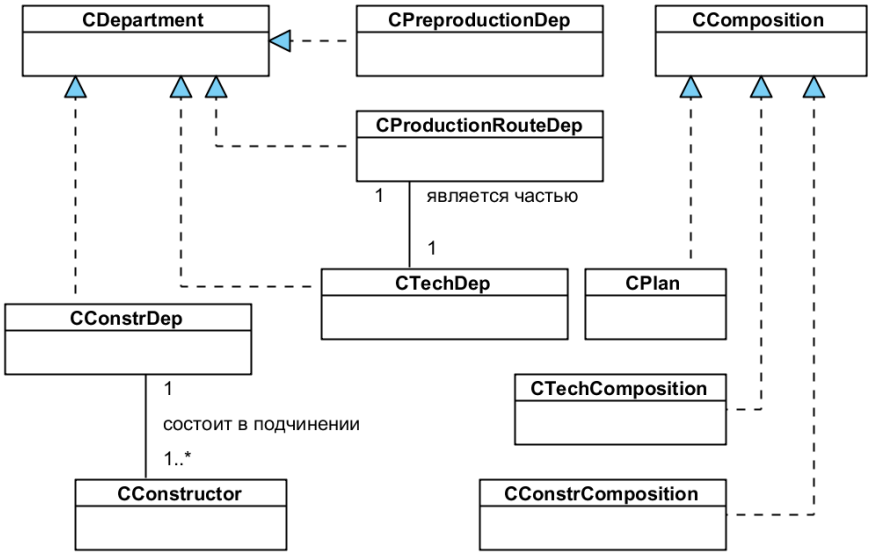


Рисунок 11 – Базовые классы графовой модели программной архитектуры и их наследование

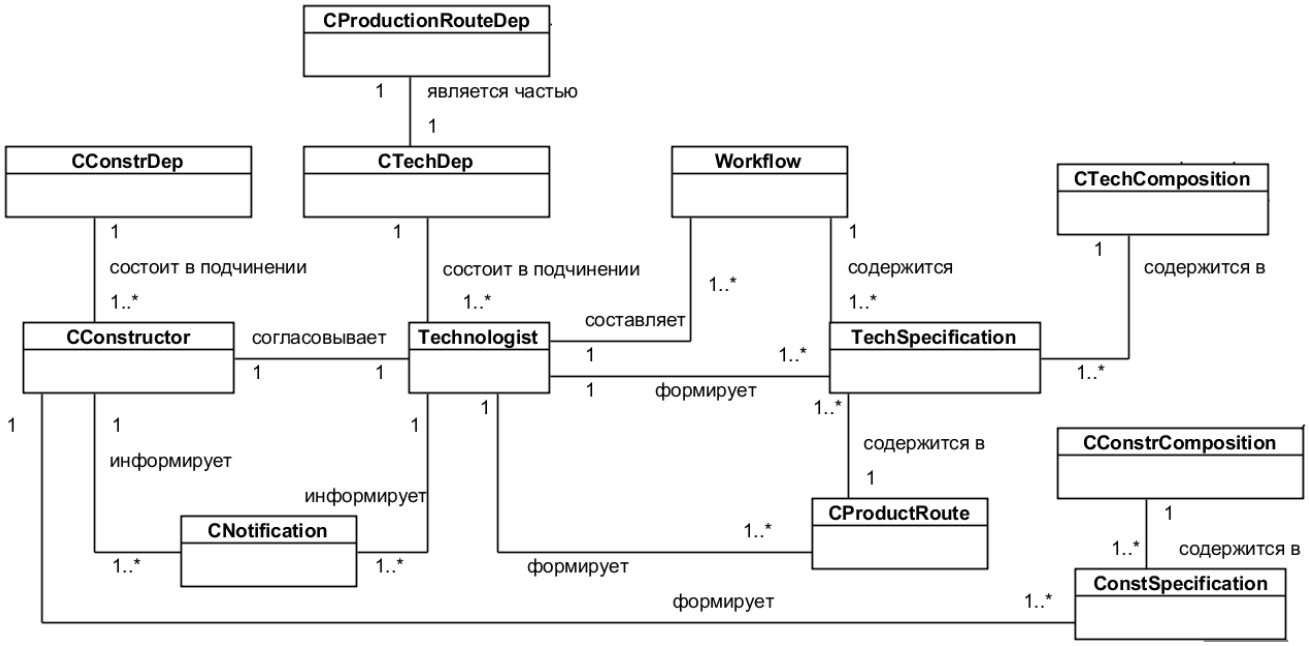


Рисунок 12 – Взаимоотношение конструкторских и технологических спецификаций

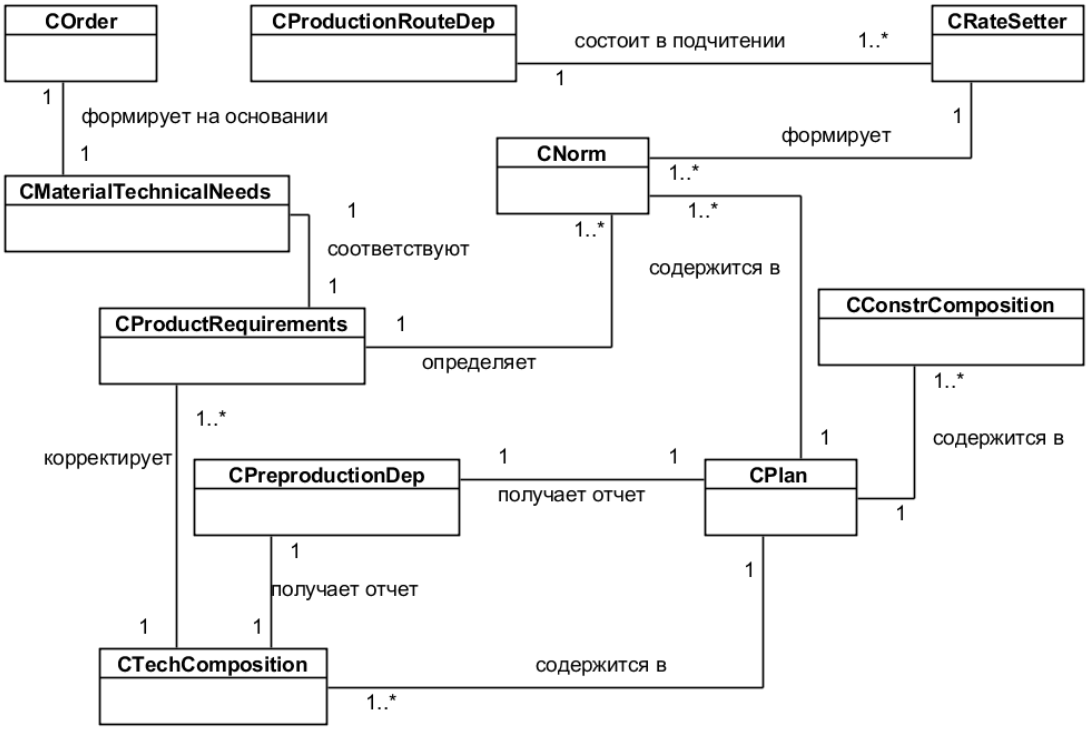


Рисунок 13 – Формирование плана производства на основании материально-технической потребности

Приложение Д

Код подпрограмм на языке PROMELA. Спецификация программного модуля

Код процедуры A1

```
proctype A1()  {
A1States=0;
A1Events=0;
        do
        ::(A1States==0) ->
                do
                        ::A1States=1; A1Events=24;
                        ::A1States=10; A1Events=102;
                        ::break
                od;
        ::(A1States==1) ->
                A1Events=25;
                A1States=2;
        ::(A1States==2) ->
                do
                        ::A1Events=22;A1States=2;
                        ::A1Events=21;A1States=3;
                        ::A1Events=23;A1States=4;
                        ::break
od;
… и т.д. …
```

Тело основного цикла автомата AForwardSM

```
do
        ::(A1States==0) ->
                do
                        ::A1States=1; A1Events=1;
                        ::A1States=16; A1Events=2;
::A1States=16; A1Events=3;
::A1States=16; A1Events=4;
::A1States=16; A1Events=12;
                        ::break
                od;
        ::(A1States==1) ->
                A1Events=11;
                A1States=2;
        ::(A1States==2) ->
```

```
            A1Events=10;
            A1States=3;
        ::(A1States==3) ->
            do
                ::A1Events=122;A1States=4;
                ::A1Events=122;A1States=5;
                ::A1Events=121;A1States=7;
                ::break;
            od;
… и т.д. …
```

Тело основного цикла автомата ABackwardSM»

```
do
        ::(A1States==0) ->
            do
                ::A1States=1; A1Events=1;
                ::A1States=10; A1Events=2;
                ::A1States=10; A1Events=3;
                ::A1States=10; A1Events=4;
                ::A1States=10; A1Events=12;
                ::break
            od;
        ::(A1States==1) ->
            A1Events=18;
            A1States=2;
        ::(A1States==2) ->
            A1Events=10;
            A1States=3;
        ::(A1States==3) ->
            do
                ::A1Events=82;A1States=6;
                ::A1Events=81;A1States=4;
                ::break
            od;
                                    … и т.д. …
```

Спецификация программного модуля

```
<?xml version="1.0"?>
<!--SpecificationFile-->
<Specification>
  <Function1 Name="func1">
```

```
    <Modificator Value="public static" />
    <Return Value="string[]" />
    <Parameters P1="ref int" P2="double" />
   </Function1>
   <Function2 Name="func2">
    <Modificator Value="public static" />
    <Return Value="void" />
    <Parameters P1="" P3="string" />
   </Function2>
</Specification>
```

Приложение Е

Этапы метода оценки функциональной сложности программного обеспечения информационной системы

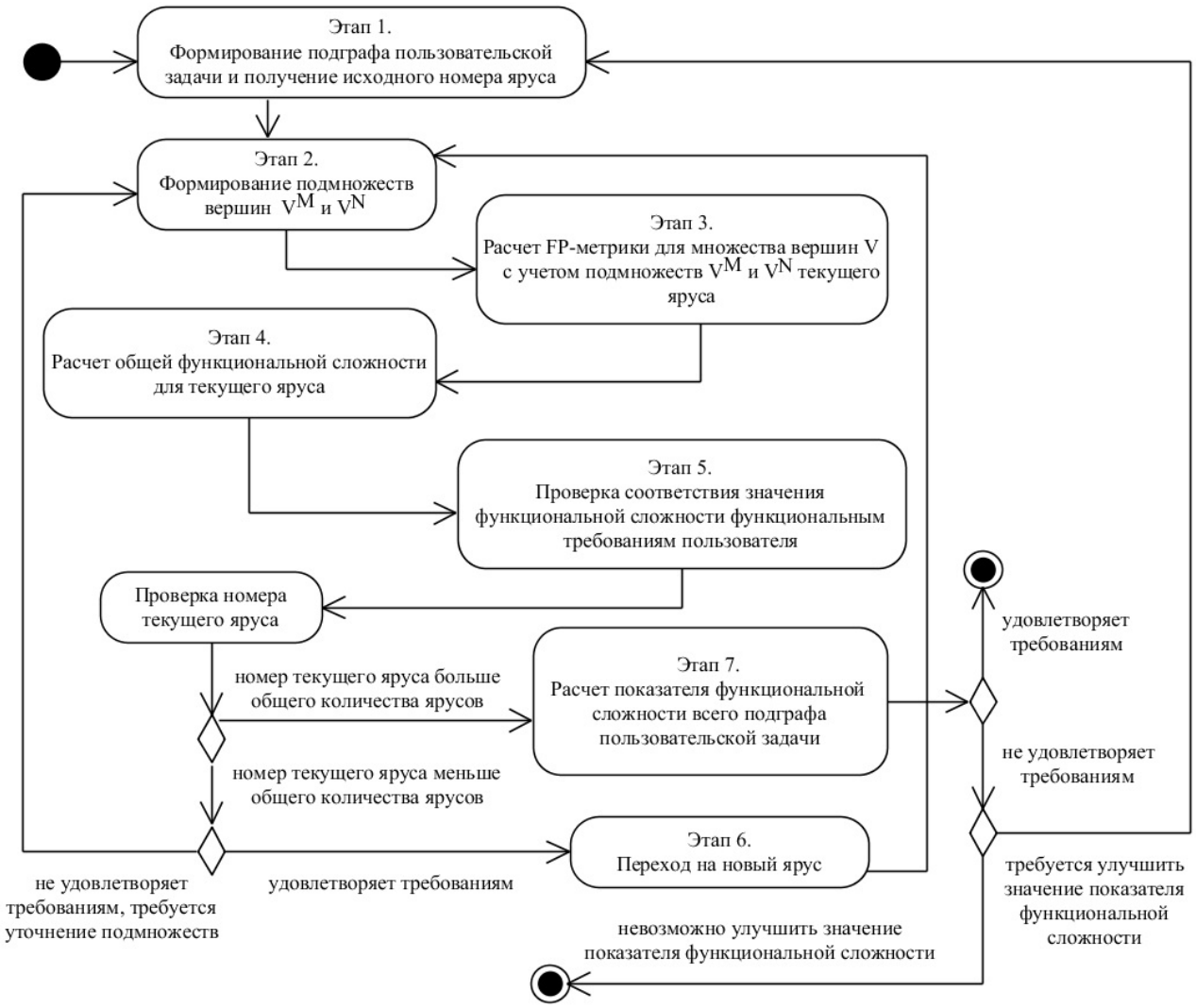


Рисунок 1 – Этапы метода оценки функциональной сложности ПО ИС

Приложение Ж

Акты внедрения результатов диссертационной работы

УКРАЇНА

**ДЕРЖАВНЕ ПІДПРИЄМСТВО
ХАРКІВСЬКИЙ НАУКОВО-
ДОСЛІДНИЙ ІНСТИТУТ ТЕХНОЛОГІЇ
МАШИНОБУДУВАННЯ**

Україна, 61016, м. Харків-16,
вул. Кривоконівська, 30
Р/р 260023001773 в ХФ АБ «Таврика»,
МФО 351953, ЄДРПОУ 14311070
тел./факс: (057) 372-40-50
E-mail: tehmash@ukr.net

УКРАИНА

**ГОСУДАРСТВЕННОЕ ПРЕДПРИЯТИЕ
ХАРЬКОВСКИЙ НАУЧНО-
ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ
ТЕХНОЛОГИИ МАШИНОСТРОЕНИЯ**

Украина, 61016, г. Харьков-16,
ул. Кривоконевская, 30
Р/с 260023001773 в ХФ АБ «Таврика»,
МФО 351953, ОКПО 14311070
тел./факс: (057) 372-40-50
E-mail: tehmash@ukr.net

Исх. № 11 от "9" января 2014 года

«УТВЕРЖДАЮ»:
Директор Государственного предприятия «Харьковский научно-исследовательский институт технологии машиностроения» к.т.н., доцент
_____ ***В.В. Косенко***

АКТ
о практическом использовании результатов диссертационной работы соискателя кафедры СТ ХНУРЭ Солодовникова Андрея Сергеевича

Комиссия в составе: председатель комиссии – Мовшович Александр Яковлевич – ученый секретарь ГП «ХНИИТМ», Лауреат Государственной премии Украины, д.т.н., профессор;

члены комиссии: Кобзев Александр Сергеевич – начальник научно-технического отдела ГП «ХНИИТМ», к.т.н., старший научный сотрудник; Свиридов Юрий Митрофанович – начальник отдела ГП «ХНИИТМ», составила данный акт о том, что результаты диссертационной работы «Модель, методы и информационная технология структурного синтеза программной архитектуры информационной системы на основе графовой модели», представленной на соискание ученой степени кандидата технических наук по специальности 05.13.06 – «Информационные технологии», использованы в проекте «Проектирование производственного процесса сложных изделий электронной техники» в виде программного обеспечения, реализующего следующие функции:

1) Формирование и выполнение запросов на отображение служебной информации об информационных зависимостях между вычислительными процессами различных подразделений объекта автоматизации и целевой процедурой определенного пользователя.
2) Контроль выполнения задач в соответствии с сетевым графиком.
3) Отображение истории вычислений и возврат состояния программной системы к заданному моменту времени при обнаружении недостатков и ошибок в результате работы отдельных предшествующих процедур или подразделений в целом.
4) Организация параллелизма на уровне задач, в случае наличия такой возможности.

Технический эффект от использования полученных результатов определяется увеличением скорости выполнения вычислительных процессов, снижением процента участия конечного пользователя в работе программного обеспечения и его конфигурировании.

Данный акт не может служить основанием для финансовых претензий к организации подписи.

Председатель комиссии: _____ А.Я. Мовшович

Члени комиссии: _____ А.С. Кобзев

_____ Ю.М. Свиридов

УТВЕРЖДАЮ
Зам. директора ИФВЭЯФ ННЦ ХФТИ
доктор техн. наук, профессор
_____М.А. Хажмурадов
"18" квітня 2016 г.

АКТ

о внедрении научных результатов диссертационной работы
Солодовникова Андрея Сергеевича

Комиссия в составе зам. нач. отдела канд. техн. наук Мартынова С.А. и членов комиссии канд. техн. наук с.н.с. Прохорец С.И. и начальника группы Круголя М.С. подтверждает, что результаты диссертационной работы Солодовникова А.С. внедрены в Институте физике высоких энергий и ядерной физики Национального научного центра «Харьковский физико-технический институт» (ИФВЭЯФ ННЦ ХФТИ) при выполнении темы «Исследования физических процессов и оптимизация параметров исследовательских установок методами математического моделирования» на 2011 – 2015 год (государственный регистрационный номер 0111U009294) в виде программного обеспечения для моделирования ядерно-физических процессов в средах сложного химического состава и сложной геометрической формы, а также оптимизации параметров физических и технологических установок.

Председатель комиссии : Мартынов С.А.

Члены комиссии: Прохорец С.И.

Круголь М.С.