Санкт-Петербургский государственный политехнический университет

Факультет технической кибернетики

Кафедра компьютерных систем и программных технологий

ДИССЕРТАЦИЯ

на соискание учёной степени

МАГИСТРА

Тема: Разработка методов и средств автоматизации реинжиниринга устройств, заданных HDL-спецификациями

Студент гр. 6081/12 О.В. Ненашев

Санкт-Петербург

2011

Санкт-Петербургский государственный политехнический университет
Факультет технической кибернетики
Кафедра компьютерных систем и программных технологий

Диссертация допущена к защите
Зав. кафедрой
_____В.Ф. Мелехин
«____»_____ 2011 г.

ДИССЕРТАЦИЯ
на соискание ученой степени
МАГИСТРА

Тема: ***Разработка методов и средств автоматизации реинжиниринга устройств, заданных HDL-спецификациями***

Направление: 220200 - Автоматизация и управление

Магистерская программа: 220200.68.01 - Управление в технических системах

Выполнил студент гр. 6081/12 _____ О.В. Ненашев

Руководитель,
ст. преп. _____ С.Л. Максименко

Консультанты:

по научной части
к.т.н., доцент _____ А.С. Филиппов

по нормоконтролю
ст. преподаватель _____ С.А. Нестеров

Санкт-Петербург

2011

Санкт-Петербургский государственный политехнический университет
Факультет технической кибернетики
Кафедра компьютерных систем и программных технологий

УТВЕРЖДАЮ

“__” _____ 2011 г.

Зав. кафедрой _____

ЗАДАНИЕ
по выпускной работе магистра
студенту Ненашеву Олегу Вячеславовичу

1. Тема работы *Разработка методов и средств автоматизации реинжиниринга устройств, заданных HDL-спецификациями*

2. Срок сдачи студентом законченной работы *1 июня 2011*

3. Исходные данные к проекту (работе)
Рассматриваются вопросы автоматизации структурных преобразований устройств; должен быть проведён обзор существующих средств реинжиниринга; строится методика представления устройства; разрабатывается архитектура средства; программная реализация прототипа ведётся на языке Java; используется библиотека vhdl_ast; рассматривается пример внесения структурной избыточности в устройства.

4. Содержание расчётно-пояснительной записки
Исследование возможностей автоматизации реинжиниринга архитектуры устройства, описанного на языке VHDL. Разработка языков представления и трансформации устройства. Разработка архитектуры расширяемого программного средства реинжиниринга. Разработка прототипа программного средства, демонстрирующего базовые возможности разработки. Исследование возможностей применения разработки.

5. Дата выдачи задания _____

Руководитель _____ (Максименко С.Л.)
(подпись руководителя) (Фамилия и инициалы)

Задание принял к исполнению “__” _____ 2011 г.

_____ (Ненашев О.В.)
(подпись исполнителя) (Фамилия и инициалы)

РЕФЕРАТ

Отчёт, 138 стр., 31 рис., 15 табл., 44 ист., 9 прил.

АВТОМАТИЗАЦИЯ РЕИНЖИНИРИНГА УСТРОЙСТВ, АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА, МЕТОДИКА ПРЕДСТАВЛЕНИЯ УСТРОЙСТВА, ПРЕОБРАЗОВАНИЕ СТРУКТУРЫ УСТРОЙСТВА, ПРОГРАММИРУЕМОЕ СРЕДСТВО, ПРОТОТИП, РАСШИРЯЕМЫЙ ИНСТРУМЕНТАРИЙ РЕИНЖИНИРИНГА

В работе рассматриваются вопросы автоматизации реинжиниринга цифровых устройств, заданных на специализированных языках описания устройства (HDL). Под реинжинирингом в работе понимается улучшение характеристик устройства повышение надёжности, производительности, снижение энергопотребления и так далее.

Рассмотрены общие подходы к реинжинирингу устройства, его место в жизненном цикле устройства, обоснована актуальность автоматизации реинжиниринга при помощи программных средств.

Проведено исследование, в ходе которого построена классификация средств поддержки реинжиниринга; проведён обзор существующих методики по представлению и трансформации устройства; рассмотрены существующие средства автоматизации реинжиниринга. По результатам обзора выявлены области, не покрытые существующими средствами, обоснована задача разработки программируемого средства реинжиниринга устройства.

В работе предложен новый подход к построению программируемого средства реинжиниринга устройства. Сформированы требования к подобному средству, разработаны методики представления и трансформации архитектуры устройства, предложена архитектура средства.

Разработан прототип программного средства, в котором были реализованы предлагаемая архитектура, методы представления и трансформации устройства. На нескольких практических задачах подтверждена применимость предложенных подходов. Проведён анализ разработанных методик и прототипа, после чего сформированы дальнейшие направления исследований.

ABSTRACT

Report, 138 pages, 31 figures, 15 tables, 44 references, 9 apps

ARCHITECTURAL TRANSFORMATION, EXTENSIBLE TOOLKIT ARCHITECTURE, HARDWARE REENGINEERING AUTOMATION, HARDWARE DESCRIPTION LANGUAGE, EXTENDABLE HARDWARE REPRESENTATION, MODEL OH DEVICE, PROGRAMMABLE REENGINEERING TOOLKIT, PROTOTYPE

Reengineering is used to improve device characteristics (performance, power effectiveness, reliability, etc) or to add new features. Modern systems are very complex and there's actual task of reengineering automation. This work approaches to build an extensible toolkit, based on which can be resolved custom tasks of reengineering. Survey of the existing tools shows, that there's no fully programmable tools of reengineering automation tools and this area remains very perspective.

This work presents new model of device for representation, analysis and transformation of digital systems that described on hardware description languages (HDL). This model is based on architecture graph with limited number of basic node types. Graph can be divided into tree-like structural description and links between elements. Hardware representation includes mechanisms for extension and navigation. A minimal command set for model transformation was developed.

In the work was designed architecture of hardware reengineering toolkit that meets basic requirements (programmable, extensible, embeddable). Toolkit is built in a modular fashion with minimal kernel that implements device model, basic operations and application programming interface (API). All others features (including HDL's input-output) should be implemented in extensions that kernel supports.

Prototype of programmable reengineering toolkit was developed as a part of research. Results of conducted experiments confirm applicability of proposed model and system architecture.

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

АСГ	-	Абстрактный Синтаксический Граф
АСД	-	Абстрактное Синтаксическое Дерево
ИС	-	Информационная Система
КСПТ	-	кафедра Компьютерных Систем и Программных Технологий
ПЛИС	-	Программируемая Логическая Интегральная Схема
ПО	-	Программное Обеспечение
САПР	-	Система Автоматизированного Проектирования
САР	-	Средство Автоматизации Реинжиниринга
СБИС	-	Сверхбольшая Интегральная Схема (в тексте - интегральная схема)
СППР	-	Средство Поддержки Принятия Решений
СПР	-	Средство Поддержки Реинжиниринга

AMS	-	Analog Mixed-Signal Смешанные аналого-цифровые сигналы
API	-	Application Programming Interface Интерфейс программирования приложения
ASG	-	Abstract Semantic Graph Абстрактный семантический граф
AST	-	Abstract Syntax Tree Абстрактное синтаксическое дерево
EDA	-	Electronic Design Automation Автоматизация проектирования электронных приборов
EDIF	-	Electronic Design Interchange Format Формат обмена проектами электронных приборов
GUI	-	Graphical User Interface Графический интерфейс пользователя
HDL	-	Hardware Description Language Язык описания устройства
IDE	-	Integrated Development Environment Интегрированная среда разработки
IEC	-	International Electrotechnical Commission Международная электротехническая комиссия
IP	-	Intellectual Property (IP-module) Сторонние функциональный модуль устройства
ISO	-	International Organization for Standardization Международная организация по стандартизации

RTL	-	Register Transfer Level Описание устройства на уровне регистровых передач
TCL	-	Tool Command Language Командный язык инструментов
UML	-	Unified Modeling Language Унифицированный язык моделирования
VHDL	-	VHSIC (Very high speed integrated circuits) HDL Язык описания высокоскоростных интегральных схем
XML	-	eXtensible Markup Language Расширяемый язык разметки

ВВЕДЕНИЕ

В течение жизненного цикла системы управления, как и любой другой сложной системы, требуется её доработка для обеспечения соответствия новым требованиям: по надёжности, точности, энергоэффективности или прочим показателям. Также часто для повторного использования предыдущих разработок требуется вносить в них некоторые изменения. Подобные задачи обычно называют задачами реинжиниринга системы. Сам термин определяется следующим образом:

Реинжиниринг - это систематическая трансформация существующей системы с целью улучшения ее характеристик качества, поддерживаемой ею функциональности, понижения стоимости ее сопровождения, вероятности возникновения значимых для заказчика рисков, уменьшения сроков работ по сопровождению системы [5].

В настоящее время большинство систем управления представляют собой программно-аппаратные комплексы. Реинжиниринг обеих составляющих производится параллельно, но используются различные методики преобразования. Поэтому, уместно рассматривать реинжиниринг аппаратной составляющей, далее называемой устройством, отдельно. Можно выделить следующие примеры задач, когда производится реинжиниринг аппаратной части системы:

- портирование устройства между аппаратными реализациями (например, перенос устройства с ПЛИС на заказную СБИС);
- добавление в устройство средств самодиагностики;
- повышение надёжности устройства;
- повышение производительности.

Проведение реинжиниринга требует больших временных затрат и высокой квалификации разработчика, так как требуется производить анализ исходной реализации и принимать решения по её модификации. В то же время, разработчику приходится решать множество рутинных задач: восстановление архитектуры, её трансформация и последующая реализация. Поэтому, актуальна задача частичной или полной автоматизации процесса реинжиниринга системы, что позволило бы снизить стоимость разработки в целом.

Задача автоматизации реинжиниринга устройства не нова, так как реинжиниринг устройства является одной из ключевых составляющих жизненного цикла системы. Тем не менее, до последнего времени стремились автоматизировать лишь частные задачи реинжиниринга, так как каждая конкретная задача требует отдельного подхода.

В то же время, было бы полезно иметь некоторый базовый набор инструментов, который можно было бы адаптировать для конкретных задач,

которые ставятся перед разработчиком. Наличие подобного инструментария позволило бы снизить затраты на автоматизацию реинжиниринга, так как оно избавило бы разработчика от реализации базовых модулей, позволив тратить время только на реализацию нужных ему алгоритмов трансформации.

Данная работа посвящена проблемам разработки программируемого средства реинжиниринга цифровых устройств, которые описываются при помощи специализированных языков описания аппаратуры (Hardware Description Language или HDL). Число публикаций по данным вопросам крайне мало, и все они носят прикладной характер. Поэтому, в работе сделан упор на фундаментальное исследование предметной области.

Работа состоит из пяти разделов. В разделе 1 проведено исследование предметной области. Рассмотрено понятие реинжиниринга, его в жизненном цикле системы, рассмотрены общие положения по средствам поддержки реинжиниринга. После этого, ставятся задачи для дальнейшего исследования.

Раздел 2 посвящён исследованию предметной области. В нём приведена построенная классификация средств поддержки реинжиниринга, проведён обзор существующих средств реинжиниринга и методик представления устройства, после чего обоснована актуальность построения автоматизированного средства реинжиниринга. В конце раздела сформулированы требования к подобному средству и выбраны ограничения при разработке.

В разделе 3 описывается предлагаемый подход по построению средства поддержки реинжиниринга, которое бы удовлетворяло сформированным требованиям. Предложена новая методика представления средства, сформирован список необходимых операций преобразования. В конце раздела предложена расширяемая архитектура программного средства реинжиниринга.

В рамках работы был разработан прототип средства реинжиниринга. Основные этапы разработки описаны в разделе 4. При построении прототипа были использованы предложения из третьего раздела. Кроме самого средства, реализованы дополнительные библиотеки, которые решают специфические задачи реинжиниринга.

В разделе 5 подведены итоги по проведённой работе. Проведена апробация разработанного средства, подтверждена применимость разработанного средства. Рассмотрены возможности применения средства для некоторых частных задач реинжиниринга устройства, выделены недостатки разработанного прототипа и предложены пути их устранения.

1. ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Перед началом иследования требуется разобраться, чем же на самом деле является реинжиниринг, каково его положение в жизненном цикле системы каковы его цели и задачи. Данные вопросы универсальны для любой области применения, поэтому первые пункты главы ориентированы на абстрактный объект преобразования. Также рассмотрены общие подходы к построению средств поддержки реинжиниринга.

Пункт 1.3 раздела посвящён проблемам реинжиниринга цифровых устройств, описанных на HDL. Рассмотрена постановка задачи, выявлены особенности данного процесса и его связи с разработкой устройства. В конце пункта рассмотрены некоторые типовые задачи реинжиниринга устройства.

По итогам введения сформирован список задач для дальнейшего исследования, которое описано в следующем разделе диссертации.

1.1. Введение в реинжиниринг

В технической области проблема реинжиниринга возникла с момента появления самих разработок. Любая система (техническая, информационная или экономическая) со временем устаревает: появляются более эффективные подходы, меняются требования к системе, появляются новые области применения.

В течение некоторого времени соответствие системы предъявляемым требованиям можно обеспечить за счёт последовательной её доработки. Данный процесс называется сопровождением системы.

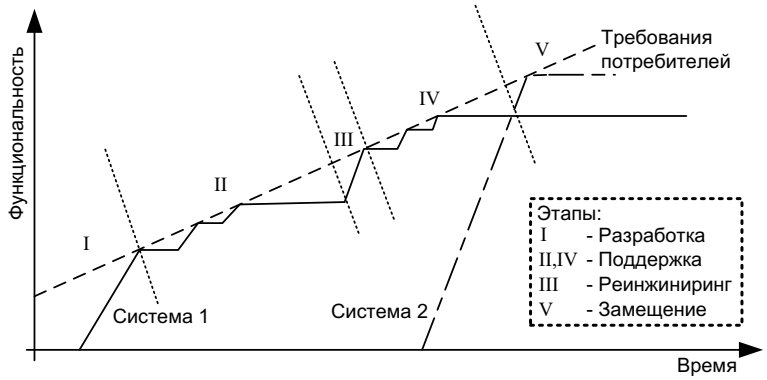

Рис. 1.1. Жизненный цикл системы [1]

Рано или поздно наступает момент, когда малые изменения системы при сопровождении становятся неэффективными, и требуется кардинальная доработка (или замена) некоторых её составляющих. Данная задача часто называется “модернизацией” системы или реинжинирингом. На рис. 1.1 за счёт реинжиниринга удаётся продлить период эксплуатации системы.

1.1.1. Понятие реинжиниринга

В ранних публикациях (до 1989 г.) реинжиниринг редко выделялся в отдельный этап, а рассматривался как повторная разработка после обратного инжиниринга (реверс-инжиниринга) архитектуры системы и корректировки её под новые требования. Кроме термина “reengineering”, в иностранных работах использовались синонимичные термины “renovation” и “reclamation”.

Обратным инжинирингом (реверс-инжинирингом) называется процесс извлечения информации о системе путём анализа её структуры, функций и поведения [13]. Реверс-инжиниринг не предполагает внесения изменений в исходную систему, а служит лишь для получения исходных данных.

Как писали Chikofsky и Cross в 1990-м году, “Обратный инжиниринг развивается как основное связующее звено в жизненном цикле программного обеспечения (ПО), но его росту препятствует путаница в терминологии” [13]. В данной статье была сделана первая попытка систематизировать понятие реинжиниринга и определить его роль в жизненном цикле системы. Авторами была предложена следующая схема процессов:

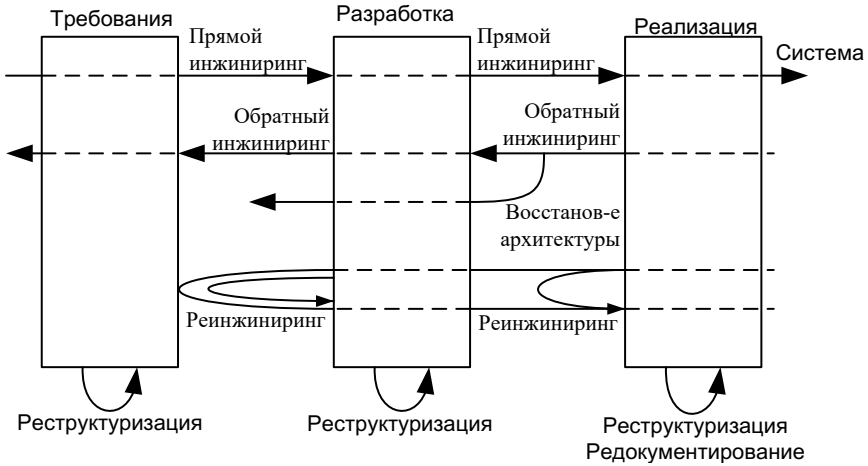

Рис. 1.2. Место реинжиниринга в жизненном цикле изделия [13]

В комментариях к схеме на рис. 1.2 авторами было предложено рассматривать реинжиниринг как отдельный процесс, после чего было предложено следующее определение:

Реинжиниринг - анализ и модификация целевой системы с целью представления ей в новой форме и последующей реализации.

В предложенном определении, на взгляд автора работы, неточно указаны цели проведения реинжиниринга. Цели реинжиниринга в современной трактовке были сформулированы М. Хаммером (1990г.) при описании методов, направленных на радикальное улучшение бизнес-процессов предприятия. Им было предложено следующее определение:

Реинжиниринг - это фундаментальное переосмысление и радикальное перепроектирование деловых процессов для достижения резких, скачкообразных улучшений главных современных показателей деятельности компании, таких, как стоимость, качество, сервис и темпы [16].

Несмотря на экономическую ориентацию, предложенные Хаммером подходы базируются на техническом анализе и системном подходе. В своё время это вызвало критику в экономических кругах, так как не учитывался человеческий фактор. Для задач, связанных с реинжинирингом технических систем данная проблема места не имеет.

В последующих работах реинжиниринг приобрёл значение самостоятельного процесса в рамках жизненного цикла продукта. Данный процесс рассмотрен в п. 1.1.3, здесь же лишь приведены основные определения:

Реинжиниринг - это систематическая трансформация существующей системы с целью улучшения ее характеристик качества: расширения поддерживаемой ею функциональности, понижения стоимости ее сопровождения, вероятности возникновения значимых для заказчика рисков, уменьшения сроков работ по сопровождению системы [11].

Реинжиниринг — это процесс создания новой функциональности или устранения ошибок в ранее реализованных технических решениях на действующем объекте [5].

Надо отметить, что в большинстве случаев реинжиниринг работает не с целевым объектом, а некоторым его описанием (например, исходные коды), которое отражает востребованные средством характеристики преобразуемого объекта (функциональность, структуру, алгоритмы поведения). По аналогии с базами данных описание решено называть представлением.

Таким образом, представление объекта – это формализованное описание, которое содержит совокупность информации об объекте, необходимую для процесса обработки.

Отличия реинжиниринга от поддержки системы

Принципиальным вопросом является разделение реинжиниринга и поддержки системы. В [29] Баринов В.А. выделяет ряд различий между реинжинирингом и поддержкой системы (см. табл. 1.1). Основным отличием второго является эволюционный подход с малым охватом изменений.

Таблица 1.1

Различия эволюционной модернизации и реинжиниринга [29]

Параметр	Поддержка системы	Реинжиниринг
Подход	Эволюционный	Революционный
Исходные данные	Существующий процесс	«Чистая доска»*
Частота изменений	Непрерывно/единовременно	Единовременно
Длительность изменений	Малая	Большая
Направление изменений	Снизу вверх	Сверху вниз
Охват	Узкий — на уровне функций (функциональный подход)	Широкий — межфункциональный

В предлагаемом сравнении Баринов В.А. указывает, что реинжиниринг ведётся с “чистого листа”. Очевидно, что при реинжиниринге за основу берётся ранее существовавшая система. Так что исходные данные включают её архитектуру, реализацию и известные характеристики. По мнению автора диссертации, в этом плане реинжиниринг полностью соответствует поддержке системы.

1.1.2. Задачи реинжиниринга системы

Как было сказано в предыдущем пункте, основной задачей реинжиниринга является достижение требуемых характеристик преобразуемой системы. На практике, могут существовать и другие задачи, связанные с преобразованием не архитектуры, а её представления. На рис. 1.2 данное преобразование обозначено малой дугой, не заходящей в фазу разработки. Итак, можно выделить следующие классы задач реинжиниринга:

- улучшение характеристик системы путём трансформации архитектуры;
- рефакторинг существующего представления;
- перенос системы из одного представления в другое.

Трансформация архитектуры системы

Трансформация системы с целью улучшения её характеристик является основной группой задач реинжиниринга, вынесенной в её определение. Про неё имеет смысл говорить только в том случае, когда известна сама система. Например, в пункте 1.4.2 рассмотрены задачи реинжиниринга применительно к цифровому устройству, описанному на HDL.

Рефакторинг представления

По определению Мартина Фаулера, рефакторинг – это процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы [44].

Отношение рефакторинга к реинжинирингу является достаточно спорным. Некоторые авторы с этим соглашаются (например, в [10] или [44]), другие же относят рефакторинг в отдельный процесс [11]. В пользу последнего можно привести следующие аргументы:

- рефакторинг не меняет характеристики устройства;
- рефакторинг часто является непрерывным процессом по улучшению качества представления системы (в т.ч. и в процессе разработки);
- рефакторинг преобразует реализацию системы, а не её архитектуру.

Несмотря на перечисленные выше замечания, в соответствии с большинством источников, в данной работе рефакторинг трактуется как один из случаев реинжиниринга.

Перенос между представлениями

Перенос представлений является промежуточной задачей между рефакторингом и улучшением характеристик. С одной стороны, сохраняется функциональность системы, но полностью меняется внешнее представление. Например, можно конвертировать спецификацию на UML (Unified Modeling Language) в исходные коды на некотором из языков программирования.

Несомненно, данная задача гораздо глубже рефакторинга, и должна рассматриваться отдельно.

1.1.3. Процесс реинжиниринга

Некоторые авторы указывают, что процесс реинжиниринга схож с процессом разработки. По их мнению, основное различие состоит в том, что при реинжиниринге исходными данными являются не только требования к системе, но и некоторая исходная система, не соответствующая данным требованиям в полной мере [10].

В [1] рассматриваются следующие основные фазы реинжиниринга:

- оценка показателей проекта по реинжинирингу, в том числе характеристик унаследованной информационной системы (фаза оценки);
- принятие решения о необходимости проведения работ по реинжинирингу или сопровождению/разработке ИС;
- осуществление реинжиниринга (выполнения работ по реинжинирингу);
- внедрение системы, трансформированной в результате проведения реинжиниринга.

Фаза осуществления реинжиниринга базируется на так называемой модели «подковы» [12]. В основу данной модели положены следующие виды деятельности:

- анализ существующей системы, основанный на одном или более ее логических описаниях;
- трансформация этих логических описаний в новое, улучшенное логическое описание системы.
- разработка новой системы, основанной на новых логических описаниях системы.

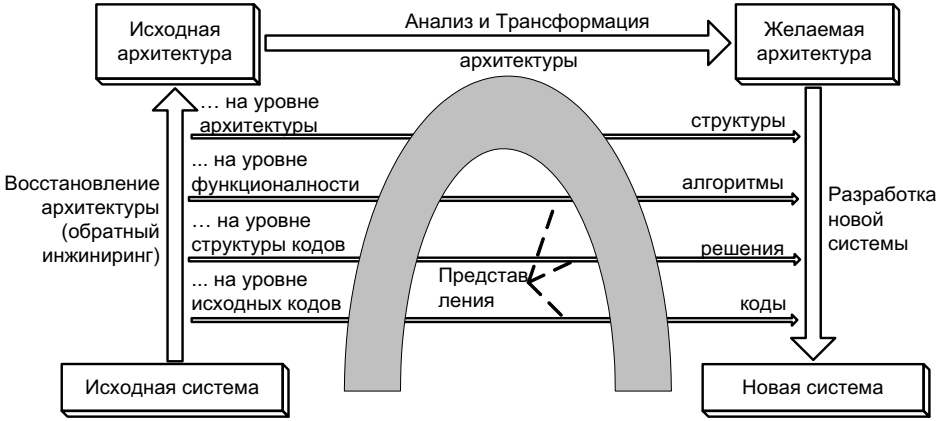

Рис. 1.3. Схема реинжиниринга системы в соответствии с моделью "подковы" [1]

На первом этапе восстанавливается архитектура существующей системы посредством обратного инжиниринга.

На втором этапе полученная архитектура анализируется на предмет соответствия её характеристик поставленным требованиям, после чего принимается решение о трансформации архитектуры. Этапы анализа и трансформации продолжаются до тех пор, пока построенная архитектура не будет соответствовать поставленным требованиям.

Третий этап включает деятельность по разработке системы, соответствующей новой архитектуре. Решаются вопросы декомпозиции элементов системы по пакетам, осуществляется выбор стратегий взаимодействия между компонентами системы компонентов системы [1]. На данном этапе производится добавление в новую систему неизменённых частей исходной системы.

Существуют и другие методики реинжиниринга, но в большинстве своём они близки к модели “подкова”.

1.1.4. Положение реинжиниринга в жизненном цикле системы

Процесс реинжиниринга тесно вплетён в жизненный цикл изделия. В статье [1] перечислены следующие задачи, с которыми связывается понятие реинжиниринга:

- прямой инжиниринг (Forward engineering);
- редокументирование (Redocumentation);
- рефакторинг (Refactoring);
- реструктуризация (Restructuring);
- обратный инжиниринг (Reverse engineering);
- сопровождение программных продуктов (Software maintenance);
- трансляция исходного кода (Source Code Translation).

В тексте работы приведены определения наиболее важных из перечисленных процессов. Определения прочих можно посмотреть в соответствующих источниках (например, в [33]).

Стандарт ISO/IEC 12207:1995 (см. [25]) определяет множество различных процессов жизненного цикла системы. Реинжиниринг в него как отдельный процесс не вошёл, но он может рассматриваться как основной процесс. На рис. 1.4 приведена иерархия процессов жизненного цикла по стандарту ISO, где штриховкой обозначены процессы, связанные с реинжинирингом.

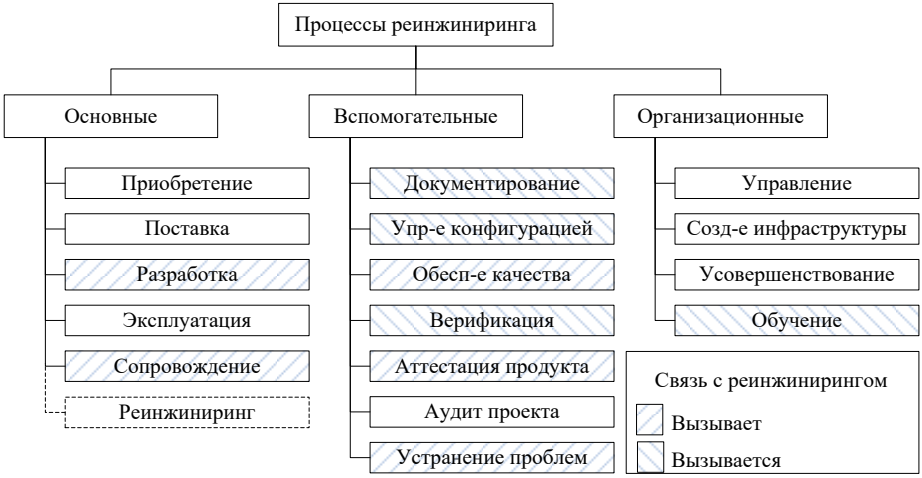

Рис. 1.4. Связи реинжиниринга с процессами жизненного цикла системы по стандарту ISO

Реинжиниринг системы требуется, когда исчерпаны возможности по поддержке системы путём наращивания функциональности. Во многих случаях

его можно начать заранее, чтобы подготовить новую систему до появления серьёзного отставания старой от требований пользователей.

Может получиться и так, что на каком-то этапе разработки система перестанет соответствовать предъявляемым требованиям. Это может быть обнаружено во время контроля качества или верификации системы. Кроме того, недостатки могут быть обнаружены пользователями уже во время эксплуатации.

Задача реинжиниринга может возникать и в процессе разработки нового продукта, например, при использовании ранее существовавшего модуля, для которого требуется произвести некоторую доработку. В таком случае процесс реинжиниринга запускается не для всей системы, а только для её отдельного компонента.

После проведения реинжиниринга требуется провести повторную верификацию и внести изменения в конфигурации разработанной системы. Также требуется отобразить изменения в документации (редокументирование). Если затронута пользовательская часть, то потребуется обучить пользователей новым возможностям.

1.2. Средства поддержки реинжиниринга

Средства поддержки реинжиниринга (СПР) – это средства разработки, функциональность которых позволяет решать некоторые задачи реинжиниринга системы: восстановления, анализа и трансформации архитектуры, реализации новой архитектуры.

СПР позволяют снизить затраты на проведение реинжиниринга. В большинстве случаев, СПР являются расширениями или дополнениями средств автоматизированного проектирования (САПР), используемыми в процессе разработки.

Типовая схема средства реинжиниринга была предложена Chikofsky и Cross в 1990-м году (см. рис. 1.5). В этой схеме средство реинжиниринга разбивалось на три модуля, два из которых обеспечивают считывание представления и генерацию выходных данных. Вся логика преобразования скрыта в блоке преобразований.

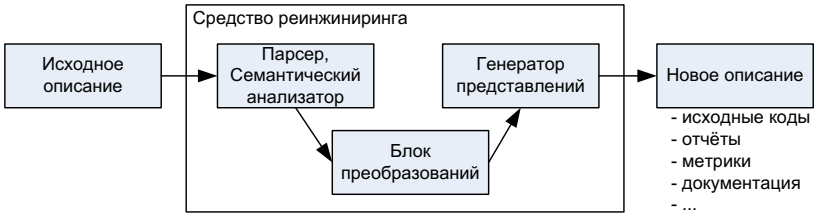

Рис. 1.5. Типовая схема средства реинжиниринга [13]

Предложенная схема может быть применена к любому функциональному блоку, на входе и выходе которого стоят преобразователи данных. К сожалению, широкий спектр задач реинжиниринга исключает возможность построения более детального описания.

Существуют задачи, которые конкретному разработчику приходится решать не один раз, а с некоторой периодичностью.

> Например, некоторая фирма занимается прототипированием печатных плат, используя для этого специализированный формат описания устройств. Заказчики же предоставляют описания в стандартных форматах, и задача их преобразования во внутренний формат возникает с большой частотой.

В предложенном случае актуально полностью автоматизировать процесс преобразования. Подобные средства в работе предлагается называть средствами автоматизации реинжиниринга (CAP).

1.3. Проблемы разработки устройств

В работе в качестве проблемной области заявлен реинжиниринг устройств. Термин “устройство” означает искусственный объект, имеющий сложную внутреннюю структуру и используемый для выполнения определённых функций. Данная трактовка очень широка, и под неё попадают практически все технические объекты: от отдельных микросхем до электромеханических устройств и океанских лайнеров. В пункте 1.3.1 сделана попытка ограничить значение данного термина.

В данном пункте также рассмотрен процесс разработки устройства и особенности форматов представления устройств. Описание ведётся применительно к реинжинирингу. Особенности реинжиниринга устройства будут рассмотрены в следующем пункте.

1.3.1. Подход к трактовке устройства в рамках задачи реинжиниринга

В работе под устройством понимается некоторая аппаратная или программно-аппаратная система, используемая для решения задач управления. Упор сделан на цифровые системы, но при необходимости предлагаемые подходы могут быть расширены и на аналоговые устройства. Ниже в тексте приведены обоснования выбранного подхода.

Предлагается трактовать устройство следующим образом:
- устройство решает задачи управления;
- рассматривается только аппаратная составляющая;
- рассматриваются цифровые устройства;
- устройство описывается на HDL.

Отдельно рассмотрим проблему реинжиниринга программно-аппаратных систем.

Проблема программно-аппаратных систем

Большинство современных устройств являются программно-аппаратными системами. При этом программная и аппаратная части имеют различные представления, что затрудняет их совместный реинжиниринг.

Обычно реинжиниринг аппаратуры и ПО разделяют на две параллельные задачи, связывая предварительным этапом, на котором формируются требования для обеих составляющих системы (см. рис. 1.6). Подобный подход применяется и в задачах разработки программно-аппаратных устройств [3].

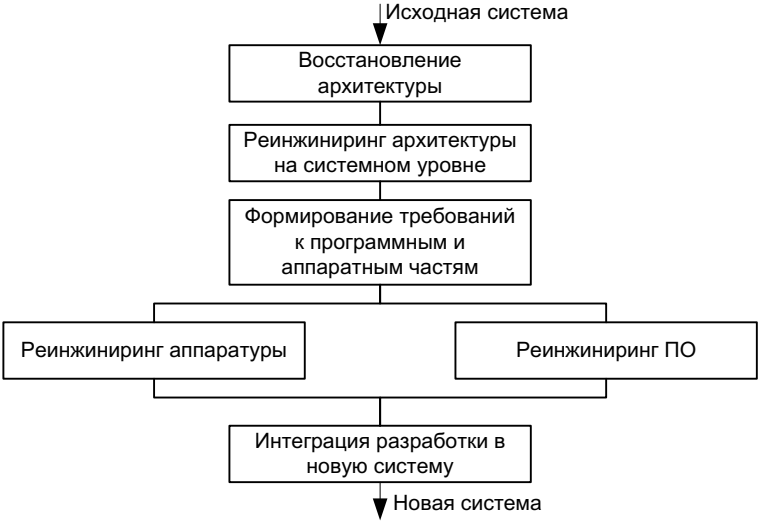

Рис. 1.6. Порядок проведения реинжиниринга программно-аппаратной системы

Реинжиниринг программного обеспечения является отдельной областью знаний, которая популярна в настоящее время и для которой предлагаются свои методики ведения процесса. Существует ряд фундаментальных работ по данному вопросу (например, [10]). Поскольку процессы реинжиниринга ПО и аппаратуры можно разделить, то в дальнейшем мы будем касаться только реинжиниринга аппаратной части системы.

В дальнейшем под термином “устройство” понимается выбранное в данном пункте подмножество.

1.3.2. Особенности описания устройства на HDL

В настоящее время существует множество способов описания устройств. При этом используются структурные и принципиальные схемы, блок-схемы, текстовые нотации, модели и пр. Для цифровых систем в настоящее время наиболее распространены описания при помощи специализированных языков программирования, называемых языками описания аппаратуры (Hardware Description Language или HDL).

Языки описания аппаратуры (HDL)

HDL не реализуют устройство, а лишь описывают требования к нему. Мы можем описать структуру устройства, требуемое нам поведение, функции преобразования сигналов и так далее. Само устройство генерируется из подобных спецификаций на этапе трансляции (синтеза).

Задача синтеза устройства из HDL решается отдельными средствами разработки. Во многом данный процесс схож с компиляцией программного обеспечения, но большинство авторов настаивают на принципиальном различии.

В настоящее время наибольшее распространение получили языки VHDL (HDL for very high speed integrated circuits) и Verilog. Они поддерживают как структурное, так и поведенческое описания. Подробнее о данных языках можно прочитать в [4].

Благодаря своей функциональности HDL удобны для спецификации устройства разработчиком. Однако, они плохо подходят для передачи данных между модулями СПР или машинной обработки. Это объясняется следующими причинами:

- существование нескольких редакций для каждого из языков;
- сложность и неоднозначность синтаксиса языков;
- возможность описания одного объекта многими способами.

В большинстве случаев для организации взаимодействия между модулями СПР используются упрощённые описания, называемые нетлистами (от англ. netlist – список соединений).

Нетлисты

Нетлисты можно рассматривать как множество примитивных, чаще всего структурных, HDL. Они описывают иерархию модулей и связи между ними, используя минимальный набор функциональных преобразований. На нижнем уровне иерархии находятся функциональные блоки целевой платформы: логические вентили и ячейки, встроенные аппаратные компоненты и т.д.

На рис. 1.7 приведена последовательность разработки устройства с указанием типов нетлистов, которые передаются между средствами, реализующими различные стадии процесса реинжиниринга.

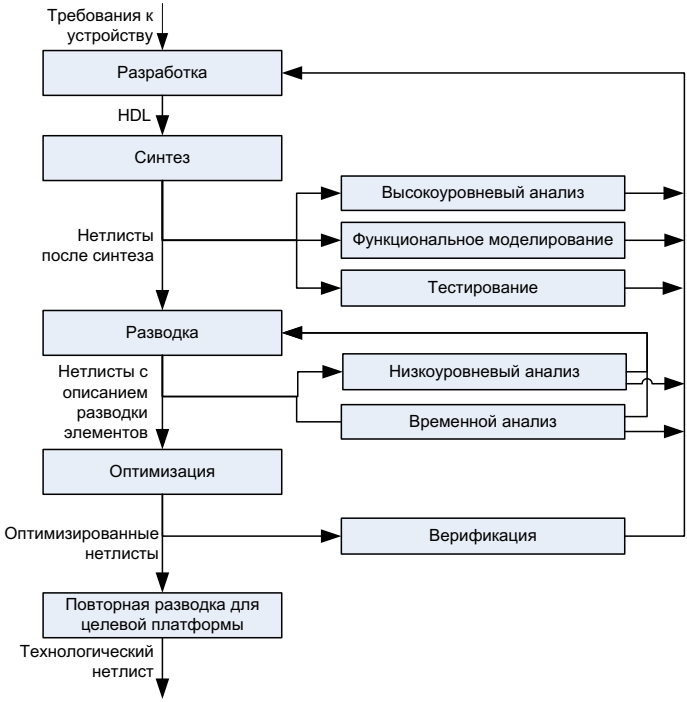

Рис. 1.7. Использование нетлистов для передачи данных между стадиями процесса разработки [26]

Средство реинжиниринга устройства могут интегрироваться в процесс разработки устройства. В этом случае, в качестве внешнего представления актуально использовать не только исходные описания, но и промежуточные нетлисты.

1.3.3. Процесс разработки устройства

Существует множество подходов к описанию процесса разработки устройства. Обычно они предполагают четыре стадии разработки: формирование требований, разработку, трансляцию и завершение. Данная модель соответствует классической каскадной модели (см. [21]), рекомендованной Институтом Управления Проектами (PMI), хотя именования фаз различаются.

На рис. 1.8 приведены основные этапы процесса разработки устройства и их составляющие. В тексте приведено краткое описание основных этапов.

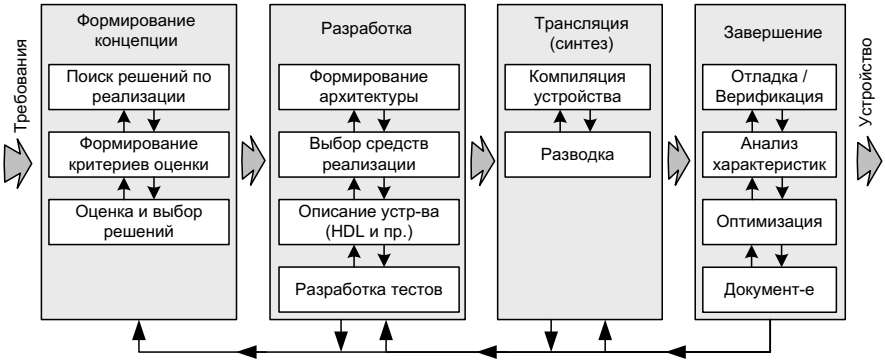

Рис. 1.8. Основные этапы разработки устройства [15]

Формирование концепции

Данный этап соответствует стадии НИОКР (Научные Исследования и Опытно-Конструкторская Разработка). Производятся предварительные исследования, которые нацелены на поиск методологических и архитектурных подходов, позволяющих выполнить поставленные требования. Формируются критерии, на основании которых выбираются решения для реализации в устройстве.

Разработка устройства

В классической модели жизненного цикла на данном этапе должны формироваться требования к реализации устройства. Однако описание на HDL и является детализированной спецификацией на устройство, поэтому данное описание устройства разрабатывается на этом этапе.

Трансляция устройства

На этапе трансляции производится преобразование представления устройства в описание его физической реализации. При этом производится оптимизация связей устройства, адаптация архитектуры под аппаратную платформу, физическая разводка соединений и так далее.

Для устройства этап трансляции крайне важен, так как данный этап практически полностью определяет характеристики будущего устройства. Если для ПО трансляция идёт автоматически и занимает минуты/часы, то трансляция сложного устройства с учётом оптимизации скоростных характеристик может занимать месяцы [17].

Завершение разработки

На данном этапе осуществляется отладка устройства, анализ его характеристик и их доводка под поставленные требования. В случае успешного завершения проекта осуществляются приёмо-сдаточные испытания, подготовка сопроводительной документации и прочие вещи, которые не представляют интереса в рамках реинжиниринга.

Процессы контроля качества

В соответствии со стандартной моделью проекта параллельно с процессом разработки идут и другие: управление требованиями и изменениями, разработка документации и т.д. [21].

С точки зрения реинжиниринга, наиболее важным процессом является процесс управления качеством, который может своевременно дать сигнал о том, что разработка отклоняется от требований, вследствие чего требуется реинжиниринг существующей архитектуры.

В исследовании [15] авторы выделяют два процесса контроля качества: анализ характеристик (Performance Analysis на рис. 1.9) и валидацию. Первый процесс занимается оценкой характеристик системы, второй же – для подтверждения соответствия устройства поставленным требованиям

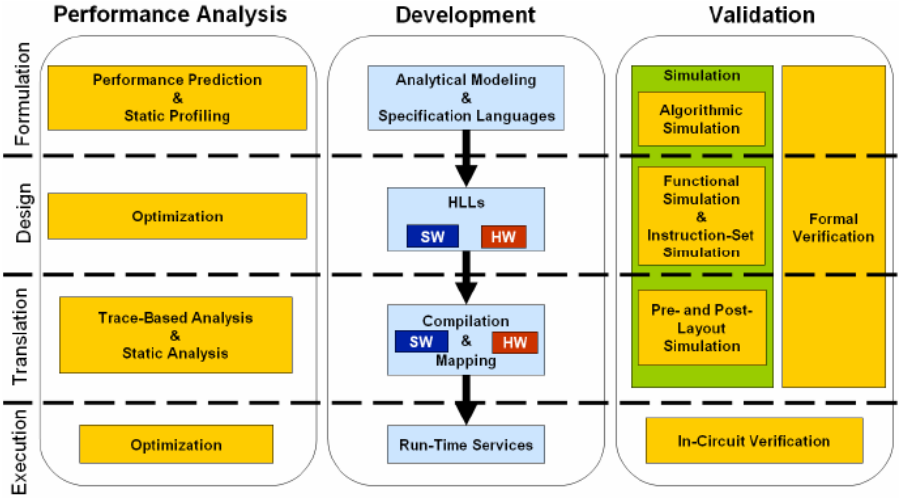

Рис. 1.9. Этапы разработки устройства и сопутствующие процессы контроля качества[15]

1.4. Реинжиниринг устройств

В предыдущих пунктах были описаны общие вопросы реинжиниринга. При этом рассматривалась некоторая абстрактная система. Тем не менее, в разных областях различаются задачи, методы и средства реинжиниринга, поэтому требуется рассмотреть аспекты реинжиниринга устройств, чему посвящён данный пункт работы.

1.4.1. Постановка задачи реинжиниринга устройства

В соответствии с определением, реинжиниринг предполагает улучшение характеристик устройства или его представления. Для устройства возможны все три класса задач, которые были рассмотрены в пункте 1.1.2:

- улучшение характеристик устройства;
- рефакторинг существующего представления устройства;
- перенос устройства из одного представления в другое.

Трансформация архитектуры устройства может потребоваться для изменения функциональности устройства или же улучшения характеристик устройства при её сохранении. Ниже приведены некоторые примеры характеристик, задача улучшения которых может ставиться при реинжиниринге:

- функциональность (поддержка новых возможностей и т.п.);
- производительность;
- надёжность (отказоустойчивость, время наработки на отказ и т.п.);
- аппаратные затраты (число ЛЭ, вентилей);
- энергопотребление.

В пункте 1.4.2 рассмотрены типовые задачи реинжиниринга для каждой из перечисленных групп. Сейчас же требуется разобраться, что может инициировать процесс реинжиниринга, и какие задачи требуется при этом решать.

В соответствии с [15] параллельно с разработкой решаются задачи контроля характеристик и валидации устройства (рис. 1.9). Если на каком-либо этапе данных процессов оказывается, что устройство не соответствует поставленным требованиям, то может потребоваться доработка. С другой стороны, в процессе сопровождения устройства к нему могут появиться новые требования. Во всех случаях, требуется провести реинжиниринг устройства, то есть выполнить следующие шаги:

- извлечь архитектуру устройства из описаний;
- провести анализ устройства и требований, по результатам которого сформировать задачи по доработке архитектуры устройства;
- трансформировать архитектуру (преобразовать HDL-описание);
- синтезировать устройство.

Поскольку устройство описывается на HDL, то восстановление архитектуры не требуется, так как исходные коды на данных языках и являются спецификацией архитектуры. Для анализа и синтеза можно использовать стандартные средства разработки, как и в процессе реинжиниринга. На рис. 1.10 приведён пример подобной организации

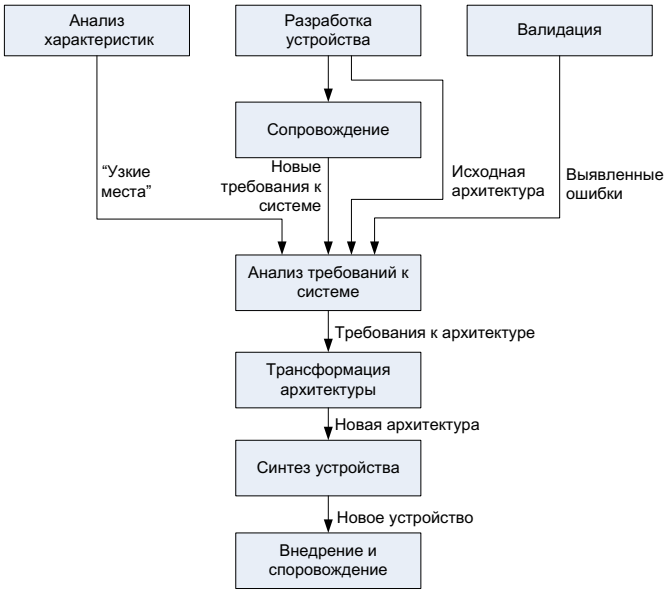

Рис. 1.10. Включение реинжиниринга в процесс разработки устройства

Таким образом, в минимальном варианте требуется решить только задачу трансформации архитектуры. Она может решаться как вручную, так и с помощью средств поддержки реинжиниринга, которые будут рассмотрены в следующем разделе.

1.4.2. Типовые задачи реинжиниринга устройства

В данном пункте перечислены некоторые типовые задачи реинжиниринга устройства. Они разбиты на категории в соответствии с общим подходом к реинжинирингу устройства (см. пункт 1.1.2).

Задачи изменения функциональности

Расширение функциональности:

- добавление новых модулей и функций;
- добавление средств диагностики (добавление отладочных интерфейсов, вывод тестовых сигналов, и т.д.);
- выделение модулей в отдельные устройства.

Задачи улучшения характеристик устройства

Повышение производительности:
- конвейеризация;
- введение суперскалярности;
- аппаратная акселерация программных алгоритмов:
- повышение предельной тактовой частоты устройства.

Повышение надёжности:
- введение структурной избыточности;
- введение временной избыточности;
- замена элементов их отказоустойчивыми аналогами;
- использование помехоустойчивых кодов в памяти и шинах сигналов;

Снижение энергопотребления:
- введение регуляторов частоты тактирования;
- введение доменов тактирования;
- добавление средств управления питанием.

Снижение аппаратных затрат:
- удаление избыточных модулей;
- оптимизация архитектуры устройства;
- удаление отладочных и диагностических сигналов.

Задачи рефакторинга

Как было сказано в пункте 1.1.2, рефакторинг – это процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы

Если адаптировать определение для устройства, то рефакторинг – это процесс изменения представления устройства, не влияющий на характеристики устройства, но упрощающий обработку его архитектуры. Таким образом, результаты синтеза до и после рефакторинга должны быть близки. Нельзя гарантировать полное соответствие, так как изменения описания могут повлиять на решения, принимаемые при автоматическом синтезе.

Для устройства достаточно сложно разделять задачи рефакторинга и модификации устройства. Рефакторинг не должен модифицировать функциональность, т.е. изменения в представлении устройства не должны повлиять на связи в нём. К рефакторингу можно отнести следующие задачи:
- переименование элементов описания (сигналов, модулей и пр.);
- реструктуризация кода;
- изменение иерархии модулей в описании;
- комментирование исходных кодов представления;
- инкапсуляция методов.

Задачи переноса представлений

Перенос описания устройства между представлениями может быть вызван самыми различными причинами: сменой аппаратной платформы, языка описания устройства или иной потребностью. Ниже рассмотрены примеры задач для каждой из групп.

Задачи смены аппаратной платформы:
- замена специализированных аппаратных модулей их эквивалентами (актуально при переходе с ПЛИС на заказные микросхемы);
- оптимизация устройства за счёт использования аппаратных средств (например, умножителей и регистров в ПЛИС);

Задачи смены HDL:
- конвертация между поведенческим и структурным описанием устройства;
- генерация HDL из других форм описания (например, генерация тестов из поведенческих описаний на RT-UML);
- объединение в устройство модулей, описанных на различных языках;
- конвертация нетлистов в синтезируемые HDL;

Прочие задачи:
- смена программных реализаций модулей (замена блоков с одинаковыми входными и выходными портами);
- реверс-инжиниринг IP-модулей (например, для последующего улучшения характеристик);
- реверс-инжиниринг исходных кодов, подвергнутых обфускации (приведение кодов к виду, затрудняющему анализ и модификацию).

1.4.3. Пример реинжиниринга устройства

Рассмотрим пример, когда требуется повысить отказоустойчивость некоторого модуля, например, для использования в условиях жесткого излучения в космосе. В таких условиях существует вероятность полного отказа отдельного модуля

Одним из путей решения проблемы является введение структурной избыточности, т.е. реализации нескольких одинаковых модулей со средствами выявления отказов в них и формирования корректного сигнала. На рис. 1.11 приведён пример введения избыточности в некоторый модуль. Для упрощения восприятия показаны не все внутренние сигналы и соединения.

В типовом подходе к введению аппаратной избыточности создаётся несколько копий исходного модуля с одинаковыми управляющими сигналами, после чего на каждый выходной сигнал ставится голосователь, по некоторому закону определяющий выходной сигнал модуля [14].

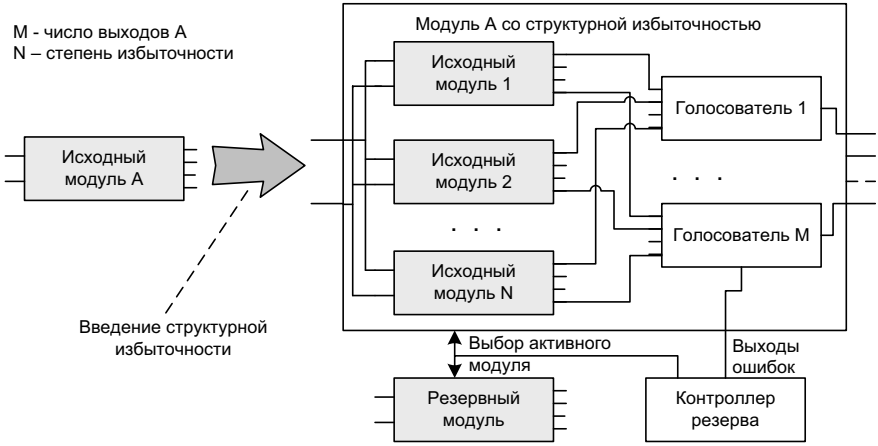

Рис. 1.11. Пример трансформации архитектуры с аппаратным резервированием

Кроме объединения выходов, голосователь может формировать сигнал о возникновении ошибки, когда один из модулей начинает выдавать сигналы, противоречащие остальным. В этом случае, при помощи специальных механизмов можно обеспечить замену вышедшего из строя модуля резервным блоком. Для этого в устройство добавляется дополнительная логика, называемая на рис. 1.11 “контроллером резерва”.

Дополнительные проблемы может добавить ситуация, когда средство анализа возникающих ошибок находится на ином уровне иерархии (например, при использовании единого модуля контроля резерва). В этом случае сигналы ошибок потребуется провести через все вложенные уровни иерархии, добавив в модули новые порты.

Все перечисленные преобразования в устройстве относятся к задачам трансформации архитектуры, описанным ранее. При ручной трансформации выполнение рассмотренного примера займёт много времени. В то же время рассматриваемые задачи трансформации можно формализовать, поэтому их относительно легко автоматизировать при наличии СПР, поддерживающего программируемые преобразования.

1.5. Постановка задач по исследованию СПР

Проведённое в первой главе исследование позволяет сформировать общее представление о реинжиниринге и проблемах его автоматизации. С точки зрения научной работы, наибольший интерес представляет разработка новых методов автоматизации реинжиниринга и новых представлений устройства, удобных для автоматизации обработки.

Рассматриваемая область достаточно широка, поэтому для формирования конкретных задач требуется предварительное исследование. Предполагается выделить основные проблемы автоматизации реинжиниринга и выбрать поле для дальнейшей разработки.

Как уже было сказано, автоматизации реинжиниринга как научная область является достаточно новым направлением. Число публикаций по данной тематике невелико, и в большей мере они рассчитаны на решение частных задач. Отсутствуют фундаментальные работы, в которых были бы описаны основы построения автоматизированных систем реинжиниринга. Хотелось бы как-либо систематизировать рассматриваемую область, поэтому первой задачей исследования является построение классификации СПР.

Имея классификацию, можно провести обзор существующих средств и выделить области, не покрытые существующими решениями. Целесообразно вести дальнейшую разработку именно в этих областях, так как это обеспечивает научную новизну разработки и перспективу практического применения результатов. Выбрав область работ, можно будет сформулировать основные требования к программному средству.

Ещё одной проблемой является внутреннее устройство самого средства реинжиниринга. Вызывают интерес его архитектура, используемые методики представления и трансформации устройства. Пусть число работ по данной тематике и невелико, но хотелось бы провести обзор и анализ уже существующих предложений.

Итак, при исследовании потребуется решить следующие задачи:
- сформировать классификацию СПР;
- исследовать существующие средства поддержки реинжиниринга;
- выбрать область для дальнейшего исследования и разработки;
- исследовать существующие методики представления и трансформации архитектуры устройства;
- сформировать требования к средству поддержки реинжиниринга.

Перечисленным задачам посвящена следующая глава диссертации.

2. ПРЕДВАРИТЕЛЬНОЕ ИССЛЕДОВАНИЕ СРЕДСТВ ПОДДЕРЖКИ РЕИНЖИНИРИНГА

В данном разделе рассматриваются общие вопросы по построению средств поддержки реинжиниринга устройства. Построена классификация средств поддержки реинжиниринга, рассмотрены методики представления и трансформации архитектуры устройств.

Основной задачей предварительного исследования является выбор проблемной области, в которой будет вестись автоматизация реинжиниринга. В пункте 2.3 проведён обзор автоматизированных СПР в котором показано, что задача программного преобразования устройства существующими средствами не решается. В 2.5 сформирован список задач, которые необходимо решить для построения программируемого СПР.

2.1. Построение классификаций средств поддержки реинжиниринга

Исходя из пункта 1.1.3, при реализации новой архитектуры приходится решать те же задачи, что и при разработке. Поэтому, в некоторой мере к средствам реинжиниринга можно отнести и средства разработки (компиляторы, анализаторы, средства моделирования и пр.).

Для подобных средств разработки существует множество различных классификаций (например, в [15] или [41]), которые можно применить и для средств поддержки реинжиниринга. Однако данные классификации не отражают специфику СПР с точки зрения самого процесса реинжиниринга. В работе сделана попытка ввести собственную классификацию, которая приведена в данном пункте.

2.1.1. Классификация СПР по решаемым задачам реинжиниринга

В СПР могут решаться любые задачи реинжиниринга. Если опираться на модель “подковы” (см. рис. 1.3), то можно выделить следующие группы задач:

- средства восстановления архитектуры (реверс-инжиниринга);
- средства поддержки принятия решений (СППР);
    - анализаторы;
    - экспертные системы;
- средства трансформации представления;
    - модификации архитектуры;
    - рефакторинга;
- средства контроля качества системы;
    - средства тестирования;
    - верификаторы;
- средства формирования выходных данных;
- средства разработки общего назначения.

Средства реверс-инжиниринга

Данные средства извлекают описание архитектуры объекта из имеющихся представлений: исходных кодов, нетлистов и т.п. В общем случае реверс-реинжиниринг можно рассматривать как преобразование объекта из одного описания в другое, более удобное для использования в процессе реинжиниринга.

Они нужны в том случае, если разработчик изначально не имеет доступа к описанию архитектуры. Такое может быть, если он использует недокументированный legacy-код или же закрытый IP-модуль.

Средства поддержки принятия решений

Реинжиниринг системы требует её детального анализа. Поэтому, на первый план выходят средства поддержки принятия решений (СППР), которые помогают разработчику выбрать оптимальные способы трансформации системы. Существует целый пласт подобных систем в области экономики, но данные системы можно отразить и на технические задачи.

В группу СППР входят анализаторы, которые позволяют выявить проблемы в текущей архитектуре системы. Например, в IDE Quartus II, используемой при разработке устройств на ПЛИС фирмы Altera, при компиляции проекта производится оценка временных характеристик устройства. С её помощью можно выявить “узкие места”, которые ограничивают повышение тактовой частоты и производительности устройства.

Экспертные системы, по сути, представляют собой базу знаний, которая содержит рекомендации для решения тех или иных проблем. Например, в случае недостатка производительности вычислительного модуля подобная система могла бы порекомендовать реализовать “конвейерную обработку”. Подобные системы позволяют накапливать историю проблем и их типовых решений, что в первую очередь полезно для разработчиков с малым опытом работы в области поставленной задачи.

Средства трансформации представления

Задачей данных средств является преобразование архитектуры системы на основании исходной архитектуры и принятых решений по реинжинирингу. В пункте 1.1.2 были выделены три группы задач:

- рефакторинг представления;
- трансформация архитектуры;
- перенос архитектуры между представлениями.

Из всех средств, рассматриваемых в данной классификации, только задачи трансформации представления относятся к самому реинжинирингу, а не поддержке данного процесса.

Средства контроля качества системы

Средства тестирования и верификации, по сути, являются анализаторами, но используются для проверки соответствия системы её спецификации. В процессе трансформации системы в неё вносятся модификации, которые могут внести в систему ошибки. Поэтому, контроль соответствия системы её спецификации является такой же неотъемлемой составляющей процессов реинжиниринга и разработки.

Средства формирования выходных описаний

После трансформации архитектуры требуется преобразовать её в требуемое выходное представление. Данный этап совпадает с реализацией в процессе разработки, поэтому в нём можно использовать обычные САПР. Выходным представлением могут быть исходные коды на HDL, нетлисты, текстовые спецификации, фотошаблоны и пр.

Ещё выходным форматом могут быть отчёты о результатах обработки (статистика, рекомендации к трансформации и пр.).

Средства разработки общего назначения

К данной группе в классификации относятся те средства, которые обычно применяются в процессе разработки, но могут быть применены и при реинжиниринге устройства. Из текущей классификации к данной группе можно отнести средства верификации и формирования выходных данных.

2.1.2. Классификация СПР по области применения

Средства поддержки реинжиниринга могут применяться в самых различных задачах. Средства могут быть универсальными или специализированными, а также ориентироваться на области, не связанными с реинжинирингом. На рис. 2.1 приведена более подробная классификация, которая будет пояснена далее.

Универсальным средством является то, которое охватывает все этапы процесса реинжиниринга системы и пригодно для решения любых задач. К данной группе относятся средства реинжиниринга, которые позволяют решать любые задачи из списка в пункте 1.4.2. Специализированными средствами называются те, которые решают только некоторые задачи из списка.

Под узкоспециализированными средствами понимаются те, которые реализуют лишь один алгоритм преобразования (например, заменяют все элементы памяти на их отказоустойчивые аналоги). В других же задачах данные средства неприменимы.

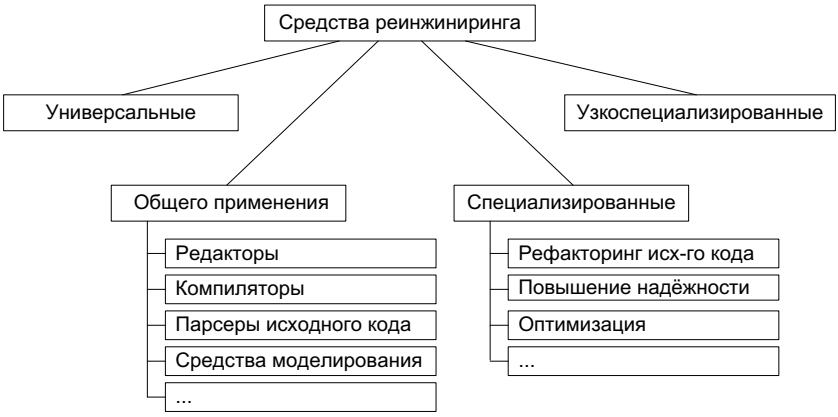

Рис. 2.1. Классификация СПР по области применения

2.1.3. Классификация СПР по интеграции с другими средствами

Средства поддержки реинжиниринга могут быть как завершёнными средствами (т.е. полнофункциональными модулями), так и набором некоторых компонентов, на базе которых может быть построено полноценное средство для решения пользовательских задач (см. рис. 2.2).

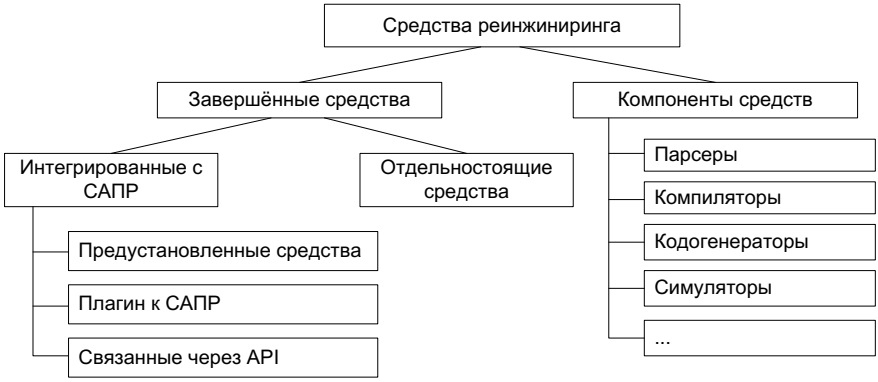

Рис. 2.2. Классификация по возможности интеграции с другими средствами

Завершённые СПР могут быть отдельностоящими средствами, когда пользователю приходится все операции выполнять в одной среде. В то же время возможно часть функциональности связать со средой разработки, позволив тем самым использовать функциональность самой среды (например, для визуализации архитектуры или компиляции). При этом из самой СПР возможен быстрый доступ к средствам реинжиниринга (например, актуально для рефакторинга кода).

СПР может быть встроенным, когда некоторые базовые функции реинжиниринга предоставляет сама среда разработки. Возможно реализовать средство реинжиниринга в виде отдельного включаемого модуля (плагина), который может быть добавлен в среду разработки при необходимости. Таким образом, среда предоставляет некоторый межпрограммный интерфейс (в англ. Application Programming Interface или API), через который модуль реинжиниринга получает доступ к её средствам.

Возможен и обратный подход, когда API предоставляется самим СПР, а уже среда разработки взаимодействует с ним. Такой подход распространён в модульных средствах, когда этапы разработки реализуются отдельными средствами, а задачей среды является вызов данных обработчиков в соответствии с конфигурацией.

2.1.4. Классификация СПР по степени автоматизации

Средства поддержки реинжиниринга могут быть:
- частично автоматические (автоматизированные);
- автоматические;
- программируемые.

Частично-автоматические средства автоматизируют лишь некоторые этапы реинжиниринга. Например, для реинжиниринга устройства можно использовать любую САПР. Этим можно полностью автоматизировать этап компиляции устройства, но трансформация архитектуры и модификация исходных кодов должна производиться пользователем. Например, пользователь может дать команду “троировать все элементы памяти” в одном из блоков, после чего все необходимые операции будут выполнены автоматически.

Автоматические средства реинжиниринга могут выполнить некоторые задачи без вмешательства пользователя. Ключевой особенностью данных средств является то, что автоматизируется принятие решений, а пользователю лишь нужно задать требования. Например, он может дать команду “минимизировать вероятность ошибок в памяти”, а система уже сама принимает решение на основании имеющихся системных ресурсов.

Программируемые средства являются, на взгляд автора, отдельным классом СПР. В них возможен любой уровень автоматизации, но сначала требуется задать алгоритм для выполнения пользовательской команды. Но после этого пользователь получает средство, в точности соответствующее его задачам, и в дальнейшем он всегда может его модифицировать.

2.1.5. Классификация СПР по представлению устройства

Для выполнения задач реинжиниринга требуется обработка архитектуры устройства. Поэтому, необходимо выбрать такое внутреннее представление, которое обеспечило бы возможность выполнения поставленных задач. Возможны следующие подходы:

- использование входного представления;
- использование специализированного представления;
    - связь с исходным представлением;
    - отсутствие связи с исходным представлением;
- использование специального представления для каждой задачи;
    - связь с исходным представлением;
    - отсутствие связи с исходным представлением.

Входные представления используют, например, средства рефакторинга, которые вносят изменения в файлы исходных кодов. Для подобных средств возможно каскадное применение, когда над одними и теми же данными последовательно выполняются несколько различных типов преобразований.

Тем не менее, у подобных средства много проблем, связанных с необходимостью опираться на представление в исходных файлах. Например, при переименовании сигнала в текстовых HDL нужно найти все упоминания в исходных файлах, проверить отсутствие конфликтов и только потом изменить имена. С усложнением преобразований увеличивается сложность связей, что требует использования специализированного формата представления данных.

Используя внутреннее представление, можно относительно легко выполнить все требуемые задачи, после чего сгенерировать выходное представление архитектуры в требуемом формате. Подобный подход прост, но имеет следующие недостатки:

- требуется обеспечить экспорт представления в выходной формат;
- разрывается связь между входными и выходными данными.

Иногда сохранение связи с входным представлением очень важно. Например, при рефакторинге исходных кодов пользователь рассчитывает получить ограниченные задачей изменения. Если же не сохранять исходное представление, то могут быть потеряны порядок следования объявлений, табуляция кода, комментарии и т.п. Таким образом, несмотря на сохранение функциональности, пользователь получит новое описание, что затруднит дальнейшую разработку. Подобный подход хорош для преобразований, не требующих соответствия выходного представления входному: анализаторов, оптимизаторов и т.п.

Компромиссным вариантом является использование внутреннего представления, которое каким-либо образом ссылается на входное. Тогда связь

сохраняется на протяжении всего преобразования, и на выходе можно получить представление, близкое к исходному. Такой подход требует модификации сразу двух представлений в процессе обработки, что в ряде случаев может быть достаточно сложным.

Также в СПР для каждой задачи может использоваться специализированное представление. Такой подход позволяет добиться более простого решения отдельных задач, но плохо подходит для расширяемого средства реинжиниринга, где решаемые задачи заранее неизвестны.

Альтернативным подходом может быть наличие нескольких представлений, когда пользователь может выбрать то из них, которое ему более подходит. При этом задачи синхронизации представлений между собой должны решаться самим средством.

Заключение

Можно сформировать набор классификаций по используемым методикам представления и трансформации устройства, внутренней архитектуре и прочим признакам.

Подобные вопросы требуют дополнительного исследования. В некоторой мере данные классификации будут рассмотрены в последующих пунктах данной главы. Текущей информации достаточно, чтобы перейти к обзору методик их построения.

2.2. Исследование существующих решений по автоматизации реинжиниринга устройства

2.2.1. Методики представления устройства при реинжиниринге

Классические представления

К данной группе представлений устройства относятся текстовые описания, блок-схемы, принципиальные схемы, диаграммы соединений и прочие тексто-графические описания. Подобные способы представления применяются при низкоуровневом проектировании системы, разработке спецификаций и технических заданий.

Подобные представления хороши для человека и используются при документировании устройства. Для машинной обработки они не подходят из-за низкой степени формализации, что приводит к большому числу неточностей в описаниях. Тем не менее, существуют средства, которые позволяют конвертировать данные описания в HDL (например, [7]).

Описание на HDL

Если устройство уже специфицировано на каком-либо HDL, то можно использовать данное описание и для реинжиниринга. Данный подход в первом приближении кажется наиболее простым, так как не требует преобразования одного представления устройства в другое.

Использование исходных кодов для машинной обработки затруднено наличием в них незначащих для машинной обработки составляющих кода: пользовательских комментариев, форматирования текста, синтакса объявления операндов. Всё это усложнит алгоритмы обработки. Поэтому, на практике первым этапом обработки является парсинг исходных кодов и формирование некоторого абстрактного представления.

Представление кода в виде абстрактных графов

Машинная обработка исходных кодов (в т.ч. и HDL) редко ведётся напрямую. Обычно строится некое дерево, которое в той или иной мере отображает исходное содержимое. Можно выделить следующие виды деревьев:

- абстрактное синтаксическое дерево (AST);
- абстрактный семантический граф (ASG);
- объектный (архитектурный) граф.

Абстрактное синтаксическое дерево – это ориентированное дерево, вершины которого сопоставлены с элементами исходного описания (кода) [8]. Представление в виде AST позволяет абстрагироваться от составляющих исходного описания, не используемых при реинжиниринге. На рис. 2.3 приведён пример AST-дерева VHDL для объявления сущности (Entity) голосователя с переменным числом входных сигналов.

Синтаксическое дерево достаточно громоздко и не содержит связи между элементами. Например, при использовании переменной в коде она объявляется в одном месте, а используется в другом. В случае модификации данного сигнала в представлении AST нам придётся пройти по всему дереву, найти все ссылки на данный сигнал и модифицировать их. Поэтому, представление в AST не подходит для использования в средстве реинжиниринга.

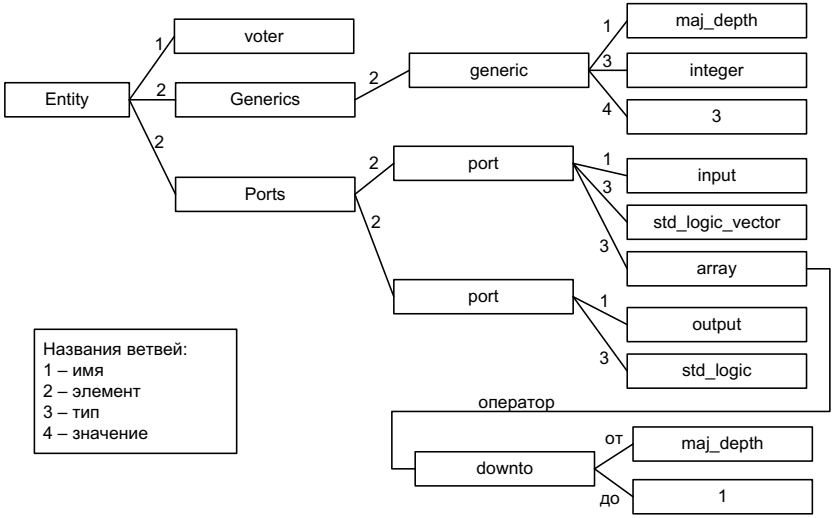

Рис. 2.3. Пример абстрактного синтаксического дерева для VHDL

Абстрактный семантический граф (АСГ) – это расширенное представление синтаксического дерева, рассмотренного выше. При формировании такого графа из вершин могут быть добавлены ссылки между связными элементами, удалены неинформативные вершины и т.п.

Объектный (архитектурный) граф

В программной реализации СПР представление АСГ можно расширить, введя вместо ячеек графа полноценные объекты со своими свойствами, методами и иерархией наследования. Подобный подход называется объектным [23] или архитектурным [24] графом. Подход позволяет реализовать описание устройства, которое достаточно тяжело представить пользователю, но зато автоматизированную обработку произвести достаточно легко.

Примером использования объектного графа является методика представления AIRE/CE (Advanced Intermediate Representation with Extensibility – Продвинутое промежуточное представление с возможностью расширения), разработанная компанией FTL Systems и университетом Цинцинатти (см. [27])

Данное представление ориентировано на представление языков VHDL и VHDL-AMS и программную реализацию на языке C++. Расширяемость представления достигается за счёт средств языка реализации: наследования и виртуальных функций. Данная методика представления ориентирована на построение средств анализа исходных кодов, никаких дополнительных решений по трансформации архитектуры авторами не предлагается.

Структурное представление устройства

Все предыдущие методы применимы для любого описания, в том числе и для устройства. Но существуют специфические подходы, в частности: структурное и поведенческое описание, которые будут рассмотрены далее.

Структурное описание устройства является наиболее простым с точки реализации, так как все HDL в той или иной мере его поддерживают. В идеале, в структурном описании присутствует только иерархическое описание модулей и связей между ними.

К сожалению, даже самые простые описания устройства (нетлисты), используемые для передачи между различными средами, содержат элементы функционального описания – операторы, функции и т.п. Например, формат VHDL-нетлистов поддерживает группированные объекты и присвоения по группам (шинам) [37]. Поэтому, при импорте представления из HDL требуется произвести преобразование всех элементов в некоторое структурное подобие.

Поведенческое описание устройства

Поведенческое описание устройств является более новым направлением, чем структурное. Его поддерживают далеко не все HDL, поэтому при использовании подобных представлений в средстве реинжиниринга потребуется обеспечить конвертацию структурных описаний в поведенческие (и наоборот). Принципиально, данная задача относится к переносу архитектуры между представлениями, что является одной из задач реинжиниринга. Для её решения существуют специальные методики, во многом схожие с процессом синтеза устройства из поведенческих описаний [18].

Поведенческое описание наиболее распространено в работах, посвящённых динамическому анализу и верификации устройства, где необходимо симулировать поведение устройства. В случае с трансформацией архитектуры поведенческое описание является второстепенным, но существует ряд методик по рефакторингу на основе данного представления.

Одним из наиболее проработанных подходов является представление, разработанное в университете Цинцинатти [28]. В настоящее время оно используется в нескольких средствах анализа (SAVANT, TyVIS и пр.). Подход заключается в том, что всё описание укладывается в четыре группы: сигналы, иерархии портов, присвоения сигналов и операторы временных зависимостей. Авторами были полностью формализованы методики построения и трансформации архитектуры.

К сожалению, введённое авторами представление не содержит методик для структурных преобразований, и, ввиду недостатка элементов структурного описания, данная задача кажется трудновыполнимой.

Смешанное описание

Использование смешанного структурно-поведенческого описания является наиболее сложным подходом, так как одну и ту же конструкцию можно описать различными способами. В то же время требуется каким-то образом связать между собой два принципиально разных описания, чтобы обеспечить их совместную обработку.

При проведении обзора существующих СПР не было выявлено ни одного серьёзного инструмента, который одновременно использовал бы оба представления. В большинстве же своём, средствами анализа используются не исходные коды на HDL, а более простые структурные описания из нетлистов.

Надо отметить, что построение средства реинжиниринга с поддержкой обоих представлений является весьма актуальной задачей, так как это позволило бы восстанавливать не только структуру устройства, но и его поведенческое описание, что уже является практически применимым результатом. В данном направлении ведутся академические работы, но в рамках магистерской работы подобная задача представляется чересчур тяжёлой.

2.2.2. Методики автоматизации преобразований

Методики трансформации устройства тесно связаны с используемыми представлениями и определяют алгоритмы преобразования представления. Поэтому, на данном этапе невозможно рассмотреть существующие решения. Рассмотрим лишь общие подходы.

Наиболее интересно, каким образом организуется передача алгоритмов в САР, т.е. его программирование. Всего можно выделить четыре подхода:

- использование фиксированного набора алгоритмов;
- трансформация по набору правил;
- программное управление трансформацией;
- управление трансформацией из внешней программы.

Фиксированные алгоритмы обработки

Фиксированные алгоритмы обработки используются в узкоспециализированных средствах, решающих лишь некоторые, заранее определённые задачи. Данные алгоритмы определяются внутри САР, и интерфейсы для программирования не требуются.

Трансформация по набору правил

По аналогии с предыдущим подходом, в САР реализуются фиксированные алгоритмы обработки. Однако, данный подход предлагает использовать используют некоторые входные правила, которые адаптируют поведение алгоритма под конкретную задачу.

Например, пусть требуется с помощью САР переименовать группы элементов в устройстве. Можно реализовать универсальный алгоритм, который принимает на вход список элементов для преобразования и последовательно осуществляет переименование для каждого элемента. В этом случае список элементов можно рассматривать как набор правил преобразования.

Программируемая трансформация

В данном случае пользователю предоставляется возможность самостоятельно реализовать алгоритм обработки данных. Далее посредством компиляции программа превращается в набор команд средства или же интерпретируется в процессе обработки. Язык реализации алгоритма в работе предлагается называть языком управления преобразованиями.

Актуальна задачи поддержки средством некоторых универсальных языков программирования (C/C++, Java, Python, …). При этом пользователь получает доступ ко всей функциональности языка (или к значительной его части). Программа управления преобразованиями может взаимодействовать с внешними средствами, если в САР для этого предусмотрен специальный шлюз (см. рис. 2.4).

Интеграция средства в существующую оболочку

Данный подход обратен предыдущему. Вместо разработки нового языка управления преобразованиями можно включить вызов функций САР в уже существующий инструментарий (например, в виде дополнительного вызова) или язык программирования.

> Например, при интеграции средства в командную оболочку (например, bash), пользователю становится доступна вся его функциональность: строковые процессоры для обработки данных и отчётов других средств.

Таким образом, пользователь сразу получает полный доступ к внешней функциональности системы, но при этом затрудняется доступ к внутреннему представлению. Поэтому, данный подход удобно комбинировать с предыдущим: связанные с представлением вещи можно реализовать внутри специальных программ средства, а внешнюю обработку оставить снаружи. На рис. 2.4 приведена структура подобного средства.

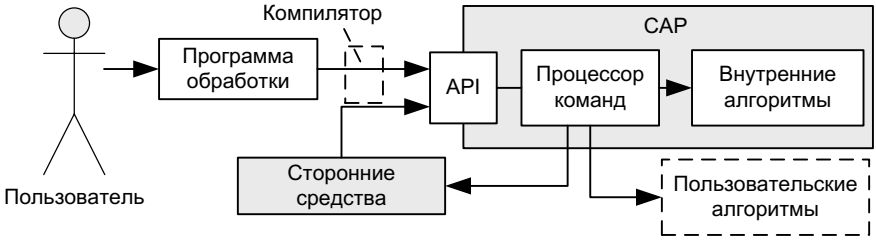

Рис. 2.4. Схема работы САР с различными источниками алгоритмов обработки устройства

2.3. Обзор средств автоматизированного реинжиниринга устройства

При предварительном исследовании предметной области был проведён обзор существующих средств поддержки реинжиниринга для следующих категорий:

- средства разработки общего назначения;
- средства рефакторинга представления устройства;
- универсальные программируемые СПР;
- узкоспециализированные СПР.

Для каждой из групп было выявлено некоторое количество средств разработки. В работе решено не приводить полный обзор средств, а рассмотреть по нескольку примеров из каждой группы. Исключение составляет группа программируемых СПР, так как для неё удалось найти всего лишь одно средство.

2.3.1. Средства общего назначения

К данным средствам относятся те средства, которые используются в процессе разработки: редакторы, компиляторы, симуляторы и так далее. Они не ориентированы на реинжиниринг, но могут быть использованы для отдельных задач процесса. Обычно средства разработки строятся по модульному принципу, после чего объединяются в цепочки средств (toolchain), каждое из которых решает отдельную задачу. Подобные пакеты программ выпускают следующие компании:

- Cadence;
- Mentor Graphics;
- Synopsys;
- Altera;
- Xilinx;
- …

Перечисленные выше компании выпускают средства для всех этапов проектирования и различных уровней сложности проектов, их номенклатура очень велика. Например, Cadence предлагает более 50 различных продуктов [36].

Очевидно, что даже такие комплексные пакеты не могут решить все возможные задачи разработки. Поэтому, все указанные пакеты имеют интерфейсы для подключения внешних средств автоматизации разработки (Electronic Design Automation или EDA), которые могут решать отдельные задачи процесса разработки. Например, среда Quartus II фирмы Altera поддерживает следующие группы средств:

- синтез;
- симуляция;
- временной анализ;
- формальная верификация;
- синтез исходных кодов [9].

Инструменты взаимодействия с EDA

Для упрощения взаимодействия оболочки со сторонними средствами автоматизации разработки устройства (Electronic Design Automation или EDA) последние должны обладать универсальными интерфейсами для передачи команд и данных.

Проведённый обзор показал, что в реальности механизмы взаимодействия не стандартизированы, и существует широкий спектр форматов, многие из которых поддерживаются лишь одним средством. Наиболее распространёнными форматами являются три типа нетлистов: EDIF, Verilog- и VHDL-нетлисты.

2.3.2. Средства рефакторинга

Автоматизация рефакторинга исходных кодов является одной из наиболее распространённых задач реинжиниринга. Поэтому существует достаточно много средств, которые включают в себя те или иные возможности рефакторинга. В большинстве подобные средства входят в состав средств разработки.

Sigasi HDT

Sigasi HDT – это полноценная среда разработки для VHDL, поставляемая компанией Sigasi. На официальном web-сайте проекта основными преимуществами называются динамическое выявление ошибок, контекстная подстановка имён из базы знаний, а также наличие модуля рефакторинга кода [43]. Данный модуль реализует следующие возможности:

- переименование элементов;
- добавление портов в Entity;
- инкапсуляция выражений в пакеты (package);
- присоединение Entity к внешнему сигналу (с созданием порта).

Последняя из перечисленных возможностей, в соответствии с принятым в работе разграничением (см. 1.4.1), может быть отнесена к реинжинирингу.

AMIQ Design and Verification Tool (DVT)

ANIQ DVT – это среда разработки на SystemVerilog и *e* (внутренний объектно-ориентированный HDL), выполненная в виде плагина к IDE Eclipse. Среда поддерживает такие простейшие функции рефакторинга, как переименование элементов описания и инкапсуляцию имён. Кроме того, для SystemVerilog присутствует поддержка рефакторинга под управлением скриптового файла [34], за что среда и привлекла внимание автора.

Скриптовый XML-файл содержит список элементарных действий, которые следует выполнить в процессе рефакторинга. Действия включают переименования различных элементов устройства, имён исходных файлов, добавление меток (например, TODO) и комментариев.

Формат скриптового файла не поддерживает сколько-либо сложную выборку объектов для преобразования (например, через регулярные выражения), поэтому возможности подобного рефакторинга крайне ограничены. Тем не менее, при помощи данного средства можно, например, произвести перенос устройства между некоторыми базовыми библиотеками со схожими интерфейсами.

Таким образом, примитивное средство рефакторинга наподобие DVT может решить и некоторые, очень узкие задачи реинжиниринга. Однако, сложность реализации подобных преобразований невелика, и в случае возможности следует использовать подобные средства.

RAMS

Рефакторинг описаний на HDL возможен не только для цифровых, но и для смешанных устройств. Например, средство RAMS позволяет производить рефакторинг описаний на языке VHDL-AMS [19]. В процессе преобразований RAMS использует объектное дерево, которое генерируется из АСГ (см. рис. 2.5). Разбор грамматики производится с использованием стандартных библиотек с открытыми исходными кодами, а сам RAMS реализует только рефакторинг.

Подход к трансформации VHDL-AMS в RAMS полностью соответствует рассмотренным в пункте 2.2 методикам. Можно заключить, что средство реинжиниринга, построенное с использованием рассмотренных методик, можно будет перенести и на описания цифро-аналоговых устройств.

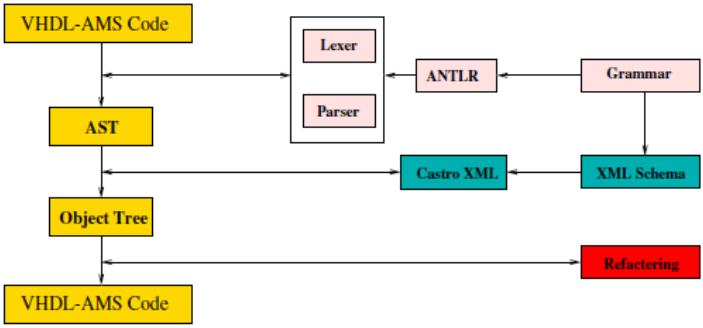

Рис. 2.5. Процесс рефакторинга VHDL-AMS с использованием RAMS [19]

2.3.3. Программируемые средства реинжиниринга устройства

DMS Software Reengineering Toolkit

Данный продукт заявлен как универсальное средство реинжиниринга программного обеспечения [42]. Средство поддерживает более двадцати языков программирования, в том числе VHDL и Verilog. Архитектура средства (см. рис. 2.6) имеет модульную структуру. Средство состоит из четырёх основных модулей: парсера исходных кодов, анализатора, модуля трансформации и генератора выходных файлов, что полностью соответствует структуре СПР из пункта 1.4.1.

Автоматизация в средстве достигается за счёт набора правил обработки, которые компилируются отдельным модулем (Компилятор правил) из пользовательских программ обработки, которые могут быть написаны на множестве языков (в т.ч. C++, Python, Java).

Получается, что рассматриваемое средство позволяет полностью автоматизировать обработку и реинжиниринг устройства, если пользователь корректно специфицирует правила преобразования. Тем не менее, у средства есть ряд ограничений:

- отсутствие интерфейса к сторонним средствам;
- представление устройства в виде семантического графа.

Главным недостатком DMS SRT является его закрытость. Пользователь может лишь косвенно управлять процессом преобразования, программируя правила. Но при этом невозможно встроить вместо модуля-анализатора некий свой модуль со специфичным внутренним представлением.

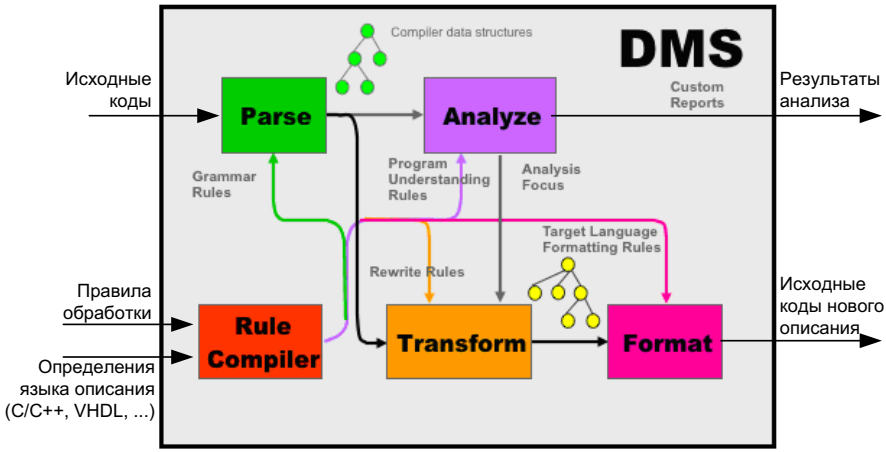

Рис. 2.6. Схема работы DMS Software Reengineering Toolkit [42]

По мнению автора диссертации, в случае DMS Software Reengineering Toolkit универсальность подхода и поддержка многих языков являются скорее недостатком, так как средство ориентировано не на реинжиниринг целевого объекта, а на реинжиниринг его исходных кодов. Тем не менее, оно является единственным из рассмотренных средств, на базе которого может быть построено полностью автоматическое средство реинжиниринга устройства с программируемой функциональностью.

2.3.4. Специализированные средства автоматизации реинжиниринга

Специализированные средства автоматизации реинжиниринга составляют большинство существующих средств. Причиной этому является то, что в большинстве случаев разработчику (даже в рамках фирмы) требуется реализовать небольшое число алгоритмов обработки. Естественно, в этом случае разработка универсального средства отдельному разработчику (да и целому предприятию) невыгодна из-за больших затрат.

Покупка и адаптация универсальных средств возможна, если их суммарная стоимость владения будет ниже, чем стоимость разработки нового специализированного средства. К сожалению, подобное условие редко выполняется даже для областей с более широким рынком (как в случае с математическими пакетами).

В результате получается, что разрабатываются внутренние средства, которые решают лишь узкие задачи, специфичные для определённой компании. Подобные решения, если и выходят на рынок, то имеют низкий спрос и мало поддерживаются, в результате чего на их последующую универсализацию

рассчитывать не приходится. Ниже приведён пример для одного из специализированных средств.

BYI EDIF Tools [38]

Данное средство предоставляет API(Application Programming Interface) для редактирования и анализа EDIF-нетлистов. С точки зрения реинжиниринга, средство включает следующие функции:

- генератор синтезируемых кодов на JHDL;
- объединение нескольких устройств в один нетлист;
- троирование элементов.

Средство использует внутреннее представление в виде АСД, поэтому, несмотря на наличие API, его применение для задач реинжиниринга затруднительно. Ещё одним ограничением является использование JHDL в качестве выходного представления, так как поддержка данного языка средствами разработки ограничена.

2.4. Выбор области для дальнейшего исследования

Проведённый обзор существующих средств показал, что большинство из них нацелено на решение узкоспециализированных задач реинжиниринга или же рефакторинг исходных кодов.

Было обнаружено несколько средств, в которых имеется возможность программирования средства реинжиниринга при помощи набора правил (DMS Reengineering Toolkit, DVT). В DVT возможно только программирование рефакторинга, а в DMS функциональность ограничена универсализацией средства, из-за чего внутреннее представление в средстве близко к АСГ, чего для многих задач реинжиниринга недостаточно.

Исходя из сказанного, можно выделить два возможных направления разработки: реализация специализированного средства для решения частных задач (например, повышения отказоустойчивости) или же разработка универсального программируемого САР.

Задача построения универсального САР более привлекательна с точки зрения научной новизны и практической ценности, но в рамках магистерской работы её решение невозможно ввиду большой сложности. С другой стороны, возможно реализовать не полноценное САР, а некоторый инструментарий, на базе которого пользователь мог бы реализовать и автоматизировать собственные задачи реинжиниринга устройства.

Данный инструментарий должен предоставлять следующие возможности:

- представлять устройство в виде, удобном для анализа и трансформации;
- включать набор команд для трансформации представления устройства;

- предоставлять возможность внесения пользовательских алгоритмов обработки;
- позволять повторное использование предыдущих наработок (алгоритмов, модулей устройства и т.п.):
- быть относительно простым в эксплуатации.

На основании данных предпосылок были сформированы требования к программному средству, которые приведены в пункте 3.1.

2.5. Постановка задач по дальнейшей разработке

В данном разделе построена классификация и проведён обзор существующих средств поддержки реинжиниринга. Результаты обзора показали, что автоматизируются лишь частные задачи реинжиниринга, в то время как ни одно существующее СПР не претендует на универсальность. Поэтому, данная область представляет наибольший интерес для дальнейших исследований.

Проведённый обзор показал, что в настоящее время не существует полноценных программируемых средств реинжиниринга устройств. Тем не менее, данная задача актуальна с точки зрения научной новизны и практической применимости, поэтому она была выбрана для дальнейшей разработки.

Будем пытаться строить программируемое средство реинжиниринга устройств в соответствии с подходами, которые были описаны в первой главе. В первую очередь потребуется сформировать требования к средству, выбрать методики внутреннего представления и трансформации устройства. После этого можно будет сформировать архитектуру средства и перейти к программной реализации.

В пункте 2.3 были рассмотрены существующие методики представления и трансформации устройства. Оказалось, что большинство работ носят исследовательский характер, а предлагаемые ими методы ориентированы лишь на частные задачи реинжиниринга.

Для структурных преобразований наиболее актуальным является использование архитектурного графа. К сожалению, не было найдено методик по построению данного представления, ориентированного на решение задач трансформации архитектуры. Поэтому, при разработке САР потребуется реализовать собственное представление и, соответственно, механизм трансформации для него.

Очевидно, что для программного средства реинжиниринга следует использовать некое объектно-ориентированное представление, но вопрос набора элементов и архитектуры элементов остаётся открытым остаётся открытым.

Разрабатываемое средство будет встраиваемым, то есть его можно будет интегрировать со сторонними средствами разработки. Поэтому, в основном будут рассматриваться вопросы управляемой трансформации архитектуры и ввода-вывода представлений. Все прочие операции предлагается оставить

3. РАЗРАБОТКА ПОДХОДОВ К ПОСТРОЕНИЮ АВТОМАТИЗИРОВАННОГО СРЕДСТВА РЕИНЖИНИРИНГА

В данном разделе рассмотрены основные этапы разработки подходов к построению САР устройства. Сформированы требования, на основании которых разработаны внутреннее представление устройства, язык управления преобразованиями и архитектура средства реинжиниринга.

В работе строится не универсальное средство реинжиниринга, а базовый инструментарий, через который можно реализовать пользовательские задачи. Таким образом, не ставится задача разработки методов и алгоритмов реинжиниринга, но предлагаются подходы по автоматизации процесса с использованием программного средства.

В данном разделе не ставится задача построения полной спецификации на САР, а только разработка подходов. Это возможно лишь после апробации подходов, которым посвящены четвёртый и пятый разделы диссертации. Построение спецификаций на САР как завершённый программный продукт и его реализация в работу не входят.

3.1. Разработка требований к программному средству

3.1.1. Требования по функциональности

Как уже было сказано, перед пользователем средства могут стоять самые различные задачи. Невозможно предусмотреть все из них. Поэтому, для универсального СПР, прежде всего, важна не конечная функциональность, а удобство адаптации средства под пользовательские задачи.

В соответствии с постановкой задачи, рассматриваемое в работе САР прежде всего должно автоматизировать процесс трансформации архитектуры устройства. Однако, для полноценного средства одной этой возможности недостаточно. В базовой конфигурации требуется обеспечить выполнение основных операций, а именно:

- импорт и экспорт внутреннего представления в стандартные форматы;
- программируемое преобразование устройства;
- возможность доступа ко всей информации об устройстве;
- верификация внутреннего представления.

Так как средство универсально с точки зрения решаемых задач, то неизвестно, какая информация об устройстве потребуется для пользовательских функций. Нужно предоставить пользователю доступ ко всей информации из внутреннего представления. Дальнейшая ей обработка и анализ остаётся на пользователе.

Верификация внутреннего представления является важным этапом при внесении изменений в архитектуру устройства. Прежде всего, должна контролироваться связность графа, чтобы впоследствии из него можно было синтезировать устройство.

Программируемость

Разрабатываемое средство должно быть программируемым, чтобы пользователь мог реализовать в нём свой алгоритм преобразования. Естественно, для этого должен существовать некоторый язык описания, называемый в работе “языком управления преобразованиями”. Требования к данному языку рассмотрены в пункте 3.1.4.

Идеальным вариантом был бы случай, когда возможности программирования в средстве позволяют реализовать любой алгоритм обработки, т.е. достигается полнота по Тьюрингу [20].

Режимы работы средства

Средство обязано поддерживать автоматический режим исполнения, когда преобразования производятся по некоторой заданной программе без участия пользователя.

Кроме того, нужно предоставить возможность последовательного выполнения отдельных команд. Это полезно для проверки отдельных команд при разработке или отладке программы реинжиниринга. При наличии подобного режима средство можно интегрировать с внешними средствами разработки, которые будут управлять преобразованиями. Подобный режим можно предоставить и пользователю. В будущем его предлагается называть интерактивным.

Навигация по представлению

Для упрощения обработки дерева внешними средствами предлагается ввести механизмы навигации. Прежде всего, хотелось бы иметь возможность получить доступ к элементам представления по абстрагированному текстовому описанию, которое предлагается называть путём к элементу. Для каждого элемента данный путь должен быть уникален, чтобы были исключены конфликты при доступе.

В ряде случаев требуется реализовать последовательную обработку элементов представления (в частности, при анализе). Поэтому, в CAP хотелось бы предусмотреть механизм, которые позволит сохранять текущее положение в представлении и вести доступ к другим элементам относительно выбранного элемента. Таким образом, требуется реализовать косвенную адресацию к составляющим представления.

Хранение истории команд

При работе в интерактивном режиме может потребоваться сохранить использованную последовательность команд, чтобы в будущем была возможность её повторения. Для этого в средстве должна быть реализована возможность хранения и вывода историю команд.

Отмена действий

Может возникнуть ситуация, когда требуется отменить результаты выполнения последней команды или группы команд. Для этого могут быть следующие причины:

- при выполнении команды возникла ошибка;
- результаты трансформации архитектуры не соответствуют требованиям или ожиданиям пользователя;
- в интерактивном режиме пользователем была вызвана неверная команда преобразования.

Хотелось бы, чтобы в средстве реинжиниринга был предусмотрен механизм для отмены команд с восстановлением исходного состояния. Подобный механизм имеется в большинстве средств разработки (например, команда Undo в большинстве редакторов).

3.1.2. Требования к внутреннему представлению

При исследовании были сформулированы следующие требования к внутреннему представлению:

- структурное описание устройства;
- использование архитектурного графа;
- ограниченный набор базовых ячеек;
- параметризация ячеек;
- наследование ячеек через расширение свойств;
- поддержка одновременной работы с несколькими устройствами;
- близость к одному из HDL.

Далее в тексте приведено обоснование поставленных требований.

Представляемая информация

Существует два основных подхода к представлению устройства: структурное и поведенческое описания. Как было отмечено ранее, данные представления можно конвертировать между собой. Поэтому, в СПР не обязательно поддерживать оба варианта представления, ведь задачу конвертации представлений можно перенести в отдельные модули.

С точки зрения реинжиниринга, структурное представление гораздо проще для преобразования представлений. Кроме того, его поддерживают все

HDL, в том числе и структурные описания из нетлистов. Последнее может быть очень полезно в рамках оптимизации уже готового устройства. Очевидно, что универсальное СПР должно поддерживать структурное представление.

В HDL не вся информация является синтезируемой, т.е. используемой для компиляции устройства. Например, в VHDL к несинтезируемому подмножеству относятся комментарии, временные задержки и пр. Подобная информация может быть использована в СПР для удобства восприятия пользователем, автоматической генерации тестов или при анализе временных задержек в устройстве. Тем не менее, для трансформации архитектуры несинтезируемое подмножество не является обязательным, и во внутреннем представлении устройства подобные данные отображать не обязательно.

Использование архитектурного графа

Внутреннее представление, прежде всего, должно быть ориентировано на автоматическую обработку. Должен существовать простой механизм для получения информации об архитектуре устройства, составных элементах и связях между ними.

Из рассмотренных в п. 2.2.1 подходов наиболее удачным кажется использование архитектурного графа. Данный подход изначально ориентирован на программную реализацию в виде связанного набора объектов, каждый из которых определяет сущность устройства. В рамках САР данный подход удобен как для анализа, так и для трансформации устройства.

Близость к одному из языков описания аппаратуры

Внутреннее представление рассчитано, в первую очередь, на структурное описание устройства, которое поддерживается практически всеми HDL. Для упрощения основания средства целесообразно во внутреннем представлении использовать семантику дерева, близкую к одному из популярных языков (например, VHDL или Verilog).

Это позволит упростить анализ внутреннего представления и его сравнение с исходными кодами на HDL. Например, за основу можно взять подмножество некоторого языка, используемое в нетлистах.

Ограниченный набор базовых ячеек

При использовании представления в виде объектного графа, появляются широкие возможности по заданию узлов этого дерева. Описывая структуру устройства, мы можем на каждый элемент (сигнал, модуль и т.п.) создать свой специфический объект.

Надо учитывать, что средство реинжиниринга будет строиться по модульному принципу. Если каждый модуль будет добавлять в представление

свои объекты, то нельзя гарантировать что сторонний модуль обработки сможет их корректно обработать. Поэтому, в средстве должен быть задан фиксированный набор типов ячеек, на базе которых формируется представление устройства. Дополнительная функциональность может быть реализована через расширение отдельных типов, о котором речь пойдёт далее.

Параметризация ячеек

Требуется обеспечить получение информации о ячейках и навигацию по ним. Для этого необходимо хранить и предоставлять информацию об имени, типе и специальных параметрах ячеек. Предлагается для каждой ячейки хранить набор параметров и предоставить пользователю возможность расширения числа параметров для описания вводимой им функциональности.

Расширяемость представления

Наиболее популярные языки описания устройства VHDL и Verilog появились примерно в то же время, что и языки объектно-ориентированного программирования (ООП). Однако, данные языки в явном виде не поддерживают наследования свойств одного объекта другим, что было бы удобно для некоторых задач реинжиниринга, связанных с расширением функциональности через добавление новых модулей.

Наиболее часто встаёт задача замены одного аппаратного модуля другим с аналогичными интерфейсами. Возможна замена и модификация только части блоков. Например, из внутреннего модуля иерархии выводится некоторый диагностический сигнал A. Во всех уровнях до обработчика этого сигнала надо добавить порты и подключить сигнал, для чего в VHDL потребуется ввести новые Entity и архитектуры к ним. В то же время хотелось бы сохранить связь создаваемых элементов с исходными.

В этом случае было бы удобно реализовать наследование некоторых свойств базового объекта и их сохранение при модификации. Будем называть это наследованием архитектуры.

В ООП существует множество подходов к организации наследования. Наиболее простой формой наследования является расширение архитектуры, когда потомок лишь расширяет архитектуру объекта-предка. Как пишет Бадд: “В то время как порождение подкласса для обобщения модифицирует или расширяет существующие функциональные возможности объекта, … Расширение просто добавляет новые методы к родительским, и функциональные возможности подкласса менее крепко привязаны к существующим методам родителя” [2]. Ещё большим упрощением является использование лишь одного класса-прародителя.

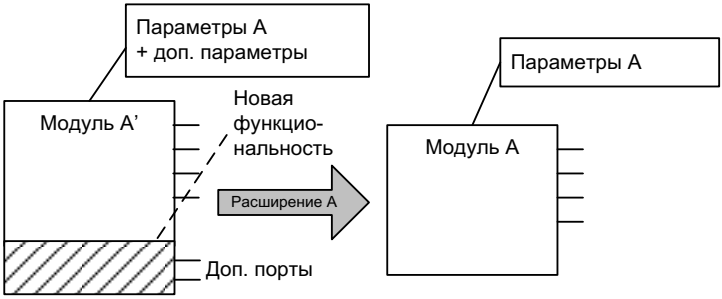

Рис. 3.1. Пример наследования архитектуры элементами через расширение

Если взять структурное подмножество HDL, то можно предложить три варианта наследования через расширение:

- расширение интерфейса (добавление новых сигналов);
- расширение структуры (добавление новых модулей);
- добавление новых параметров.

Поддержка нескольких устройств

Возможен случай, когда пользователю требуется вести параллельную обработку сразу нескольких устройств. Например, такой вариант может быть удобен при копировании элементов из одного устройства в другое или их объединении.

Корневым элементом внутреннего представления должно быть не устройство, а некоторая синтетическая ячейка. В этом случае на верхний уровень можно будет поместить и пользовательские библиотеки, которые также необходимо отображать в представлении. Удобно было бы предусмотреть случай, когда для одного устройства все другие в иерархии рассматриваются как пользовательские библиотеки. Схема предлагаемой иерархии приведена на рис. 3.2.

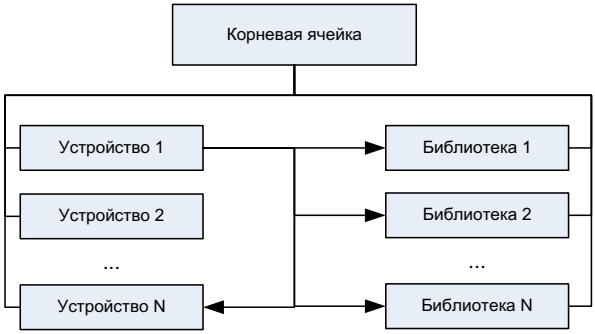

Рис. 3.2. Иерархия элементов верхнего уровня при наличии корневого элемента

3.1.3. Требования к архитектуре средства

Ключевым требованием к архитектуре является её расширяемость и встраиваемость, поэтому предлагаемая архитектура во многом повторяет принципы построения современных IDE, где реализуются схожие механизмы расширения функциональности.

Встраиваемость

Средство автоматизации реинжиниринга может быть отдельностоящим продуктом, но актуальна задача его интеграции с другими средствами. Одним из путей является его полное включение (встраивание) в некоторую среду разработки. Это позволяет при реинжиниринге использовать ресурсы самой среды, прежде всего, средства синтеза и анализа.

Возможна как жёсткая интеграция средства со средой, так и динамическое подключение (механизмы плагинов и расширений). При разработке требуется лишь обеспечить возможность подобного использования.

Наличие API

Разрабатываемое средство реинжиниринга может быть использовано в различных процессах разработки, использующих свои среды проектирования. Нужно предоставить для данных средств некий межпрограммный интерфейс (API), через который САПР смогут взаимодействовать с разрабатываемым САР.

При наличии API, возможно будет автоматизировать не только сам реинжиниринг, но и вызов данного процесса (см. пример на рис. 3.3). Например, выявив узкие места по производительности в анализаторе, можно из этого же средства вызвать САР с необходимыми параметрами, чтобы оптимизировать архитектуру.

Поддержка механизма событий

Несмотря на то, что управление преобразованиями будет производиться через API, тяжело обеспечить передачу всех необходимых результатов. Например, при отправке команды “ввести в устройство отказоустойчивую память” САР потребуется вернуть информацию обо всех изменённых элементах, после чего внешнее средство должно будет обновить свои внутренние представления.

Подобный подход кажется не очень удобным, и для решения проблемы предлагается ввести в средство реинжиниринга дополнительный интерфейс, который будет работать в направлении “САР→клиентское средство”. Через данный интерфейс внешнее средство будет извещаться об изменениях в представлении и других аналогичных событиях. Использование подобного механизма позволит разбить обработку результатов выполнения команды на

составляющие, что упростит разработку. Также станет возможным возврат информации в клиентское средство до завершения обработки, что удобно при выполнении длительных операций.

На рис. 3.3 приведён пример для случая, когда взаимодействующее с CAP средство отображает структуру внутреннего представления. При этом синхронизация двух представлений производится через механизм событий.

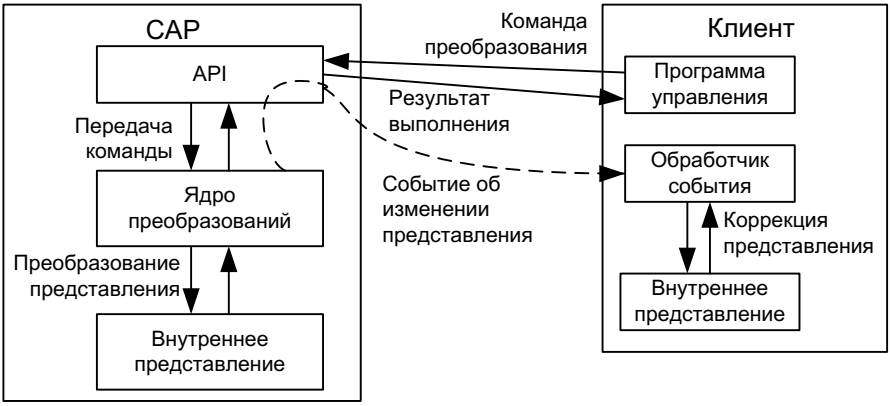

Рис. 3.3. Пример взаимодействия клиентского приложения со средством реинжиниринга через API

Поддержка пользовательских библиотек

При выполнении реинжиниринга пользователю зачастую требуется использовать в устройстве модули, изначально отсутствующие в описании устройства. Такое может быть, например, при переносе устройства между различными аппаратными платформами. Было бы удобно, если бы пользователь имел возможность группировки элементов в некоторые описания, которые в дальнейшем можно было бы повторно использовать в средстве реинжиниринга.

Подобные элементы предлагается называть пользовательскими библиотеками. Они будут ещё одним внешним представлением, поддерживаемым средством реинжиниринга.

Поддержка проектов

При формировании требований к функциональности было сказано, что должен быть режим последовательного выполнения команд. Если подобный режим используется пользователем, то для него была бы удобной возможность сохранения текущего контекста средства: внутреннего представления, подгруженных библиотек и прочего динамического окружения.

Перечисленные вещи невозможно просто экспортировать во внешнее представление (например, в исходные коды HDL), так как многие вещи специализированного внутреннего представления не могут быть отражены на внешнее. Примером является наследование. Кроме того, пользователю может потребоваться сохранить внутренний контекст в незавершённом состоянии, когда внутреннее представление не проходит валидацию. Подобные вещи экспортировать куда-либо также невозможно.

Исходя из перечисленного, было бы удобно в средстве реинжиниринга организовать экспорт контекста в некий формат, который можно было бы легко экспортировать и импортировать. Данный формат, по аналогии с контекстами IDE, предлагается называть “проектами”.

3.1.4. Требования к языку управления преобразованиями

Язык управления преобразованиями (система команд) является основным интерфейсом к внутреннему представлению устройства. Через него будут осуществляться все операции преобразований, поэтому данный механизм должен быть детально проработан на этапе разработки.

Поскольку язык управления преобразованиями тесно связан с методиками представления устройства, то на данном этапе можно выделить лишь основные требования к нему:

- функциональная полнота преобразований;
- возможность добавления пользовательских команд;
- возможность реализации полного цикла реинжиниринга;
- поддержка внешнего языка программирования;
- возможность смены контекста.

Функциональная полнота преобразований

Под функциональной полнотой СПР понимается возможность трансформации представления архитектуры устройства из любого начального состояния в любое другое посредством операций из стандартного набора СПР. Дополнительным требованием может быть возможность построения внутреннего представления “с нуля” без импорта представления. В этом случае СПР можно рассматривать и как средство разработки.

Пользовательские команды

При проведении программируемой трансформации требуется реализация пользовательских алгоритмов. Логично отражать их на систему команд средства, чтобы разработанные алгоритмы можно было повторно использовать в новых разработках. Для этого в средстве потребуется реализовать механизм, которые будут добавлять команды из пользовательских библиотек в общую систему команд.

Автоматическое выполнение преобразований

Для построения автоматического средства реинжиниринга недостаточно функциональной полноты преобразований. Требуется, чтобы можно было автоматизировать все этапов реинжиниринга: извлечение архитектуры, анализ, реализацию новой архитектуры, синтез. Эти задачи могут решаться не только средством реинжиниринга, но и сторонними средствами, поэтому в системе команд требуется предусмотреть возможность произвольных операций. Реализация подобных команд, естественно, ложится на пользователя.

Смена контекста

При переходе между операциями может потребоваться смена контекста. Например, при вызове функции может вестись преобразование только определённой области представления (например, одного модуля) и доступ к другим элементам должен быть ограничен. В языках программирования аналогом являются области видимости переменных.

Поддержка внешнего языка программирования

В ряде случаев может оказаться, что функциональности внутреннего языка недостаточно. Несмотря на то, что при выполнении требований по функциональной полноте можно реализовать что угодно, удобнее использовать уже разработанные алгоритмы на других языках программирования.

Было бы полезно интегрировать внутренний язык с каким-либо из стандартных языков программирования. Наиболее интересным подходом является использование скриптовых языков (Tcl, Perl, Python, Bash-скрипты) с добавлением интерпретатора в само средство. Существует много открытых реализаций подобных интерпретаторов, поэтому перед разработчиком будет стоять лишь задача интеграции данного модуля, что можно сделать через планируемый механизм расширений.

3.2. Разработка языка представления устройства

3.2.1. Общая концепция языка представления

Язык представления устройства является одним из основных компонентов САР, так как на его основании строятся архитектура средства и языки управления преобразованиями.

Прежде всего, требуется выбрать, на какой язык будет опираться внутреннее представление, т.е. какой синтаксис будет взят за основу. Решено взять за основу VHDL, являющийся одним из наиболее популярных языков описания устройства.

Предлагаемый язык представления устройства основан на архитектурном графе. При этом строится основная древовидная иерархия, которую предлагается называть деревом элементов. Для связи элементов иерархии между собой используются дополнительные механизмы, которые будут описаны в последующих пунктах.

В соответствии с разработанными в п. 3.1.2 требованиями, при разработке представления требуется выполнить следующее:

- определить базовые типы ячеек в дереве элементов и их иерархию;
- разработать механизмы связи элементов представления вне иерархии (наследование, ссылки и пр.);
- разработать механизмы параметризации элементов дерева;
- разработать механизмы группировки элементов;
- определить механизмы навигации по дереву элементов.

Кроме разработки самого языка представления, нужно сформировать подходы по его извлечению из входных данных. Для формирования полноценного представления необходимо знать входные форматы данных, что в случае универсального средства невозможно. Поэтому, будет описан порядок формирования дерева. При этом будем полагать, что на вход подаётся структурное описание, из которого возможно извлечь иерархию устройства.

Задачи вывода представления не рассматриваются, так как подобная задача реализуется пользовательскими алгоритмами для требуемых ему форматов.

3.2.2. Дерево элементов

На рис. 3.4 приведена структура элементов внутреннего представления с указанием возможных включений и ссылок между различными типами ячеек. Части, связанные со структурным описанием устройства во многом наследованы от VHDL. В таблице 3.1 сопоставлены элементы внутреннего представления и представления в VHDL.

Подробное описание элементов приведено далее в тексте. Основными контейнерами сигналов в иерархии являются блоки и библиотеки. Первые являются элементами описания устройства (его модулями), вторые же реализуют группировку элементов для работы с ними.

Далее в тексте приведена краткая информация о различных типах ячеек. Описываются механизмы использования данных ячеек, затрагиваются вопросы их параметров и механизмов верификации.

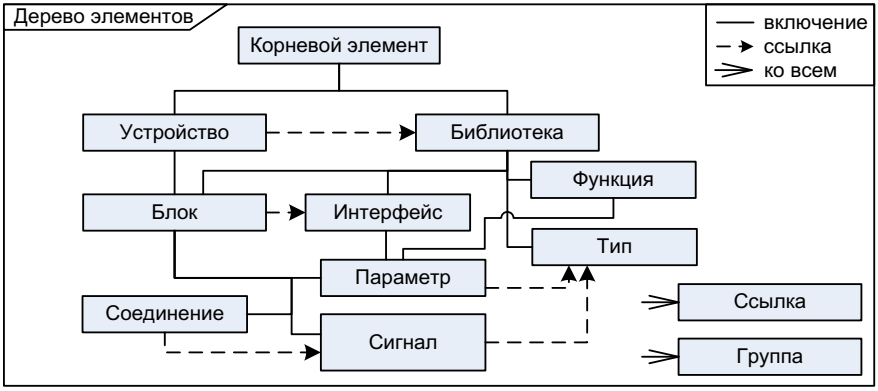

Рис. 3.4. Структура дерева элементов в разработанном представлении устройства

Таблица 3.1

Таблица соответствия элементов описания в VHDL и внутреннем представлении

VHDL	Внутреннее представление	Комментарий
Entity	Интерфейс	
Architecture	Устройство, Блок	
Component	Блок	
Package, Library	Библиотека	
Port, Generic, Signal, Const Value	Сигнал	Используется внутренняя типизация
Type, Subtype	Тип	Use реализован через ссылки
Array	Группа	std_logic_vector является группой
Assigment	Соединение	
Function, Operator	Функция	
Поведенческое описание	Отсутствует	М.б. реализовано через функции

Корневой элемент (Root)

Данный элемент используется для определения списка библиотек и устройств в рамках контекста. Данная ячейка не имеет никакой функциональности, не синтезируется в выходные представления, и её параметры не могут быть модифицированы командами преобразования.

Устройство (Device)

Данный элемент представляет устройство. Устройство является синтетическим компонентом для выделения верхних уровней иерархии. Данный объект может быть размещён только в корневом элементе дерева, что позволяет

одновременно работать с множеством устройств. Устройство может включать следующие элементы:

- корневой блок root;
- внутреннюю библиотеку;
- список ссылок на используемые внешние библиотеки.

Корневой блок является элементом типа Block (см. далее) и реализует всю внутреннюю архитектуру устройства. Подобная организация позволяет работать с устройством как с обычным блоком.

Имя внутренней библиотеки совпадает с именем устройства. В ней определяются все блоки и интерфейсы устройства, которые загружаются при импорте устройства и не описаны во внешних библиотеках.

Интерфейсы (Interface)

Интерфейсы определяют внешнее представление блоков, устройств и функций. В интерфейсы входят описания портов и параметров (подтипы сигналов). Данный элемент является аналогом Entity в VHDL, но в разработанном представлении допускается множество синтезируемых реализаций интерфейса (в VHDL - одна), что позволяет в определённых рамках использовать понятие интерфейса из объектно-ориентированного программирования.

Выбранный подход позволяет легко заменять одни реализации модулей другими, если у них совпадают внешние интерфейсы. Также предполагается организация наследования интерфейсов через расширение списка внешних портов.

Блоки (Block)

Блоки являются основным элементом иерархии устройства. Каждый блок включает ссылку на интерфейс, который определяет его внешние связи, и некоторый набор компонентов, описывающих его внутреннюю структуру. Блок может включать следующие элементы:

- блоки нижнего уровня;
- сигналы, порты, generic-параметры;
- описания соединений между элементами.

Если сравнивать с VHDL, то сам блок реализует архитектуру, а ссылки на него – объявления компонентов.

Библиотеки (Library)

Библиотеки используются для хранения компонентов, используемых в дереве элементов. Библиотека может включать любые элементы, кроме соединений. Возможно построение иерархии библиотек, что может быть

использовано для построения вложенных пространств имён (VHDL поддерживает подобный механизм).

Доступ к элементам библиотеки идёт через механизм ссылок (см. далее). При этом изменение элемента в библиотеке должно автоматически приводить и к изменению его ссылки.

Сигналы (Signal)

Ячейки сигналов в дереве элементов определяют все элементы, которые могут быть присвоены некоторые значения. К данной группе относятся как элементы физических сигналов, так и параметры блоков и функций. Для разделения типов введены подтипы сигналов, которые описаны в таблице 3.2.

Таблица 3.2

Подтипы ячейки сигналов в дереве элементов

Тип	Реализуемые элементы VHDL	Возможность соединения*			
		V	P	G	C
Variable	- сигналы в архитектуре; - внутренние переменные функций.	+	+	-	-
Port	- порты сущностей; - параметры и возвращаемые значения функций;	+	-	-	-
Generic	- generic-параметры Entity;	+	+	+	-
Constant	Константы, значения по-умолчанию	+	+	+	-
*Variable, Port, Generic и Constant соответственно. “+” ставится, если возможно соединение сигнала с типом из горизонтальной строки к типу из вертикальной					

В большинстве HDL используется типизация сигналов, поэтому её требуется реализовать и в разрабатываемом представлении. Для этого в дерево элементов введён отдельный тип ячеек (см. далее).

Каждый сигнал обладает следующими параметрами:
- имя;
- подтип ячейки;
- тип сигнала;
- флаг поддержки двунаправленных соединений (BIDIR).

Соединения (Connection)

Соединения используются для организации связи между сигналами, т.е. они связывают элементы иерархии устройства. Каждое соединение содержит ссылки на входной и выходной сигналы. Реализация двунаправленных соединения (соединения BIDIR-сигналов в VHDL) возможна при помощи специального параметра.

При верификации соединений проверяется корректность направлений соединений (см. табл. 3.2), а также соответствие типов сигналов. Двунаправленные соединения допускаются только между портами и переменными, причём требуется, чтобы в параметрах порта был установлен флаг поддержки BIDIR-сигналов.

Функции (Function)

Под функциями в дереве элементов понимается любая сущность, обладающая входными и выходными сигналами, в которой описывается связь вход-выход. Из языка VHDL к рассматриваемому классу ячеек относятся функции, операторы, обращения к массивам через индексы и прочее.

В настоящей архитектуре не предполагается дополнительных средств для модификации внутреннего содержания функций. Всё внутреннее содержание представляется в виде параметра дерева.

Ссылки (Reference)

Ссылки не являются отдельным типом в дереве элементов, а используются для организации связи между элементами и наследования архитектуры через расширение. Механизм ссылок подробно описан в пункте 3.2.3.

Типы сигналов (Type)

В большинстве HDL существует типизация сигналов. При этом, типы могут быть как описаниями физических сигналов (std_logic, bit, и т.д.), так и объявлениями массивов, специфическими типами (integer, boolean), перечислениями или целыми структурами.

Список поддерживаемых типов варьируется в зависимости от HDL, а в некоторых из них пользователь может вводить свои типы. Поэтому, внутреннее представление должно поддерживать типы как отдельный элемент описания. Для этого вводится дополнительная ячейка, которая может быть размещена внутри библиотеки.

Ячейка типа является составной, т.е. его описание набирается из других ячеек типов и параметров. Данный механизм во многом приближен к синтаксическому дереву, так как при отсутствии чётко выраженных границ сложности типов невозможно сформировать структуру ячейки. Для простых типов в рамках нетлистов, языков VHDL и Verilog предлагаемого подхода достаточно.

Группы и агрегации (Group, Aggregation)

Данные типы ячеек были введены для реализации механизма группировки, который описан в пункте 3.2.4. В предлагаемом представлении группы и агрегации представляют собой некоторый массив ссылок на элементы с возможностью индексного доступа.

Группы реализуют шины сигналов и объединяют однотипные элементы (например, триггеры могут быть объединены в некоторое подобие регистра). Агрегация является особым типом группы. Прежде всего, она отражает временное объединение элементов для выполнения какой-либо операции. Примером могут быть агрегации сигналов в языке VHDL.

3.2.3. Механизм ссылок

Часто бывает, что при реинжиниринге в устройстве нужно хранить сразу несколько экземпляров одной ячейки. Например, в устройстве может быть несколько одинаковых блоков (регистры, логические вентили, и т.д.). Имеет смысл хранить только одну копию подобных объектов, а для всех остальных указывать, что они являются подобием другого объекта в дереве.

Предложенный выше механизм предлагается называть ссылками на элементы. При помощи него предлагается реализовать все ссылки вне древовидной иерархии, реализуя горизонтальные связи архитектурного графа.

Ссылка полностью повторяет внешнее поведение ячейки, на которую ссылается (в т.ч. принимает тип данной ячейки). Таким образом, ссылки становятся прозрачными для внутреннего представления и команд трансформации устройства, что удобно при считывании и анализе, но требует дополнительного внимания. Можно выделить два способа использования ссылок:

- создание связанной копии объекта;
- указание на использование другой ячейки.

В дереве элементов возможно создание полной копии ячейки (внутренние описания совпадают), но при этом разрывается связь с исходной ячейкой. Это может быть удобно при последующей модификации одной из частей, но в других случаях удобно взять ссылку. За счёт её использования сохранится связь с исходным элементом, и при его изменении автоматически обновятся и все связанные копии (ссылки). Выбор между двумя механизмами копирования остаётся за пользователем.

Ссылки также используются для указания зависимостей между ячейками дерева элементов. Например, через ссылку указывается тип сигнала для ячейки сигнала. В таблице 3.3 приведены некоторые примеры использования ссылок в ячейках иерархии устройства.

Таблица 3.3

Использование ссылок в различных ячейках дерева элементов

Ячейка дерева	Использование
Устройство	ссылки на используемые библиотеки
Блок	ссылка на реализуемый интерфейс
Интерфейс	указание на расширяемый интерфейс
Библиотека	ссылки на используемые библиотеки
Сигнал	указание типов сигнала
Соединение	ссылки на входной и выходной сигналы
Функция	ссылка на реализуемый интерфейс
Тип	ссылки на используемые типы нижнего уровня
Группа	ссылки на содержимое элементы

3.2.4. Механизм параметров

Для каждой ячейки дерева элементов требуется набор параметров, которые позволяют описать некоторые её свойства. Каждый параметр должен обладать, как минимум, именем и значением. Предлагается следующий подход к построению системы параметров:

- реализуется древовидная иерархия параметров;
- в рамках одного уровня имя параметра уникально;
- ведётся верификация значений параметров:
- поддерживаются пользовательские параметры;
- значения параметров являются строковыми выражениями.

Параметры могут быть двух типов: встроенные и дополнительные. Под встроенными параметрами понимаются те, что отражают положение ячейки во внутреннем представлении, генерируются и обновляются автоматически. К ним относятся имя, тип ячейки, ссылка на родительскую ячейку и т.п.

Все остальные параметры элементов дерева являются дополнительными. При необходимости они создаются, изменяются и используются в алгоритмах трансформации архитектуры.

3.2.5. Механизм группировки элементов

В ряде случаев может потребоваться группировка элементов. Примером может быть объединение сигналов в шину или же объединение однотипных блоков (например, триггеров памяти в регистр). Поэтому, для удобства работы с деревом элементов было решено ввести тип ячеек “группа”.

Группа является отдельной ячейкой дерева и хранит внутри себя ссылки на элементы, в неё входящие. Сами ячейки в группу не добавляются для того,

чтобы имелась возможность включить одну и ту же ячейку в несколько различных групп.

В настоящей реализации группы представляют индексированные наборы элементов. Это позволяет включать в них одноимённые элементы, что не поддерживается для прочих типов ячеек.

3.2.6. Механизмы навигации по дереву элементов

В разработанном представлении требуется обеспечить адресацию к элементам структурного описания и параметрам ячеек. Поскольку параметры определены только в рамках ячейки и не имеют внешних связей, решено выделить доступ к параметру в отдельный механизм адресации, который работает в пределах ячейки. В итоге появляется два механизма адресации к элементам.

Именование ячеек

В представлении ячейки обладают именем, типом и индексом положения в группе (если ячейка включена в группу). Триада данных параметров уникальна в пределах ячеек одного уровня иерархии и может быть использована в качестве уникального ключа, который позволяет произвести доступ к ячейке. В текстовом виде ключ предлагается записывать следующим образом:

[идентификатор_типа]имя_ячейки[индекс_в_группе] *(3.1)*

В качестве идентификатора типа предлагается использовать не имя типа, а односимвольный маркер. В таблице 3.4 приведён список выбранных маркеров для ячеек различных типов. При записи имени допускается ряд исключений:

- если ячейка не входит в группу, то индекс можно не указывать;
- если идентификатор не указан, то ячейка считается блоком;
- если после имени ячейки следует двоеточие, то ячейка считается устройством.

В имени ячейки запрещено использование пробелов и специальных символов, которые перечислены в таблице 3.5.

Таблица 3.4

Соответствие маркеров имён типам ячеек

Индекс	Тип	Индекс	Тип
R	Корневой элемент	I	Интерфейс
D	Устройство	F	Функция
B	Блок	l	Библиотека
C	Соединение	g	Группа
S	Сигнал	a	Агрегация
T	Тип		

Адресация к ячейкам

Поддерживается три механизма адресации к ячейкам дерева элементов: прямая и косвенная через путь, а также прямая через идентификатор (см. далее). Пути к элементам описываются в рамках древовидной иерархии дерева элементов. Каждый уровень пути именуется в соответствии с форматом (3.1)

При прямой адресации путь к ячейке записывается относительно корневого элемента или устройства. В случае косвенной адресации в начале пути ставится метка ‘.’ или ‘..’. Расчёт пути при этом начинается от элемента, который в настоящее время выбран в контексте.

Допускается использование меток косвенной адресации в любом количестве и месте пути. Следует помнить, что проверка типов для данных элементов может быть произведена лишь динамически во время обработки пути. При попытке перехода выше корневого элемента метки пути игнорируются.

Таблица 3.5

Специальные метки при навигации по дереву элементов

Модуль	Описание
.	Текущий элемент
..	Родительский элемент
:	Устройство
[x]	Индекс группы, если указан в конце имени Тип ячейки, если указан в начале имени
/ , \	Разделители уровней. Если он указан в начале пути, то адресация считается абсолютной
~	Начало модификаторов имени*
_	Разделитель модификаторов имени*
@	Символ доступа по идентификатору
* - Допускается использование символов в имени устройства	

Адресация к параметрам

В предлагаемом представлении параметры ячеек обладают иерархической структурой, поэтому и доступ к параметру должен производиться через некоторый путь. Отсутствует нужда в косвенной адресации к параметрам или их типам, поэтому путь к параметру можно записать в виде иерархии

Адресация по идентификатору

При использовании данного механизма адресации каждой ячейке в представлении присваивается уникальный номер, по которому к ней можно обратиться. Желательно, чтобы при переносе ячейки её индекс сохранялся, а при

удалении – не занимался другими ячейками. В этом случае индексы можно хранить отдельно, например, для отложенной обработки или ускорения доступа. Примеры записи пути для всех типов адресации приведены в таблице 3.6.

Таблица 3.6

Примеры адресации к ячейкам и параметрам дерева элементов

Тип адресации	Примеры
Абсолютная адресация	\[I]basic\[I]not\[S]in\ test:\clk~clkctrl\[S]outclk\ [D]test\[B]clk~clkctrl\[S]outclk\
Относительная адресация	. .. .\ clk~clkctrl\[S]outclk\ ..\ clk~clkctrl\[S]outclk\ ..\..\ [S]outclk\ ..\clk~clkctrl\..\ clk~clkctrl..\clk~clkctrl\[S]outclk
Адресация по идентификатору	@11235813 @42
Адресация к параметрам	name ports/clk/type

3.2.7. Методика построения внутреннего представления

Разработанная методика представления достаточно сложна, поэтому задача формирования внутреннего представления на основе внешних описаний требует дополнительного рассмотрения.

Предположим, что представление строится на основе структурного описания, из которого возможно извлечь иерархию устройства или набор элементов для библиотеки. Предлагаемый порядок построения для данного случая приведён на рис. 3.5.

Предлагаемые алгоритмы предполагают последовательное считывание иерархии с последовательной её детализацией. Это сделано для того, чтобы исключить возникновение ситуации, когда некоторые исходные данные не готовы к добавлению в иерархию (например, для соединения считан только выходной сигнал, или интерфейс для считываемого компонента ещё не загружен в библиотеку).

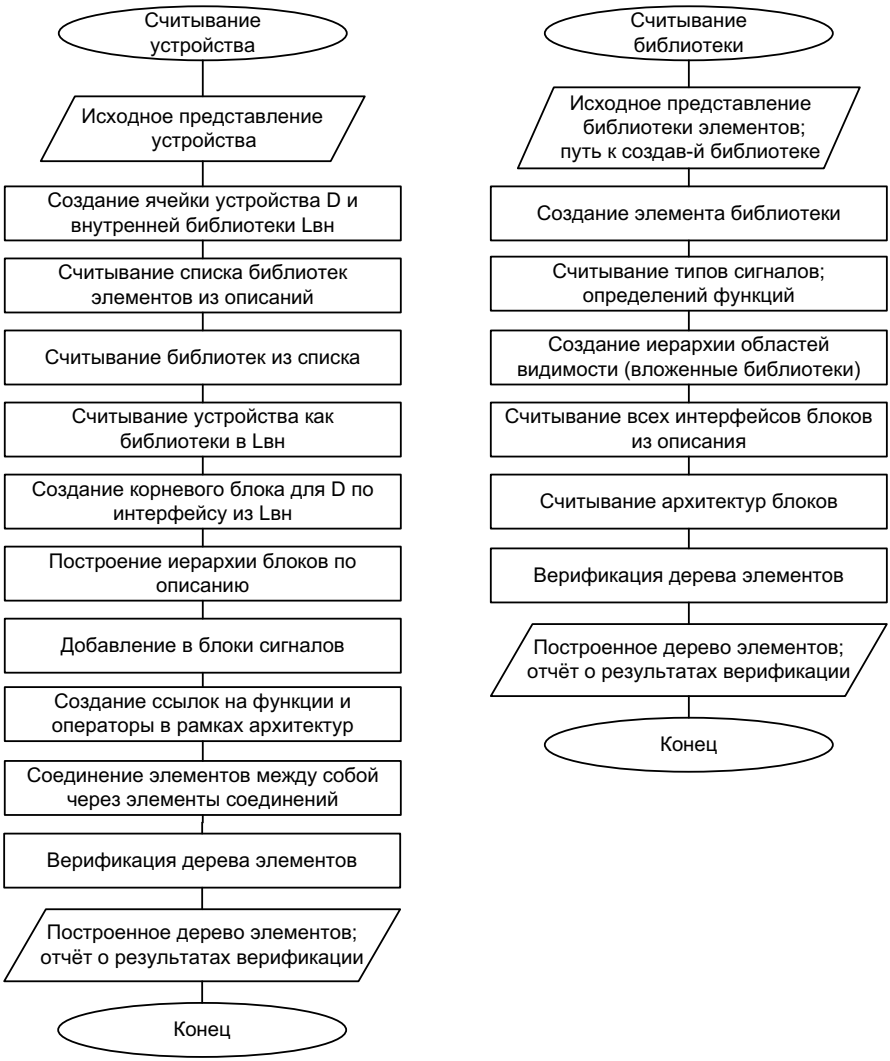

Рис. 3.5. Рекомендуемый порядок считывания представления устройств и библиотек из внешнего структурного описания

В предлагаемых алгоритмах не указана обработка ошибок считывания, которые могут возникать на любом этапе алгоритма. В данном случае есть четыре варианта решения, выбор между которыми остаётся за разработчиком модулей импорта:

- переход к следующим этапам импорта, чтобы максимально считать представление;
- остановка считывания и возвращение частичной архитектуры;
- отмена модификаций дерева элементов, сделанных при выполнении операции считывания;
- попытка исправления дерева элементов путём исправления ошибки (например, создание несуществующего интерфейса).

Выбранная реализация алгоритмов может потребовать нескольких проходов по синтаксическому дереву считываемого представления. Для сложных устройств это может быть достаточно длительным процессом, поэтому могут быть разработаны алгоритмы с однократным проходом. Это приведёт к росту сложности реализации алгоритмов, но конечный выбор остаётся за разработчиком модулей ввода-вывода.

3.3. Разработка языка трансформации устройства

3.3.1. Формирование системы команд языка

Независимо от интерфейса доступа к менеджеру команд ядра и языка программирования средства, в ядре требуется реализовать базовые команды, которые обеспечат функциональную полноту преобразования в разработанном дереве. Можно выделить следующие группы команд:

- команды модификации дерева ячеек;
- команды получения параметров ячеек;
- команды навигации по дереву;
- команды ввода-вывода;
- команды управления средством реинжиниринга.

Целесообразно выделить минимальный набор операций, без которого проектируемое средство не сможет функционировать. Все перечисленные команды потребуется реализовать в планируемом прототипе средства реинжиниринга. Логично сделать для этого библиотеку ядра, которая будет подгружаться при запуске средства реинжиниринга.

Команды модификации дерева элементов

Минимальный набор включает следующие команды:

- создание новой ячейки;
- создание ссылки на ячейку;
- удаление ячейки;
- копирование ячейки;
- изменение значения параметра.

Команды получения информации об ячейках

В соответствии с требованиями, должна быть возможность получения всей информации из дерева элементов. Для этого в CAP должны быть реализованы следующие команды:

- вывод типа ячейки (в т.ч. выявление ссылки);
- вывод списка ячеек нижнего уровня;
- вывод списка параметров ячейки;
- получение значения параметра.

Команды навигации по дереву элементов

Должны быть следующие команды:

- переход к указанной ячейке;
- вывод пути к текущей ячейке.

Команды ввода-вывода представления

Средство должно обеспечивать работу с внешними описаниями устройства и последующий вывод результатов преобразования. Как минимум, требуется обеспечить ввод-вывод на уровне устройства, т.е. полное считывание и полную запись архитектуры. Кроме того, нужно обеспечить загрузку пользовательских библиотек элементов.

Также было бы удобно иметь возможность вывода отдельных модулей. Например, ячейку “Блок” вполне можно вывести во внешнее представление как полноценное устройство, если она содержит внутреннюю реализацию, а не ссылается на библиотечный элемент.

Команды управления средством реинжиниринга

В соответствии с разработанными требованиями, в CAP должна быть возможность управления пользовательскими библиотеками, вызовами программ обработки и прочими пользовательскими функциями. Предлагается все эти функции отразить на систему команд, чтобы обеспечить программное управление не только преобразованиями, но и самим средством.

Данную группу предлагается называть системными командами. Они предоставляют доступ функциям к возможностям ядра, т.е. реализуют шлюзы между пользовательскими командами и внутренней функциональностью ядра. Естественно, расширение подобных команд невозможно, поэтому набор команд данного типа должен быть детально проработан на этапе проектирования системы. В рамках существующих требований необходимо реализовать следующие команды:

- запуск программы обработки;
- вывод списка доступных библиотек;
- загрузка пользовательской библиотеки;

- вывод истории команд (в виде программы обработки);
- отмена последней выполненной команды;
- сохранение и загрузка проекта;
- завершение преобразований (выход).

Дополнительные команды

Команды, перечисленные в предыдущих пунктах, являются минимально необходимым набором для решения задач любой сложности. Тем не менее, многие задачи можно решить более просто, введя дополнительные команды. Список данных команд приведён ниже:

- вывод списка доступных команд;
- вывод справки по команде;
- повтор отменённой команды;
- перенос ячейки в указанное место;
- вывод списка ссылок на ячейку (не надо собирать ссылки по всему дереву).

Отдельный класс составляют команды модификации дерева элементов, которых можно придумать сколько угодно. Тем не менее, базовый набор команд предполагается сделать минимальным, так как его всегда можно расширить за счёт пользовательских библиотек.

3.3.2. Внутренний язык управления средством

Разрабатываемое средство ориентировано на программную реализацию алгоритмов обработки. Тем не менее, для самого средства была бы полезна некоторая нотация, описывающая отдельные команды: вызов операции, передачу ей параметров и присвоение результатов.

Подобный язык удобен и для практического использования в средстве: с его помощью можно реализовать интерактивный режим преобразований, отладку алгоритмов, хранение истории команд и многие другие операции.

Предлагается использовать синтаксис языка, близкий к одному из скриптовых языков программирования. Например, при разработке прототипа (см. раздел 4) был использован синтаксис, схожий с вызовами в командной оболочке Unix. Формат вызовов приведён ниже:

имя_команды [опции] [путь_к_ячейке_дерева] [>приёмник] *(3.2)*

В перечисленных выше применениях внутреннего языка достаточно программы в виде последовательностей вызовов команд. Реализация более сложных алгоритмов (ветвления, циклы, и т.д.) остаются на внешнем языке программирования. В приложении G приведены примеры скриптовых программ на внутреннем языке программирования прототипа.

3.4. Разработка архитектуры расширяемого программного средства

В пункте 3.1.3 были сформулированы общие требования к архитектуре расширяемого средства. После была разработана методика представления устройства и предложен язык управления средством. Теперь требуется построить архитектуру средства, которая бы поддерживало разработанные методики и в то же время соответствовало бы поставленным требованиям.

При разработке будем полагать, что САР будет реализовано на объектно-ориентированном языке программирования, что позволит упростить расширение архитектуры за счёт использования средств, предоставляемых самим языком программирования.

3.4.1. Общая концепция

Для построения САР предлагается модульный подход. При этом формируется некоторое “ядро”, которое включает минимально необходимую функциональность средства и динамический контекст преобразования. Всё остальное выносится в дополнительные модули-расширения, которые могут загружаться ядром в случае необходимости.

Ядро и расширения вместе образуют средство реинжиниринга, которое, в свою очередь, может быть использовано в пользовательских целях: интегрировано с САПР, адаптировано для специфических задач и т.п. На рис. 3.6 приведена схема средства реинжиниринга с указанием основных его составляющих. Подробно все перечисленные модули будут рассмотрены далее.

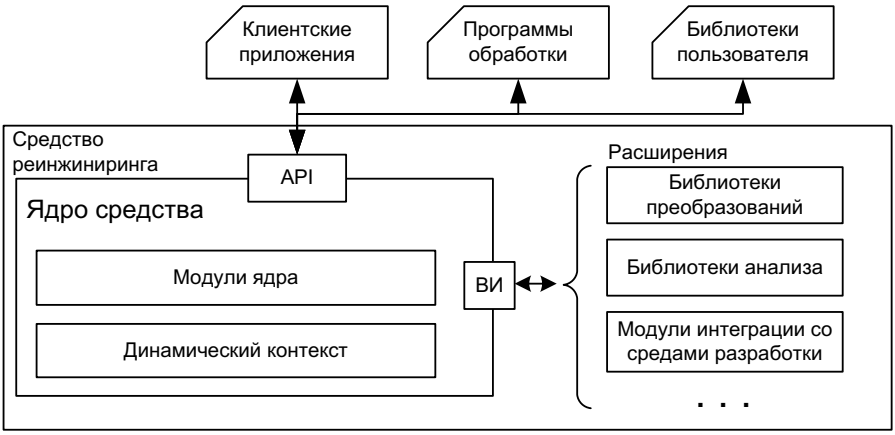

Рис. 3.6. Концепция построения средства реинжиниринга

Контекст средства

Контекст CAP хранит всю динамическую информацию, которая используется при выполнении задач реинжиниринга. В него входят как внутренние представления устройств, так и пользовательские элементы.

Контекст средства реинжиниринга включает следующие элементы:
- внутреннее представление устройства;
- информация о загруженных библиотеках и расширениях;
- списки обработчиков событий;
- указатели на текущее положение в дереве (для косвенной адресации к элементам).

Для управления элементами контекста будут использоваться специальные модули, называемые менеджерами.

Пользовательские расширения

Под пользовательскими расширениями понимаются любые модули, которые подключаются через внутренний интерфейс (ВИ). Само ядро, кроме интерфейса, содержит механизмы для загрузки модулей и организации их взаимодействия. Чтобы исключить конфликты с библиотеками элементов, предлагается называть данные расширения библиотеками ядра. Подробнее механизмы расширений рассмотрены в 3.4.6.

3.4.2. Ядро средства

Ядро CAP является основной частью, через которую связываются все прочие составляющие системы. Данный модуль, с одной стороны, должен быть максимально компактен, но при этом он должен поддерживать все возможности, описанные в 3.1.3. Можно выделить следующие основные элементы, которые должны присутствовать в ядре CAP:
- средство хранения контекста преобразования;
- API для подключения пользовательских библиотек
- механизм генерации событий;
- библиотека системных команд для управления ядром.

На рис. 3.7 приведена примерная схема взаимодействия модулей ядра с внешними элементами. Для упрощения контроля над преобразованиями предлагается весь доступ к контексту преобразования производить через группу менеджеров отдельных составляющих, вместе называемых “исполняемой средой”.

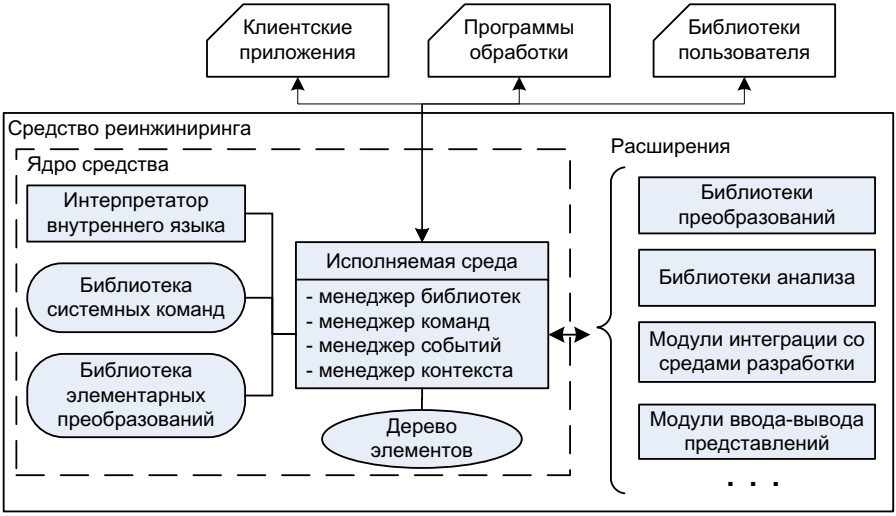

Рис. 3.7. Ядро средства реинжиниринга

Кроме необходимых элементов, в ядро средства решено добавить ещё два элемента: интерпретатор команд внутреннего языка и библиотеку элементарных преобразований. Основанием к этому стала необходимость использования данных модулей в большинстве задач реинжиниринга. Далее мы рассмотрим общие подходы к построению каждого из элементов ядра.

Дерево элементов

Дерево элементов является основной составляющей контекста программного средства. Данная группа элементов реализует представление в виде архитектурного графа в соответствии с языком представления, разработанном в пункте. 3.2.

Дерево элементов представляет собой иерархию связей разнотипных ячеек. Каждая ячейка представляет собой отдельный объект и содержит следующие данные:

- имя, тип, уникальный идентификатор;
- контейнер внутренних ячеек;
- древовидный контейнер параметров;
- список ссылок на ячейку.

В дерево элементов входит и функциональность по модификации представления, на основе которой строятся команды языка управления преобразованиями. В конечном счёте, данный модуль можно реализовать как отдельную библиотеку, в которой определено представление средства и базовые методы его преобразования. Само ядро при этом можно рассматривать как

расширение данного представления для решения задачи автоматизации преобразований.

Команды преобразований

Работа ядра средства реинжиниринга основана на передаче команд между его модулями. С учётом требований и архитектуры, объект команды в средстве должен включать следующее:

- реализацию алгоритма трансформации устройства;
- описание параметров команды;
- описание правил разбора командной строки;
- алгоритмы отмены выполнения команды;
- методы проверки параметров;
- справочную информацию для генерации отчётов.

Под командами в средстве понимаются самые разные группы операций, которые оказывать различное влияние на архитектуру или не оказывать его вовсе. Средство реинжиниринга может не иметь возможности проанализировать алгоритмы и оценить их влияние, поэтому предлагается ввести в описание команды следующие флаги:

- команда модифицирует дерево элементов;
- команда может быть отменена;
- команду требуется фиксировать в истории команд.

Менеджеры исполняемой среды

Менеджеры исполняемой среды используются для организации взаимодействия между командами из библиотек ядра и деревом элементов. Также они представляют шлюз к внутренней функциональности для внешних приложений (через API). Менеджеры ядра тесно связаны друг с другом, и разделение идёт по решаемым задачам. Сами задачи будут рассмотрены далее.

Интерфейсы ядра

В архитектуре средства реинжиниринга предусмотрено два типа интерфейсов:

- межпрограммные интерфейсы (API);
- внутренние интерфейсы для расширений.

Межпрограммные интерфейсы, в соответствии со сформированными требованиями (см. пункт 3.1.3), должны обеспечить передачу команд в CAP и возвращение результатов. Кроме этого, должен присутствовать механизм для оповещения внешнего средства о возникновении событий. Внутренние интерфейсы используются для предоставления расширенного доступа к функциональности библиотекам ядра.

Основной задачей интерфейсов ядра является предоставление контролируемого доступа к менеджерам исполняемой среды, которые обеспечивают исполнение передаваемых команд.

Интерпретатор внутреннего языка управления преобразованиями

Интерпретатор команд является дополнительным модулем, который позволяет считывать и исполнять программы, реализованные на внутреннем языке. Модуль решает следующие задачи:

- разбор команды (определение имени команды, подготовка параметров);
- формирование вызова менеджера команд для исполнения;
- последовательное исполнение группы команд (например, при исполнении скрипта).

Основным элементом модуля, используемым в интерпретаторе, является конфигурируемый парсер команды, который должен провести разбор и анализ командной строки с учётом того, что каждая команда имеет свои опции и аргументы. Модуль парсера, в частности, используется менеджером команд при исполнении своих вызовов.

3.4.3. Хранение контекста

В требованиях к средству реинжиниринга предусмотрена навигация по дереву элементов с хранением текущей ячейки для косвенной адресации, хранение истории команд. Имеет смысл выделить задачи управления контекстом в специальный модуль, называемый менеджером контекста.

Менеджер контекста исполняет следующие задачи:

- хранение ссылки на текущую выбранную ячейку;
- хранение истории переходов между ячейками;
- хранение и обновление истории команд;
- выявление изменений в дереве элементов и генерация сообщений.

Менеджер контекста необходим для формирования событий об изменении дерева, так как в самих командах обработки может быть недостаточно данных для формирования отчёта. В итоге, при изменении элемента сначала вызывается менеджер контекста, который передаёт всю необходимую информацию в менеджер событий.

3.4.4. Выполнения команд в ядре

Всякая программа управления преобразованиями устройства разбивается на некоторую последовательность вызовов, запускающих те или иные команды трансформации. В ядре средства должен быть модуль, с помощью которого осуществляется управление исполнением программ. Этот модуль предлагается называть менеджером команд. В его задачи входят:

- хранение списка доступных команд и описаний их параметров;
- проверка поступающих команд на исполнимость (существование обработчика команды, корректность параметров);
- вызов обработчика команды;
- контроль порядка исполнения команд и исключение конфликтов;
- возвращение результатов выполнения команды;
- генерация отчётов об ошибках.

При хранении команд предлагается определять их имена в менеджере команд, а не в самих обработчиках. Это позволит при необходимости изменить имя команды, чтобы избежать конфликта имён между двумя командами из различных библиотек (например, заменить конфликтные команды A на lib1.A и lib2.A). Также возможно привязать одну команду сразу к нескольким вызовам.

Перед выполнением любой команды менеджер команд при помощи парсера извлекает имя команды, после чего загружает её описание из своей базы. Менеджером производится извлечение и проверка параметров команды, после чего запускается функция преобразования. Результаты выполнения команды возвращаются наружу.

Загрузка списка команд в менеджер команд производится при инициализации ядра или же менеджером расширений при добавлении новой библиотеки.

3.4.5. Механизм событий

Механизм событий должен информировать внешние модули обо всех изменениях в контексте устройства, чтобы те могли адаптировать своё представление под текущий контекст ядра реинжиниринга (см. пример на рис. 3.3). Можно выделить следующие группы событий:

- изменение структуры дерева элементов;
- изменения параметров у ячейки дерева элементов;
- изменение выбранного элемента;
- загрузка и отключение расширений;
- возникновение исключительных ситуаций в ядре.

В таблице 3.7 приведён список необходимых для CAP событий. В колонке слева приведены описания событий и их символьные идентификаторы, которые предлагается использовать в дальнейшем.

Для передачи событий в сторонние модули предлагается использовать механизм, когда сами модули предоставляют некоторый API, через который им может быть передана информация о событии. Абстрагируясь от интерфейса передачи, будем называть подобный интерфейс “обработчиком событий”.

Таблица 3.7

Список основных событий, генерируемых ядром

Имя события	Описание
NODE_ADDED	В дерево элементов добавлена новая ячейка
NODE_DELETED	Из дерева элементов удалена ячейка
NODE_CHANGED	Заменена ячейка дерева элементов
PARAM_ADDED	В ячейку добавлен новый параметр
PARAM_DELETED	Из ячейки удалён дополнительный параметр
PARAM_CHANGED	Изменено значение параметра
LIB_LOADED	Добавлена новая библиотека ядра
SEL_CHANGED	Изменён выбранный элемент для навигации
CORE_EXIT	Прекращение работы средства реинжиниринга

Для обеспечения взаимодействия с обработчиками в ядре средства предлагается реализовать специальный модуль – Менеджер событий. Данный модуль должен решать следующие задачи:

- регистрация и настройка обработчика события;
- отключение обработчика событий;
- вызов обработчика при возникновении события;
- предоставление списка текущих обработчиков.

Модули сами должны регистрировать свои обработчики, так как само ядро о них может не знать. Аналогично, должна быть возможность отключения обработчика, чтобы при изменении состояния модуля (например при отключении), не тратить системные ресурсы на обработку событий. Для данной задачи в API должна быть предоставлена соответствующая функциональность.

Число генерируемых событий при реинжиниринге велико, поэтому предлагается их каким-либо образом группировать, чтобы при возникновении конкретного события не приходилось вызывать все обработчики, а только те, которым действительно требуется данное событие.

3.4.6. Механизм расширений (библиотек ядра)

Разработка библиотек ядра является основным путём расширения функциональности CAP. Вместе с библиотекой ядра в средство могут быть загружены следующие элементы:

- новые команды обработки;
- расширения базовых ячеек дерева элементов;
- модули поддержки ввода-вывода в различные форматы представления;
- модули поддержки внешних языков программирования;
- дополнительные API для доступа к средству;
- пользовательские библиотеки элементов (в виде библиотек элементов).

Таким образом, механизм расширений является крайне важным с точки зрения встраиваемости, программируемости, расширяемости разрабатываемого средства. При реализации средства реинжиниринга он должен быть реализован в первую очередь.

Менеджер библиотек

Управление расширениями производится при помощи отдельного модуля, называемого менеджером библиотек. В данном модуле решаются следующие задачи:

- хранение списков загруженных и доступных библиотек;
- загрузка новых библиотек;
- начальная инициализация библиотек (загрузка их компонентов);
- редактирование и сохранение списка доступных библиотек.

Задача выгрузки библиотеки крайне сложна, так как дерево элементов может содержать элементы, построенные с использованием расширений, что может привести к непредсказуемым последствиям. Полезность подобного механизма сомнительна, и на текущем этапе решено, что единственным путём выключения библиотеки будет перезапуск средства с полной повторной инициализацией.

Зависимости между библиотеками

Библиотеки ядра могут зависеть друг от друга. При этом, от менеджера библиотек требуется обеспечить загрузку всех необходимых библиотек. Для реализации подобного механизма требуется реализовать:

- контроль версий библиотек;
- хранение списка ссылок на используемые библиотеки (с указанием диапазона версий);
- автоматический выбор порядка загрузки библиотек с учётом зависимостей.

Контроль версий необходим, так как библиотеки могут обновляться независимо друг от друга, в результате чего может оказаться, что загруженная по зависимости библиотека не содержит нужной команды или, что хуже, её поведение изменилось

Состав библиотеки

Исходя из вышесказанного, сущность библиотеки должна включать следующие элементы:

- идентификатор библиотеки (тип + версия);
- список используемых библиотек;
- программа инициализации библиотеки;
- пользовательское наполнение (команды, элементы, и т.д.).

3.4.7. Внутренние библиотеки ядра

В ядре средства реинжиниринга предусмотрено две внутренних библиотеки: системных команд и базовых операций. Они добавлены в ядро CAP, так как им требуется расширенный доступ к его составляющим, а использование команд из данных библиотек необходимо практически во всех операциях. Подробное описание команд приведено в приложении D.

Внутренние библиотеки подгружаются при инициализации средства реинжиниринга, так как в противном случае не будет выполняться требование функциональной полноты системы команд.

Библиотека системных команд (SystemLib)

Библиотека системных команд реализует команды управления средством реинжиниринга, которые были описаны в пункте 3.3.1. Команды данной библиотеки реализуют управление менеджерами среды и средством в целом. Поэтому, данной библиотеке требуется предоставить более широкие права доступа, чем для пользовательских библиотек. Функционирование CAP без данной библиотеки невозможно, так как через её средства подключаются другие задачи реинжиниринга.

Библиотека базовых операций (BasicLib)

Библиотека базовых преобразований также требует расширенного доступа, но уже к дереву элементов. Через команды базовой библиотеки должны осуществляться преобразования в пользовательских расширениях, сами же команды должны работать с деревом элементов напрямую.

В библиотеку базовых преобразований входят команды модификации дерева элементов и получения информации о ячейках. Также в группу добавлены команды вывода текстовых описаний устройства (см. команду ls).

3.5. Организация ввода-вывода представлений

Для работы CAP требуется обеспечить импорт и экспорт внешнего представления на одном из языков описания аппаратуры. Для различных задач реинжиниринга могут использоваться разные представления, поэтому функции ввода-вывода являются специфичными для каждой задачи и должны быть вынесены за пределы ядра средства реинжиниринга.

В предлагаемой архитектуре средства ввода-вывода будут реализованы в отдельных библиотеках ядра. При этом для организации взаимодействия будут использоваться стандартные интерфейсы ядра.

Для упрощения модулей считывания и генерации кода предлагается разделить их на две ступени. Пример такой организации приведён на рис. 3.8. На

первом этапе считывания представления генерируется АСД, который затем конвертируется во внутреннее представление.

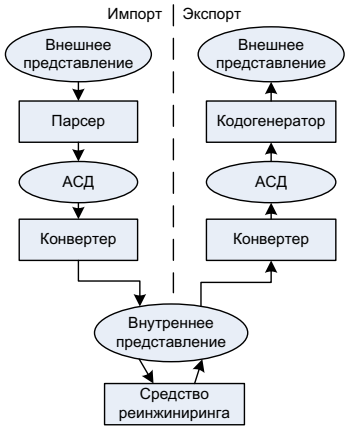

Рис. 3.8. Средства ввода-вывода данных в CAP с использованием промежуточного АСД

Использование двухуровневого подхода, кроме упрощения модулей, позволяет использовать различные конвертеры. Таким образом, при необходимости пользователь сможет реализовать свои библиотеки экспорта и импорта представления, опираясь на предоставляемые ему механизмы работы с синтаксическим деревом.

3.6. Резюме по разделу

В данном разделе были сформированы предложения по построению программируемого CAP. При этом была разработана собственная методика представления устройства, основанная на архитектурном графе устройства. Была сформирована функционально полная система команд для представления, которая впоследствии была дополнена командами управления устройством.

Разработанная архитектура средства во многом основана на подходах к построению интегрированных сред разработки. При этом упор сделан на модульность организации, расширяемость и наличие полнофункционального API.

Количество сформированных предложений очень велико, и тяжело произвести анализ эффективности подходов без их апробации на практических задачах. Поэтому, решено разработать прототип CAP в соответствии с большинством предлагаемых подходов, о чём пойдёт речь в следующей главе.

4. РАЗРАБОТКА ПРОТОТИПА ПРОГРАММНОГО СРЕДСТВА РЕИНЖИНИРИНГА

Для апробации подходов, сформированных в предыдущем разделе, решено было разработать некий прототип автоматизированного средства реинжиниринга. Очевидно, что качественная реализация всех предложений требует больших временных затрат, которые не соответствуют объёмам магистерской работы. Поэтому, было решено выделить и реализовать лишь основные предложения.

В разделе приведена информация обо всех этапах разработки прототипа CAP: от постановки задач до тестирования. Поскольку архитектура средства была описана ранее, то в первых пунктах сформированы ограничения и отступления от данной архитектуры, которые позволили реализовать функционирующий прототип средства.

От разработки полного комплекса программной документации в соответствии с государственными стандартами решено отказаться, так как разрабатываемый прототип не является самостоятельным программным продуктом, а лишь одной из составляющих научного исследования. Разработка программной документации возможна в случае успешной апробации и принятия решения о развитии средства до полноценного программного продукта.

4.1. Постановка задач на разработку прототипа

4.1.1. Формирование концепции прототипа

Как уже было сказано, основной целью разработки прототипа является демонстрация практической применимости идей из предыдущего раздела. Хотелось бы показать следующее:

- проведение полного цикла реинжиниринга с использованием средства для нескольких частных задач;
- возможность интеграции средства с другими САПР;
- возможность разработки пользовательских библиотек и расширений.

Для демонстрации данных задач недостаточно одного ядра прототипа, для которого велась разработка в предыдущем разделе. В минимальном варианте прототип CAP должен:

- использовать разработанное внутреннее представление;
- реализовывать разработанную внутреннюю архитектуру;
- считывать описание устройства из внешнего представления;
- генерировать синтезируемое выходное представление;
- поддерживать хотя бы один способ программирования;
- поддерживать хотя бы один способ интеграции со сторонними средствами;

- иметь хотя бы одну пользовательскую библиотеку, решающую специфическую задачу реинжиниринга;
- иметь возможность добавления пользовательских библиотек;
- иметь хотя бы один пользовательский интерфейс.

В качестве решаемой задачи реинжиниринга предлагается рассматривать некоторый пример из области повышения надёжности. Например, можно взять пример по введению аппаратной избыточности, рассмотренный в пункте 1.4.2. Более того, можно ограничиться лишь мажорированием некоторого блока устройства с добавлением голосователя. Пример минимальной реализации прототипа приведён на рис. 4.1.

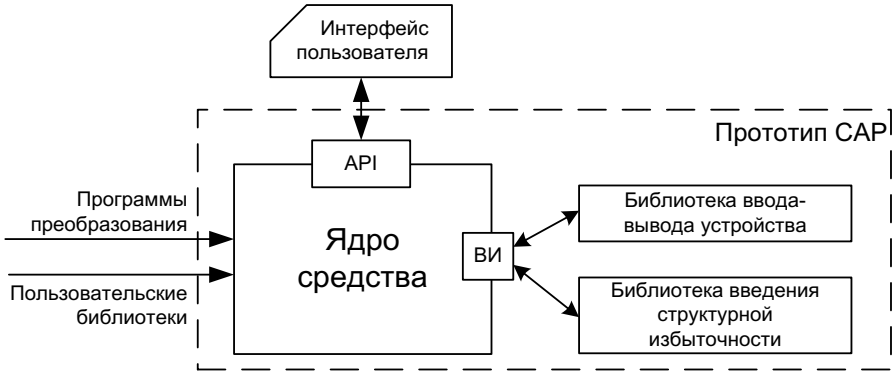

Рис. 4.1. Состав минимальной реализации прототипа САР

Основной целью разработки прототипа САР является подтверждение применимости подходов, предлагаемых в работе. Разрабатываемый прототип средства должен быть максимально прост в тех частях, которые не представляют интереса с точки зрения исследовательской задачи. В последующих пунктах рассмотрены ограничения и технические решения, которые позволили снизить сложность реализации прототипа.

4.1.2. Выбор внешнего представления устройства

Чтобы можно было продемонстрировать полный цикл реинжиниринга с использованием прототипа САР, требуется обеспечить поддержку импорта и экспорта внешнего представления устройства. Невозможно обеспечить поддержку множества форматов внешнего представления устройства, так как реализация модулей ввода-вывода для каждого из них требует достаточно больших временных затрат. Поэтому, решено было выбрать по одному основному формату входных и выходных данных.

С учётом того, что разрабатываемое средство планируется интегрировать с другими САПР, имеет смысл рассматривать только два подхода к представлению устройства:

- использование полноценных описаний на HDL;
- использование нетлистов.

Каждый из подходов имеет свои преимущества и недостатки, основные из которых описаны в таблице 4.1.

Таблица 4.1

Преимущества и недостатки использования внешних представлений

Формат	Способ внешнего описания устройства	
	HDL	нетлисты
входной	+ различные описания; + наличие пользовательских комментариев и меток; - сложность конвертации во внутреннее представление; - сложность интерпретации конструкций языка (макросы, условная генерация);	+ простота обработки; + независимость от внешнего представления; - привязка к САПР; - привязка к аппаратной платформе;
выходной	+ произвольная сложность генерации; + универсальность;	+ простота генерации; - жёсткая привязка к САПР;

Для упрощения импорта представления целесообразно использовать нетлисты. При этом разрабатываемое САР будет функционировать лишь совместно с некоторой средой разработки, которая сможет сгенерировать нетлисты на основе исходных HDL-описаний. С другой стороны, мы получаем полную независимость от внешних описаний устройства. Прототип САР сможет опосредованно работать со всеми форматами представления (в том числе и мультиязычными), которые поддерживает базовая САПР.

Проблемы привязки к аппаратной платформе

Наибольшую проблему при использовании нетлистов в качестве входного формата представляет то, что нетлисты формируются для передачи данных в средство реализации очередного этапа разработки устройства, будь то анализ или синтез. Поэтому нетлисты генерируются с использованием системных библиотек, которые привязаны к аппаратной платформе.

Для ПЛИС в нетлисты входят описания внутренних элементов кристалла: логических элементов, памяти, умножителей и прочих аппаратных акселераторов. Данные компоненты являются элементами иерархических описаний нижнего уровня и включают некоторую функциональность. Поэтому,

полноценный анализ устройства требует учитывать аппаратный базис. Можно предложить три подхода:

- добавить в библиотеку импорта представления слой, преобразующий аппаратную реализацию в абстрактное представление за счёт замены блоков на функциональные аналоги;
- вести работу в рамках исходного представление, а задачу трансформации перенести на функции экспорта;
- оставить решение по организации преобразований пользователю.

Первый подход является наиболее корректным, так как принуждает использовать внутри средства абстрактное представление, что обеспечит совместимость пользовательских библиотек между собой. С другой стороны, перенос архитектуры в абстрактное представление является отдельной задачей и достаточно сложной задаче реинжиниринга, поэтому её реализация для внешнего представления имеет смысл только в том случае, если это действительно необходимо (например, при переносе реализации со ПЛИС на заказные микросхемы).

В разрабатываемом прототипе решено отказаться от введения дополнительного слоя преобразований, так как не планируется решать задачи переноса архитектуры между различными архитектурами и аппаратными платформами. Подобная задача может быть решена при помощи дополнительных библиотек ядра.

Вывод нетлистов

С выводом нетлистов складывается несколько иная ситуация. Критерий простоты здесь неактуален, так как для любого сложного HDL мы можем использовать лишь некоторое его подмножество. В тоже время, при необходимости, мы можем использовать всю функциональность, заложенную разработчиками языка. Используя стандартный HDL, мы гарантируем то, что результаты обработки могут быть использованы впоследствии не только базовой САПР, но и любой другой. Поэтому, в качестве выходного формата решено использовать некоторый полноценный язык описания устройства. С точки зрения универсальности наиболее привлекательны VHDL или Verilog.

Итак, по результатам анализа выбраны различные входные и выходные представления. Имеет смысл использовать нетлисты и HDL со схожим синтаксисом, чтобы в прототипе можно было частично реиспользовать программную функциональность. В этом случае нам доступны пары Verilog/Verilog-netlist и VHDL/VHDL-netlist. Данные пары для нас практически равнозначны, поэтому выбор будет сделан на основании степени поддержки тех или иных форматов в выбранной САПР.

4.1.3. Интеграция прототипа в процесс разработки

Процесс реинжиниринга во многом использует средства разработки (синтез, анализ, моделирование). В прототипе, как и в конечном средстве, не имеет смысла реализовывать весь цикл разработки. Было принято решение интегрировать разрабатываемое средство с одной из сред проектирования и использовать её ресурсы для решения всех задач, кроме трансформации архитектуры и ввода-вывода представлений.

Для реализации прототипа была выбрана среда Quartus II компании Altera. Причиной выбора стали:

- опыт автора диссертации;
- возможность вызова средства через TCL;
- наличие встроенных средств анализа устройства;
- интеграция всей функциональности в одной оболочке.

Процесс разработки в среде Quartus II приведён на рис. 4.2. Он включает в себя все стадии разработки устройства. Среда поддерживает замену стадий сторонними EDA, что обеспечивает высокую конфигурируемость данной среды разработки.

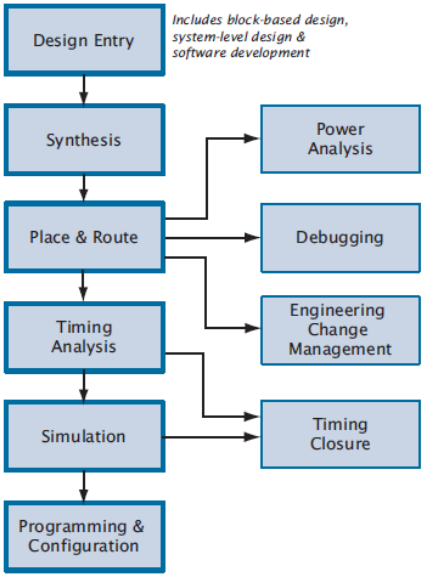

Рис. 4.2. Этапы разработки устройства в среде Quartus II [9]

Интеграция разрабатываемого средства реинжиниринга с Quartus II позволяет использовать встроенные средства синтеза и анализа, а также

использовать Quartus как транзитную программу при доступе к средствам моделирования (например, к ModelSim).

Среда Quartus II позволяет генерировать нетлисты в четырёх форматах: EDIF, VQM и в виде редуцированных структурных описаний на VHDL и Verilog. При этом исходными данными являются проекты среды QuartusII, которые могут включать описания в виде блок-диаграмм, полноценных VHDL и Verilog, а так же многих других форматах. Это позволяет расширить возможности применения средства.

На рис. 4.3 приведена схема использования прототипа совместно со средой QuartusII. Средство реинжиниринга встраивается внутрь процесса разработки, используя результаты первого синтеза в среде Quartus2 и порождая исходные коды для второго.

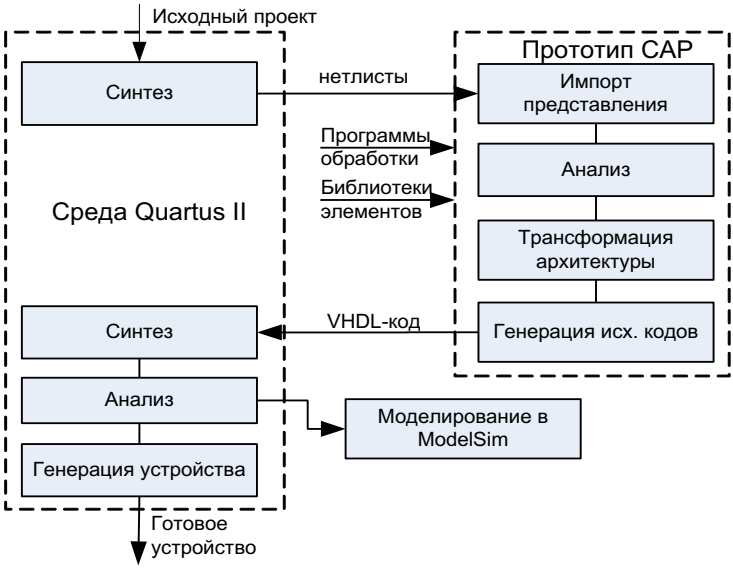

Рис. 4.3. Схема использования прототипа совместно со средой Quartus II

Среда QuartusII не позволяет реализовать подобный вариант генерации при помощи стандартных средств. Однако в данной САПР присутствует поддержка программ на Tcl, т.е. можно запрограммировать её на выполнение цикла реинжиниринга по предлагаемому алгоритму. Тем не менее, на этапе разработки прототипа задача автоматизации взаимодействия двух сред является опциональной.

4.1.4. Решения по внутренней реализации прототипа

Ядро средства

Данный модуль является наиболее важной и сложной составляющей средства реинжиниринга. Поэтому, даже в прототипе требуется его детальная проработка, иначе прототип окажется неработоспособным и неприменимым для частных задач реинжиниринга.

Ядро средства решено реализовывать как включаемую библиотеку. Подобная реализация позволяет относительно просто интегрировать его в сторонние приложения в виде некоторого статического объекта. При этом, очень просто организовать взаимодействие с ядром и возможно редуцировать API, так как возможен прямой доступ к объектам контекста.

В списке ниже перечислены основные приоритеты разработки прототипа, отсортированные по убыванию значимости:

- реализация дерева элементов;
- реализация ядра средства, базовой и системных библиотек;
- разработка тестовых интерфейсов и библиотек;
- разработка прикладных библиотек для САР;
- разработка полноценного API;
- поддержка внешнего языка программирования.

Далее в тексте описаны основные решения по программной реализации отдельных элементов САР.

Дерево элементов

Разработанное дерево элементов тесно связано с архитектурой САР и языком трансформации устройства. Поэтому, данная составляющая должна быть реализована максимально полно. Как минимум, должны быть реализованы все типы ячеек.

В прототипе средства можно отказаться от идеи наследования свойств через расширение, так как данный подход позволяет упростить обработку дерева в сложных пользовательских функциях, перенося всю обработку на ядро средства. При необходимости, наследование может быть реализовано позднее.

Аналогично, нет необходимости в механизмах группировки всех элементов в дереве. Можно ограничиться лишь группировкой сигналов в шины, что поддерживается не только всеми HDL, но и большинством форматов нетлистов.

Интерфейсы ядра

В разрабатываемом прототипе решено реализовать как командный интерфейс ядра, так и механизм событий. В предлагаемой архитектуре для обоих случаев предполагается, что доступ к контексту самого CAP будет осуществляться через “Менеджер команд”. Таким образом, получение любой информации должно осуществляться через некоторый шлюз, что позволит контролировать целостность дерева элементов.

Пользуясь тем, что в разрабатываемом прототипе средство реализуется на Java, решено предоставить пользовательским библиотекам прямой доступ к исполняемой среде, т.е. пользовательские библиотеки ядра будут иметь доступ ко всей функциональности средства. Такой подход некорректен с точки зрения безопасности, но позволяет упростить реализацию пользовательских функций.

Аналогично будет реализован и API прототипа. В результатах выполнения команд и сообщениях, кроме сформированных отчётов, будут передаваться и ссылки на объекты дерева элементов. Таким образом, внешние средства получат полный доступ к внутренней функциональности средства.

Механизмы программирования прототипа

Программируемость – одно из основных предъявленных требований. Данное свойство планируется обеспечить за счёт разработки библиотек ядра и поддержка языка программирования для управления преобразованием внутри самого средства реинжиниринга.

Библиотеки ядра будут реализованы программно с использованием исходных кодов средства реинжиниринга, после чего будут подключаться к CAP на этапе компиляции (или динамически, если это позволяют выбранные средства разработки).

Кроме программирования библиотек ядра, на базе внутреннего языка управления преобразованиями предполагается реализовать простой скриптовый язык для внешнего управления преобразованиями. Он будет использован для сохранения и воспроизведения истории пользовательских команд, выполнения инициализационных скриптов для средства и прочих задач. Данный язык не будет обладать полнотой по Тьюрингу, т.е. построение произвольных программ обработки на нём будет невозможно.

Пользовательские элементы архитектуры

При разработке прототипа решено отказаться от большинства внешних элементов архитектуры: не будут поддерживаться проекты реинжиниринга с сохранением промежуточного контекста, внешние языки программирования (только примитивные скрипты). API средства будет ориентирован на использование CAP в качестве включаемого модуля на Java.

Все перечисленные элементы могут быть реализованы при помощи механизма расширений, поэтому при необходимости могут быть добавлены в прототип без модификации внутренней архитектуры.

4.1.5. Формирование требований к дополнительным модулям

Одного ядра CAP недостаточно для того, чтобы продемонстрировать выполнение требований к средству. Требуется разработать дополнительные модули, которые обеспечат импорт нетлистов, экспорт VHDL, а также дадут возможность управления преобразованиями.

Для демонстрации решения прикладных задач реинжиниринга решено выбрать область повышения надёжности устройства. Как минимум, хотелось бы реализовать операцию по внесению аппаратной избыточности в устройство (см. пример на рис. 1.11).

4.2. Организация процесса разработки прототипа

Разработка прототипа средства велась на протяжении года. В разработке участвовали несколько человек, что потребовало не только детальной проработки архитектуры прототипа, но и организации совместной разработки прототипа.

4.2.1. Выбор средств разработки прототипа

Для прототипа выбор средств разработки принципиальным не является, так как реализуется лишь прототип средства реинжиниринга. Основным требованием к нему является демонстрация применимости подхода, поэтому основным критерием при выборе является скорость разработки. Параметры по производительности, надёжности и прототипа отходят на второй план.

Во многом скорость разработки определяется не средствами разработки, а опытом в их использовании, поэтому выбор производился из числа известных автору средств разработки.

Средства разработки архитектуры

Для разработки архитектуры и первичной спецификации прототипа было решено использовать UML (Unified Modeling Language) и среду разработки Enterprise Architect. Разработку архитектуры решено проводить по методологии ICONIX (см. [40]), изначально ориентированную на UML.

Выбранная методология предполагает нисходящее проектирование с последовательной детализацией отдельных модулей, параллельным использованием структурного и поведенческого описаний [22].

Язык программной реализации

Для программной реализации прототипа было решено использовать язык Java. Причиной выбора, кроме знания автором данного языка, стали:

- развитая концепция ООП в языке;
- простота разработки;
- наличие интегрированных средств разработки (IDE);
- простота повторного использования библиотек.

Решающим фактором при выборе языка программирования стало наличие плагинов для разработки на VHDL и Verilog в среде Eclipse. Технически, в неё можно встроить средство реинжиниринга, интегрировав его со средствами разработки в рамках одной IDE. Поскольку в Eclipse наиболее просто встроить модули, реализованные на Java, то и прототип целесообразно разрабатывать на нём.

Выбор среды разработки

В качестве среды разработки решено взять интегрированную среду разработки NetBeans IDE версии 6.9. Она поддерживает язык Java 1.6, имеет встроенные средства отладки и модульного тестирования. Будут использоваться следующие возможности средства:

- редактирование и отладка программ на Java;
- рефакторинг исходных кодов;
- модульное тестирование проектов при помощи JUnit4;
- профилирование программ на Java;
- генерация документации на исходные коды в стандарте Javadoc.

4.2.2. Выбор средств для работы с нетлистами VHDL

При разработке прототипа хотелось бы минимизировать затраты на разработку модулей ввода-вывода. В случае предлагаемого двухэтапного подхода, логично использовать некоторое отдельностоящее средство для считывания АСД или АСГ и последующей генерации исходных кодов. Можно предложить следующие пути:

- разработать библиотеку для ввода-вывода самостоятельно;
- заказать разработку библиотеки у третьих лиц;
- использовать готовое стороннее средство;
- извлечь парсер и кодогенератор из другого проекта с открытым исходным кодом и соответствующей лицензией;

Ввиду ограничений по временным ресурсам нежелательно было нежелательно вести разработку модуля самостоятельно, так как это привело бы к меньшей проработке основных модулей САР. Также проект находился на

самофинансировании, поэтому вариант с покупкой коммерческого продукта или же разработкой на заказ не рассматривался.

Из оставшихся вариантов наиболее приемлемым было использование уже готового модуля с открытыми исходными кодами, даже несмотря на то, что для данных вариантов отсутствуют всякие гарантии работоспособности библиотек. Среди подобных вариантов оказалась и библиотека vhdl_ast, разработанная на кафедре Компьютерных Систем и Программных Технологий (КСПТ) с целью автоматизации задач по повышению отказоустойчивости. Ввиду схожести тематики и возможности прямого контакта с автором библиотеки решено было остановиться на данном варианте.

Библиотека vhdl_ast основана на SIGNS (изначально Simple Gate Net Simulator, т.е. “Простой Симулятор Вентильного Уровня”). В конечном счёте, проект развился в САПР полного цикла разработки на языке VHDL с открытыми исходными кодами [35].

Из проекта Signs разработчиками vhdl_ast были взяты сигнатуры языка VHDL и первичная реализация парсера и генератора, которая в свою очередь основана на технологии JavaCC (Java Compiler Compiler). В vhdl_ast реализация парсера была доработана для лучшего представления синтаксического дерева.

Библиотека vhdl_ast позволяет конвертировать данные между представлением в АСД и исходными кодами в редуцированном подмножестве VHDL. Таким образом, при помощи выбранного средства организуется первый этап ввода-вывода из схемы в пункте 0. Задача формирования внутреннего представления будет решаться средствами прототипа, для чего будет разработана отдельная библиотека ядра. На рис. 4.4 приведена схема организации ввода-вывода данных с использованием библиотеки vhdl_ast и средой Quartus II, которая была выбрана ранее.

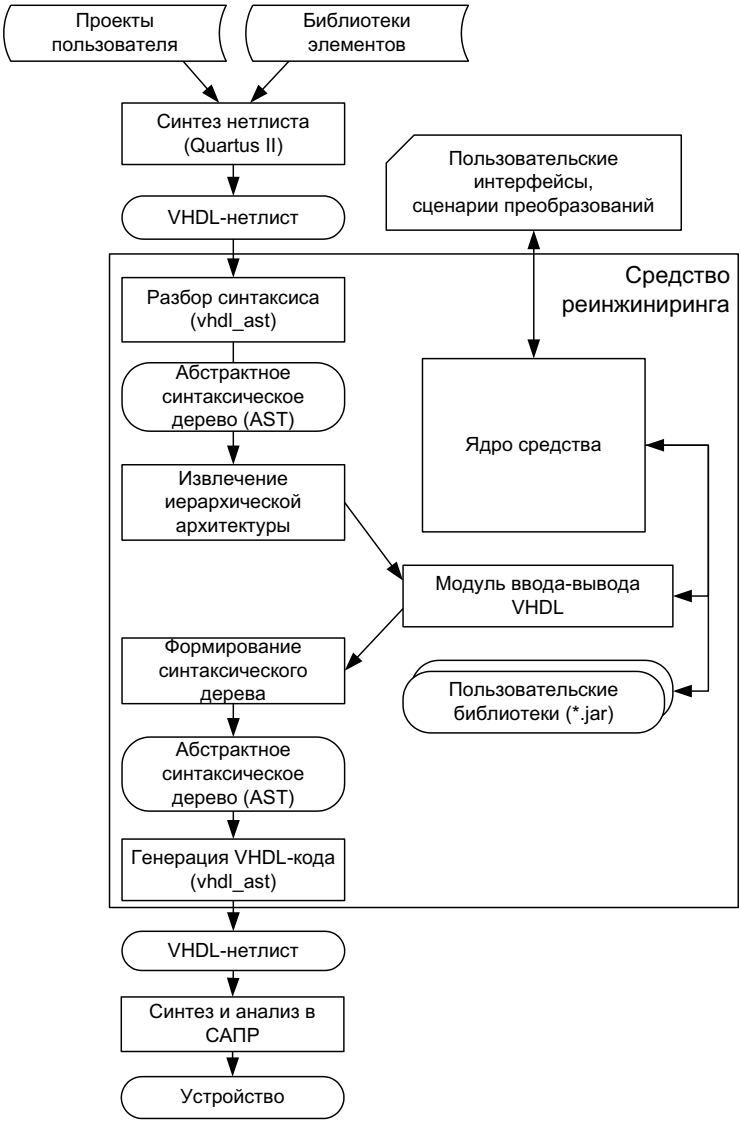

Рис. 4.4. Организация ввода-вывода представления в средстве реинжиниринга с использованием библиотеки vhdl_ast и среды Quartus II

4.2.3. Выбор средства разбора команд внутреннего языка

В архитектуре разрабатываемого средства вводится внутренний язык управления преобразованиями. Решено, что данный язык будет использовать стандартный синтаксис вызова в командных оболочках UNIX. Поэтому, для разбора команд можно использовать любой парсер командной строки.

После обзора решений был выбран открытый проект JSAP (Java Simple Argument Parser), который обладает следующими свойствами:

- открытые исходные коды;
- простота использования;
- поддержка различных типов параметров и опций (флаги, значения, списки и т.п.);
- возможность добавления пользовательских типов аргументов (с проверкой на этапе разбора);
- автоматическая генерация справки на функцию [39].

4.2.4. Выбор и развёртывание средств управления проектом

В разработке прототипа программного средства принимали участие несколько студентов и преподавателей кафедры КСПТ, в том числе и автор данной магистерской диссертации. Большая часть работы происходила удалённо, поэтому возникла необходимость в использовании средств управления проектами для организации взаимодействия участников. Подобная система должна реализовывать следующие функции:

- наличие средств контроля версий для исходных кодов;
- управление задачами;
- хранение файлов;
- предоставление краткой информации о проекте;
- доступ ко всем ресурсам через Интернет.

Ввиду малого масштаба проекта отсутствует необходимость в использовании сложных систем для решения отдельных задач. Поэтому, было решено выбрать некоторую интегрированную систему управления проектами с Web-интерфейсом. Затраты времени на администрирование системы также были признаны неэффективными, поэтому было принято решение разместить систему на удалённом хостинге, предлагающем готовые решения по управлению проектами. В результате проведённого обзора была выбрана система Redmine и хостинг Sourcerepo (www.sourcerepo.com).

Redmine считается наиболее функциональной системой из открытых систем своего класса [6]. Кроме перечисленных возможностей, она включает форумы, блоги, привязку ревизий к задачам и т.п. Решающими факторами при выборе стали поддержка системы контроля версий Mercurial и возможность

построения иерархии проектов в рамках одного хостинга. Также не последнюю роль в выборе сыграл опыт автора в использовании и администрировании Redmine.

Система управления проектом была развёрнута на хостинге в ноябре 2010-го года. В течение месяца была произведена её полная настройка (она выполнялась в фоновом режиме), после чего к системе был открыт ограниченный публичный доступ (см. [32]). В настоящее время в системе присутствует большая часть информации по проекту, пользователи прототипа имеют доступ к информационным страницам, форуму проекта. Кроме того, имеется возможность занесения информации об ошибках и пожеланиях в систему управления задачами.

Функциональность системы управления проектами в настоящее время избыточна, но при дальнейшем развитии проекта подобная система окажется востребованной.

4.3. Программная реализация прототипа

В предыдущих пунктах были сформированы требования к средству и ограничения, которых решено придерживаться при реализации прототипа. Также были предложены архитектура средства, методика внутреннего представления и язык (набор) трансформации устройства. Все перечисленные данные достаточно хорошо описывают требования к архитектуре средства. С учётом размеров проекта и командной разработки, требуется разработать требования к программной реализации средства.

Прототип средства не является самостоятельным программным продуктом, поэтому решено отказаться от разработки технического задания и полной спецификации на программную реализацию. Вместо этого по методологии ICONIX была разработана частичная UML-спецификация (см. [30]).

В данном пункте кратко описаны основные особенности программной реализации прототипа. Более полную информацию можно получить в документации на исходные коды [31].

4.3.1. Реализация дерева элементов

Дерево элементов средства реинжиниринга реализует представление, описанное в пункте 3.2. Исходя из перечисленных функций интерфейса, всё дерево элементов строится на базе трёх групп объектов: ячеек (и ссылок на них), параметров и контейнеров ячеек нижнего уровня. Все три типа реализуются при помощи разных иерархий классов, которые подробно описаны в документации на исходные коды [31].

Далее, рассмотрим более подробно программную реализацию каждой из составляющих дерева элементов. Также более подробно остановимся на ячейках групп и ссылок.

Ячейки дерева элементов

Все ячейки в дереве элементов представляют собой отдельные объекты и реализуют интерфейс DeviceNode, который определяет основные функции доступа к ячейкам. К данным функциям относятся:

- получение имени и типа ячейки;
- получение ссылки на родительскую ячейку;
- доступ к контейнеру ячеек нижнего уровня;
- доступ к контейнеру параметров ячейки;
- получение списка ссылок на ячейку.

На каждый тип ячейки дерева элементов (см. пункт 3.2) создано по одному базовому классу, который реализует хранение информации о ячейках и механизмы модификации данных.

Кроме самих ячеек, для некоторых типов (блоки, интерфейсы и пр.) потребовалось создать дополнительные классы для ссылок на данные ячейки, что было обусловлено необходимости хранения внутренних данных. Например, в случае, когда блок или функция являются ссылкой на библиотечный элемент, соединения должны подключаться к портам ссылки, т.е. данный виртуальный элемент должен содержать в себе и контейнеры портов.

Механизмы навигации

В прототипе реализована как абсолютная, так и относительная адресация в соответствии с механизмами именования, описанными в пункте 3.2.6. В программной реализации пути элементов разрешаются при помощи классов DevNodePath и DevNodePathItem, которые включают механизмы разбора строковых путей, анализ пути и методики его модификации. Разрешение путей к параметрам производится при помощи отдельного класса ParameterPath, который включает список параметров по уровням и механизмы его разбора.

Все программные модули средства и внутренние вызовы команд используют перечисленные выше классы, а не строковые представления. Поэтому, при необходимости возможна замена механизмов разрешения путей без переработки программных модулей.

Контейнеры ячеек

Для хранения ячеек нижнего уровня была реализована собственная иерархия контейнеров. Основные типы контейнеров описаны в табл. 4.2. Различие контейнеров состоит в способе хранения данных, механизмах доступа

к ячейкам и поддерживаемых типах ячеек. Все контейнеры реализуют один интерфейс, который предоставляет следующие возможности:

- доступ к элементу по описанию пути (в т.ч. и косвенная адресация);
- добавление элемента по унифицированному описанию;
- итерация по всем элементам контейнера;
- верификация контейнера.

Таблица 4.2

Основные типы контейнеров ячеек в программной реализации прототипа

Имя контейнера	Описание
DevNodeContainer	Абстрактный интерфейс для всех типов ячеек
MultitypeNodeCollection	Контейнер, доступ к ячейкам которого осуществляется по парному ключу (Имя, Тип)
NodeArray	Контейнер, доступ к ячейкам которого производится только по индексу группы
UniqueNameNodeList	Список ячеек одного типа с уникальными именами
EmptyNodeContainer	Пустой контейнер. Используется для ячеек нижнего уровня, которые не могут включать другие ячейки (сигналы, соединения, функции и пр.).
SingleTypeNodeArray	Индексированный контейнер ячеек одного типа (например, используется в шинах сигналов)

4.3.2. Реализация ядра средства реинжиниринга

Ядро средства реинжиниринга было реализовано в соответствии с разработанной архитектурой (см. п. 3.4). Каждый модуль ядра является группой классов в отдельном пакете (package), общая функциональность вынесена в дополнительные пакеты. В таблице 4.3 приведена краткая информация о расположении данных модулей и их составе. Более полная информация приведена в документации на исходные коды [31].

Ядро CAP является статическим объектом, который доступен через интерфейс Instanced, что позволяет обращаться к нему из любой точки приложения. Такой механизм выбран, чтобы не усложнять программную реализацию пересылками указателей на исполняемую среду и прочие модули. Подобная реализация исключает использование нескольких ядер CAP в рамках одного процесса.

В ядре средства реализована иерархия исключений, с помощью которых распространяются ошибки, возникающие при выполнении операций. При корректном доступе к ядру через API попадание исключений наружу исключено: они заменяются событиями ядра.

Таблица 4.3

Основные составляющие программной реализации ядра (пакет obl_core)

Пакет	Описание
.element_tree	Программная реализация дерева элементов (см. выше)
.event_manager	- менеджер событий ядра; - описания типов событий; - интерфейсы обработчиков событий;
.command_manager	- менеджер команд; - определение класса команды; - средства разбора строковых команд;
.library_manager	- менеджер библиотек; - базовые классы для пользовательских библиотек ядра; - список библиотек по-умолчанию;
.io	- абстрактные классы и интерфейсы для средств ввода-вывода представления - средства вывода представления в текстовой форме
.util	- описания исключений в средстве реинжиниринга; - описания механизмов разрешения путей;
.commands	- библиотека системных команд ядра; - шаблонные классы для команд;

Менеджер контекста в прототипе как отдельный модуль не реализован. Функции хранения истории переданы в менеджер команд, а указатели на выбранную ячейку – в дерево элементов.

4.3.3. Реализация механизмов программирования прототипа

Было реализовано два механизма программирования средства реинжиниринга: через реализацию библиотек ядра на Java или через примитивный скриптовый язык. Из-за ограничений по времени от поддержки функционально-полного внешнего языка программирования решено было отказаться.

Программирование через библиотеки ядра

Данный механизм является основным для разрабатываемого средства. Он был реализованы в соответствии с архитектурой, описанной в пункте 3.4.6. Расширения прототипа получают полный доступ к ядру средства. Реализация алгоритмов преобразования ведётся на языке Java.

Скрипты управления преобразованиями

При разработке прототипа было принято решение отказаться от полноценного внешнего языка программирования. Вместо этого на основе интерпретатора команд была реализована возможность выполнения программ преобразований, заданных во внешнем скриптовом файле. Для запуска скриптов преобразования в библиотеку SystemLib была добавлена команда run (см. приложение D). Также была добавлена возможность вызова команд через обращение к командной оболочке ОС (Команда system).

Программная реализация механизма предусматривает только последовательное исполнение команд из файла. Примеры скриптов преобразования приведены в приложении G.

4.3.4. Доработка библиотеки vhdl_ast

В ходе разработки модуля ввода-вывода нетлистов в библиотеке vhdl_ast был обнаружен ряд недоработок и ошибок, список которых приведён в приложении E. Фактически, парсер vhdl_ast работоспособен только для простых нетлистов (устройства типа счётчиков и сумматоров), но для сложных нетлистов он оказался неприменим.

При обращении к разработчикам vhdl_ast оказалось, что поддержка библиотеки не ведётся, поэтому необходимую функциональность пришлось реализовывать самостоятельно. Для доработки в системе управления проектами Redmine был создан отдельный проект, в который заносились выявленные ошибки и история их исправления.

Доработка парсера

Для обеспечения возможности работы с нетлистами потребовалось доработать парсер, исправив в нём ряд ошибок и введя новую функциональность. Кроме исправления ошибок, были добавлены следующие возможности:

- импорт объявлений пакетов в AST;
- импорт вызовов функций в дерево AST;
- доступ к элементам массивов внутри вызовов функций.

Доработка кодогенератора

Генератор нетлистов в vhdl_ast отвечает за генерацию исходных кодов по АСД. В библиотеке данный модуль ориентирован на работу с VHDL-нетлистами. В то же время, выходным форматом у прототипа являются VHDL-совместимые описания, которые генератором не поддерживаются.

Пришлось доработать механизмы генерации, добавив промежуточный слой преобразований, который формирует синтаксическое дерево, совместимое с VHDL. Были реализованы:

- замена имён элементов на VHDL-совместимые;
- устранение соединений между сигналами различного уровня;
- устранение несинтезируемых сигналов.

В приложении Е, кроме выявленных ошибок vhdl_ast, отмечены те, которые были исправлены в ходе данного этапа работы. Незапланированная работа по исправлению vhdl_ast отняла достаточно много времени, так как вместе с парсером приходилось править и рассчитанный на него модуль ввода-вывода нетлистов.

4.3.5. Реализация модуля ввода-вывода представления

В соответствии с двухуровневой схемой ввода-вывода был реализован программный модуль, который обеспечивает возможность ввода VHDL-нетлистов и вывода VHDL-описаний. Модуль ввода-вывода не входит в ядро, и вся его функциональность была реализована в виде двух библиотек ядра: VHDL и Netlist. Библиотека Netlist ссылается на VHDL, которая в свою очередь реализует базовые механизмы работы с VHDL-файлами, используя библиотеку vhdl_ast.

Библиотеки в настоящее время содержат только функции ввода-вывода данных. Если с выводом VHDL-файлов всё относительно просто, то организация ввода нетлистов потребовала решения большого числа частных задач, некоторые из которых перечислены ниже:

- реализация алгоритма считывания в соответствии с п. 3.5;
- исключение несинтезируемых сигналов из нетлистов;
- восстановление иерархии по объявлениям блоков;
- восстановление интерфейсов по неполным объявлениям компонентов;
- реализовать различные алгоритмы считывания для плоских и иерархических нетлистов;
- восстанавливать иерархии компонентов;
- …

Подробнее о проблемах ввода нетлистов можно прочитать в приложении А, а о реализации алгоритма – в описании команд библиотеки. За время разработки прототипа не удалось обеспечить поддержку всех возможных конструкций в VHDL-нетлистах прототипа. Поэтому, разработанный модуль поддерживает не все возможности типы нетлистов, а считывание некоторых устройств может происходить с ошибкой.

4.4. Разработка дополнительных модулей

Кроме базовых модулей средства реинжиниринга, необходимых для обеспечения требований к функциональности прототипа, в процессе разработки были созданы дополнительные модули, которые облегчили разработку средства и позволили проверить различные возможности его использования.

В первую очередь, были разработаны пользовательские интерфейсы к средству, которые позволили использовать его как независимое приложение. Разработанные интерфейсы прежде всего использовались для отладки, но они вполне подходят для выполнения простых задач реинжиниринга.

Также был разработан плагин для NetBeans, который позволил упростить разработку библиотек ядра для средства. Впоследствии, при помощи данного плагина были разработаны прототипы специализированных библиотек ядра по сбору статистики и повышению надёжности, которые также описаны в данном пункте.

4.4.1. Разработка консольного пользовательского интерфейса

Целью разработки было продемонстрировать возможность как управляемого (автоматизированного), так и полностью автоматического реинжиниринга устройства. Разработанный консольный интерфейс поддерживает оба режима. Они соответственно называются режимами управляемого преобразования и исполнения программы.

Выбор режима осуществляется по наличию опции – пути к скриптовому файлу с командами. Таким образом, строка вызова приложения выглядит следующим образом:

путь_к_программе [путь к исполняемому файлу]

Вывод результатов выполнения команд осуществляется через стандартный поток вывода, поэтому при необходимости вывод результатов может быть перенаправлен при помощи средств командной оболочки. Это удобно, например, при накоплении отчётов о результатах или автоматизированном анализе результатов выполнения программы.

Интерактивный режим

В интерактивном режиме программа считывает пользовательские команды и передаёт их в менеджер команд ядра, после чего результаты выводятся в стандартный поток вывода (stdout). Выход из приложения осуществляется автоматически при вводе пользователем команды quit или exit (см. систему команд ядра).

Режим автоматической обработки

В данном режиме осуществляется запуск скриптового файла, переданного пользователем в параметрах вызова приложения. Интерпретатор менеджера команд построчно исполняет набор команд, указанный в файле при помощи стандартной команды системной библиотеки ‘run’. При возникновении ошибки результат выводится на экран.

Данный режим удобно использовать для автоматического вызова некоторого фиксированного преобразования, что может быть использовано, например, в скриптах компиляции устройства из исходных кодов или при автоматическом добавлении диагностических сигналов при отладке.

4.4.2. Разработка графического пользовательского интерфейса

Несмотря на удобство использования консольного интерфейса в автоматическом режиме, он плохо подходит для задач, где требуется анализ архитектуры устройства. Поэтому, было принято решения разработать примитивный графический интерфейс, который отражал бы внутреннее представление устройства и предоставлял бы интерфейс для выполнения пользовательских команд.

Как и ядро средства, GUI был разработан на Java с использованием графической библиотеки Swing. Главное окно приложения (рис. 4.5) включает следующие элементы:

- иерархия ячеек представления устройства в виде древовидного списка;
- иерархия параметров выбранной ячейки в виде древовидного списка;
- окно атрибутов выбранного параметра (имя, тип, значение);
- консоль для ввода команд (аналогична консольному интерфейсу);
- окно с результатами валидации устройства;
- меню с возможностями управления импортом и экспортом представлений, вызовом проверки дерева и выводом справки.

Графический интерфейс связывается с ядром средства реинжиниринга через API. Для управления преобразованиями используется механизм команд, обновление информации в интерфейсе происходит на основании событий, генерируемых ядром средства реинжиниринга.

Древовидный список, который на рис. 4.5 в левом верхнем углу, формируется инкрементально в процессе изменения элементов и просмотра ячеек пользователем. Это сделано для того, чтобы минимизировать время загрузки и отображения устройства, которое для крупных проектов может занимать несколько минут.

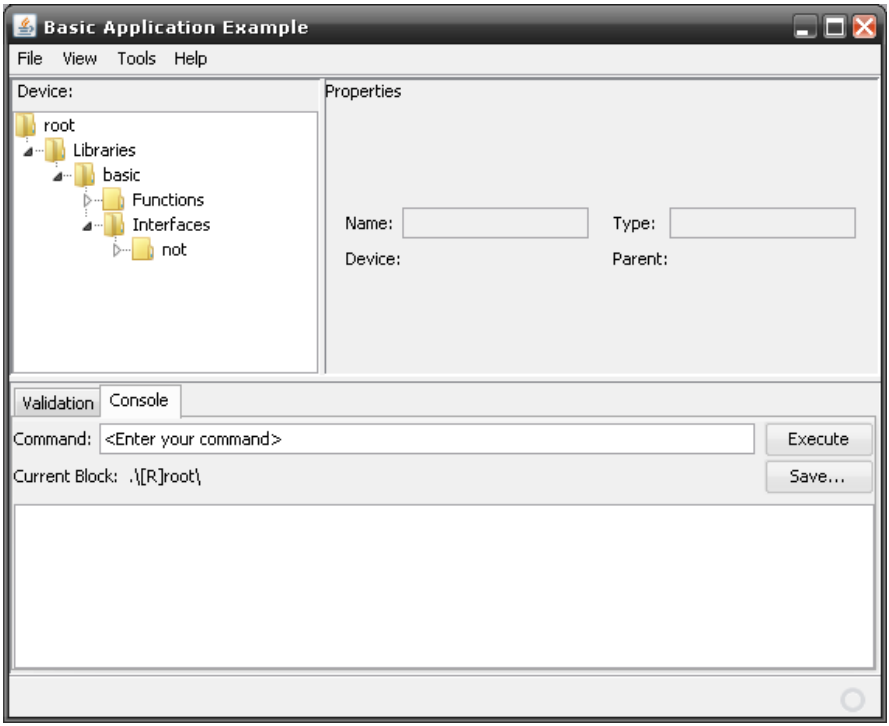

Рис. 4.5. Изображение главного окна GUI к средству реинжиниринга

4.4.3. Разработка плагина средства для IDE NetBeans

Для упрощения разработки пользовательских библиотек ядра решено было реализовать в NetBeans плагин, который будет включать в себя базовые вещи для разработки средства реинжиниринга.

В настоящее время плагин включает в себя шаблоны библиотек ядра и команд к ним, а также сборки ядра и базовых библиотек ядра с документацией на исходные коды CAP в формате Javadoc. Примеры шаблонов приведены в приложении G.

Данный модуль не является самостоятельным результатом, но может наращиваться вплоть до формирования для NetBeans полнофункционального плагина реинжиниринга устройства.

4.4.4. Библиотека сбора статистики по дереву элементов

Библиотека сбора статистики появилась в результате слияния функций анализа, которые изначально располагались в базовой библиотеке работы с

деревом элементов, с командами, разработанными Егоровым И.В. в рамках научно-исследовательской работы на кафедре КСПТ.

Библиотека сбора статистики включает следующие функции:
- вывод всех ссылок на ячейку;
- вывод всех ячеек дерева элементов;
- получение числа ячеек нижнего уровня различных в элементе;
- получение всех ячеек, реализующих указанный интерфейс;
- получение списка всех элементов по интерфейсу;
- получение статистики по портам интерфейса;
- получение цепочек соединений между сигналами.

Перечисленный функционал, прежде всего, может быть полезен при программном управлении преобразованиями, когда решение о проведении операции принимается по результатам анализа дерева. В конечном итоге, данная библиотека может быть расширена для проведения структурного анализа устройства.

4.4.5. Библиотека введения структурной избыточности

Данное расширение было запланировано как многосторонняя библиотека, которая решала бы задачи введения структурной избыточности в устройство, что может быть одним из путей повышения надёжности.

В рамках проекта в библиотеке Reliability реализована команда введения аппаратной избыточности, которая работает с блоками любой сложности. При выполнении данной задачи было сделано следующее:
- создана библиотека с элементами повышения надёжности (отказоустойчивая память, голосователи);
- созданы методы для генерации ячейки голосователя по заданным пользователем параметрам;
- реализован алгоритм внесения структурной избыточности (см. рис. 4.6).

Параметры вызова команды описаны в приложении D (п. 2.3.). Пример введения структурной избыточности с использование данной команды приведён в приложении Н. Ввиду отсутствия специализированных команд модификации, многие операции в алгоритме пришлось выполнять “с нуля”. При помощи команд из стандартных библиотек были выполнены следующие задачи:
- копирование блоков;
- создание блоков по интерфейсам;
- подключение групп сигналов, добавления портов к блокам;
- переносы соединений;
- добавление сигналов.

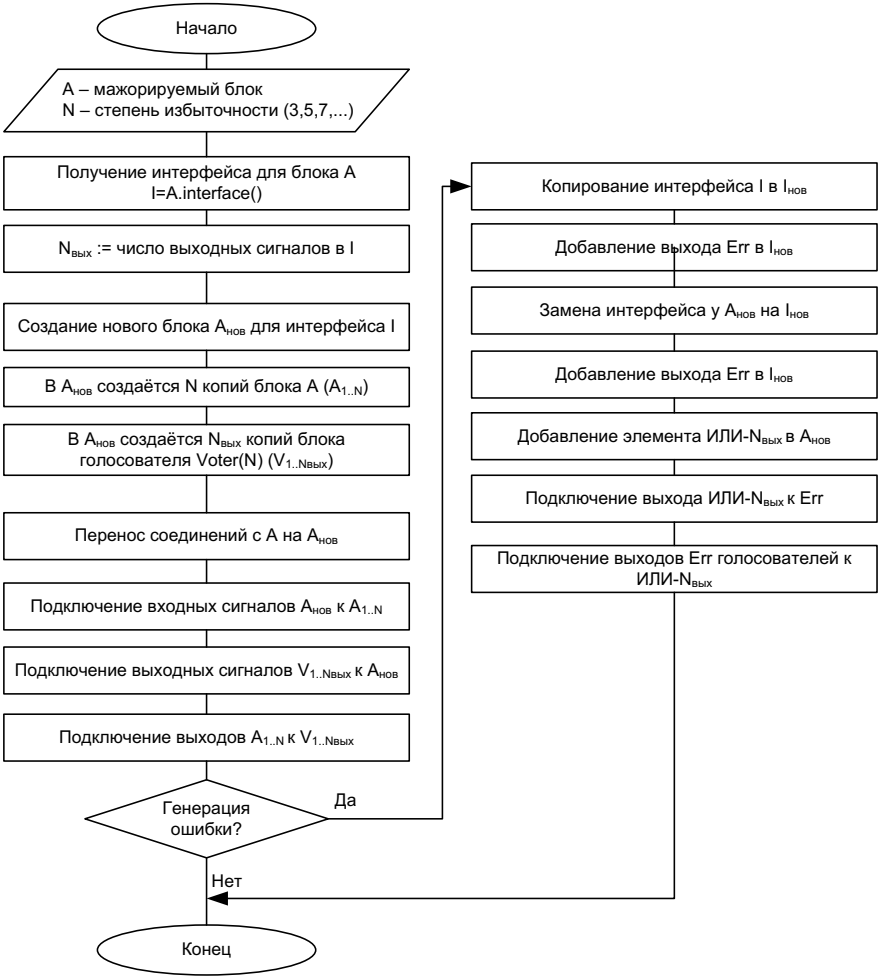

Рис. 4.6. Порядок действий при внесении аппаратной избыточности в устройство с использованием функций базовой библиотеки CAP

4.5. Тестирование разработанного средства

Основной целью тестирования CAP является подтверждение работоспособности прототипа, т.е. способности выполнять требуемые операции по преобразованию на тестовых примерах. Задача полного устранения ошибок в прототипе не ставилась, так как полное тестирование программной реализации заняло бы много времени из-за большого объёма проекта (около сорока тысяч строк).

При тестировании использовались следующие подходы:
- модульное тестирование;
- функциональное тестирование;
- верификация дерева элементов;
- ручное тестирование.

Модульное тестирование

Модульное тестирование разработанного прототипа затруднено большим числом элементов в исходных кодах (например, в коде около 250 классов). При разработке прототипа тестами были покрыты далеко не все классы и методы, а только наиболее низкие элементы описания. От тестирования модулей преобразований решено отказаться, так как для каждой задачи преобразования в дереве пришлось бы формировать исходное представление, что представляется достаточно сложной задачей.

Для организации модульного тестирования была использована библиотека JUnit 4, интегрированная в среду NetBeans.

Автоматизация функционального тестирования

Автоматизация тестов производилась при помощи программ управления преобразованиями, которые вызывались в менеджере команд. Данное тестирование было интегрировано в систему модульного тестирования NetBeans при помощи дополнительного класса SystemScriptTest (см. [31]).

В тест передаётся список скриптовых программ, которые исполняются в средстве. Если хотя бы одна команда возвращает ошибку, то тест считается неудачным, и пользователю возвращается отчёт об ошибке, сгенерированный менеджером команд.

Верификация дерева элементов

Разработанное средство имеет встроенные механизмы верификации дерева элементов. Пусть они реализованы лишь частично, но с их помощью возможно выявить некоторые нарушения в структуре дерева элементов, связанные с некорректными преобразованиями. Поэтому, в скриптах и автоматизированных тестах дополнительно вызывается команда валидации дерева элементов.

Ручное тестирование

Пользовательские интерфейсы и прочие дополнительные приложения прототипа CAP не были покрыты автоматическими тестами, так как задача их тестирования по сложности сравнима с реализацией, а временные затраты на данные модули хотелось минимизировать. Частичное тестирование подобных модулей осуществлялось в ручном режиме при использовании в процессе

разработки ядра прототипа и его библиотек. Выбранные подходы к тестированию не гарантируют исправность средства, но позволяют выявить ошибки в основных трассах выполнения, чего достаточно для демонстрационного прототипа.

5. ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ ПРИМЕНЕНИЯ СРЕДСТВА

В данном заключительном разделе подведены основные итоги по проведённой работе. Проведена апробация прототипа САР, на основании которой сделан вывод о применимости разработанных подходов. Также в рассмотрены возможности практического применения разработки и выделены основные направления дальнейшего исследования.

В ходе работы разработки прототипа у предлагаемых подходов был выявлен ряд недостатков, появились новые идеи по расширению архитектуры прототипа и пр. Эти вопросы также рассмотрены в разделе.

5.1. Апробация разработанного прототипа

В данном пункте приведены основные результаты апробации средства по примерам задач, которые были сформулированы в пункте 4.1.

Решение задач реинжиниринга при помощи прототипа

В прототипе средства реинжиниринга была разработана и реализована библиотека Reliability (см. 4.4.5), которая позволила решить задачу введения аппаратной избыточности в устройство. С помощью данной библиотеки решено продемонстрировать применимость средства для решения практических задач.

Решено провести эксперимент с выполнением полного цикла реинжиниринга некоторого простого устройства: от исходного проекта в Quartus II до синтеза и моделирования устройства, сгенерированного прототипом САР. Задачей реинжиниринга будет введение структурной избыточности через мажорирование некоторого модуля исходного устройства. Подробное описание проведённого эксперимента и анализ результатов приведены в приложении Н.

Результаты показали, что возможно проведение полного цикла реинжиниринга с использованием средства, построенного по предлагаемой архитектуре. Таким образом, предлагаемые методики пригодны хотя бы для автоматизации структурных преобразований устройства. О прочих задачах говорить пока рано.

Проверка поддержки сложных устройств

При тестировании средства также проверена возможность импорта, представления и трансформации, процессорных ядер, которые являются примером сложных цифровых устройств. В таблице 5.1 перечислены некоторые из проверенных процессоров.

В результате проблем с библиотекой vhdl_ast, на момент написания диссертации не удалось реализовать полный цикл реинжиниринга процессорных ядер. Тем не менее, удалось успешно считать и трансформировать архитектуру данных устройств. Проблемы вывода устройств будут устранены в дальнейшем.

Таблица 5.1

Список процессорных ядер, на которых тестировался прототип

Процессор	Архитектура	Число логических элементов*	Комментарий
e8051	8051	6500	Проверены плоские и иерархические описания
dalton		6588	
T51		6146	
oregano		5700	
LEON3	SPARC v8	12400	
* из отчёта о результатах синтеза в среде Quartus II			

Интеграция САР с другими средствами разработки

В явном виде интеграции САР с другими средствами произведено не было, так как был отдан приоритет расширению ядра средства реинжиниринга. Тем не менее, был проведён ряд экспериментов, которые подтвердили возможность интеграции с САПР Quartus II:

- по списку VHDL-файлов сгенерированы файлы проектов Quartus;
- средство реинжиниринга было запущено из среды Quartus II через встроенный Tcl;
- средство реинжиниринга было запущено на этапе синтеза устройства с передачей ему нетлиста;
- из прототипа запущена среда Quartus II и запущен синтез проекта.

Проведённые эксперименты показали, что возможно организовать взаимные вызовы прототипа САР и среды Quartus II даже без использования полноценного API.

Возможности расширения средства

Апробация возможностей расширения разработанного прототипа для решения пользовательских задач реинжиниринга была произведена студентом кафедры КСПТ Егоровым И.В. в рамках научно-исследовательской работе. Им была разработана библиотека анализа дерева элементов, в которую позже были перенесены функции анализа, реализованные в системной библиотеке при разработке прототипа.

В рамках работы Егоров И.В. использовал разработанный плагин для среды NetBeans, который позволил частично автоматизировать генерацию исходных кодов для его библиотеки. При этом были разработаны следующие функции:

- получение списка всех элементов по интерфейсу;
- получение статистики по портам интерфейса;
- получение цепочек соединений между сигналами.

Таким образом, CAP было успешно расширено для решения частных задач анализа устройства.

Заключение

К сожалению, не удалось провести всех запланированных экспериментов. Причиной тому стали проблемы и импортом и экспортом представлений (см. пункт 4.3.4). Тем не менее, удалось провести минимально необходимый набор опытов, которые позволили оценить применимость предлагаемых подходов к реинжинирингу устройства.

Проведённая апробация прототипа показала, что разработанное средство позволяет решать задачи реинжиниринга, соответствует общим требованиям по встраиваемости и расширяемости. Это говорит о том, что методики, на базе которых был разработан прототип, применимы для решения поставленных задач. Тем не менее, был выявлен ряд недостатков подходов, которые рассмотрены в пункте 5.3.

5.2. Анализ соответствия разработанного прототипа поставленным требованиям и предложениям.

При разработке прототипа не планировалось обеспечить соответствие всем требованиям, поставленным в пункте 3.1. Был выбран ряд ограничений и технических решений по реализации прототипа, которые позволили достаточно быстро реализовать проект.

Несмотря на ограничения, реализация прототипа потребовала больше времени, чем ожидалось, поэтому некоторые из запланированных возможностей пришлось отказаться на этапе реализации прототипа. Прежде всего, в жертву были принесены интерфейсные части. Не были реализованы следующие моменты:

- автоматизация взаимодействия средства со средой QuartusII;
- внешний язык управления преобразованиями;
- подключение средства в виде плагина к IDE.

Всё перечисленные пункты являются чисто инженерными задачами, не представляющими научного интереса в рамках данной магистерской работы. Поэтому, отсутствие данных возможностей, скорее, ограничило возможности практического применения прототипа.

В исходных кодах многие составляющие архитектуры были реализованы лишь частично. Примерами этому являются:

- подключение пользовательских библиотек элементов возможно только через тип “устройство”;
- типизация сигналов;
- верификация ячеек дерева элементов реализована частично.

Несмотря на существующие недостатки реализации, разработанный прототип был успешно применён для решения нескольких частных задач реинжиниринга, а его внутренняя архитектура в большинстве своём соответствует предложениям из третьего раздела. Поэтому, выполнена основная задача – реализация работоспособного прототипа на основании сформированных предложений.

5.3. Формирование предложений по доработке архитектуры средства и методик представления устройства

При реализации прототипа был выявлен ряд недостатков, который может затруднить практическое использование средства. Анализ недостатков показал, что некоторые из них вызваны недоработками в рамках предложенных архитектуры и методик представления.

В данном пункте рассмотрены основные недостатки и предложены пути их устранения.

5.3.1. Недостатки программной реализации прототипа

При разработке прототипа был выявлен ряд принципиальных проблем в предлагаемом подходе к реализации средства реинжиниринга. К ним относятся следующие проблемы:

- в приложениях может быть только одно ядро средства;
- настоящая реализация не является потокобезопасной;
- отсутствие защиты от пользовательских действий;
- неполная верификация дерева элементов;
- отсутствует возможность отката команды при неполном её выполнении;
- сложность обработки возвращаемых командами данных.

Ниже приведён краткий анализ проблем и предложены варианты их решения.

Ядро средства является статическим объектом

Для упрощения реализации прототипа исполняемая среда и прочие элементы ядра были заменены статическими объектами. Это позволило обращаться к модулям средства через интерфейс Instanced, не занимаясь передачей ссылок на ядро. Следствием модификации стало то, что внутри приложения может быть только одно ядро. Многоядерность может потребоваться при параллельной трансформации нескольких устройств.

Для решения проблемы может быть разработана серверная реализация ядра, когда само ядро и системные библиотеки будут находиться в отдельном приложении. Поскольку число механизмов API у ядра невелико, то его реализация большой сложности не представляет.

Безопасность по потокам

В настоящей реализации ядра отсутствует какая-либо потоковая защита средства. Поэтому, пользователь вполне может вызвать параллельные команды в различных потоках, что может нарушить работу ядра и привести к некорректным результатам.

Очевидным путём является перенос всей функциональности ядра в отдельный поток. При этом можно добавить очередь обработки команд, и команды, поступающие от разных потоков помещать в неё.

Дополнительным преимуществом такого подхода является возможность асинхронного выполнения команд. Но появляется и новая проблема - механизм событий, которые при текущей реализации будут вызываться в потоке ядра.

Защита от пользовательских действий

Сейчас пользователь, подключивший библиотеку ядра, получает полный доступ к содержимому ядра. Принципиально, сейчас ничто не мешает ему обратиться к дереву элементов, не используя механизм команд. Таким образом, при некорректных операциях в пользовательской функции может быть безвозвратно нарушено дерево элементов.

Есть два пути решения проблемы:
- запрет прямого доступа к ядру;
- разрешение доступа только для чтения;

В первом случае пользователь сможет получать информацию о дереве, используя встроенные команды ядра. Такой подход гарантирует сохранность дерева элементов, но фатально влияет на производительность, так как все операции будут происходить через дополнительный слой команд.

Сложность отмены изменений при ошибке

В настоящее время, если в процессе выполнения команды произошла ошибка, то откат уже произведённых ею изменений не производится. Это особенно плохо для скриптов и пользовательских команд, сложность которых может быть велика, а промежуточный результат может нарушить работу устройства.

> Пример: команда majorize из библиотеки Reliability
>
> Пусть мы резервируем некоторый блок дерева. При проведении мажорирования исходный блок был заменён новым объектом, в нём были созданы три копии исходного блока. После этого команда подключает голосователь, и, если его vhdl-файл отсутствует, завершает команду с ошибкой. При этом, вместо заменяемого блока у нас теперь “пустышка”, и отменить действие нельзя.

Технически, откат команды может быть реализован пользователем при помощи перехвата и анализа исключений, но делать это для каждой команды очень тяжело.

Частично данную ситуацию можно решить путём добавления возможности сохранения и загрузки текущего контекста исполняемой среды. Т.е. мы при выполнении критической команды можем сохранить весь контекст и, если возникнет ошибка, после команды восстановить исходное состояние. Недостатком является длительность копирования и восстановления всего контекста. Но подобные команды, например, были бы удобны при ручном проведении реинжиниринга, когда пользователю хочется “сохранить проект” и продолжить его модификацию через некоторое время.

Вторым вариантом является хранение истории изменений. Т.е. при каждом изменении дерева элементов или другой составляющей проекта сохраняется информация, которая может быть использована для последующего отката. Данный механизм чем-то похож на транзакции в базах данных, по образу и подобию его можно было бы реализовать.

Недостатком последнего подхода является сложность его реализации. Она ещё более усложнится из-за поддержки расширяемости, что актуально для пользовательских библиотек, которые в процессе работы формируют свои метаданные. Тем не менее, подобный механизм необходим в случае перехода к полноценной САПР.

Сложность обработки результатов команд

В настоящее время команды предполагают только текстовый результат выполнения посредством отчёта, генерируемого на базе результатов самой команды и её подкоманд. Это хорошо для консольного интерфейса, но не средств обработки.

С моей точки зрения, было бы удобно, чтобы команды могли возвращать не простой текст, а некоторый форматированный отчёт с расширяемой структурой. Нужно результаты в неком универсальном формате, доступном сторонним средствам представления. В этом плане наиболее привлекателен язык XML, который в настоящее время является одним из стандартных решений для формирования отчётов и взаимодействия между программами.

Например, результаты анализа, выполненные при помощи средства, могут быть преобразованы из XML в форматы специальных средств обработки. Особенно с этой точки зрения интересна интеграция с OLAP (Online Analytical Processing), который является стандартным механизмом обработки структурированных данных (в т.ч. XML).

5.3.2. Возможности доработки методики представления

Большинство проблем в прототипе связано с неполной реализацией внутренней архитектуры и методики представления. Тем не менее, выявлен ряд проблем в самих предложениях. Некоторые из них рассмотрены ниже.

Проблема портов и параметров ячеек

Порты и параметры являются подтипами сигналов в дереве элементов (port и generic соответственно). Они являются не самостоятельными элементами иерархии дерева элементов, а всегда относятся к блоку и формируются в соответствии с описанием его интерфейса. Данные ячейки используются для организации соединений.

Разработка прототипа, что подобный подход является спорным, так как возникают проблемы при обновлении портов и параметров в случае изменения интерфейса. При этом требуется пройти по всем реализующим интерфейс блокам, удалить старые сигналы и добавить новые. Также требуется разрывать все устаревшие соединения. Механизм подобных преобразований в прототипе был частично реализован, но с ним связана большая часть обнаруженных на настоящее время ошибок.

Альтернативой является привязка соединений не только к сигналам, но и к элементам, их содержащим (блокам, функциям). Тогда можно в соединение добавить параметр, описывающий выход, к которому идёт подключение, а сами объекты не создавать. Это упростит структуру дерева элементов, упростит процедуру модификации интерфейсов. Минусом будет ухудшение связности графа (нельзя получить порт напрямую). Вопросы с появлением соединений с некорректным указанием вывода можно решить при помощи механизмов верификации и их использования в момент модификации интерфейса.

Реализация функций

В настоящей архитектуре не предполагается дополнительных средств и операций для модификации внутреннего содержания функций. Всё внутреннее содержание представляется в виде строкового параметра Content.

При разработке предполагалось, что функции должны заменяться структурными описаниями и примитивными операциями, как это было сделано для модуля ввода нетлистов прототипа. Подобный подход вносит дополнительные сложности с преобразованиями, так как любая операция для изменения наполнения функции должна провести разбор строки параметра и потом каким-либо образом сформировать новую строку. Переносимость подобного механизма между языками представления отсутствует.

Функции можно представить как последовательность присвоений и вызовов других функций. Таким образом, данная ячейка после преобразования будет содержать следующие элементы:

- объявления сигналов;
- присвоения сигналов;
- ссылки на функции.

Для реализации механизмов не требуется вводить в дерево элементов дополнительных типов ячеек. В тоже время, появится возможность программируемой трансформации вызовов функций без анализа внутреннего описания параметра Content, нужда в котором отпадёт.

Проблема ссылок на библиотеки

Ссылки на используемые библиотеки хранятся только на уровне библиотек или устройства. Поэтому, при переносе элементов описания из одного объекта в другой (например, перенос часто используемого элемента в библиотеку) есть риск, что в нём будут присутствовать ссылки на элементы из библиотек, использование которых не указано. Это в свою очередь может привести к ошибкам в обработке устройства и его генерации.

Для решения проблемы предлагается внести в систему команд требование по автоматической модификации списка ссылок на верхнем уровне или же вообще отказаться от его хранения, генерируя его при помощи дополнительной системной команды.

Типизация параметров

В разработанной методике представления для описания значений параметров предложено использовать строковые переменные, а для контроля значений реализовать для них механизм верификации. Параметры могут использоваться в пользовательских алгоритмах трансформации устройства, которые не вызывают верификацию дерева элементов на каждом шаге, что в случае ошибки может привести к непредсказуемому поведению с плохим распространением ошибки.

Поэтому, было бы удобно реализовать контроль типов сигналов в соответствии, введя специальные функции в программную реализацию или ссылаясь на ячейки типов дерева элементов.

5.3.3. Возможности доработки архитектуры средства

Разработанная архитектура средства описывает лишь общие принципы построения средства. Все они были подтверждены по результатам апробации, но существует ряд частных проблем, которые хотелось бы подробно описать в последующих версиях.

Предоставление доступа к дереву элементов для чтения

В настоящее время архитектура предполагает, что всё взаимодействие пользовательских команд с деревом элементов будет вестись через исполняемую среду при помощи внутренних библиотек базовых и системных команд. Такой подход корректен, но при анализе дерева элементов потребуется совершать множество запросов, что негативно скажется на производительности из-за сложных механизмов обработки команд.

Предлагается добавить в библиотеку ядра механизм прямого доступа к дереву элементов для считывания данных. Подобный механизм может потребовать существенных доработок в случае, когда библиотеки ядра реализованы в виде отдельных приложений. В этом случае придётся реализовать механизм для пересылки требуемого программой участка описания.

В текущей реализации есть команда ls, которая выводит структурную иерархию ячеек с указанием связей. Этого механизма недостаточно, так как необходимо предоставлять ещё и содержимое внешних ссылок.

Управление правами доступа

Одной из проблем средства является необходимость предоставления расширенного доступа к составляющим ядра для пользовательских расширений. В прототипе такой проблемы нет – там сразу предоставлен полный доступ, что может привести к повреждению контекста ядра внешними средствами.

Предлагается добавить в средство дополнительный менеджер, через который библиотеки ядра будут запрашивать разрешения на расширенный доступ. Это позволит исключить случайный доступ к функциям ядра. Сам же модуль будет протоколировать доступ к данной функции, что в случае возникновения ошибок позволит быстрее определить их источник.

Защита дерева от модификаций

В ряде случаев при реинжиниринге используются статические элементы, модификация которых невозможна. Тем не менее, система команд и внутреннее преставление не предусматривают механизмов защиты от их изменения, и после трансформации архитектуры устройство может оказаться несинтезируемым.

Предлагается ввести механизм запрета модификации отдельных элементов: параметров, ссылок, блоков или целых библиотек. Данный механизм должен иметь отображение на дерево элементов. Предлагается в каждый элемент описания добавить флаг запрета модификации, который будет влиять на всю нижестоящую иерархию устройств.

Для данного механизма в системную библиотеку придётся добавить команды управления доступом (например, lock и unlock), а в прочие вызовы – механизмы для контроля возможности модификации.

5.4. Подходы к использованию разработанного прототипа

Несмотря на то, что в настоящее время прототип до конца не доработан, сформировано работоспособное ядро средства и механизмы пользовательских расширений. Поэтому, возможно использование разработанного средства для решения частных задач реинжиниринга.

Апробация прототипа показала, что средство принципиально подходит для решения любых задач реинжиниринга устройства. При этом предоставляется функционал по трансформации архитектуры, а задачи анализа решаются пользователем практически с нуля. Хотелось бы понять, что потребуется от пользователя для автоматизации некоторых практических задач реинжиниринга.

В настоящее время библиотеки ядра получают неограниченный доступ к его ресурсам, поэтому при помощи расширений могут быть реализованы любые дополнительные возможности: новые алгоритмы обработки и трансформации дерева элементов, API, языки программирования, механизмы импорта-экспорта нетлистов.

Фактически, единственным незаменимым элементом CAP является дерево элементов, т.е. внутреннее представление устройства. Всего возможно четыре уровня использования прототипа в сторонних средствах:

- использование дерева элементов;
- использование ядра CAP;
- использования ядра CAP и дополнительных модулей;
- использование ядра, модулей и интерфейсов как САПР.

Далее рассмотрим некоторые примеры использования прототипа CAP. Отдельным случаем является использование CAP как расширения к другому средству. Этот вариант будет рассмотрен отдельно.

Использование дерева элементов без ядра

При необходимости можно отказаться от всей функциональности ядра, и использовать дерево элементов как внутреннее представление устройства в своей разработке. При этом сохранятся возможности расширения представления, но все механизмы работы с деревом элементов пользователю придётся реализовать самостоятельно, так как базовые команды из системных библиотек привязаны к ядру средства.

Многие функции обработки реализованы вне команд, поэтому пользовательское приложение сможет получить к ним доступ, то есть на

конечной функциональности по преобразованиям отсутствие ядра скажется слабо.

Использование средства как включаемой библиотеки

В настоящей реализации ядро средства и базовый набор системных библиотек представляют собой единый jar-файл, который может быть подключен к проектам как на java, так и на других языках (обеспечивается сторонними средствами).

После подключения библиотека пользователю требуется создать объект, наследованный от obl_core.OblCore. В данном классе реализуется вся начальная инициализация ядра, после которой пользователь может приступить к использованию средства.

После подключения ядра пользователь должен обеспечить его интеграцию со своим приложением. Он может:

- активировать стандартные библиотеки ядра;
- подключить свои специализированные библиотеки;
- зарегистрировать обработчики событий ядра.

Для всех перечисленных действий предусмотрены стандартные команды, поэтому пользователю не потребуется вникать во внутренние методы ядра.

После подключения пользователь может взаимодействовать с ядром посредством команд ядра. Уровень сложности команд неограничен. К примеру, пользователь может при помощи одной команды блокировать ядро и работать с деревом элементов напрямую. При этом, вся ответственность за результаты модификации лежит на пользователе.

Текущая реализация менеджера команд позволяет пользователю внутри своей команды вызывать другие команды (в т.ч. и незарегистрированные в самом менеджере). Это позволяет избежать стандартных операций, а также добавляет возможности программной рекурсии и т.п.

Использование средства как расширения для среды разработки

Разработанный прототип легко встроить в любое средство, которое поддерживает расширения на языке Java. В противном случае потребуется разработка дополнительного API, а в случае Java к CAP можно обращаться напрямую. Примерами таких IDE являются NetBeans и Eclipse.

При разработке расширения, кроме самого CAP, потребуется некоторый проект плагина, в котором будет реализован интерфейс к средству реинжиниринга. Он может строиться по тем же принципам, что и пользовательские интерфейсы, описанные в пункте 4.4.

Возможно, потребуется реализовать библиотеку ядра, которая будет обеспечивать взаимодействие CAP с внешней средой через её API. Таким образом, оба средства будут рассматривать друг друга как расширения.

Особый интерес представляет использование CAP совместно с расширениями для разработки на HDL, что позволяет интегрировать весь цикл разработки в одну среду. Например, возможно дополнить возможности Sigasi HDT (см. пункт 2.3) по рефакторингу.

5.5. Исследование возможностей применения средства для решения практических задач реинжиниринга устройства

В рамках работы были рассмотрены следующие возможные направления практического применения средства:

- повышение надёжности устройства;
- введение конвейеризации в устройство;
- реверс-инжиниринг архитектуры устройства;
- преобразование HDL высокого уровня.

Возможности применения средства для повышения надёжности

В начале научно-исследовательской работы по тематике рассматривались вопросы повышения надёжности устройства и автоматизации данного процесса. Со временем, работы отклонились от данного направления в сторону более универсального средства. Тем не менее, повышение надёжности устройства остаётся одной из наиболее важных задач реинжиниринга, поэтому хотелось бы рассмотреть возможности повышения надёжности при помощи разработанного средства.

Повышение надёжности в готовом устройстве является одним из наиболее распространённых примеров реинжиниринга. Чаще всего она решается для сложных устройств, где выполнение ручного реинжиниринга затруднено.

В настоящее время в средстве реализована библиотека Reliability, которая в своём составе имеет средства для введения структурной избыточности в устройство, что является одним из подходов к повышению надёжности. Тем не менее, хотелось бы расширить данную библиотеку следующими возможностями:

- автоматическое введение избыточности для элементов одного типа (например, всех регистров);
- введение временной избыточности;
- замена элементов памяти на отказоустойчивые аналоги;
- расширение элементов памяти и шин для передачи помехозащищённых кодов;

Также библиотека может включать некоторые элементы анализа для выявления узких мест и оптимальных уровней введения структурной избыточности (системная, модульная и т.п.)

Введение конвейеризации в устройство

В рамках исследования была рассмотрена проблема конвейеризации обработки при помощи разработанного прототипа. Были рассмотрены два класса устройств: простые описания устройства на HDL и устройства, использующие мегафункции Altera с управляемой глубиной конвейеризацией.

В случае с мегафункциями анализ посредством CAP затруднён, так как их исходные генерируются для заданных параметров, и явных механизмов конвейеризации (например, для выделения стадий конвейера в отдельные модули) там не прослеживается. Для модулей вне мегафункций прослеживается иерархия компонентов и видны элементы нижнего уровня: триггеры, регистры и т.п. Возможно разделение устройства на составляющие по циклам тактирования, для которых впоследствии может быть построен конвейер (каждую фазу можно разбить на составляющие).

Текущая функциональность не даёт особых преимуществ при решении задачи введения конвейеризации в устройство. Вероятно, потребуется использовать дополнительное представление, а все реализация алгоритмов анализа остаётся на пользователе.

Реверс-инжиниринг архитектуры устройства

Как уже было сказано, в разработке устройств широко распространено использование сторонних компонентов (IP-модулей). Данные компоненты добавляются при синтезе устройства, поэтому их описания будут включаться в нетлисты, сформированные средством разработки (в т.ч. и Quartus). К IP-модулям часто применяются механизмы защиты интеллектуальной собственности, такие как обфускация, то есть преобразование исходных кодов для затруднения их анализа и преобразования. Также данные модули часто поставляются в виде нетлистов, что дополнительно скрывает внутреннюю архитектуру.

При реинжиниринге устройства с IP-модулями требуется извлечь их архитектуру, и хотелось бы частично автоматизировать данный процесс. При реверс-инжиниринге могут решаться следующие задачи:

- исключение незначимых сигналов, соединений и уровней архитектуры;
- выявление сигналов тактирования;
- выявление элементов памяти: триггеров, регистров и т.п.
- выявление управляющих конечных автоматов;
- определение сложных структур по шаблонам.

Преобразование HDL высокого уровня

В работе в основном рассматриваются такие языки описания устройства, как VHDL и Verilog. Они являются наиболее популярными на текущий момент, но существует ряд более современных и функциональных языков описания устройства. Примерами могут быть SystemC, JHDL, UML-спецификации, описания на базе функциональных языков программирования (Lava, Hydra).

Все перечисленные языки перспективны, но из-за их сложности прямой синтез устройства затруднён. Поэтому, обычно данные описания на данных языках конвертируются в языки низкого уровня, и только потом синтезируются в средствах разработки.

При помощи CAP возможно обеспечить данное преобразование. При этом потребуется обеспечить только считывание описания во внутреннее представление, если модуль для вывода в стандартный язык ранее был реализован. При должном уровне абстракции внутреннего представления подобный двухэтапный перенос может оказаться проще для разработчика, так как алгоритмы ввода-вывода делятся на две более простых стадии.

Рефакторинг описаний

Разработанная архитектура прототипа разрывает связи между входными и выходными представлениями, из-за чего теряется информация об элементах описания, не используемых во внутреннем представлении: комментариях, форматировании текста, файловой организации описания. Это делает разработанный прототип практически бесполезным для рефакторинга, так как на выходе будет получено описание, в котором разработчику потребуется снова разобраться.

Для решения задач рефакторинга посредством CAP предлагается использовать механизм параметров, добавляя в них всю необходимую информацию об исходном представлении устройства. При этом потребуется реализовать отдельные модули ввода-вывода, которые будут использовать данную информацию.

5.6. Перспективы дальнейших исследований

Возможно расширение разработанных методик на смежные области реинжиниринга. Прежде всего, представляют интерес возможности работы с поведенческими описаниями и смешанными цифро-аналоговыми устройствами. Данные области требуют предварительного исследования, предпосылки и задачи которых описаны ниже.

Поддержка полных HDL-описаний

При разработке средства реинжиниринга внутреннее представление устройства было адаптировано под упрощенные структурные описания (нетлисты). Подобный подход позволяет достаточно просто реализовать модуль ввода представлений, но при этом ставит средство в зависимость от среды разработки. Кроме того, при синтезе и генерации нетлистов теряется часть информации, которая могла бы быть использована при реинжиниринге (комментарии, исходные способы описания и пр.). Поэтому, актуален переход к поддержке внутреннего представления, совместимого с полными описаниями на HDL.

В разработанном представлении для поддержки требуется расширить структурное описание и добавить поддержку поведенческого описания.

Наибольшую проблемы в структурном описании являются возможность задания разных архитектур в рамках одного блока. При этом изменения могут касаться не только внутренней архитектуры, но и интерфейсов, что в корне противоречит понятию интерфейса в предложенной методике представления. Проблему представляют следующие элементы описания:

- переменная размерность шин интерфейсов (в зависимости от значения generic-параметра);
- макроподстановки в Verilog (в т.ч. и условная генерация портов);
- условная генерация в VHDL.

Для перечисленных проблем в некоторой степени работает механизм расширения ячеек через наследование; в ряде случаев можно сгенерировать описания устройств для всех комбинаций параметров. Тем не менее, в общем случае существующие подходы непригодны, поэтому актуальна задача доработки представления для поддержки архитектуры, зависимой от значений параметров.

Для поддержки поведенческих описаний требуется расширить внутреннее представление устройства. Опираясь на [28], можно сказать, что достаточно добавить ячейки процесса (Process) и оператора ожидания (Wait). Однако остаются проблемы с распаковкой внутреннего содержимого: циклов, case-селекторов и пр.

Исходя из перечисленных проблем, поддержка совместного структурно-поведенческого описания требует серьёзной проработки, но построение подобного представления вполне возможно.

Совместный реинжиниринг поведенческих и структурных описаний

В средстве автоматизированного реинжиниринга недостаточно просто представить устройство. Требуется также предоставить механизмы для анализа и трансформации архитектуры. В случае, когда внутри средства используются сразу поведенческое и структурное представления, задача реинжиниринга резко усложняется, так как требуется параллельно работать сразу с обоими представлениями, т.е. обеспечить совместный реинжиниринг.

Прежде всего, проблему вызывает анализ связей сигналов в дереве. Если в структурном описании связи фиксированные, то для поведенческого они возникают в зависимости от условий. В итоге, не синтезируя элементов из поведенческого описания, тяжело реализовать совместный анализ, причём для каждой задачи решать проблему придётся пользователю. Надо полагать, что в чистом виде использование двух равноправных описаний невозможно, и требуется неявное приведение одного типа к другому, чтобы поведенческое описание было прозрачным для методов, использующих только структурное.

Поддержка аналоговых и смешанных устройств

Для описания аналого-цифровых устройств существуют специализированные языки описания, например, VHDL AMS (Analog Mixed-Signal) или Verilog AMS. Чаще всего подобные языки являются расширениями HDL для цифровых устройств.

При разработке архитектуры CAP и методики внутреннего представления не было использовано ни одной предпосылки о том, что средство является цифровым. Поэтому, ожидается, что предложенные подходы будет достаточно легко распространить на цифро-аналоговые устройства, но точного вывода требуется провести дополнительные исследования.

В случае перехода к цифро-аналоговым описаниям, скорее всего, потребуется расширить списки параметров сигналов и соединений. Также придётся добавить дополнительные типы сигналов. Перечисленные изменения касаются расширяемых элементов представления, что подтверждает простоту поддержки цифро-аналоговых описаний.

ЗАКЛЮЧЕНИЕ

Автоматизация реинжиниринга устройств управления является одним из перспективных направлений в области автоматизации процессов разработки. В современном мире подобные задачи решаются при помощи построения программных приложений. При этом возникают проблемы выбора реализуемой функциональности, внутреннего представления, архитектуры самого средства.

На этапе предварительного исследования оказалось, что в настоящее время фактически отсутствует такая область, как автоматизация реинжиниринга устройства. Мало рассматриваются даже более общие задачи реинжиниринга системы, хотя на практике подобные задачи решаются постоянно. Поэтому в работе много места уделено именно общим вопросам реинжиниринга.

Были рассмотрены общие вопросы реинжиниринга системы: понятие, его место в жизненном цикле системы, проблемы автоматизации процесса. Применительно к устройствам была рассмотрена постановка задачи реинжиниринга, типовые примеры частных задач. На примере была показана актуальность автоматизации процесса реинжиниринга и, в частности, трансформации архитектуры устройства.

В работе был проведён исследование существующих средств поддержки реинжиниринга. Выявлено, что большинство из них решают частные задачи: анализ устройств, рефакторинг или однотипные преобразования. В то же время проблема реализации пользовательских алгоритмов трансформации устройств существующими средствами не решается. Исключением является средство реинжиниринга устройства DMS Software Reengineering Toolkit, но оно не учитывает специфики самого устройства.

Для решения проблемы в работе предложено строить программируемое средство автоматизации реинжиниринга (CAP). При этом требуется обеспечить интеграцию реинжиниринга с процессами разработки устройства, так с их помощью можно решить большинство частных задач. На основании данных требований были сформированы требования к архитектуре средства, методике внутреннего представления и языку управления данными представлениями.

Обзор существующих методик представления не выявил методики, удобной одновременно для анализа устройства и его трансформации. Была разработана своя методика представления структуры устройства, основанная на архитектурном графе. Удалось уложиться в небольшое число базовых типов ячеек, расширение которых возможно через механизмы наследования, ссылки и параметры. Также был предложен функционально полный набор команд для преобразования устройства.

В соответствии с требованиями была разработана архитектура средства реинжиниринга. Ключевой её особенностью является модульная архитектура с

ядром средства, реализующим минимальную функциональность, и поддержкой расширений, через которые может быть реализована всё остальное: от алгоритмов управления преобразованиями до API и модулей ввода-вывода для любых описаний устройства.

За счёт предлагаемой архитектуры пользователь может не только реализовать произвольные алгоритмы реинжиниринга устройства, но и включить средство в любые другие процессы разработки, что расширяет возможности его использования. Единственным ограничением является внутреннее представление

Для подтверждения применимости разработанных решений в работе было решено разработать прототип программируемого САР. Из-за большого объёма разработки пришлось отказаться от реализации всех предложений, реализовав ядро средства и минимально необходимую обвязку: модуль ввода-вывода представления устройства и примитивный пользовательский интерфейс.

В качестве средства разработки устройства для интеграции с прототипом была взята среда Quartus II компании Altera. В качестве входного формата данных выбраны VHDL-нетлисты, а выходного – синтезируемые структурные описания на подмножестве VHDL. Основной причиной выбора стала возможность расширения модуля для поддержки полного VHDL.

Был разработан модуль вывода с использованием библиотеки vhdl_ast, разработанной на кафедре КСПТ. При разработке оказалось, что формат VHDL-нетлистов оказался достаточно сложным для обработки, а vhdl_ast содержала в себе много ошибок и недоработок. Поэтому, разработка модуля одного лишь ввода-вывода представлений потребовала больше времени, чем разработка самого ядра средства.

Для апробации средства был разработан ряд библиотек, решающих частные задачи реинжиниринга, благодаря чему подтверждена возможность расширения средства. К сожалению, из-за задержки в реализации прототипа не хватило времени на его апробацию для решения комплексных задач (например, синтеза отказоустойчивого процессора).

В работе был проведён полный цикл реинжиниринга на примере задачи внесения структурной избыточности. К сожалению, из-за ошибок и недоработок в модуле ввода-вывода не удалось полностью автоматизировать генерацию VHDL-кода. Тем не менее, результаты показали, что предлагаемый подход подходит для автоматизации задач анализа и структурных преобразований устройства по пользовательским алгоритмам

При разработке прототипа создан ряд инструментов, направленных на упрощение разработки расширений для САР. В частности, был разработан модуль для NetBeans с шаблонами основных пользовательских классов,

достаточно подробная документация на исходные коды. Также была развёрнута система управления проектами на базе Redmine. Хоть она и оказалась невостребованной из-за низкой активности прочих участников проекта, но в будущем возможно её использование для организации взаимодействия с пользователями средства.

В завершении работы был проведён анализ соответствия результатов разработки поставленным требованиям. Сделано заключение о том, что и предлагаемая архитектура, и прототип соответствуют общим требованиям к САР по программируемости, расширяемости и встраиваемости. Для частных требований анализ выявил ряд отклонений, которые могут быть устранены при дальнейшей разработке. Также при разработке прототипа были выявлены новые проблемы, прежде всего связанные с контролем обеспечением целостности дерева элементов и удобства совместной работы многих библиотек. В работе предложены пути по решению данных проблем, что может быть дальнейшим предметом разработки.

Также развитием работы может быть введение полноценных механизмов наследования элементов в представлении, введение полноценного внутреннего API и более функционального языка программирования. Для прототипа требуется полностью реализовать предлагаемую архитектуру и методику представления. Актуальной остаётся задача перехода к полноценной поддержке HDL в средстве. Можно также расширять функциональность прототипа за счёт разработки дополнительных библиотек ядра. Если прототип покажет свою эффективность на практических задачах, то возможна его доработка до полноценной САПР.

Для дальнейшего исследования предлагаются также методики представления и реинжиниринга поведенческих описаний, методик их переноса в структурные описания. Отдельной задачей является обеспечение совместного реинжиниринга структурных и поведенческих форм представления устройства. Рассмотрение этих вопросов является необходимым этапом для поддержки современных языков описания устройства (VHDL, Verilog, SystemC) и перспективных UML-описаний на уровне внутреннего представления средства реинжиниринга.

В любом случае, автор диссертации планирует продолжить разработку прототипа САР и довести его до практического применения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахтырченко К.В., Сорокваша Т.П. Методы и технологии реинжиниринга ИС // Институт Системного Программирования РАН. 2003. 15 с.

2. Бадд Т. Объектно-ориентированное программирование в действии. СПб: Питер, 1997. 464 с.

3. Васильев А.Е. Микроконтроллеры. Разработка встраиваемых приложений. СПб: БХВ-Петербург, 2008. 302 с.

4. Поляков А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. Солон-Пресс, 2003. 320 с.

5. Шестаков А.В. Реинжиниринг // Энциклопедический словарь экономики и права. 2000. с. 568.

6. Яремчук С.В. Обзор систем управления проектами // Системный администратор. 2010. № 5. 37-49 с.

7. Akehurst D.H. и др. Compiling UML State Diagrams into VHDL: An Experiment in Using Model Driven Development // ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Genova, Italy (submitted).

8. Alfred V., Sethi R., Jeffrey D.U. Compilers: Principles, Techniques and Tools. Addison-wesley, 1986. 870 с.

9. Altera Corporation. Introduction to the Quartus® II Software v10.0 // 2010.

10. Arnold R. Software reengineering. Los Alamitos Calif.: IEEE Computer Society Press, 1993. 645 с.

11. Bergey J. и др. A Reengineering Process Framework. Pittsburgh: Software Engineering Institute, 1995. 840 с.

12. Carriere S.J., Woods S., Kazman D. Software Architectural Transformation. Pittsburgh: Software Engineering Institute, 1999. 820 с.

13. Chikofsky E.J., Cross J.H. Reverse engineering and design recovery: A taxonomy // IEEE software. 1990. 13-17 с.

14. Eusgeld I. и др. Hardware Reliability // Dependability Metrics / под ред. I. Eusgeld, F.C. Freiling, R. Reussner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. с. 59-103.

15. Gonzalez I. и др. Classification of Application Development for FPGA-Based Systems // Aerospace and Electronics Conference, 2008. NAECON 2008. IEEE National. с. 203–208.

16. Hammer M. Reengineering the corporation: a manifesto for business revolution. New York NY: Harper Business, 1994. вып. 1. 680 с.

17. Hennessy J., Patterson D. Computer organization and design : the hardware software interface. San Francisco Calif.: Morgan Kaufmann Publishers, 1998. вып. 2nd ed. 751 с.

18. Jerraya A. Behavioral synthesis and component reuse with VHDL. Boston: Kluwer Academic Publishers, 1997. 265 с.

19. Kaiping Z., Huss S.A. RAMS: A VHDL-AMS Code Refactoring Tool Supporting High Level Analog Synthesis // IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05). Tampa, FL, USA. с. 266-267.

20. Kozen D. Theory of computation. London: Springer, 2006. 429 с.

21. Project Management Institute. A guide to the project management body of knowledge (PMBOK® Guide). Newtown Square Pa.: Project Management Institute, 2008. вып. 4th ed.

22. Rosenberg D. Agile development with ICONIX process : people, process, and pragmatism. Berkeley CA: Apress, 2005. 570 с.

23. Rosenberg L.H., Hyatt L.E. Software re-engineering // Unisys Federal Systems. 1998. 17 с.

24. Schürr A. Reengineering & Refactoring with Graph Transformations // TU Sarmstadt. 2004. 16 с.

25. Singh R. International Standard ISO/IEC 12207 software life cycle processes // Software Process Improvement and Practice. 1996. т. 2. 35-50 с.

26. Wawrzynek J. Components and Design Techniques for Digital Systems. Lection 6: Hardware Description Languages // Berkeley University. 2003. 42 с.

27. Willis J. AIRE/CE. Internal Intermediate Representation (IIR) Specification Version 4.6 Including Digital VHDL & VHDL-AMS support // University of Cincinnati. 1999. 573 c.

28. Wilsey P.A., Benz D.M., Pandey S.L. A model of VHDL for the analysis, transformation, and optimization of digital system designs. University of Cincinnati, 2002. c. 611–616.

29. Баринов В.А. Реинжиниринг: сущность и методология [Электронный ресурс]. URL: http://www.ippnou.ru/article.php?idarticle=002369 (дата обращения: 09.05.2011).

30. Ненашев О.В. UML-спецификация на прототип средства реинжиниринга // 2010.

31. Ненашев О.В. Документация на исходные коды прототипа средства реинжиниринга // 2011.

32. Ненашев О.В. Сайт проекта по разработке средства автоматизированного реинжиниринга устройства [Электронный ресурс]. URL: https://nenhome-apps.sourcerepo.com/redmine/nenhome/ (дата обращения: 29.05.2011).

33. Орлик С. Инструменты и методы программной инженерии// Основы Программной Инженерии (по SWEBOK) [Электронный ресурс]. URL: http://swebok.sorlik.ru/9_software_engineering_tools_and_methods.html (дата обращения: 09.05.2011).

34. AMIQ company. DVT System Verilog User Guide [Электронный ресурс]. URL: http://www.dvteclipse.com/help.html?documentation/sv/Tips_and_Tricks.html (дата обращения: 22.05.2011).

35. Bartsch G., Holst S. The SIGNS project [Электронный ресурс]. URL: http://signs.sourceforge.net/ (дата обращения: 28.05.2011).

36. Cadence Design Systems, Inc. Cadence Products A - Z [Электронный ресурс]. URL: http://www.cadence.com/products/Pages/all_products.aspx (дата обращения: 24.05.2011).

37. EDIF overview// Elgris Technolodgies [Электронный ресурс]. URL: http://www.elgris.com/content/edif_overview.html (дата обращения: 15.05.2011).

38. Brigham Young University. FPGA Reliability Studies - BYU EDIF Tools [Электронный ресурс]. URL: http://reliability.ee.byu.edu/edif/ (дата обращения: 08.06.2011).

39. Lamb M. JSAP: The Java Simple Argument Parser [Электронный ресурс]. URL: http://www.martiansoftware.com/jsap/ (дата обращения: 08.06.2011).

40. Rosenberg D. Agile development with ICONIX process : people, process, and pragmatism. Berkeley CA: Apress, 2005.

41. SAMATE team. Tool Taxonomy// SAMATE project [Электронный ресурс]. URL: http://samate.nist.gov/index.php/Tool_Taxonomy.html (дата обращения: 17.05.2011).

42. Semantics Designs inc. DMS Software Reengineering Toolkit [Электронный ресурс]. URL: http://www.semdesigns.com/Products/DMS/DMSToolkit.html (дата обращения: 04.11.2010).

43. Sigasi nv. Sigasi HDT for VHDL [Электронный ресурс]. URL: http://www.sigasi.com/sigasi-hdt (дата обращения: 22.05.2011).

44. Venners B. Refactoring with Martin Fowler [Электронный ресурс]. URL: http://www.artima.com/intv/refactor.html (дата обращения: 04.11.2010).

ПРИЛОЖЕНИЕ А. ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ VHDL-НЕТЛИСТОВ В СРЕДЕ QUARTUS II И ПРОТОТИПЕ

В данном приложении рассмотрены вопросы использования VHDL-нетлистов, которые были выбраны в средстве реинжиниринга устройства в качестве входного формата описания. Перечислены параметры нетлистов, используемые в прототипе средства реинжиниринга. Описания в данном пункте ориентированы на версии 9 и 10 среды Quartus II.

1. Включение нетлистов в процесс разработки Quartus II

При разработке прототипа требовалось интегрировать его с внешней средой разработки, чтобы использовать её возможности по синтезу и анализу устройства. На рис. 1. приведён пример процесса разработки в среде Quartus II. Он допускает взаимодействие (в т.ч. автоматическое) со различными группами средств разработки, поэтому возможно связать его с различными средствами реинжиниринга.

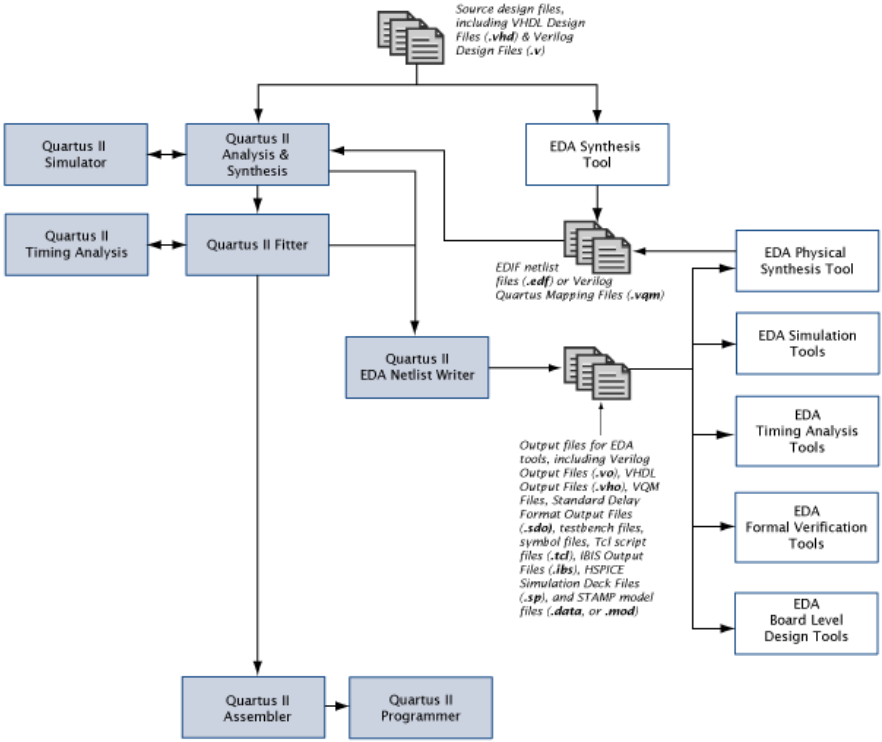

Рис. 1. Этапы процесса разработки устройства в среде Quartus II с использованием сторонних средств разработки [9]

Среда Quartus II поддерживает некоторое количество входных и выходных форматов. Для прототипа средства реинжиниринга в качестве входного формата решено взять VHDL-нетлисты, а выходного – структурное подмножество VHDL. Это позволяет использовать схему с двойным синтезом, предложенную в пункте 1.4.

Надо отметить, что для различных типов внешних EDA-средств IDE Quartus II поддерживает различные форматы выходных файлов, что связано с ориентацией САПР на уже существующие средства разработки. Поэтому, генерация VHDL-нетлистов возможна только для средств симуляции, и конфигурируется из соответствующей панели. При этом функциональность средства не ограничивается (т.е. возможно не только моделирование).

Настройки генерации VHDL-нетлистов

Среда Quartus II поддерживает различные варианты формирования VHDL-нетлистов. В таблице 1 описаны основные настройки генерации выходных данных для симулятора, которые можно задать в меню Settings/EDA Tools/Simulation. Приведены только настройки, влияющие на структурное описание устройства. Настройки симуляции в таблице не приведены. Некоторые параметры будут подробно рассмотрены далее.

Таблица 1

Настройки формирования VHDL-нетлистов в среде Quartus II

Параметр	Тип	Описание
Format for output netlist	Список	Формат выходного нетлиста. Для генерации VHDL-нетлистов требуется выбрать
Architecture name in VHDL	Строка	Имя архитектуры в выходном файле
Maintain hierarchy	On/Off	Параметр задаёт, требуется ли отображать в нетлисте иерархию элементов устройства. Форматы нетлистов описаны в п. 2 приложения
Map illegal HDL characters	On/Off	Исключение из имени нетлистов символов, не поддерживаемых стандартом VHDL.*
Bring out device-wide set/reset signals as ports	On/Off	Добавление глобальных сигналов запуска и перезапуска в порты устройства
Flatten busses into individual nodes	On/Off	Развёртывание шин сигналов в отдельные ячейки. Также влияет на добавление агрегаций в описания присвоений сигналов.
Don’t write top-level HDL entity	On/Off	Запрет вывода Entity для корневого элемента
Truncate long hierarchy paths	On/Off	Ограничение длины путей по иерархии в именах ячеек

Для задания путей в уровнях иерархии расширяются имена ячеек. При этом, для разделения уровней иерархии используются специальные метки (см. примеры в табл. 2). В случае, если включена опция “Map illegal HDL characters” все символы заменяются на “_a”. Это делает устройство потенциально синтезируемым, но исключает возможность анализа, так как комбинации “_a” могут встречаться и в именах.

Таблица 2

Примеры объявлений в нетлистах ячеек при различных значениях опции “Map illegal HDL characters”

Значение опции	Имя элемента в объявлении	Комментарий
On	SIGNAL \counter|s[1]~6\ : std_logic;	Объявление сигнала
	\counter|s[2]~7\ : cycloneii_lcell_comb	Добавление компонента
Off	SIGNAL counter_as_a1_a_a6 : std_logic;	Объявление сигнала
	counter_as_a2_a_a7 : cycloneii_lcell_comb	Добавление компонента

2. Форматы VHDL-нетлистов

VHDL-нетлисты представляют собой некоторое подмножество описаний языка описания устройства VHDL. При этом, форматы данных файлов не полностью соответствуют стандартам языка, поэтому синтез нетлистов данного типа в среде Quartus II не всегда возможен.

Нетлисты могут включать:
- использование библиотеки IEEE и пакета std_logic_1164;
- объявление библиотек и пакетов, содержащих компоненты для ячеек нижнего уровня языка;
- объявления типов внутри пакетов (PACKAGE);
- описания сущностей, используемых в нетлисте:
- описания архитектур сущностей.

В сущностях и архитектурах могут использоваться сигналы и порты могут иметь типы std_logic, std_logic_vector или типы на их основе, определённые внутри нетлиста. Описания сущностей могут содержать не только порты, но и generic-параметры.

В архитектурах допускается использование лишь некоторого подмножества структурного описания устройств. При этом, в архитектуре могут быть использованы:
- объявления компонентов;
- присвоения как сигналов и их шин;
- использование агрегаций в присвоениях;
- доступ к элементам шин по индексу;
- использование оператора NOT.

Плоские и иерархические нетлисты

Фактически, Quartus II поддерживает генерацию нетлистов в двух форматах: с добавлением архитектуры и без неё. Данные типы нетлистов соответственно называются иерархическими и плоскими.

В иерархических нетлистах каждый модуль устройства представляет собой отдельную сущность (entity) и архитектуру, которые связываются без дополнительного объявления компонентов. Данное описание наиболее близко к чистому VHDL, но имеется ряд отличий, который исключает прямой синтез:

- допускаются соединения между сигналами различных уровней иерархии;
- допускаются глобальные сигналы.

Плоские нетлисты являются более простой версией, где всё описание устройства приводится в виде одной entity и одной архитектуры. При этом в объявлениях сигналов и компонентов указывается полный путь в исходной иерархии, что позволяет восстановить её во время импорта.

3. Поддержка VHDL-нетлистов в модуле ввода-вывода прототипа

Разработанный прототип средства реинжиниринга поддерживает далеко не все комбинации приведённых выше настроек. В таблице 3 приведены списки настроек, поддерживаемых в прототипе.

Таблица 3

Требуемые настройки генерации нетлистов для прототипа САР

Параметр	Тип	Описание
Format for output	Список	Всегда VHDL
Architecture name	Строка	произвольно имя
Maintain hierarchy	On/Off	Поддерживаются и плоские (Off), и иерархические (On) нетлисты.
Map illegal HDL characters	On/Off	Всегда Off. Спец. символы используются для распознавания уровней иерархии.
Bring out device-wide set/reset signals as ports	On/Off	Значение м.б. любым, так как предусмотрено в модуле ввода-вывода прототипа предусмотрено исключение несинтезируемых сигналов devoe, devclrn и devpor.
Flatten busses into individual nodes	On/Off	Значение м.б. любым, там как модуль ввода-вывода поддерживает преобразование шин и агрегаций в группы.
Don’t write top-level HDL entity	On/Off	Должен быть off, иначе не будет считан интерфейс корневой ячейки.
Truncate long hierarchy paths	On/Off	Должен быть off, иначе восстановление архитектуры окажется невозможным.

Кроме перечисленного, в текущей реализации модуль ввода-вывода не поддерживает следующие возможности:

- загрузку определений компонентов из указанных библиотек;
- обработку описаний внутри пакетов;
- обработку объявлений пользовательских типов.

В настоящей версии прототипа при загрузке игнорируются указания на используемые библиотеки элементов, и по-умолчанию всегда используется cyclone_ii, т.е. поддерживаются только ПЛИС семейства Cyclone II, производимые фирмой Altera.

Прочие проблемы возникали только при импорте устройств, использующих мегафункции Altera, генерируемые средой Quartus II. Для сложных устройств (в т.ч. микропроцессорных ядер) в нетлистах объявления данных компонентов не встречались, поэтому проблема с поддержкой компонентов была выявлена лишь на этапе тестирования.

Перечисленные выше ограничения относительно легко устранить, так как разработанная методика представления поддерживает все необходимые механизмы для работы с библиотеками и типами. Доработка модуля ввода-вывода является одной из перспективных задач дальнейшей разработки, необходимой для реализации полноценной САПР.

ПРИЛОЖЕНИЕ В. КРАТКАЯ ИНФОРМАЦИЯ О ПРОЕКТЕ РАЗРАБОТКИ ПРОТОТИПА

В данном приложении приведена информация о проекте разработки прототипа средства реинжиниринга. Рассмотрены вопросы организации проекта, перечислены используемые средства разработки с указанием их применения,

1. Организация проекта

В разработке прототипа программного средства принимали участие несколько студентов и преподавателей кафедры КСПТ, в том числе и автор данной магистерской диссертации. В таблицах ниже приведены классификация проекта в соответствии с руководством Института Управления Проектами (PMI), список участников проекта и информация об информационных ресурсах.

Таблица 1

Классификация проекта по PMI

Параметр	Значение
Тип	Технический
Класс	Мультипроект
Масштаб	Средний
Вид	Смешанный (инновационно-исследовательский)
Длительность	Краткосрочный
Организационная форма	Матричная
Модель жизненного цикла	Спиральная
Организационная форма	Матричная
Стиль руководства	Консультативное принятие решений
Методологии разработки	Scrum, ICONIX

Таблица 2

Участники проекта

Участник		Выполняемые задачи
Максименко С.Л.	каф. КСПТ, ст. преп.	Общее руководство проектом
Филиппов А.С.	каф КСПТ, доцент	Общее руководство проектом
Ненашев О.В.	каф. КСПТ, магистрант	- Разработка архитектуры CAP, методик представления и трансформации; - Реализация прототипа средства; - Администрирование ресурсов проекта;
Егоров И.В.	магистрант, каф. КСПТ	- Доработка библиотеки ядра средства по анализу устройства.

Информационные ресурсы проекта

Для организации взаимодействия между участниками использовалась система управления проектами Redmine. На её базе было обеспечено хранение всей информации по проектам:

- репозитории исходных кодов на базе Mercurial;
- файловое хранилище;
- управления задачами и ошибками;

Прочие возможности Redmine на настоящий момент остаются невостребованными.

2. Средства разработки прототипа

Таблица 3

Используемые средства разработки

Средство	Решаемые задачи
Средства разработки	
NetBeans IDE 6.9	- программная реализация прототипа; - тестирование и отладка прототипа; - пользовательский модуль с шаблонами для библиотек и классов;
Quartus II 10	- разработка тестовых примеров для CAP; - разработка модулей для пользовательской библиотеки повышения надёжности; - генерация нетлистов для прототипа.
ModelSim	- моделирование устройств; - проверка корректности работы устройства, сгенерированного CAP устройства
Enterprise Architect 8	- построения частичной спецификации на средство;
Организация взаимодействия	
Mercurial	- версионирование исходных кодов;
Redmine	- отслеживание ошибок в программной реализации; - управление задачами; - хранилище файлов; - wiki проекта.
Средства документирования	
MS Office 2003	- подготовка программной документации;
NetBeans Javadoc	- генерация документации на исходные коды в стандарте Javadoc.
Doxygen	- генерация документации на исходные коды; - построение диаграмм связей по исходным кодам.
Zotero	- хранение и систематизация библиографии.

Используемые языки разработки

Таблица 4

Используемые языки разработки

Язык	Решаемые задачи
Java	- программная реализация прототипа; - реализация дополнительных библиотек ядра;
VHDL	- реализация устройств для тестирования средства: - реализация специализированных блоков для пользовательских библиотек.
UML	- разработка частичной спецификации на прототип:
bash-подобные скрипты	- внешние программы управления преобразованиями; - представление истории команд в CAP;
Tcl	- тестовые программы взаимодействия со средой Quartus II;

Используемые библиотеки Java

При программной реализации прототипа были использованы различные библиотеки языка Java. Ниже приведён список использованных библиотек с указанием путей их применения в разработанном средстве.

JavaCC:
- конвертация VHDL-нетлистов в АСД;
- импорт пользовательских библиотек в АСД;
- генерация VHDL-кода по АСД, который построен модулем ввода-вывода.

JUnitTest 4:
- модульное тестирование основных классов программной реализации;
- функциональное тестирование через скриптовые программы.

JSAP:
- парсинг командной строки в консольном интерфейсе;
- интерпретатор команд для ядра CAP.

Swing:
- графический интерфейс для средства реинжиниринга.

3. Программная реализация прототипа

Структура проектов

Для улучшения переносимости кода вся разработка была разбита на несколько проектов. Краткое описание проектов приведено в таблице 5.

Таблица 5

Список проектов в программной реализации прототипа

Проект	Описание
object_lib	Проект с ядром CAP и всеми базовыми библиотеками ядра (в перспективе требуется разнести на проекты)
ObjectConsole	Консольный интерфейс для прототипа CAP
ObjectViewer	Графический интерфейс для прототипа CAP
vhdl_ast	Проект, созданный для доработки библиотеки vhdl_ast
OBL_templates	Проект модуля NetBeans с шаблонами для формирования функций и библиотек ядра.

Статистика по исходным кодам

Статистика по исходным кодам была собрана при помощи программы cloc (cloc.sourceforge.net) и приведена в таблице 6. Как видно из таблицы, ядро средства реинжиниринга, будучи основной целью разработки, вместе с системными библиотеками занимает всего половину от общего размера исходных кодов.

Таблица 6

Статистика по размеру исходных кодов в прототипе средства реинжиниринга

Модуль	Число файлов	Размер исходных кодов, строк			
		Код	Ком.	Разд.	Всего
OBL core	127	11514	5817	1650	18981
Библиотеки	37	13611	4661	1743	20015
-- SystemLib	23	1319	313	252	1884
-- BasicLib	15	1517	495	246	2258
-- StatisticsLib	14	1014	237	130	1381
-- ReliabilityLib	12	1571	364	108	2043
-- Библиотеки ввода-вывода	28	8190	3252	1007	12449
Тесты	25	820	76	340	1236
Пользовательские интерфейсы	59	6425	1097	1443	8965
-- Obl_Console	1	108	45	58	211
-- Obl_GUI	58	6317	1052	1385	8754
Netbeans module	4	170	162	53	385
Библиотеки элементов (VHDL)	27	2378	324	45	369
Всего	248	32540	12137	5274	49951

ПРИЛОЖЕНИЕ С. ПРИМЕРЫ ДИАГРАММ ИЗ UML-СПЕЦИФИКАЦИИ НА ПРОТОТИП СРЕДСТВА РЕИНЖИНИРИНГА

В данном приложении приведены примеры диаграмм, взятых из UML-спецификации на прототип средства реинжиниринга. Следует отметить, что разработка прототипа велась в несколько итераций, а спецификация была составлена только на первую из них, поэтому в диаграммах присутствуют некоторые отклонения от фактической реализации прототипа.

1. Основные диаграммы спецификации

Требования к устройству и use-case

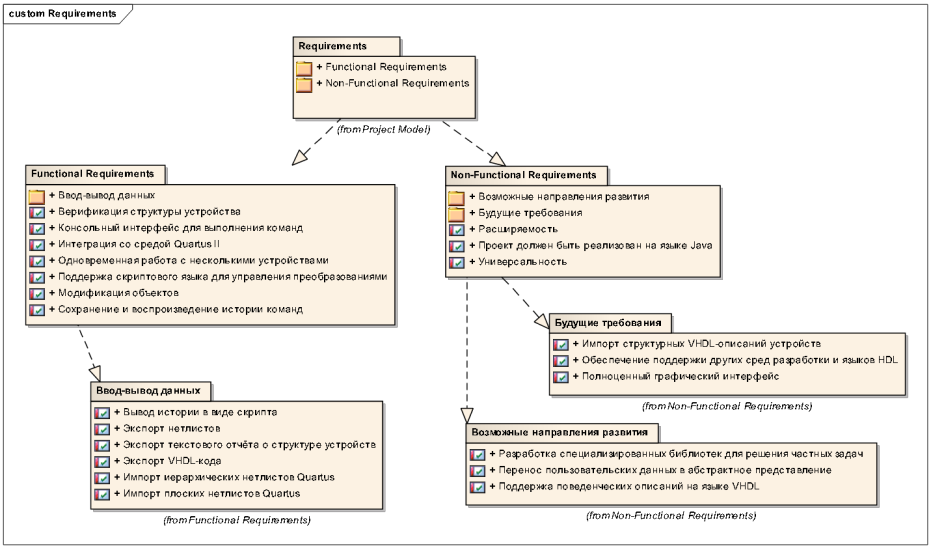

Рис. 1. Основные требования к средству реинжиниринга

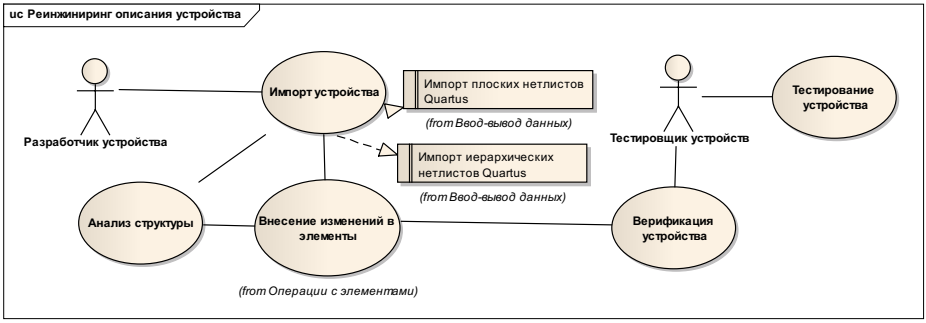

Рис. 2. Порядок реинжиниринга в прототипе

Use-case диаграммы

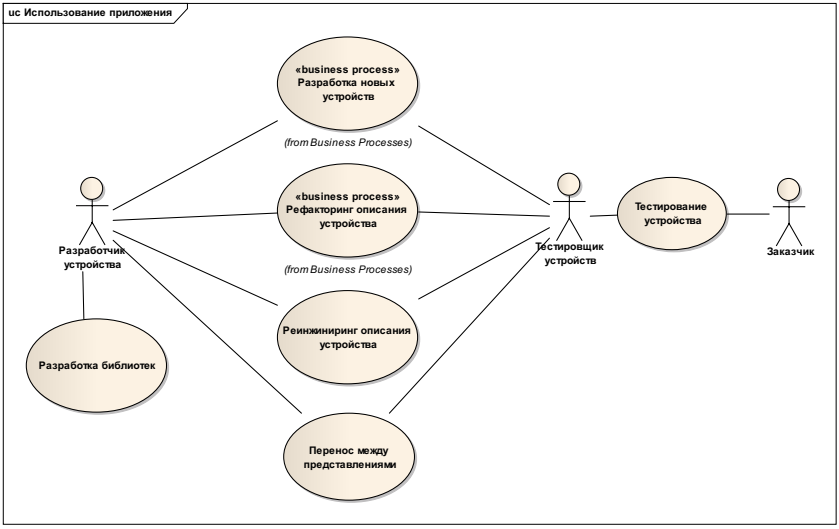

Рис. 3. Использование средства в различных задачах реинжиниринга

Структура проектов

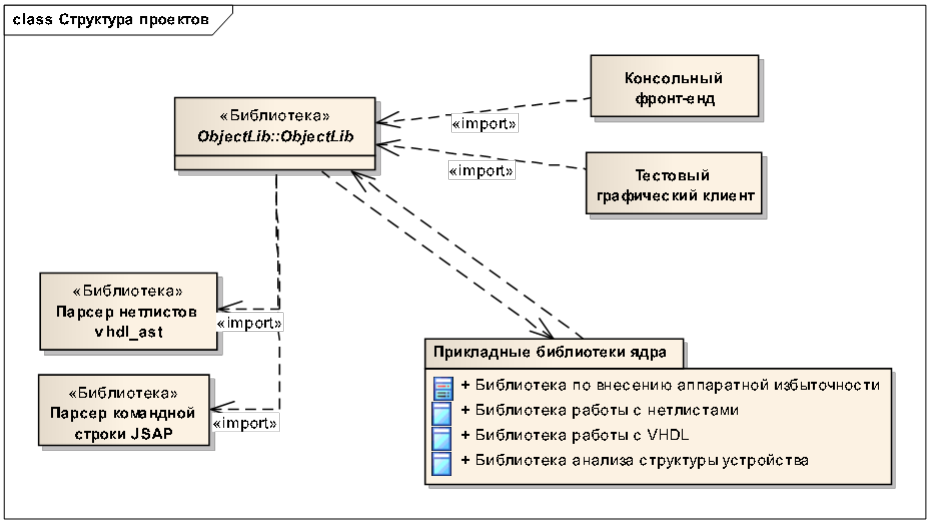

Рис. 4. Структура элементов САР

Структуры программных модулей

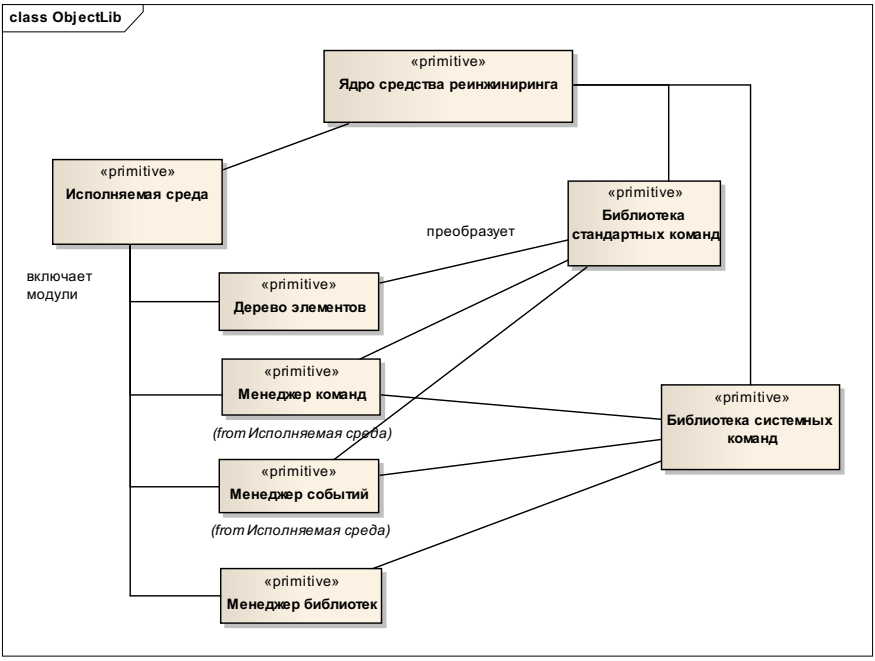

Рис. 5. Структура ядра CAP

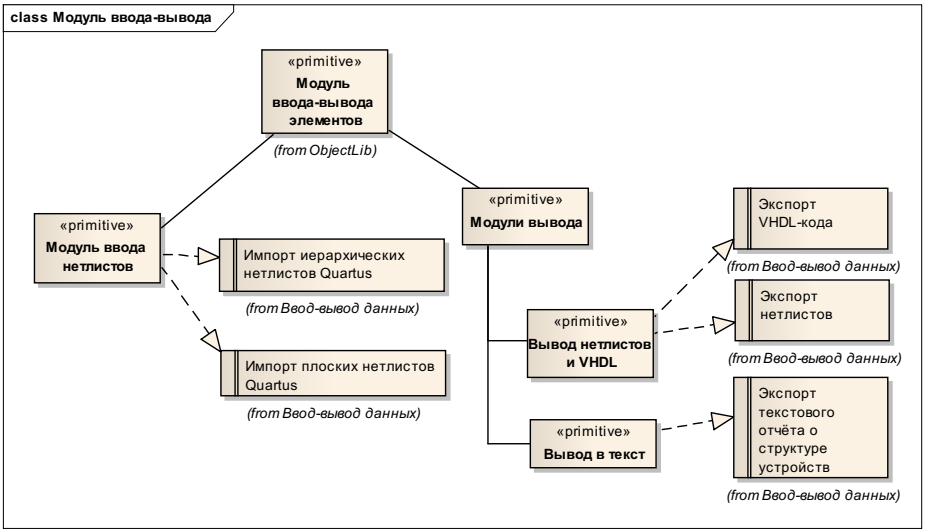

Рис. 6. Структура модуля ввода-вывода VHDL

Диаграммы классов программных модулей

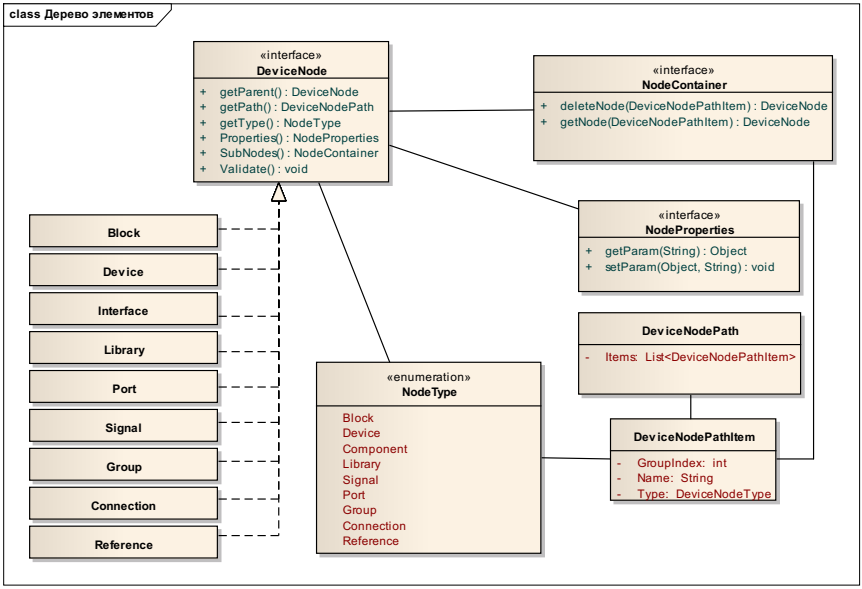

Рис. 7. Диаграмма классов дерева элементов

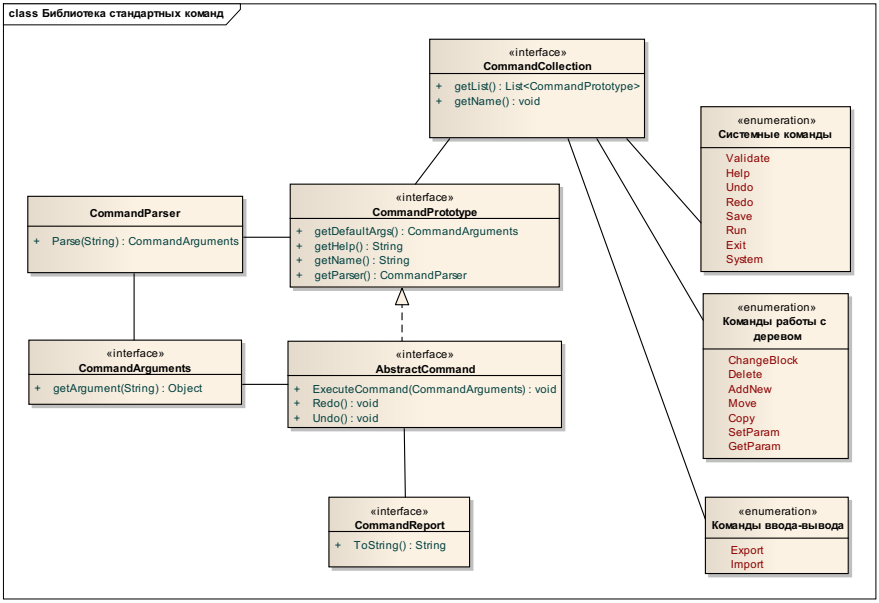

Рис. 8. Диаграмма классов работы с командами в ядре САР

Диаграммы операций

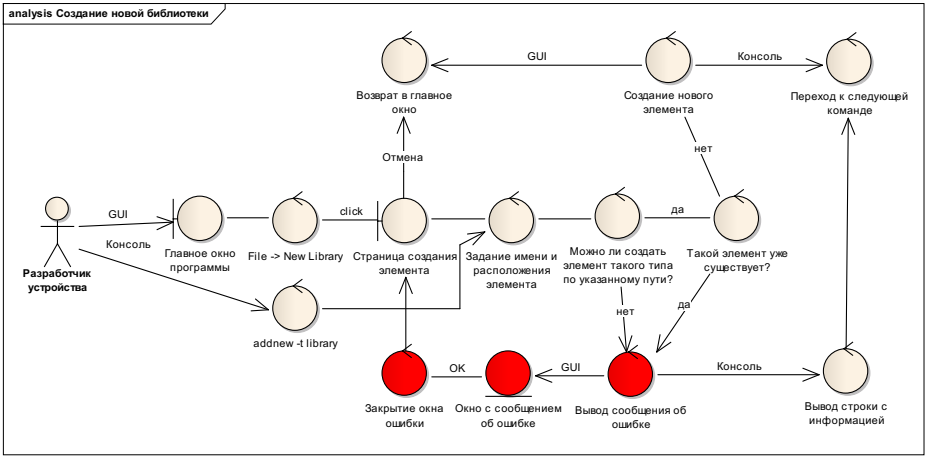

Рис. 9. Последовательность действий при добавлении библиотеки ядра в средство

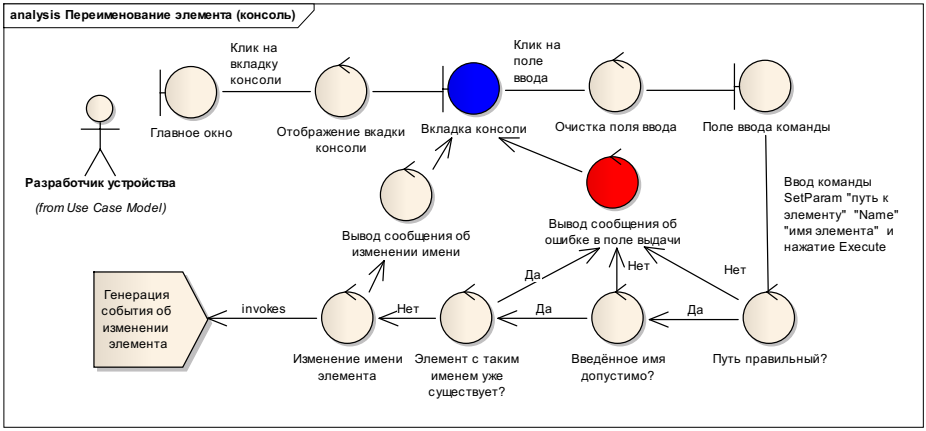

Рис. 10. Последовательность действий при переименовании элемента

2. Эскизы окон графического интерфейса

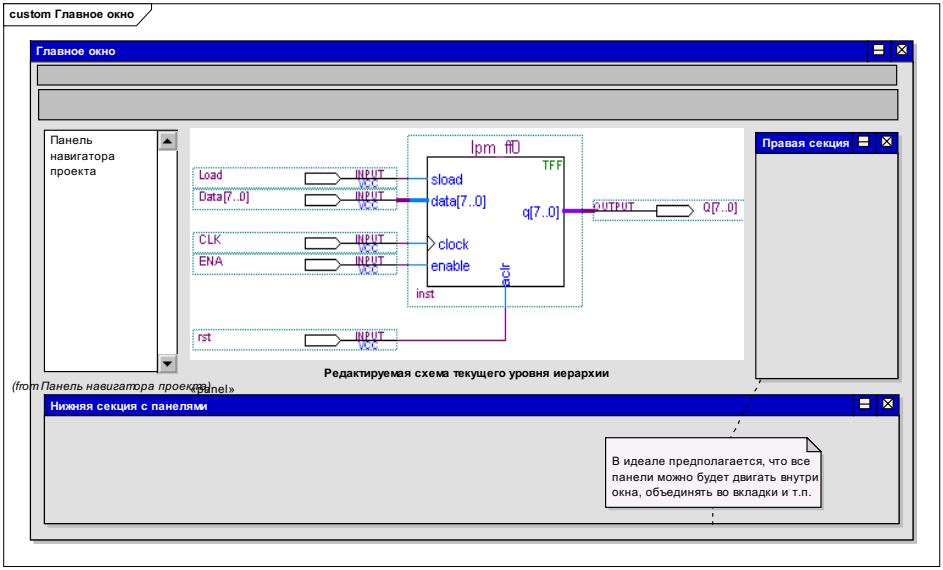

Рис. 11. Эскиз главного окна

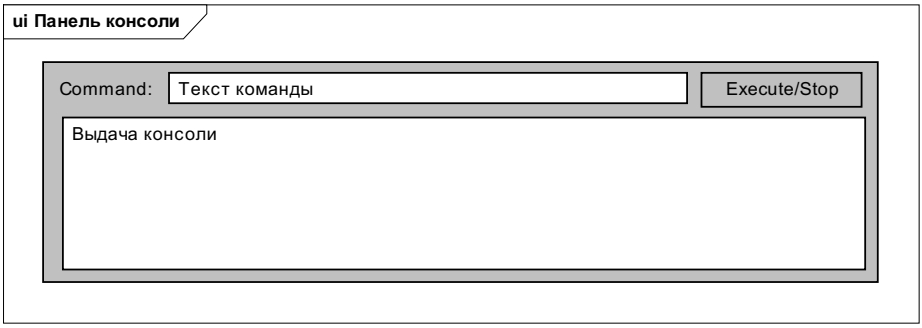

Рис. 12. Эскиз панели консоли

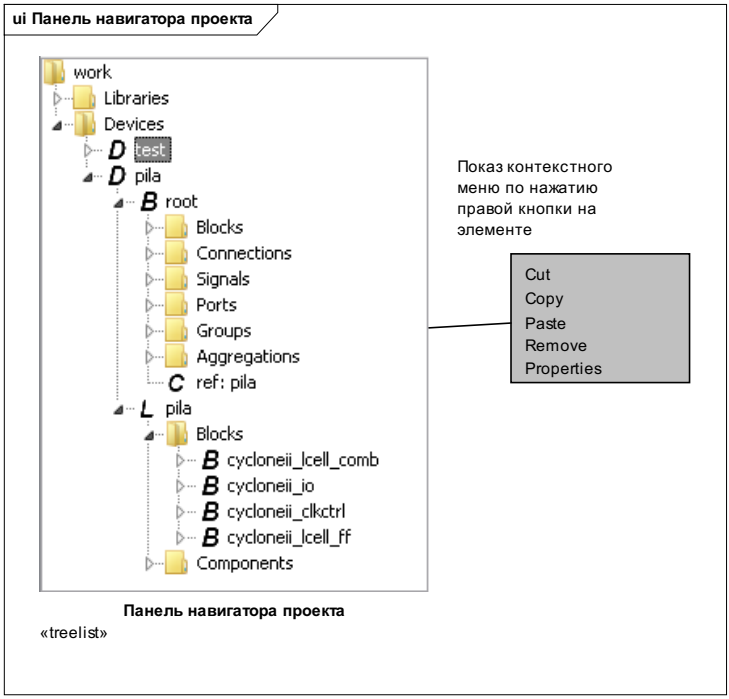

Рис. 13. Эскиз панели дерева элементов

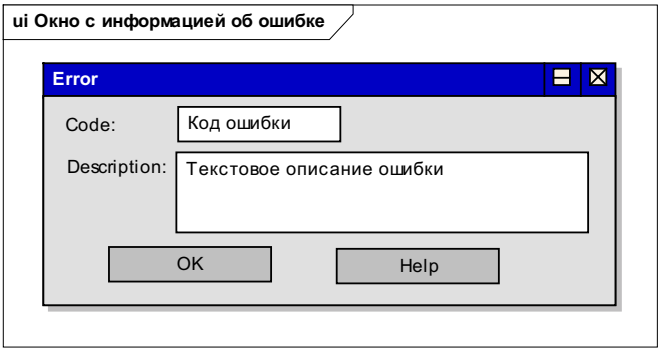

Рис. 14. Эскиз окна с информацией об ошибке

ПРИЛОЖЕНИЕ D. ОПИСАНИЕ СИСТЕМЫ КОМАНД В ПРОТОТИПЕ СРЕДСТВА РЕИНЖИНИРИНГА

В данном приложении приведено краткое описание библиотек ядра, реализованных в прототипе средства реинжиниринга. Команды из библиотек описываются частично.

1. Внутренние библиотеки ядра

1.1. Библиотека системных команд SystemLib

Библиотека SystemLib включает команды управления средсвом реинжиниринга. В данную группу входят команды запуска программ, работы с библиотеками историей команд, а также команды выхода справки и выхода из средства реинжиниринга.

Таблица 1
Список команд в библиотеке SystemLib

Команда	Описание	Флаги*		
		M	L	U
help, man, ?	Вывод справочной информации о команде или списка команд	-	-	+
undo	Отмена последнего действия, если возможно	+	-	-
redo	Повторение последнего отменённого действия	+	-	+
history	Отображение истории команд	-	-	-
save	Сохранение истории команд в скрипт по указанному пути	-	-	-
run	Запуск скрипта	+	+	-
quit, exit	Выход из программы	+	+	-
system	Выполнение системной команды через shell windows	-	+	-
clibs_info	Вывод информации о доступных и загруженных библиотеках	-	-	+
clibs_load	Загрузка библиотеки	+	+	-
* Флаги M – модифицирует контекст, L – сохраняется в истории, U – м.б. отменена командой undo (для команд модификации)				

Команда вывода справки

Команда вывода справки предназначена для вывода информации о доступных командах средства и информации об отдельных командах. Кроме того, планируется добавление в команду базовой информации о механизмах адресации и прочих элементах.

Команда help принимает один аргумент – имя команды. При отсутствии аргумента выводится список доступных команд из загруженных библиотек, иначе – справка по команде в следующем формате:

Листинг 1. Пример вывода справки по команде help

```
[R]root\>>help setparam
Sets new value for parameter
Usage:
        <nodepath> [-?|--help] [-s] <parampath> <newvalue>
Options:
 <nodepath>
    Path to device node
 [-?|--help]
    Prints this help
 [-s]
    Prints subcommand reports
 <parampath>
    Path to parameter. Ex,"ports.port_A.signal_type"
 <newvalue>
    new value
```

Команда отображения истории (history)

Команда отображения истории выводит список выполненных команд. Также выводятся команды, которые были отменены через undo.

Листинг 2. Пример вывода истории по команде history

```
pila:\>>history
Executed actions:
        run scripts\load.m
        cb pila:
        cb .\counter
        cb ..
Redo memory:
        cb .
        cb .\[S]gnd
Total commands number: 6
```

Сохранение истории (save)

Команда сохранения истории выводит историю (без отменённых команд) в виде скрипта, который затем может быть исполнен командой run.

Запуск скрипта (run)

Запускает исполнение скриптовой программы из указанного файла. Исполнение программы идёт последовательно вплоть до завершения файла или возникновения ошибки. Строки, начинающиеся на “#”, считаются комментариями и игнорируются.

Вызов системной операции (system)

Передача вызова в командную оболочку ОС. В функцию передаётся строка в вызовом команды и её аргументами. Следует помнить, что команды навигации в Shell не влияют на текущий каталог в прототипе САР.

Вывод списка библиотек (clibs_info)

Команда выводит списки загруженных или доступных для загрузки (при ключе -a) библиотек. В выводимом списке приводится информация об именах и версиях доступных библиотек.

Листинг 3. Пример вызова команды вывода списка доступных библиотек

```
pila:\>>clibs_info -a
System[0.0.0]
BasicTree[0.0.0]
VHDL[0.0.0]
Netlist[0.0.0]
Analysis[0.0.0]
Quartus[0.0.0]
Reliability[0.1.1]
```

1.2. Библиотека базовых операций с деревом элементов BasicLib

Библиотека BasicLib включает минимально необходимые команды для работы с деревом элементов. Сюда входят команды преобразования дерева, получения информации о ячейках и навигации по дереву. В таблице 2 приведён список команд данной библиотеки.

Таблица 2

Список команд в библиотеке BasicLib

Команда	Описание	Флаги*		
		M	L	U
pwd, ls	Получение пути к текущему выбранному элементу	-	-	-
cb	Переход к элементу по указанному пути	+	+	+
add	Создание новой ячейки	+	+	+
addref	Создание ссылки на указанную ячейку	+	+	+
delete	Удаление ячейки	+	+	+
move	Перенос ячейки по указанному адресу	+	+	+
copy	Копирование ячейки	+	+	+
getparam	Получение значения параметра или списка параметров	-	-	+
setparam	Установка значения параметра	+	+	+
check	Верификация указанного элемента или всего дерева	-	-	+
* Флаги M – модифицирует контекст, L – сохраняется в истории, U – м.б. отменена командой undo (для команд модификации)				

2. Пользовательские библиотеки ядра

2.1. Библиотека VHDL

Библиотека VHDL включает базовые средства для работы с VHDL и VHDL-нетлистами. Данная библиотека подключает vhdl_ast и реализует основные методы считывания внутреннего представления из AST и методики экспорта.

Библиотека VHDL включает пользовательскую команду для вывода устройства в виде единого VHDL-файла, структура которого близка к нетлисту, но при этом возможен синтез данного устройства, т.е. файл полностью соответствует стандарту VHDL. Формат вызова приведён ниже:

>vhdl_export путь_к_ячейке имя_выходного_файла

Команда не модифицирует дерева элементов, но фиксируется в истории команд и может быть отменена. В случае, если выходной файл был перезаписан, то при отмене исходный файл не восстанавливается.

2.2. Библиотека Netlist

Библиотека Netlist для работы с VHDL-нетлистами. Для работы использует библиотеку VHDL.

Команда import_netlist

Команда предназначена для считывания ячеек из плоских и иерархических нетлистов различных типов. Считывание идёт в двухпроходном режиме. Сначала считываются объявления интерфейсов, а потом – всё остальное.

Таблица 3
Опции команды import_netlist

Команда	Описание
[-?|--help]	Вывод справки
[-s]	Вывод отчёта о выполнении подкоманд
[-f]	Флаг считывания плоских нетлистов (по умолчанию - иерархические)
[-p <loadpath>]	Путь к ячейке, в которую будут сохранены считанные элементы
[-o]	Разрешение перезаписи считываемых ячеек
[-i]	Игнорирование несинтезируемых сигналов при считывании (devoe, devpor, …)
<file>	Путь к нетлисту, из которого идёт считывание

Команда export_netlist

Команда экспорта плоского или иерархического нетлиста. Была реализована до реализации вывода VHDL-нетлистов и с тех пор не поддерживается.

2.3. Библиотека Reliability

Библиотека реализует функцию введения аппаратной избыточности в указанный модуль устройства. В текущей реализации поддерживается только троирование указанного элемента. Ниже приведён формат команды:

majorize путь_к_ячейке

Порядок действий при внесении структурной избыточности данной командой приведён в 4.4.5

2.4. Библиотека Analysis

Библиотека сбора статистики (в средстве - Analysis) предназначена для извлечения дополнительной информации о дереве элементов. В таблице 4 приведён список команд, которые поддерживает библиотека.

Таблица 2
Список команд в библиотеке Analysis

Команда	Описание	Флаги*		
		M	L	U
refs	Получение списка ссылок на ячейку	-	-	+
stats	Получение статистики по числу ячеек различных типов в иерархии заданной ячейки	-	-	+
ports	Получение информации о портах ячейки или блока	-	-	+
comps	Получение списка блоков, использующих указанный интерфейс	-	-	+
* Флаги M – модифицирует контекст, L – сохраняется в истории, U – м.б. отменена командой undo (для команд модификации)				

ПРИЛОЖЕНИЕ Е. ВЫЯВЛЕННЫЕ И УСТРАНЁННЫЕ НЕДОСТАТКИ БИБЛИОТЕКИ VHDL_AST

При разработке прототипа для разбора нетлистов и генерации исходных кодов на VHDL использовалась библиотека vhdl_ast, разработанная на кафедре КСПТ. Данная библиотека генерирует объектное представление АСД, которое потом используется в модуле ввода-вывода прототипа.

В процессе разработки в парсере был выявлен ряд ошибок, некоторые из которых потребовалось устранить для обеспечения импорта и экспорта тестовых нетлистов. Ниже приведены списки ошибок и неточностей в библиотеке vhdl_ast.

Парсер

Таблица 1

Список выявленных и устранённых ошибок в парсере vhdl_ast

Описание	Устранёна?	Комментарии
нет поддержки оператора NOT	Да	
нет передачи индексированного доступа в функции	Да	Например, NOT A(5)
не считываются объявления компонентов	Да	
нет поддержки объявлений пакетов	Нет	
нет поддержки типов	Нет	
ложные assert в коде	Да	

Кодогенератор

Таблица 2

Список выявленных и устранённых ошибок в кодогенераторе vhdl_ast

Описание	Устранёна?	Комментарии
нет генерации для типов, пакетов и компонентов	Нет	
нет вывода функций (в т.ч. NOT)	Да	
вывод скобок для пустых массивов	Да	Например, a: std_logic()

ПРИЛОЖЕНИЕ F. ОСОБЕННОСТИ ПРОГРАММНОЙ РЕАЛИЗАЦИИ ПРОТОТИПА СРЕДСТВА РЕИНЖИНИРИНГА

1. Возможности реиспользования компонентов прототипа

В рамках разработки средства реинжиниринга предполагается обеспечить встраиваемость средства реинжиниринга. В рамках прототипа возможно четыре основных уровня использования наработок:

- использование дерева элементов;
- использование ядра CAP;
- использования ядра CAP и дополнительных модулей;
- использование ядра, модулей и интерфейсов как САПР.

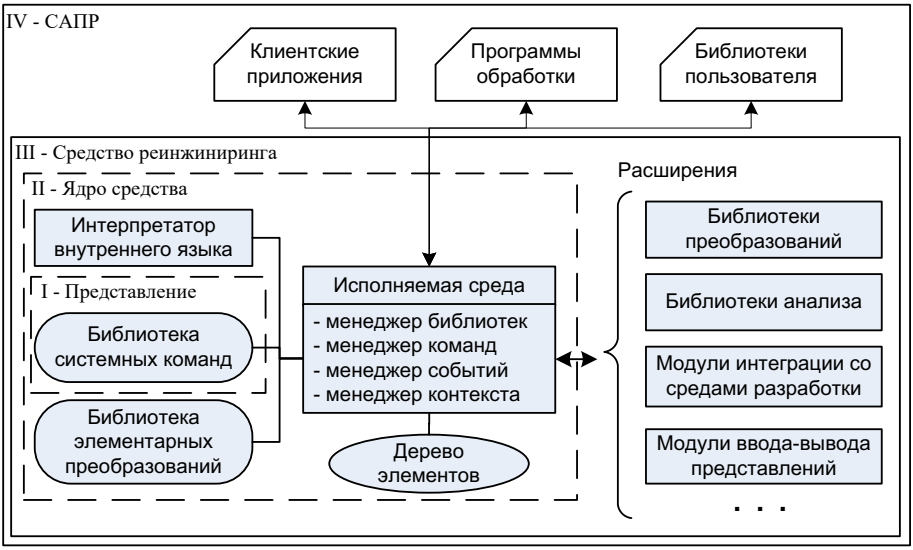

Рис. 1. Уровни использования компонентов прототипа CAP в сторонних средствах

2. Шаблоны пользовательских команд и библиотек

Для упрощения разработки пользовательских функций и библиотек был разработан включаемый модуль NetBeans, который включает шаблоны для генерации пользовательских классов команд и внутренних библиотек. Генерация производится автоматически стандартными средствами среды разработки.

Все внутренние команды и команды библиотек прототипа были разработаны с использованием шаблонов, что позволило ускорить разработку дополнительных команд обработки.

Листинг 1. Шаблон класса пользовательской библиотеки

```
/*
 * ${nameAndExt}
 * This file is CoreLib extension fo VHDL reengeneering tool
 * Created by ${user} at ${date}
 */

<#if package?? && package != "">
package ${package};
</#if>

import obl_core.library_manager.AbstractCoreLibrary;
import obl_core.library_manager.CoreLibraryException;

/**
 *TODO: add library description
 * @author ${user}
 */
public class ${name} extends AbstractCoreLibrary
{
   /**Конструктор библиотеки*/
   public ${name}()
   {
      //TODO: Correct library description
      super("${name}",0,0,0);

      //TODO: Add Commands and relations
      // ex, commandSet.AddCommand(YourCommand.class, "my_command");
   }

   @Override
   public void LoadToWI() throws CoreLibraryException
   {
      super.LoadToWI();

      //TODO: add specific initialization
   }
}
```

Листинг 2. Шаблон класса команды

```
/*
 * ${nameAndExt}
 * This file implements command that can be executed by Command Manager from
 * Object Lib Core.
 * Created by ${user} at ${date}
 * Generated from command template for object lib
 */

<#if package?? && package != "">
package ${package};
</#if>

import com.martiansoftware.jsap.JSAPException;
import com.martiansoftware.jsap.JSAPResult;
import obl_core.command_manager.ParamCheck.ParamCheckException;

/**
 *TODO: add command's description
 *
 * @author ${user}
 */
//TODO:: Specify pattern
public final class ${name} extends Command
{
   static ${name}Specific Specific=new ${name}Specific();
   ${name}Parameters Params;

   public ${name}()
   {
      Params=new ${name}Parameters();
   }

   @Override
   protected void PerformActions()
   {
      // TODO: Add actions
      throw new UnsupportedOperationException("Not supported yet.");
   }

   ///// Parameters /////
   // <editor-fold defaultstate="collapsed" desc="Parameters">
```

```
/**
 * Class contains ${name}'s parameters. This class must be unique for
 * each instance of ${name}
 * @see ${name}
 * @see ${name}Specific
 * @author Oleg Nenashev <o.v.nenashev@gmail.com>
 */
public class ${name}Parameters extends CommandParameters
{
    @Override
    public JSAPResult Parse(String str)
    {
        JSAPResult res=super.Parse(str);
        if (res.success())
        {
            //TODO: Implement parameters parsing here:
        }
        return res;
    }

    @Override
    public void CheckParameters() throws ParamCheckException
    {
        super.CheckParameters();

        //TODO: Add parameters check here
    }

    @Override
    public String toString()
    {
        // TODO: Implement parameters output
        return super.toString();
    }
}
// </editor-fold>

///// Specific /////
// <editor-fold defaultstate="collapsed" desc="Specific">
/**
 * Class contains ${name}'s parser and static data definitions.
 * @see ${namme}
 * @see ${name}Parameters
```

```
     * @author Oleg Nenashev <o.v.nenashev@gmail.com>
     */
    public static class ${name}Specific extends CommandStaticSpecific
    {
        @Override
        protected void InitParser() throws JSAPException
        {
            super.InitParser();
            //TODO: Add specific parameters parsing
        }

        @Override
        public String getHelpHeader()
        {
            // TODO: Specify help header
            return "${name} - no info";
        }
    }
    // </editor-fold>

    //// Access ////
    // <editor-fold defaultstate="collapsed" desc="Inherited access. Don't modify">
    @Override
    public ${name}Parameters parameters()
    {
        return Params;
    }

    @Override
    public ${name}Specific specific()
    {
        return Specific;
    }
    // </editor-fold>
}
```

ПРИЛОЖЕНИЕ G. ПРИМЕРЫ ПОЛЬЗОВАТЕЛЬСКИХ ПРОГРАММ РЕИНЖИНИРИНГА В РАЗРАБОТАННОМ ПРОТОТИПЕ

Всего существует два механизма программирования для разработанного прототипа: через функции библиотек ядра и через внешний скриптовый язык программирования. Кроме того, возможна реализация внешних алгоритмов обработки на Java с использованием последовательных вызовов ядра через API. Далее будут приведены примеры для всех трёх случаев.

1. Пример функций библиотеки ядра

В данном пункте приведены листинги пользовательских функций обработки, реализованных на Java. Функции построены на базе шаблонов, предоставляемых модулем NetBeans для разработки под прототип средства реинжиниринга.

Полностью привести листинги функций тяжело, так как они используют сторонние объекты, в т.ч. и описания дерева элементов. Для демонстрации решено взять команду переноса ячейки move, которая включает весь функционал в один файл (см. листинг 1).

На примере листинга можно увидеть структуру команд. Каждая команда состоит из трёх основных частей:

- статического класса параметров, идентичного для всех экземпляров команд одного типа;
- класса с описанием параметров методов;
- тела команды, которое включает функции выполнения команды PerformActions() и её отмены sysUndo().

Статическая часть содержит общую информацию о команде: флаги модификации контекста, строку справки и параметры парсера JSAP. В классе параметров, кроме их хранения, определены механизмы проверки параметров, формирования строки параметров команды для истории. Класс параметров связан с родительской командой. Также все классы могут получить доступ к исполняемой среде через интерфейс Instanced.

Листинг 1. Листинг команды move из базовой библиотеки

```
 /**
 * Команда переноса ячейки
 * @author Oleg Nenashev <o.v.nenashev@gmail.com>
 */
public final class MoveNodeCommand extends ModifyingBackable
{
   /** Статические параметры команды
   static MoveNodeCommandSpecific Specific;
```

```
  /**Параметры команды*/
  MoveNodeCommandParameters Params;

  /**Конструктор команды*/
  public MoveNodeCommand()
  {
     Params=new MoveNodeCommandParameters();
  }

  /**
   * Конструктор команды с заданием параметров
   * @param nodepath
   * @param newpath
   */
  public MoveNodeCommand(DevNodePath nodepath,DevNodePath newpath)
  {
     Params.USEDNODE=nodepath;
     Params.newPosition=newpath;
  }

  @Override
  protected void PerformActions()throws CommandException
  {
     DeviceNode mov=WI.getElementTree().nodes().getNode(Params.USEDNODE);
     Params._oldPosition=Params.USEDNODE;

     // Удаляем ячейку из старого места
     Command com=new DeleteNodeCommand(mov);
     ExecuteSubcommand(com);

     // Добавляем ячейку в новое место
     com=new AddNodeCommand(mov,Params.newPosition);
     ExecuteSubcommand(com);
  }

  @Override
  protected CommandReport sysUndo()throws CommandException
  {
     MoveNodeCommand com=new
MoveNodeCommand(Params.newPosition,Params._oldPosition);
     ExecuteSubcommand(com);
     return Report;
```

```
}

///// MoveNodeCommandParameters /////
// <editor-fold defaultstate="collapsed" desc="MoveNodeCommandParameters">
/** Класс параметров команды MoveNodeCommand*/
public class MoveNodeCommandParameters extends CommandParameters
{
   DevNodePath newPosition;
   DevNodePath _oldPosition;

   @Override
   public JSAPResult Parse(String str)
   {
      JSAPResult res=super.Parse(str);
      if (res.success())
      {
         newPosition=(DevNodePath)res.getObject("destination");
      }
      return res;
   }

   @Override
   public void CheckParameters() throws ParamCheck.ParamCheckException
   {
      super.CheckParameters();
      ParamCheck.CheckMustNotExist("destination", newPosition);
   }

   @Override
   public String toString()
   {
      return super.toString()+" "+newPosition;
   }
}
// </editor-fold>

///// MoveNodeCommandSpecific /////
// <editor-fold defaultstate="collapsed" desc="MoveNodeCommandSpecific">
/**Класс статической информации о команде MoveNodeCommand*/
public class MoveNodeCommandSpecific extends ModifyingBackableSpecific
{
   @Override
   protected void InitParser() throws JSAPException
```

```
        {
            super.InitParser();

            UnflaggedOption newpos=new UnflaggedOption("newpath");
            newpos.setRequired(true);
            newpos.setHelp("New path to node");
            DevNodePathParser newpos_p=new DevNodePathParser();
            newpos.setStringParser(newpos_p);
            Parser.registerParameter(newpos);
        }

        @Override
        public String getHelpHeader()
        {
            return "Moves node to selected device/block";
        }

        @Override
        public boolean usesNode()
        {
            return true;
        }
    }
    // </editor-fold>

    ….
}
```

2. Примеры скриптовых программ обработки

Скриптовые программы представляют собой последовательность вызовов, которые последовательно исполняются менеджером команд. Ниже приведены примеры скриптовых программ, которые использовались для функционального тестирования прототипа САР.

Листинг 2. Скриптовая программа загрузки устройств

```
# Добавление библиотеки работы с нетлистами
clibs_load Netlist
# Загрузка тестовых устройств
import_netlist -f -p test: examples\flat\6lab.vhd
import_netlist -f -p pila: examples\flat\pila.vho
cb test:
```

Листинг 3. Скриптовая программа внесения структурной избыточности в устройство

```
# Загрузка библиотек
clibs_load Netlist
clibs_load Reliability
# Загрузка устройства
import_netlist -f -p test:\ examples\flat\6lab.vhd
cb test:
# Мажоризация
majorize -s test:\counter
# Вывод результатов
export_vhdl test: tests\
```

3. Пример использования функций взаимодействия с ядром

В листинге приведён пример взаимодействия консольного интерфейса с ядром CAP. В примере ядро инициализируется как статический объект, создаётся и регистрируется обработчик события ExitEvent, после чего программа в цикле считывает и исполняет команды.

Листинг 4. Пример использования функций взаимодействия с ядром

```
/** Ядро средства реинжиниринга */
static final OblCore ProgramCore=new OblCore();

/** Класс обработчика событий, генерируемых исполняемой средой */
static class EventListener implements WorkInstanceEventListener
{
   @Override
   public void onWIEvent(WorkInstanceEvent evt)
   {
      switch (evt.getEventType())
      {
         case ExitEvent:
            exitflag=true;
            return;
         default:
            return;
      }
   }
}
```

```
/**
 * Основная функция программы
 * @param args the command line arguments
 */
 public static void main(String[] args)
{
WorkInstance.eventManager().addListener(Listener);
…
// Вызов команды на исполнение в ядре
CommandReport rep=WI.getCommandManager().Execute(command);
if (rep.getResult()==CommandResult.FAILED)
                        System.out.println("COMMAND FAILED");
System.out.println(rep.getMessage());
…
}
```

ПРИЛОЖЕНИЕ Н. ПРИМЕР ВВЕДЕНИЯ СТРУКТУРНОЙ ИЗБЫТОЧНОСТИ В УСТРОЙСТВО

В данном приложении приведён пример пошагового выполнения реинжиниринга с использованием прототипа средства. Приведённый пример относится не к последней версии ядра CAP, так как из-за внесённых в методики представления сигналов и групп нарушилась совместимость ядра с демонстрационными библиотеками.

При тестировании ставится задача демонстрации полного цикла реинжиниринга: от синтеза исходного проекта в Quartus II до синтеза и тестирования устройства после реинжиниринга. Для проверки возьмём какое-либо простое устройство, в котором введём структурную избыточность для одного из элементов.

Итак, требуется выполнить следующие шаги:
- выбрать тестовый проект для проверки команды;
- синтезировать нетлист в Quartus 2;
- импортировать нетлист в прототипе CAP;
- вызвать команду мажорирования одного из модулей устройства;
- провести верификацию синтезированного устройства в CAP;
- экспортировать устройство в VHDL;
- синтезировать полученное устройство;
- проверить корректность работы построенного устройства.

К сожалению, в рамках прототипа не было автоматизировано взаимодействие с Quartus, поэтому передавать данные между средствами приходилось вручную.

Подготовка тестового проекта

В связи с наличием недоработок в прототипе, требуется взять простое устройство. Его решено реализовать на VHDL при помощи поведенческого описания, чтобы продемонстрировать преимущества использования внешней среды для формирования структурных описаний.

В качестве тестируемого устройства возьмём счётчик с управляемым модулем счёта, разрешением счёта, возможностью внешнего сброса и автоматического сброса после достижения требуемого значения. Исходные коды данного устройства приведены ниже в листинге 1.

Листинг 1. Исходные коды тестируемого устройства

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
-- Counter with saturation and autoreload
ENTITY sat_counter IS
	generic
	(
		width: integer:=2
	);
	port
	(
		-- Clock
		CLK:	IN	std_logic;
		-- Reset if nRst=0
		nRST:	IN	std_logic;
		-- Count enable
		ENA:	IN	std_logic:='1';
		-- Automatic reset if code=Sat
		autoRST:IN	std_logic:='1';
		-- Max code
		Sat:	IN  std_logic_vector(width-1 DOWNTO 0):=(others => '1');
		-- Code output
		Q:		OUT std_logic_vector(width-1 DOWNTO 0)
	);
END sat_counter;

ARCHITECTURE arch OF sat_counter IS
	SIGNAL SUM: std_logic_vector(width-1 DOWNTO 0);
BEGIN
clk_ev:
	PROCESS(clk,nRST)
	BEGIN
		if (nRST='1') then
			SUM <= (others => '0');
		elsif (CLK'event and CLK='1') then
			if (SUM = Sat) then
				if (autoRST='1') then
					SUM <= (others => '0');
				end if;
			elsif (ENA = '1') then
				SUM <= std_logic_vector( unsigned(SUM) + 1 );
			end if;
		end if;
	END PROCESS;
	Q <= SUM;
END arch;
```

Был произведён синтез устройства в Quartus II со стандартными настройками проекта. Схема на уровне RTL, сформированная средством RTL Netlist Viewer, приведена на рис. 1.

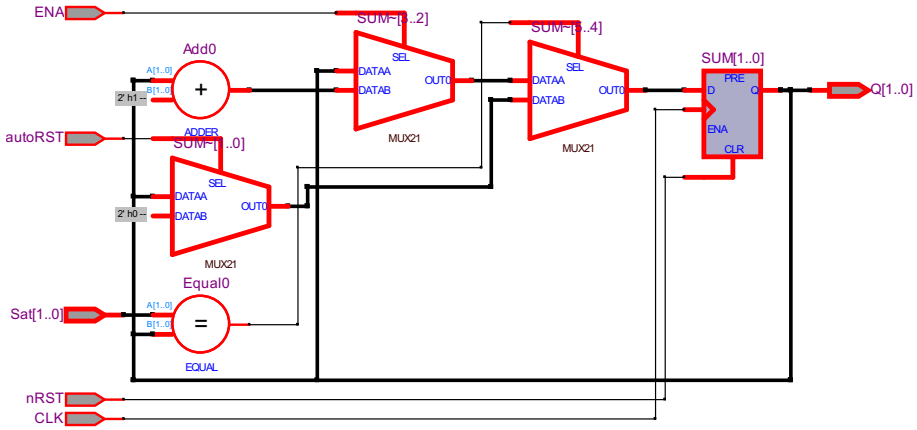

Рис. 1. RTL-схема синтезированного устройства

Как и ожидалось, для хранения накопленного результата были сгенерированы триггеры. Сравнение адреса идёт через компаратор, инкрементация – через сумматор. Код загрузки из сумматора проходит через серию мультиплексоров, которые управляются входными сигналами (в т.ч. ENA) и результатом сравнения. Асинхронный сброс подан сразу на соответствующие входы сброса триггеров.

Для проверки функционирования устройства было проведено его моделирование (см. рис. 2). Построенное устройство в соответствии со взятыми требованиями. С точки зрения практического применения, вызывает сомнение, что при изменении ограничения на код, меньший текущего, счёт идёт до переполнения (пример на 180-й нс моделирования). Для проверки прототипа САР это неважно, поэтому перейдём к синтезу нетлиста.

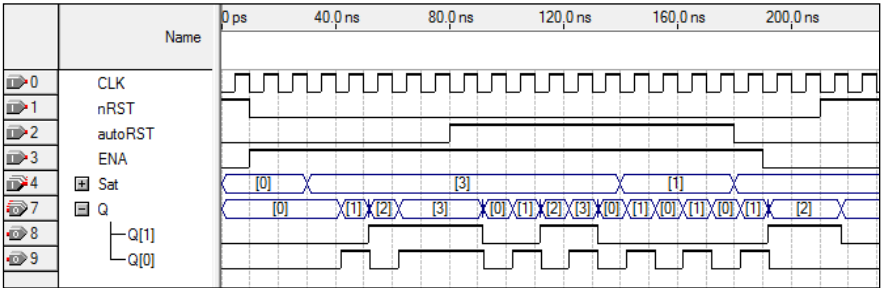

Рис. 2. Результаты моделирования устройства

Подготовка нетлистов

Для проведения моделирования требуется подготовить VHDL-нетлисты с описанием исходного устройства. Для этого нужно изменить настройки генерации нетлиста в Project Options/EDA Tool/Simulation, а потом нажать на кнопку “More EDA Netlist Writer Settings” и в открывшемся окне настроить дополнительные параметры. Указания по настройке генерации нетлистов и обоснование опций приведены в приложении А к диссертации. На рис. 3 и 4 приведены скриншоты обоих окон настройки.

После полного синтеза нетлист был успешно сформирован. Его размер составил 276 строк, поэтому в отчёте решено не приводить его листинг.

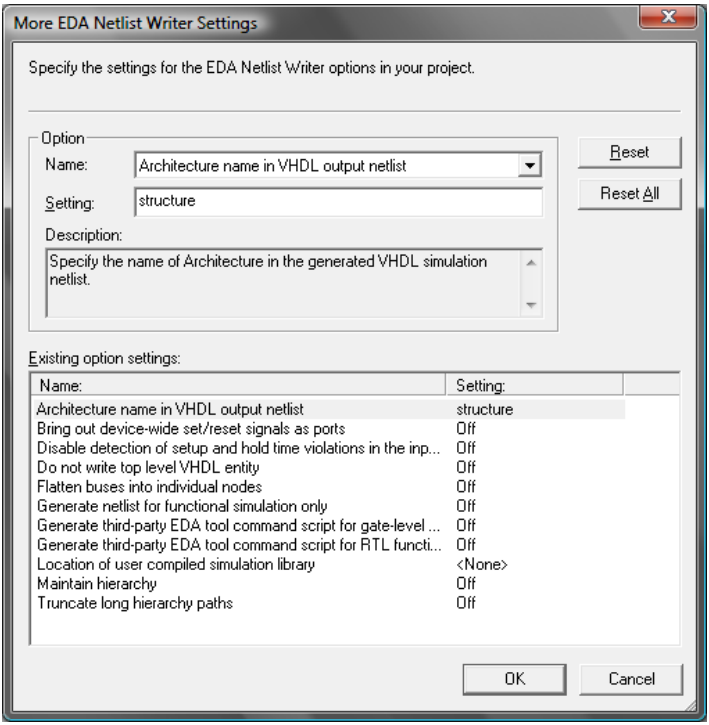

Рис. 3. Окно дополнительных настроек генерации нетлистов

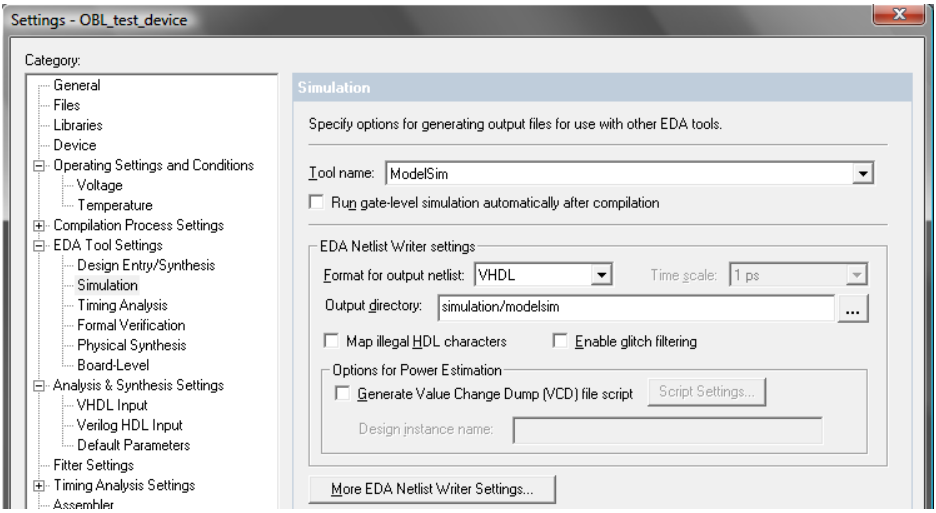

Рис. 4. Окно настроек EDA Tool/Simulation в Quartus

Проведение реинжиниринга

Для выполнения операции по внесению структурной избыточности была разработана скриптовая программа, листинг которой приведён ниже:

Листинг 2. Скрипт проведения преобразования

```
# Загрузка библиотек
clibs_load Netlist
clibs_load Reliability
# Загрузка устройства
import_netlist -f -p test_device:\ E:\Temp\obl_demo\OBL_test_device.vho
# Мажорирование устройства
majorize test_device:
check test_device:
# Вывод результатов
export_vhdl -d E:\Temp\obl_demo\test_device.vhd test_device:
```

Данная программа была вызвана в консоли графического интерфейса прототипа при помощи системной команды run. Для команды был установлен флаг s, поэтому на дисплей были выведены отчёты по всем внутренним командам программы. В листинге 3 приведён сформированный отчёт.

Листинг 3. Отчёт о выполнении скриптовой программы

```
.\[R]root\>>run -s E:\Temp\obl_demo\demo.m
Subcommands:
 >>>clibs_load Netlist
  Library Netlist[0.0.0] was loaded
 >>>clibs_load Reliability
  Library Reliability[0.1.1] was loaded
 >>>import_netlist -f -p test_device:\ E:\Temp\obl_demo\OBL_test_device.vho
  Node loaded
 >>>majorize test_device:\
  Node test_device:\ majorized.
 >>>check test_device:\
  Errors: 0  Warnings: 0  Messages: 0
 >>>export_vhdl test_device:\
  Node test_device:\ has been exported.
Script executed
```

В графическом представлении дерева (рис. 5) можно видеть, что созданы копии корневого элемента и добавлен голосователь для выходных сигналов. В Connections присутствуют все необходимые подключения между элементами, т.е. алгоритм введения структурной избыточности выполнен верно.

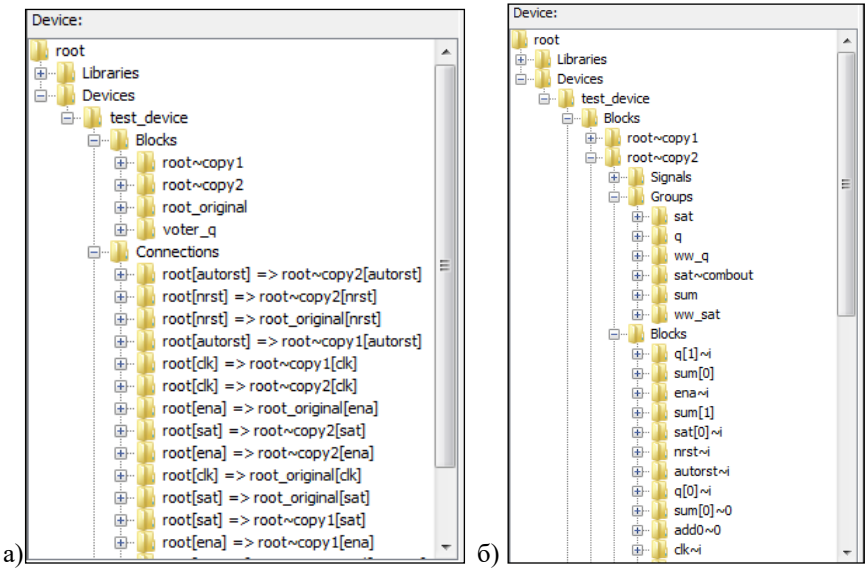

Рис. 5. Структура устройства после преобразования
(а – структура устройства; б – структура скопированной ячейки)

В листинге 4 приведён пример отчёта о структуре устройства, сформированный командой ls. Полный отчёт с рекурсией по уровням привести невозможно из-за большого числа ячеек в иерархии.

Листинг 4. Структура корневой ячейки построенного устройства

```
.\[R]root\>>ls test_device:
Device "test_device"
  -- Blocks:
        BlockReference "root~copy1" #REFERS test_device:\root_original\
        BlockReference "root~copy2" #REFERS test_device:\root_original\
        BlockInstance "root_original"
        VoterReference "voter_q" #REFERS [l]reliability\voter3\
  -- Connections:
        SignalConnection "root[autorst] => root~copy2[autorst]"
        SignalConnection "root[nrst] => root~copy2[nrst]"
        SignalConnection "root[nrst] => root_original[nrst]"
        SignalConnection "root[autorst] => root~copy1[autorst]"
        …
  -- Libraries:
        Library "test_device"
```

По отчёту и GUI видно, что был создан только один голосователь, хотя в устройство синтезировалось для двухразрядного счётчика. Таким образом, при выполнении теста в модуле ввода-вывода была обнаружена очередная ошибка. К сожалению, на момент написания диссертации устранена она не была. Чтобы закончить пример, в устройство вручную был добавлен ещё один голосователь.

После исправления ошибки с соединениями и ряда модификаций устройство успешно скомпилировалось и заработало. Результаты моделирования (см. рис. 6) совпали с результатами для исходного устройства

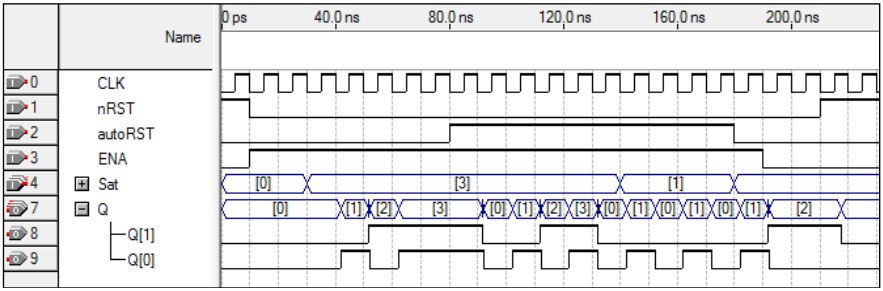

Рис. 6. Результаты моделирования устройства, сгенерированного прототипом CAP (после внесения правок)

Обеспечение компиляции VHDL-файла

Выбранный пример был экспортирован в VHDL при выполнении программы. Далее требуется провести синтез и моделирование устройства, для чего опять используется среда Quartus II. Для успешного синтеза в Quartus требуется выполнить ряд операций, которые рассмотрены ниже.

Quartus II, генерируя нетлист, на нижнем уровне использует логические элементы ПЛИС, параметры которой задаются в проекте. Подобный нетлист обрабатывается CAP, но для его компиляции требуется добавить в проект файл с описанием компонентов выбранной платы.

[корневая директория установки quartus]
\quartus\eda\sim_lib\[имя_архитектуры]_components.vhd

Для успешной компиляции файла VHDL нужно проделать следующее:
- добавить в проект файл описания компонентов для Modelsim;
- удалить сигналы data_input из padio;
- удалить ячейки maxii_io из описания внутренней архитектуры устройства (вне корневого модуля).

Подобная подход не гарантирует корректной работы устройства, так как во ячейках ввода-вывода может содержаться некоторая функциональность. Поэтому, необходимо расширение функций модуля ввода-вывода для исключения компонентов ПЛИС из внутреннего представления.

Анализ результатов

Таким образом, в настоящее время программная реализация прототипа не позволяет сформировать синтезируемый файл, но проблема вызвана ошибками в модуле ввода-вывода, а само ядро средство выполняет все требуемые операции.

Модуль ввода-вывода не является составной части ядра средства, и можно говорить лишь о частичной работоспособности разработанного прототипа. Тем не менее, были успешно пройдены фазы ввода устройства и его трансформации, что уже позволяет говорить о применимости подходов к представлению средства для реинжиниринга устройства.

ПРИЛОЖЕНИЕ I. ПРИМЕРЫ ИСХОДНЫХ КОДОВ ПРОТОТИПА

В данном приложении приведены некоторые примеры исходных кодов прототипа на языках Java и VHDL. Больше примеров можно найти на диске, который прилагается к диссертации.

1. Примеры исходных кодов на Java

Объявление интерфейса ячейки дерева элементов

В файле DeviceNode.java описан интерфейс ячейки дерева элементов, а также описан внутренний контейнер ссылок, который используется в классах-наследниках. Данный класс, как и все прочие, подробно документирован при помощи комментариев в формате Javadoc.

Листинг 1. Файл DeviceNode.java

```
package obl_core.element_tree.nodes;

import java.util.Iterator;
import java.util.LinkedList;
import obl_core.element_tree.nodes.device.Device;
import obl_core.element_tree.nodes.groups.Group;
import obl_core.element_tree.nodecontainer.DevNodeContainer;
import obl_core.element_tree.ElementTreeItem;
import obl_core.element_tree.parameters.NodeParameterContainer;
import obl_core.util.path.DevNodePath;
import obl_core.util.path.DevNodePathItem;

/**
 * Интерфейс, который должны поддерживать все ячейки дерева элементов.
 * @author Oleg Nenashev <o.v.nenashev@gmail.com>
 */
//TODO: переделать механизм группировок
//TODO: Переименовать всех наследников в [....]Node
public interface DeviceNode
        extends Cloneable,ElementTreeItem,NodeParameterContainer
{
    /**
     * Включение элемента в группу
     * @param group Группа, в которую добавляется элемент
     * @param position Индекс позиции ячейки в группе
     * @exception DeviceNodeException Ошибка при добавлении элемента в
группу
     */
```

```
void addToGroup(Group group, int position) throws DeviceNodeException;

/**
 * Копирование элемента
 * @return Копия элемента. Ячейки копируются, а ссылки - нет
 * @throws CloneNotSupportedException
 */
public DeviceNode clone()throws CloneNotSupportedException;

/**
 * @return Устройство, к которому относится ячейка;
 * null, если ячейка к устройству не относится
 */
Device getDevice();

/**
 * Получение группы, в которую входит элемент
 * @return Ссылка на группу, в которой состоит элемент. Если группы нет, то
 * возвращается null.
 */
Group getGroup();

/**
 * Получение индекса положения элемента в группе.
 * Если элемент не входит в нумерованную группу, то
 * возвращаемое значение может быть любым.
 * @return Индекс положения элемента в группе.
 */
int getGroupPosition();

/**
 * Получение родительского элемента
 * @return Ссылка на родительский элемент или null, если родительский
 * элемент отсутствует.
 */
DeviceNode getParent();

/**
 * Получение типа ячейки
 * @return Тип ячейки
 */
DeviceNodeType getType();
```

```
    /**
     * Получение контейнера ячеек, входящих в элемент (равносильно ячейкам
     * нижнего уровня).
     * @return
     */
    DevNodeContainer nodes();

    /**
     * Получение контейнера ссылок на текущую ячейку
     * @return Контейнер ссылок на ячейку
     */
    NodeReferences references();

    /**
     * Задание родительского элемента. Все преобразования структуры дерева
     * элементов при этом должны выполняться автоматически.
     * @param Parent Новый родительский элемент
     */
    void setParent(DeviceNode Parent);

    /**
     * Проверка, является ли элемент корневым. Элемент считается корневым,
если
     * у него нет родительского.
     * @return true, если элемент является корневым.
     */
    boolean isRoot();

    /**
     * Проверка, является ли элемент ссылкой на другой элемент. Используется,
     * чтобы выделить элементы типа Reference.
     * @return true, если элемент является ссылкой.
     */
    boolean isPointer();

    /**
     * Контейнер ссылок на элемент дерева
     */
    public class NodeReferences implements Cloneable, Iterable<DevNodePointer>
    {
        /**Список ссылок*/
        LinkedList<DevNodePointer>  References;
```

```
/**Указатель на ячейку, для которой хранятся ссылки*/
DeviceNode Refers;

/**
 * Создание полной копии контейнера
 * @return Новый объект NodeReferences
 * @throws CloneNotSupportedException появляться не должно
 */
@Override
protected NodeReferences clone() throws CloneNotSupportedException
{
   NodeReferences refs=(NodeReferences)super.clone();
   refs.References=null;
   refs.Refers=null;
   return refs;
}

/**
 * Добавление в список ссылки на элемент
 * @param ref Добавляемая ссылка
 */
public void addReference(DevNodePointer ref)
{
   if (References==null)
      References=new LinkedList<DevNodePointer>();
   References.add(ref);
}

/**
 * Удаление из списка указанной ссылки
 * @param ptr Удаляемая ссылка
 */
public void deleteReference(DevNodePointer ptr)
{
   References.remove(ptr);
}

/**
* Удаление всех ссылок на элемент
*/
public void removeAll()
{
   References.clear();
```

```
        }

        /**
         * Выводит список ссылающихся элементов
         * @return Строка с описанием ссылок
         */
        @Override
        public String toString()
        {
            String res="";
            if (References==null)
                return res;
            Iterator<DevNodePointer> iterator = References.iterator();
            if(iterator.hasNext())
                res+=iterator.next().getPath().toString();
            while (iterator.hasNext())
                res+="\n"+iterator.next().getPath().toString();
            return res;
        }
        @Override
        public Iterator<DevNodePointer> iterator()
        {
            return References.iterator();
        }
    }

    /**
     * Получение списка параметров ячейки
     * @return Список параметров ячейки
     */
    DeviceNodeParameters parameters();

    /**
     * Получение пути к ячейке
     * @return Путь к ячейке
     */
    DevNodePath getPath();

    /**
     * Получение элемента пути для ячейки
     */
    DevNodePathItem getPathItem();
}
```

Описание типов базовых ячеек

В прототипе описание базовых типов ячеек сформировано в виде энумерации. В неё включены описания идентификаторов и имён для множественных обозначений, что реализовано через дополнительные статические контейнеры.

Листинг 2. Файл DeviceNodeType.java

```
package obl_core.element_tree.nodes;

import java.util.HashMap;
import obl_core.util.path.DevNodePathException;

/**
 * Перечисление основных типов элементов, которые могут встречаться в
 * объектном представлении
 * @author Oleg Nenashev <o.v.nenashev@gmail.com>
 */
public enum DeviceNodeType
{
   /** Корневой элемент */
   Root("Roots","R"),
   /** Элемент устройства*/
   Device("Devices","D"),
   /**Блок*/
   Block("Blocks","B"),
   /**Соединение*/
   Connection("Connections","C"),
   /**Сигнал, порт, generic и т.п.*/
   Signal("Signals","S"),
   /**Функция*/
   Function("Functions","F"),
   /**Интерфейс*/
   Interface("Interfaces","I"),
   /**Библиотека*/
   Library("Libraries","l"),
   /**Группа*/
   Group("Groups","g"),
   /**Агрегация*/
   Aggregation("Aggregations","a");

   /**Контейнер с информацией об отдельных типах ячеек*/
   private static HashMap<DeviceNodeType,TypeDescription> Descr;
```

```
  /**Хэш-карта для сопоставления индексов и типов
   * @see TypeDescription
   */
  private static HashMap<String,DeviceNodeType> IndexMap;

  /**
   * Класс с описанием типа отдельной ячейки
   */
  public class TypeDescription
  {
     /**Имя типа во множественном числе*/
     String multipleName;
     /**Односимвольный индекс типа при использовании в путях*/
     //FIXME: Заменить на char
     String pathLabel;
     /**Строка с краткой информацией о типе*/
     String helpString;

     /**
      * Конструктор
      * @param multipleName Имя ячейки во множественном числе
      * @param pathIndex Уникальный односимвольный индекс типа
      * @param helpString Строка с краткой информацией о типе
      */
     public TypeDescription(String multipleName, String pathIndex, String
helpString)
     {
        this.multipleName = multipleName;
        this.pathLabel = pathIndex;
        this.helpString = helpString;
     }
  }
  /**
   * Конструктор
   * @param multipleName Имя ячейки во множественном числе
   * @param pathIndex Уникальный односимвольный индекс типа
   * @param helpString Строка с краткой информацией о типе
   */
  private DeviceNodeType(String multipleName,String label, String helpString)
  {
     addToMap(this, new TypeDescription(multipleName, label, helpString));
  }
```

```
 /**
 * Конструктор без информацилнной строки
 * @param multipleName Имя ячейки во множественном числе
 * @param pathIndex Уникальный односимвольный индекс типа
 */
private DeviceNodeType(String multipleName,String label)
{
   addToMap(this, new TypeDescription(multipleName, label, "No help yet"));
}

/**
 * Добавление типа в карты поиска
 * @param type Тип
 * @param descr Описание типа
 */
private static void addToMap(DeviceNodeType type,TypeDescription descr)
{
   if (Descr==null)
      InitStatic();
   Descr.put(type, descr);
   IndexMap.put(descr.pathLabel, type);
}

/**
 * Инициализация статических переменных энумерации.
 * Используется, т.к. нельзя определить статические переменные до
 * перечисления.
 */
private static void InitStatic()
{
   Descr=new HashMap<DeviceNodeType, TypeDescription>();
   IndexMap=new HashMap<String, DeviceNodeType>();
}

/**
 * Получение справочной информации по типу
 * @return Строка со справкой по типу
 */
public String getHelpString()
{
   return getDescription().helpString;
}
```

```
    /**
     * Получение индекса пути к типу
     * @return Односимвольный индекс пути к типу
     */
    public String getPathLabel()
    {
        return getDescription().pathLabel;
    }

    /**
     * Получение множественного имени типа
     * @return Строка с именем типа во множественном числе
     */
    public String toStringMul()
    {
        return getDescription().multipleName;
    }

    /**
     * Получение описания типа
     * @return Класс с описанием типа ячейки
     */
    public TypeDescription getDescription()
    {
        return Descr.get(this);
    }

    /**
     * Получение типа ячейки по её индексу
     * @param index Односимвольный индекс типа
     * @return Тип ячейки
     * @throws DevNodePathException Тип с указанным индексом не найден
     */
    public static DeviceNodeType Index2Type(String index)throws
DevNodePathException
    {
        DeviceNodeType type=IndexMap.get(index);
        if (type==null)
            throw new DevNodePathException("Type index '"+index+"' doesn't exit");
        else return type;
    }
}
```

Пример объявления системной библиотеки

В данном примере приведён класс с описанием библиотеки SystemLib, которая является составляющей ядра разработанного средства реинжиниринга.

В конструкторе определяются списки команд библиотеки и зависимости, а начальная инициализация производится в функции LoadToWI(), которая вызывается менеджером библиотек после вызова команды подключения библиотеки. При инициализации возможен полный доступ к дереву элементов, и возможно подгрузить необходимые библиотеки элементов.

Листинг 3. Файл SystemCommandsLibrary.java

```
/*
 * SystemCommandsLibrary.java
 * This file is CoreLib extension fo VHDL reengeneering tool
 * Created by Oleg Nenashev <o.v.nenashev@gmail.com> at 28.01.2011
 */

package obl_core.SystemLib;

import obl_core.library_manager.AbstractCoreLibrary;
import obl_core.library_manager.CoreLibraryException;

/**
 * Библиотека системных команд.
 * <p>Системная библиотека хранит команды, необходимые для корректной
работы
 * ядра библиотеки. В частности, через неё реализуется загрузка других
 * библиотек пользователем.</p>
 * <p>Данная библиотека по-умолчанию загружается ядром средства в
 * процессе инициализации.</p>
 * @author Oleg Nenashev <o.v.nenashev@gmail.com>
 */
public final class SystemCommandsLibrary extends AbstractCoreLibrary
{
   /**Конструктор библиотеки*/
   public SystemCommandsLibrary()
   {
      super("System",0,0,0);

      //Commands
      commandSet.AddCommand(ShowHelpCommand.class,"help","?","man");
      commandSet.AddCommand(ExitCommand.class,"exit","quit");
```

```
        commandSet.AddCommand(ShowHistoryCommand.class,"history");
        commandSet.AddCommand(UndoCommand.class,"undo");
        commandSet.AddCommand(RedoCommand.class,"redo");
        commandSet.AddCommand(SaveHistoryCommand.class,"save");
        commandSet.AddCommand(RunScriptCommand.class,"run");
        commandSet.AddCommand(ExecuteShellCommand.class,"system");
        commandSet.AddCommand(CoreLibInfoCommand.class, "clibs_info");
        commandSet.AddCommand(CoreLibLoadCommand.class, "clibs_load");
    }

    @Override
    public void LoadToWI() throws CoreLibraryException
    {
        super.LoadToWI();
    }
}
```

2. Примеры исходных кодов на VHDL

Для апробации прототипа был на VHDL был разработан ряд модулей, прежде всего рассчитанных на повышение надёжности устройства посредством введения структурной избыточности. Из-за недоработок в прототипе не удалось применить данные разработки на практике, но хотелось продемонстрировать несколько примеров.

В листингах приведены примеры VHDL-кодов для отказоустойчивого триггера и элемента памяти, построенного на его основе.

Листинг 4. Пример отказоустойчивого триггера

```
-- fo_trigger.vhd
-- Модуль отказоустойчивого D-триггера со структурной избыточностью и
-- генерацией сигнала об ошибке
-- автор- Ненашев О.В.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.my_package.all;

entity fo_trigger is
            generic  (
```

```
                maj_depth:integer:=11;
                restore:boolean:=true;
                form_error:boolean:=true;
                ext_error:boolean:=true
        );
        port  (
                clk:            in std_logic;
                D:              in std_logic;
                Ena:            in std_logic;
                Q:              out std_logic;
                f:              out std_logic;
                aclr:           in std_logic;
                aset:           in std_logic;
                noise_clr:      in std_logic;
                noise_set:      in std_logic
        );
end entity;

architecture failover_trigger of fo_trigger is
        component my_mux is
                generic  (
                        num_inputs:integer:=8
                );
                port (
                        A: in std_logic_vector(num_inputs-1 downto 0);
                        sel: in integer range 0 to num_inputs-1;
                        Q: out std_logic
                );
        end component;
        signal  qout: std_logic;
        signal  reg:std_logic_vector(maj_depth-1 downto 0);
        signal  sel: integer range 0 to 2**maj_depth-1;
        signal  muxes: std_logic_vector(2**maj_depth-1 downto 0);
begin

-- Multiple triggers with majorization
L00: if maj_depth>1 generate
        majmux:my_mux
                generic map             (2**maj_depth)
                port map                (muxes, sel, qout);

        -- initializing mux inputs
        L1:     for i in 0 to (2**maj_depth-1) generate
```

```
                process (reg)
                begin
                        if code_weight(CONV_STD_LOGIC_VECTOR
                        (i,maj_depth),maj_depth) <maj_depth/2  then
                                muxes(i)<='0';
                        else
                                muxes(i)<='1';
                        end if;
                end process;
        end generate;

        -- error output
        l4:     if form_error=true generate
                        f<= '0' when is_null(reg,maj_depth) or
                        is_ones(reg,maj_depth) else '1';
                end generate;
        sel<=conv_integer(reg);
end generate;

-- Unary trigger without error check
L01: if maj_depth=1 generate
        qout<=reg(0);
        f<='0';
end generate;

-- trigger with external error
l2:     if ext_error=true generate
        process (aclr,aset,clk,noise_clr,noise_set)
        begin
                if aclr='0' then
                        for i in 0 to maj_depth-1 loop
                                reg(i)<='0';
                        end loop;
                elsif aset='0' then
                        for i in 0 to maj_depth-1 loop
                                reg(i)<='1';
                        end loop;
                elsif clk'event and clk='1' then
                        if (ena='1') then
                                for i in 0 to maj_depth-1 loop
                                        reg(i)<=D;
                                end loop;
                        elsif (restore=true) then
```

```
                                        for i in 0 to maj_depth-1 loop
                                                reg(i)<=qout;
                                        end loop;
                                end if;
                        end if;

                        -- noise reaction
                        if noise_clr='0' then
                                reg(0)<='0';
                        elsif noise_set='0' then
                                reg(0)<='0';
                        end if;
                end process;
end generate;

-- trigger without external error
l3:     if ext_error=false generate
        process (aclr,aset,clk)
        begin
                if aclr='0' then
                        for i in 0 to maj_depth-1 loop
                                reg(i)<='0';
                        end loop;
                elsif aset='0' then
                        for i in 0 to maj_depth-1 loop
                                reg(i)<='1';
                        end loop;
                elsif clk'event and clk='1' then
                        if (ena='1') then
                                for i in 0 to maj_depth-1 loop
                                        reg(i)<=D;
                                end loop;
                        elsif (restore=true) then
                                for i in 0 to maj_depth-1 loop
                                        reg(i)<=qout;
                                end loop;
                        end if;
                end if;
        end process;
end generate;

        Q <= qout;
end architecture;
```

Листинг 5. Листинг модуля отказоустойчивой памяти

```
-- fo_memory.vhd
-- Модуль отказоустойчивой памяти
-- Автор- Ненашев О.В.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.my_package.all;

entity fo_memory is
	generic (
		size:integer:=256;
		width:integer:=8;
		maj_depth:integer:=1;
		form_error:boolean:=false;
		restore:boolean:=false;
		ext_error:boolean:=false
	);
	port (
		clk:		in std_logic;
		Data:		in std_logic_vector(width-1 downto 0);
		rd_addr:	in integer range 0 to size-1;
		wr_addr:	in integer range 0 to size-1;
		Ena:		in std_logic;
		wr_ena:		in std_logic;
		Q:		out std_logic_vector(width-1 downto 0);
		f:		out std_logic;
		rst:		in std_logic;
		noise_clr:	in std_logic;
		noise_set:	in std_logic
	);
end entity;

architecture failover_memory of fo_memory is

	type selector is array (0 to size-1) of std_logic_vector (1 to width);

	signal qout: selector;
	signal fail: std_logic_vector (0 to size-1);
	signal loads: std_logic_vector (0 to size-1);
```

```
begin

dec: my_decoder
        generic map(size,'0')
        port map(wr_ena,wr_addr,loads);

reg0: fo_register
        generic map(maj_depth=> maj_depth,restore=>restore, width=>width,
                        form_error=>form_error,ext_error=>ext_error)
        port map( clk, Data, ena,loads(0),qout(0),fail(0),rst,noise_clr,noise_set);
l1:
for i in 1 to size-1 generate
        reg:fo_register
                generic map(maj_depth=> maj_depth, restore=>restore,
                                width=>width,ext_error=>false)
                port map( clk, Data, ena,loads(i),qout(i),fail(i),rst,'1','1');
end generate;

l2:
if form_error=true generate
        process (fail)
        begin
                f<='0';
                for i in 0 to size-1 loop
                        if fail(i)='1' then
                                f<='1';
                                exit;
                        end if;
                end loop;
        end process;
end generate;

l3:
if form_error=false generate
        f<='1';
end generate;

-- Вывод сигнала
Q<=qout(rd_addr);

end architecture;
```

ПРИЛОЖЕНИЕ J. СОДЕРЖАНИЕ CD-ДИСКА С ПРИЛОЖЕНИЯМИ

К диссертации прилагается CD-диск с полной информацией по проекту. Из-за большого объёма многие вещи (например, исходные коды прототипа средства) приведены только на диске. В данном приложении описано его содержание с краткими комментариями.

В таблице 1 приложения приведена информация о содержимом корневого раздела диска. В колонке комментариев указано содержимое каждой директории или документа.

Таблица 1
Содержание диска с приложениями

Файл / директория	Комментарии
Readme.pdf	Файл с кратким описанием содержимого диска
Disser	Директория с информацией по магистерской диссертации: - магистерская диссертация; - презентация на защиту.
Additional	Дополнительные материалы по диссертации - план магистра; - библиография из Zotero; - публикации автора по тематике.
Dist	Демонстрационная сборка прототипа
Source samples	Директория с примерами исходных кодов прототипа - исходные коды прототипа на Java; - исходные коды на VHDL.
Doc	Документация на прототип - Частичная UML-спецификация; - Документация на исходные коды, сгенерированная Javadoc; - Документация на исходные коды, сгенерированная Doxygen.

Кроме диска, дополнительную информацию по диссертации можно найти в системе управления проектом, которая расположена по адресу https://nenhome-apps.sourcerepo.com/redmine/nenhome.