федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

На правах рукописи

Пенской Александр Владимирович

Разработка и исследование архитектурных стилей проектирования уровневой организации встроенных систем

05.13.12 – Системы автоматизации проектирования (приборостроение)

Диссертация на соискание ученой степени

кандидата технических наук

Научный руководитель

к.т.н.

Ключев Аркадий Олегович

Санкт-Петербург – 2016

Оглавление

Введение

Актуальность темы. При разработке сложных вычислительных систем широко зарекомендовало себя многоуровневое иерархическое проектирование. Если выделять в системе относительно независимые уровни организации вычислительного процесса, то снижается сложность и затраты на разработку системы, а также обеспечивается разделение и специализация труда (A. Sangiovanni-Vincentelli, Непейвода Н.Н.). Совокупность уровней и их взаимосвязей в вычислительной системе называется уровневой организацией. Выделение уровней обуславливается классом вычислительных задач, технологическими и организационными причинами. Наиболее успешные и перспективные из них становятся самостоятельными вычислительными платформами (ВПЛ) с собственной «экосистемой». Например, процессорные архитектуры и системы команд (ISA), языки программирования общего и специального назначения, операционные системы, библиотеки и фреймворки, программируемые логические интегральные схем (ПЛИС) и т.д.

Многоуровневое иерархическое проектирование занимает особое место во встроенных и кибер-физических системах (ВСС и КФС; S. Seshia) ввиду заказного характера разработки, специализированной аппаратуры, ограниченных ресурсов, распределённой организации, требований реального времени. Использование таких понятий как уровень и ВПЛ имеет ключевое значение в перспективных методиках проектирования ВСС. Таких, как платформно-ориентированное проектирование (Platform-Based Design), модель-ориентированная инженерия (Model-Driven Engineering), контрактно-ориентированное проектирование (Contract-Based Design; R. Passerone) и язык-ориентированное программирование (Language-Oriented Programming; С. Дмитриев, M. Voelter). Вопросы проектирования уровневой организации непосредственно поднимаются в HLD-методологии (Платунов А.Е.) и методологическом направлении совместного проектирования (CoDesign; E.A. Lee, L. Jóźwiak, Терехов А.Н.).

Требования рынка к характеристикам систем и срокам их разработки растут. Это вынуждает проектировщиков отказываться от традиционных решений в пользу более сложных, основанных на принципах реконфигурации и виртуализации, но позволяющих повысить уровень адаптивности и эффективности вычислительных ресурсов (R. Hartenstein, Каляев А.В., Каляев И.А.). Это, в свою очередь, требует детальной проработки уровневой организации ВСС и серьёзной поддержки со стороны САПР.

Формирование уровневой организации относится к слабо автоматизируемой области архитектурного проектирования (M. Shaw, F.H. Cardoso, Топорков В.В.). Основные инструменты

здесь – языки архитектурного описания (Architecture Description Language) и архитектурные стили (Architecture Style). Последние включают методики решения соответствующих задач проектирования и онтологические модели (системы понятий).

Несмотря на широкое распространение многоуровневого проектирования, специалисты отмечают следующие проблемы, большинство из которых связаны с вопросом «удержания целого» (Платунов А.Е., D. Densmore, J. Teich, C. Baldwin, P. Clements, M. West, Левенчук А.И.):

- Излишне шаблонное проектирование уровневой организации, подменяющее осознанное принятие решений на «слепое» следование традициям. Многие узкоспециализированые разработчики не видят альтернативных вариантов, что приводит к вырождению процесса исследования пространства проектных решений (Design Space Exploration).
- Недостаточное развитие методов и средств проектирования уровневой организации. Препятствует полноте и точности анализа. Снижает качество принимаемых архитектурных решений и возможность прогнозирования последствий принятых решений. Приводит к смешению интересов при проектировании и росту сложности.
- Потеря концептуальной информации при документировании и передаче проектного опыта. Наличие неопределённости в спецификациях ВСС, терминологии и онтологических моделях уровневой организации.
- Высокая сложность и стоимость создания заказных элементов уровневой организации, в том числе САПР. Эта проблема остро стоит для методологического направления совместного проектирования, реконфигурируемых вычислительных архитектур и «сквозных» (cross-level) решений.

Стремительно растущий рынок ВСС и КФС сталкивается с серьёзными ограничениями существующих методов и инструментов проектирования уровневой организации, препятствующими разработке и внедрению новых вычислительных архитектур. Это формирует следующую *актуальную научную задачу*: развитие методов и инструментов архитектурного уровня для разработки встроенных систем с многоуровневой организацией, что определило направление диссертационного исследования.

Объект исследования – процессы проектирования и разработки встроенных систем, архитектура встроенных систем в части уровневой организации.

Предмет исследования – архитектурные стили для проектирования и документирования уровневой организации встроенных систем, методики проектирования встроенных систем и методики моделирования элементов встроенных систем в САПР.

Цель диссертационной работы – повышение качества архитектурных решений в части уровневой организации встроенных систем и сокращение затрат на разработку заказных элементов уровневой организации за счёт развития научных основ построения САПР, а именно архитектурных стилей и методик моделирования встроенных систем.

В соответствии с целью, в работе ставятся и решаются следующие *задачи*:

1) Исследование архитектурных стилей, предназначенных для работы с уровневой организацией встроенных систем. Определение структуры пространства проектных решений в части уровневой организации и формирование требований к более эффективным архитектурным стилям.
2) Разработка архитектурных стилей и языков архитектурного описания для проектирования и документирования уровневой организации встроенных систем.
3) Формализация разработанных архитектурных стилей с целью создания методики моделирования многоуровневых встроенных систем и их элементов для САПР.
4) Развитие системы архитектурных абстракций для работы с уровневой организацией в соответствии с актуальными тенденциями в проектировании встроенных систем.
5) Модификация традиционной методики проектирования встроенных систем на основе предложенных архитектурных стилей, языков архитектурного описания, методик моделирования и архитектурных абстракций.

Методы исследования. При решении поставленных задач использовались методы системного, аспектного и архитектурного анализа; функционального и объектно-ориентированного программирования; теория категорий, метод структурирования функции качества, методы и приёмы моделирования высших онтологий.

Основные положения, выносимые на защиту:

1) Архитектурные стили *«модифицированный граф актуализации»*, *«системно-иерархический»* и *«модель-процесс-вычислитель»* для проектирования и документирования уровневой организации, расширяющие пространство проектных решений, повышающие качество документации архитектурного уровня и выделяющие соответствующие вопросы организации встроенных систем.
2) *Методика моделирования многоуровневых встроенных систем и их элементов*, предназначенная для проектирования компонентов САПР в составе заказных элементов уровневой организации и сокращающая затраты на их разработку.
3) *Расширение системы архитектурных абстракций* для работы с уровневой организацией встроенных систем и *методика проектирования многоуровневых*

встроенных систем, предназначенные для унификации рассмотрения уровневой организации, формализации объектов повторного использования и повышения качества проектирования многоуровневых встроенных систем.

Результаты, характеризующиеся *научной новизной*:

1) Стили *«модифицированный граф актуализации»* и *«системно-иерархический»*, являющиеся модификациями существующих. Первый отличается от своего прототипа принципом декомпозиции систем, второй – добавленными средствами представления стадий жизненного цикла встроенной системы.
2) Стиль *«модель-процесс-вычислитель»*, разработанный с использованием методов моделирования высших онтологий и отличающийся абстрагированием от способа реализации, высокой детализацией уровневой организации, отсутствием избыточности и однозначностью системы понятий.
3) *Методика моделирования многоуровневых встроенных систем и их элементов*, являющаяся формализацией архитектурного стиля «модель-процесс-вычислитель» с использованием теории категорий и отличающаяся полнотой представления уровней встроенных систем и межуровневых взаимосвязей.
4) *Расширение системы архитектурных абстракций*, включающее:
   - уточнённые понятия *вычислительной* и *системной платформы, уровня* и *уровневой организации*;
   - понятие *многоуровневого вычислительного агрегата*;
   - онтологические модели *[ре]конфигурации* и *иерархической организации*.
5) *Методика проектирования многоуровневых встроенных систем*, отличающаяся использованием предложенных архитектурных стилей, разработанной методики моделирования и расширенной системы архитектурных абстракций.

Практической ценностью характеризуются следующие результаты:

1) *Языки архитектурного описания* для предложенных архитектурных стилей. Применяются для документирования и передачи проектного опыта в области уровневой организации встроенных систем.
2) *Набор формальных моделей* элементов уровневой организации встроенных систем для применения в САПР. Включает в себя модели: языка макроассемблера, языка структурных схем, вычислительного процесса фон Неймановского процессора в

разных режимах работы, тактовые модели актуализации и виртуализации, модели специализированного сигнального процессора.

3) САПР сигнального процессора реального времени NL3 и САПР с элементами сквозного моделирования и отладки для работы со специализированным реконфигурируемым фон Неймановским процессором.

Обоснованность и достоверность научных положений обеспечены полнотой анализа теоретических и практических исследований, положительной оценкой на конференциях, семинарах и научных конгрессах, практической проверкой и внедрением полученных результатов исследований в производственную деятельность.

Реализация и внедрения результатов работы. Результаты работы были успешно использованы в следующих НИР и ОКР:

1) Разработка и исследование аспектно-ориентированных технологий проектирования на базе унифицированных элементов информационно-коммуникационной инфраструктуры активно-адаптивных энергосетей (ГК № 07.514.11.4073 от «13» октября 2011 г., Шифр: 2011-1.4-514-120-048).
2) Исследование механизмов обеспечения надежности аппаратно-резервированных информационно-измерительных систем на базе ПЛИС (Университет ИТМО, № 214434).
3) Создание бесшовных технологий проектирования встраиваемых систем и систем на кристалле на основе реконфигурируемых архитектур (Университет ИТМО, № 713564).
4) Разработка методов и средств системотехнического проектирования информационных и управляющих вычислительных систем с распределенной архитектурой (Университет ИТМО, № 610481).
5) Нелинейное и адаптивное управление сложными системами (Университет ИТМО, № 713546).
6) Система коммерческого учёта электроэнергии «ИИС Луч-ТС М» (свидетельство об утверждение типа: RU.C.34.033.A № 57631).

Также, положения работы были успешно применены при разработке: редактора архитектурных спецификаций информационно-управляющих систем (свидетельство № 2012618605 от 21.09.2012), редактора спецификаций технических и программных средств информационно-управляющих систем (свидетельство № 2012618606 от 21.09.2012), библиотеки для анализа слабо формализованных текстовых данных (свидетельство № 2015615882 от

26.05.2015), библиотеки для разработки программных интерфейсов управления мобильным оборудованием (свидетельство № 2015614306 от 14.04.2015), виртуальной машины для задач управления беспилотным летательным аппаратом.

Результаты работы использованы на кафедре вычислительной техники Университета ИТМО в учебном процессе по курсам: «Программное обеспечение встраиваемых вычислительных систем», «Информационно-управляющие системы», «Организация ЭВМ и систем», «Интерфейсы периферийных устройств», «Управление программными проектами».

Апробация работы. Научные результаты и положения диссертационной работы докладывались и обсуждались на 15 конференциях в 18 докладах: научная и учебно-методическая конференция Университета ИТМО (2011, 2012, 2013, 2014, 2015 гг.); научно-практическая конференция молодых ученых «Вычислительные системы и сети, Майоровские чтения» (2010, 2011, 2012, 2013 гг.); II Всероссийский конгресс молодых ученых (2013 г.); II Международная научно-практическая конференция «Sensorica – 2014» (2014 г.); Международная конференция Mediterranean Conference on Embedded Computing – MECO (2012, 2014 гг.); Международная конференция ICUMT 2014 - the 6th International Congress on Ultra Modern Telecommunications and Control Systems (2014 г.); Международная конференция 14th international multidisciplinary scientific geoconference – SGEM2014 (2014 г.).

Публикации. По теме диссертационной работы опубликовано 8 печатных работ, в том числе 6 работ в изданиях ВАК или входящих в список Scopus, а также 2 учебно-методические работы. Зарегистрировано 4 программных продукта.

1 Архитектурное проектирование встроенных систем

1.1 Проектирование встроенных систем

Современные встроенные вычислительные системы (ВСС) получили широкое применение во всех областях человеческой деятельности: устройства личного пользования (часы, телефоны, бытовая техника, автомобили), системы автоматизации производств (управляющие контроллеры, роботы), территориально распределённые системы (системы сбора и анализа данных, сенсорные сети, энергетические системы), глобальные системы (метеорологические системы, системы глобального позиционирования, системы связи) и многие другие. ВСС решают задачи автоматизации, управления, сбора, передачи и обработки данных. Заинтересованность индустрии в разработке новых, более функциональных, производительных, надёжных, дешёвых и энергетически эффективных ВСС отмечается множеством ведущих специалистов в области [1,2]. Как отмечается в работе [3], объём рынка ВСС растёт экспоненциально (Рисунок 1), что говорит о перспективности разработок в данной области.

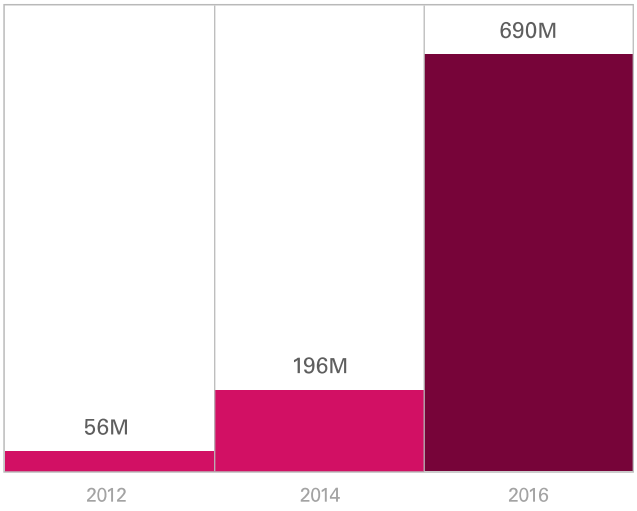


Рисунок 1 – Оценка тенденции роста объёма рынка встроенных систем

За время своего существования, класс ВСС неоднократно расширялся, делая вызов разработчикам. Ниже приведены основные вехи этого процесса:

1) Информационно-управляющие системы (ИУС [4,5]). Под ИУС обычно понимается вычислительная система, вынесенная за пределы объекта управления и описываемая классической схемой управления в кибернетике (управляющая система, управляемая система, прямая и обратная связь).
2) Распределённые встроенные системы (РВСС [6]). В узкой трактовке, под РВСС принято рассматривать ВСС, размещённую в рамках управляемого объекта, но при

этом децентрализованную. Необходимость децентрализации обусловлена размером объекта управления. Стоимость распределённой системы, как правило ниже, чем стоимость проводки кабелей от всех датчиков и исполнительных устройств. Кроме того, централизованная организация не всегда возможна из-за требований реального времени, помех, надёжности и других технических причин.

3) Кибер-физические системы (КФС [7–9]). Относительно новый класс ВСС, отличительной особенностью которого, является системообразующая роль вычислительной техники. Во ВСС и РВСС, объект управления может быть рассмотрен, как система в отсутствии вычислительной составляющей, в случае КФС – подобное рассмотрение не является корректным, так как функциональные возможности управляемой части неразрывно связаны с вычислительной составляющей. Классический пример КФС – робототехнические системы.
4) Социо-кибер-физические системы. В настоящий момент находятся на этапе становления. Для них характерно взаимодействие с социальными структурами и организациями (Physical-Cyber-Social Systems [10]).

Общее для всех приведённых классов систем – непосредственное взаимодействие с объектом управления, что соответствует определению ВСС из работы [6]:

Встроенные вычислительные системы (ВСС) — специализированные (заказные) вычислительные системы (ВС), непосредственно взаимодействующие с объектом контроля или управления [и объединённые с ним единой конструкцией].

Данное определение будет использоваться в качестве основного. Рассмотрим подробнее отличительные особенности ВСС [8,6]:

1) Непосредственная интеграция с объектом управления накладывает ограничения, связанные с температурными режимами, физическим размером, тепловыделением, энергопотреблением, средствами защиты от условий окружающей среды (пыль, вода, вибрация, температура, электромагнитные помехи) и соответствующими человеко-машинными интерфейсами (элементы управления и индикации). В совокупности, это влияет на элементную базу, ограничивая применение мощных процессоров общего назначения, графических процессоров и жёстких дисков с большим объёмом памяти.
2) Управление физическими процессами обуславливает ограничение реального времени [5]. Поэтому, ВС должна гарантировать время реакции системы на событие. Обеспечить это за счёт многократного запаса вычислительной мощности не всегда получается ввиду ограниченных ресурсов.

3) Значительная часть ВСС имеет повышенные требования [4] к надёжности и качеству, так как условия эксплуатации системы делают невозможным её обслуживание, а сбой может привести к человеческим жертвам или техногенным катастрофам.
4) В отличие от множества информационных систем, являющихся программными продуктами, для ВСС характерно использование специализированной аппаратуры. Это приводит к необходимости рассмотрения этапа «производства» и учёта технологических требований. В случае средне- и крупносерийного производства ВСС, актуальными становятся вопросы себестоимости элементной базы и процесса производства. В случае, если ВСС необходимо воспроизводить в течении многих лет, то необходимо учитывать срок жизни элементной базы, также отражающийся на стоимости проекта.
5) Сложность разработки ВСС значительно превышает сложность разработки информационных систем из-за необходимости проведения работ на большем количестве уровней организации системы [11,12] (Рисунок 2). Если для создания большинства приложений для ПК необходима работа с фреймворком и языком программирования высокого уровня, то для ВСС нужно низкоуровневое программирование с учётом специфики используемой аппаратуры (bare metal [8]), цифровая и аналоговая схемотехника.

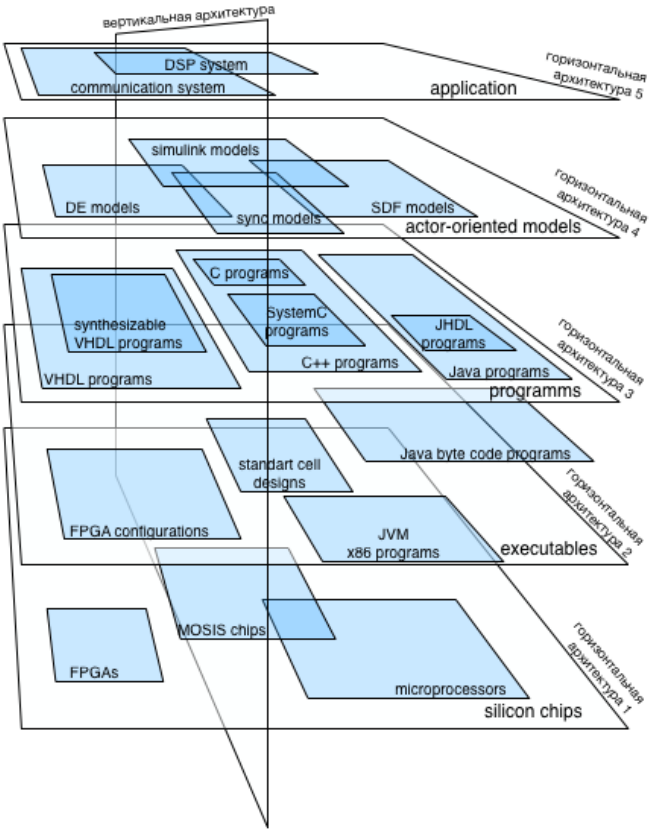


Рисунок 2 – Иерархия вычислительных платформ

1.1.1 Особенности проектирования встроенных вычислительных систем

Особенности проектирования ВСС непосредственно вытекают из их специфики. Центральное место в процессе проектирования занимает пространство проектных решений (ППР) [13–15] – многомерное пространство, осями которого являются технические вопросы, точками – комплексные технические решения. Эти технические решения формируют архитектуру и реализацию разрабатываемой ВСС.

Структурой ППР является совокупность рассматриваемых вопросов с вариантами ответов на них [16]. Пример её визуализации приведён на рисунке (Рисунок 3). Процесс проектирования рассматривается как процесс анализа вариантов и принятия решений. Структура ППР ситуативная и определяется опытом и навыками проектировщика. Как следствие, ППР не обязательно должно включать в себя все принципиально важные для успеха проекта решения или варианты («мы не рассматривали такой вариант решения задачи»). В тоже время, ППР может быть избыточным, увеличивая трудозатраты на проектирование («из-за неполных требований, мы потратили ресурсы на анализ не подходящих вариантов»).

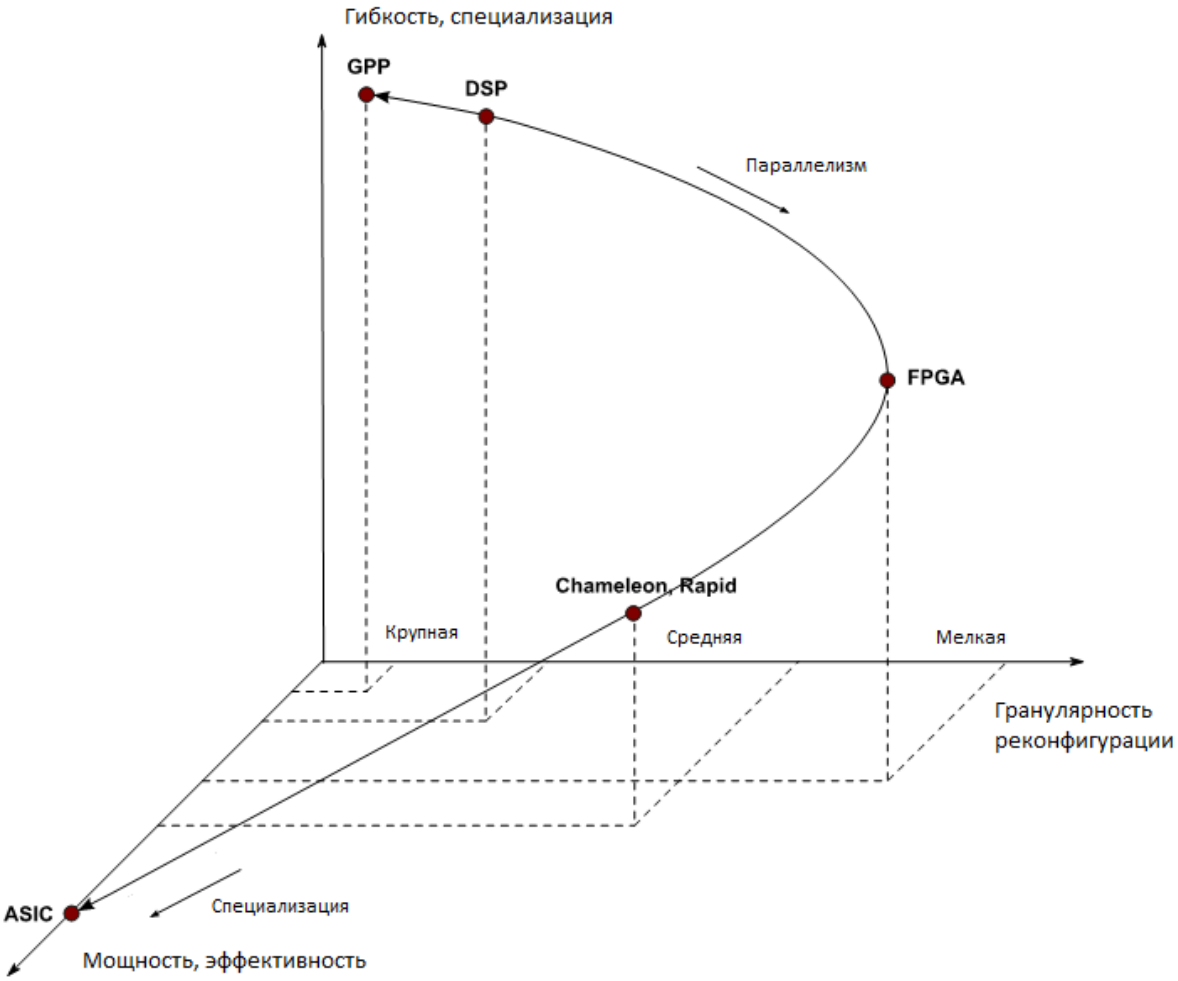


Рисунок 3 - Пример структуры пространства проектных решений

Ситуативность структуры ППР основывается на следующем:

1) На опыте проектирования и управления проектами, так как регулярно возникают ситуации нерассмотренных проблем или вариантов решений. Недостаток знаний, опыта и навыков не позволяет корректно определить структуру ППР.
2) На теоретической невозможности полного доступа к информации ввиду того, что в основе работы проектировщика лежит акт творческой деятельности –постоянно порождаются новые варианты решения известных вопросов и новые точки зрения на проблему, позволяющие выявлять новые проблемы. Детально вопрос распространения творческой информации между людьми освящён в работах австрийской школы экономики [17], где теоретически обосновывается данный тезис.

Помимо вышеизложенного, структура пространства проектных решений зависит от того, на каком этапе находится процесс разработки ВСС. На Рисунок 4 [14] показан пример классификации проектного пространства. Можно видеть, что уровни рассмотрения проектного пространства выделяются в зависимости от общности/гранулярности принимаемых решений. Верхним является уровень архитектурного рассмотрения, а нижним – частные вопросы реализации (например, имя переменной).

Рисунок 4 – Уровни проектного пространства

Типовой жизненный цикл ВСС [18] приведён на Рисунок 5. На нём показана необходимость деятельности различных специалистов (в области аппаратного и программного обеспечения). Это приводит к декомпозиции проектного пространства по группам интересов, исходя из специализации разработчика. Принимаемые решения в разных областях ППР, не

являются независимыми. Это формирует необходимость в высокоуровневых и совместных методиках проектирования ВСС (подробнее в пунктах 1.1.2.1 и 1.1.2.4), которые позволили бы минимизировать проблемы, вызванные:

- высоким уровнем гетерогенности, который приводит к росту объема ППР и сегментации его по специализациям;
- высоким уровнем специализации разработчиков (высокая сложность области);
- взаимосвязанностью элементов, разрабатываемых независимо.

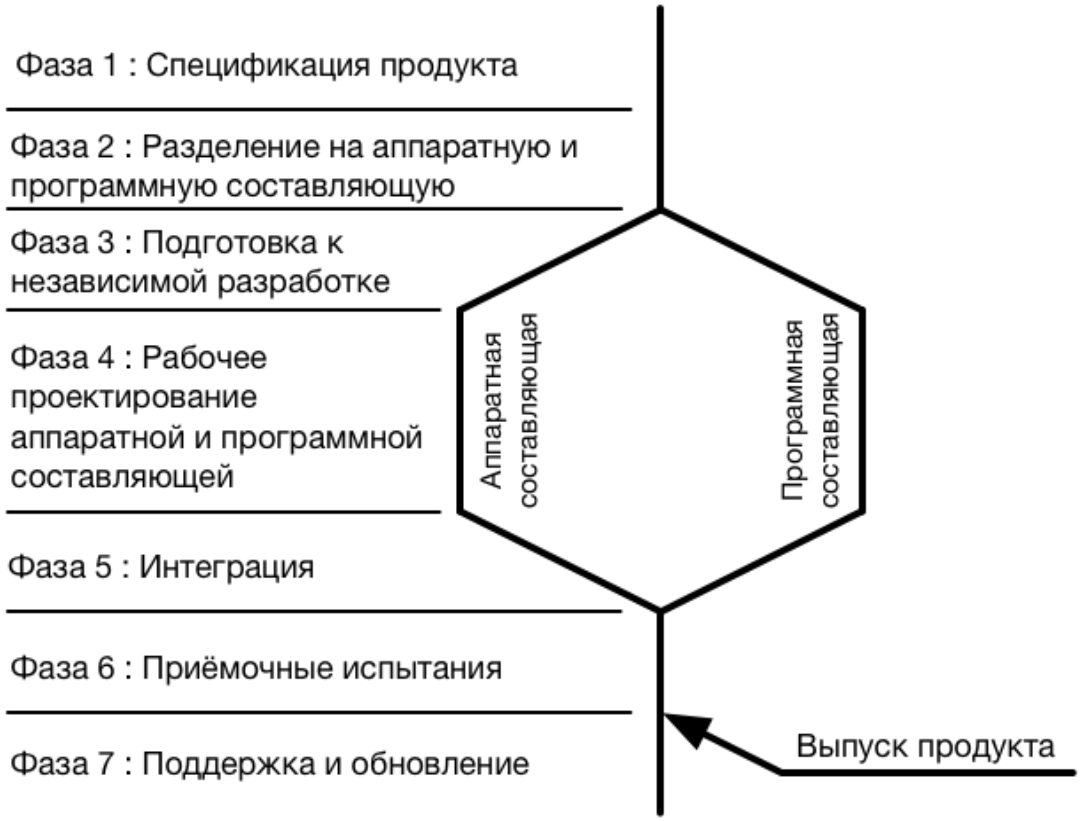


Рисунок 5 – Типовой жизненный цикл встроенной системы

Помимо обозначенного выше, особый вид принимает проблема повторного использования. Из-за того, что большинство разработок выполняется под заказ, система создаётся под конкретное, зачастую, единственное применение (например, управление насосом модели «НСБ04»). В условиях ограниченных ресурсов это заставляет идти на компромисс между повторным использованием и эффективностью. Повторное использование может быть обеспечено следующими путями:

- адаптацией существующих наработок под новый проект, что приводит к разделению кодовых баз, провоцирует некорректное повторное использование, сложности поддержки и другие проблемы;

– проектированием объектов повторного использования, при котором должны учитываться применения в других проектах и минимизироваться накладные расходы, что приводит к росту сложности реализации.

Доступные объекты повторного использования в значительной степени определяют соотношение стоимости, сроков и функциональности, разрабатываемой ВСС. Классическими объектами повторного использования являются модули, определяемые в рамках конкретных вычислительных платформ [19,20] (к примеру: подпрограмма, сопроцессор, элементная база, класс…) и шаблоны проектирования [21,22] (определяющие, как именно следует реализовывать конкретную функциональность). Кроме того, существуют более сложные и перспективные варианты: вычислительные платформы [23,24] (раздел 1.2), реконфигурируемые вычислители [25,26] и совместные виртуальные машины (пункт 1.1.3.1).

1.1.2 Проблемы проектирования встроенных вычислительных систем

Рассмотренные выше особенности ВСС и особенности процесса их проектирования позволяют определить основные проблемы, характерные для данной области. Обозначенные проблемы приводят к снижению качества принимаемых архитектурных решений при проектировании встроенных систем и должны быть устранены частично или полностью.

Часть рассмотренных в данном разделе проблем носят «вечный» характер и, судя по всему, смогут быть полностью решены лишь с появлением искусственного интеллекта. Тем не менее, работа над ними не теряет актуальности, так как многие полученные решения приводят к изменению структуры индустрии, делая процессы создания вычислительных систем более быстрыми, качественными и эффективными. Ярким примером является появление структурного программирования, объектно-ориентированного анализа, ПЛИС и так далее.

1.1.2.1 Суженное пространство проектных решений, излишне шаблонное проектирование

Сужение ППР выражается в выборе того или иного решения или шаблона, без рассмотрения и анализа альтернативных вариантов. Как правило, это вызвано [13,27]:

1) Зависимостью пространства проектных решений от ситуации и исполнителя.
2) Фиксацией решения заказчиком на уровне технического задания, вызванной текущими тенденциями и сложившимися/привычными практиками.

Сужение ППР приводит к принятию неэффективных и неоптимальных проектных решений. Оно наиболее характерно для архитектурных решений, так как работа на данном уровне требует высокой квалификации исполнителя, разностороннего рассмотрения (не только

целевой системы [28], но и процесса разработки, доступных кадров, перспективности технологий и многого другого) и при этом имеет наименее автоматизированные средства генерации и анализа решений. Кроме того, стоимость ошибок архитектурного уровня многократно превышает стоимость ошибок реализации ([29,30], Рисунок 6), а ценность самих решений видится низкой, ввиду отдалённости результата во времени [17]. В связи с этим, разработчик психологически не склонен тратить на их принятие серьёзные ресурсы.

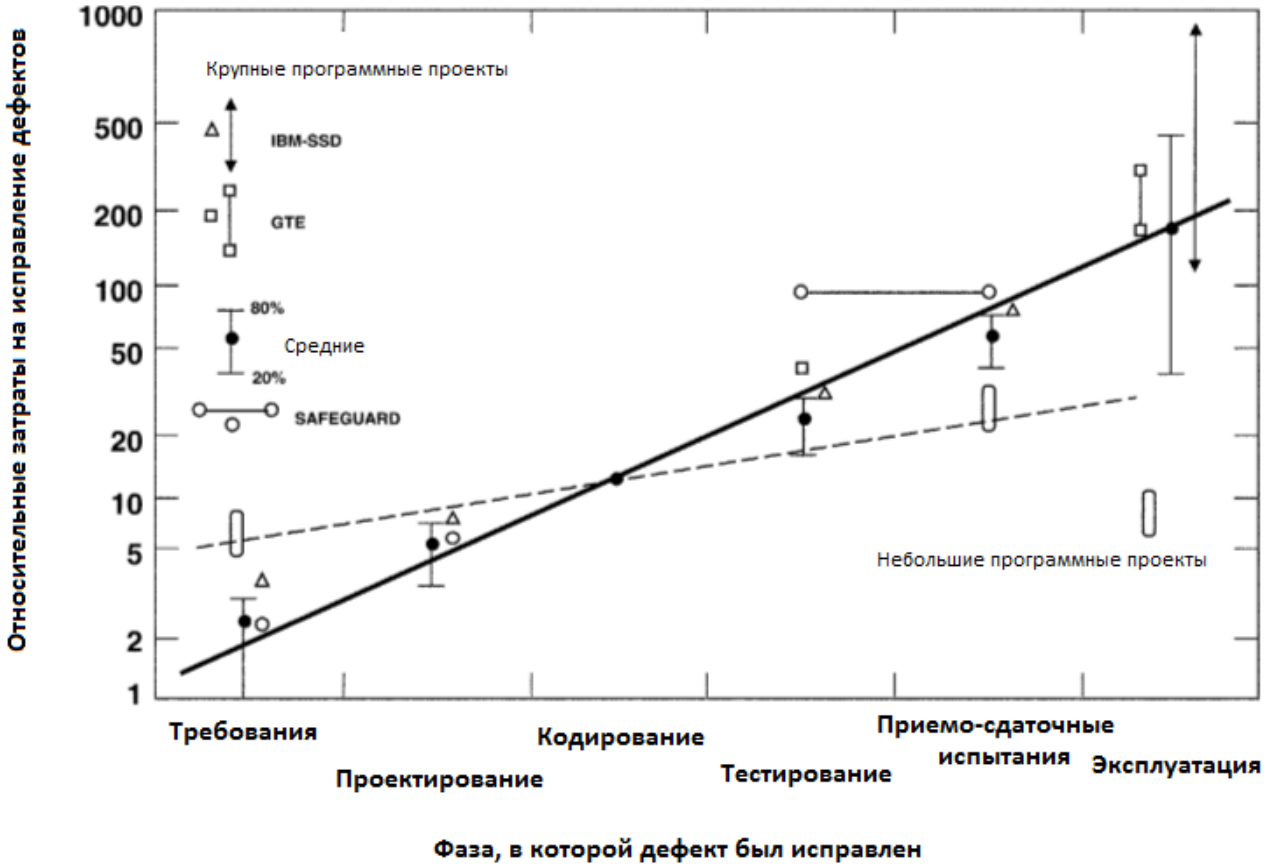


Рисунок 6 – Зависимость стоимости исправления ошибки от стадии, на которой она обнаружена

Увы, способа устранения искусственного сужения проектного пространства на сегодня не представлено. Работы по автоматизации исследования отдельных элементов проектного пространства ведутся весьма активно [14–16], но, как правило, они не выходят на общесистемный уровень и ведутся в рамках строго зафиксированного шаблона вычислительной архитектуры. Задача по исследованию ППР без фиксации заданного набора шаблонов, близка к проблеме создания сильного искусственного интеллекта.

Расширение ППР может осуществляться за счёт применения соответствующих методик проектирования, вынуждающих разработчика фиксировать принимаемые решения и, как следствие, делающие возможным их критический анализ. Описанными выше вопросами занимаются в рамках архитектурного проектирования, центральным инструментом которого являются архитектурные стили.

1.1.2.2 Сохранение и передача проектного опыта

Самые значимые технические решения при проектировании ВСС принимаются на архитектурном уровне. К сожалению, архитектурные решения и процесс их принятия имеют довольно низкий уровень формализации и автоматизации. Это обусловлено:

- высокой размерностью задачи (количество анализируемых объектов, методологические проблемы анализа их различий и многогранность вариантов их рассмотрения);
- отсутствием прямых критериев оценки эффективности принятых решений;
- сложностью установления корреляции между принятыми архитектурными решениями и интегральными характеристиками проекта (корреляция легко устанавливается для ошибочных решений, в отличие от «правильных решений»).

Часть технических вопросов организации ВСС хорошо формализована. В первую очередь это структурные описания вычислительной системы (компонентные архитектурные стили), структуры данных, классификации и процессы их взаимодействия в системе и с системой. Другие, менее распространённые представления системы, имеют крайне низкий уровень формализации и инструментальной поддержки (к примеру, аспектное программирование [31]), что формирует проблему сохранения и передачи проектного опыта, решение которой позволило бы значительно повысить эффективность разработки.

Одним из основных инструментов решающими проблему передачи проектного опыта, являются шаблоны проектирования (design patterns [22]) и языки архитектурного описания (Architectural Description Language [32]). В них аккумулируется накопленный опыт и практики проектирования.

Наличие открытых исходных кодов или документации (даже архитектурного уровня), как правило, не позволяет эффективно передать проектный опыт:

1) Работа с исходным кодом имеет высокую стоимость восстановления информации «архитектурного уровня» об организации системы.
2) Зафиксированная информация об архитектуре системы редко содержит сведения, как формировалась данная архитектура и оценку её эффективности.

1.1.2.3 Сложность встроенных систем

Специалистами отмечается постоянный рост сложности ВСС (см. Рисунок 7, [2]). Это связано с возрастающими требованиями к их функциональности и характеристикам [7], а также с развитием элементной базы, предоставляющей все большие возможности [2,33].

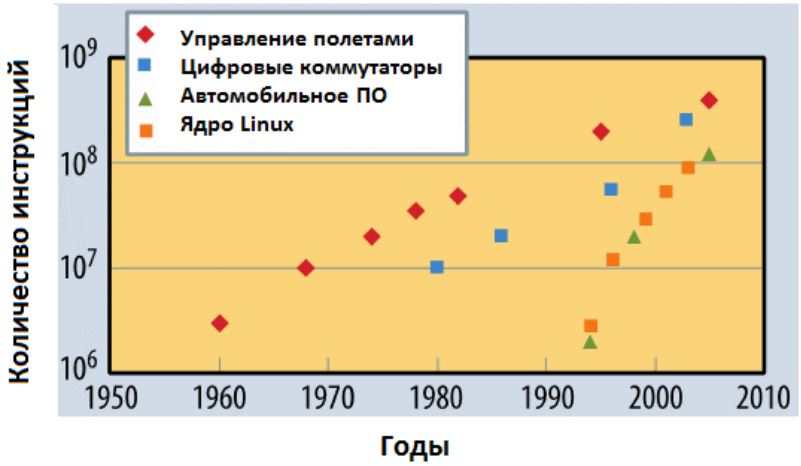


Рисунок 7 – Рост сложности встроенной системы

Основными способами решения этой проблемы являются:

1) Формирование новых моделей вычислительных систем и процессов, позволяющих сократить нагрузку на разработчика за счёт сокращения количества объектов и инвариантов [20,34,35]. Данный способ более общий чем абстракция, так как не только скрывает избыточные данные, но и позволяет сменить точку зрения (например, рассмотрение системы как конечного автомата, группы взаимодействующих процессов или физической конструкции).
2) Формализация процесса проектирования, позволяющая фиксировать в виде артефактов промежуточные результаты. Это упрощает процесс их анализа, позволяет кооперироваться с другими разработчиками и даёт возможность автоматизировать работу с промежуточными результатами. Также даёт возможность вводить формальные критерии для моделей, такие как: отсутствие ошибок типов, время исполнения, корректность реализации протокола [36], отсутствие взаимных блокировок [37] и многие другие.
3) Автоматизация процесса проектирования, перекладывающая часть задач с разработчика на компьютер (к примеру, автоматизация работы с памятью или автоматизированная трассировка печатных плат).

1.1.2.4 Смешение интересов при проектировании встроенных систем

Единовременная разработка ВСС целиком невозможна из-за размера и сложности современных систем и специализации разработчиков [38]. По этой причине работа с ППР ведётся поэтапно, сегментами разной гранулярности и общности, а также выстроенных с разных точек зрения. Выбор сегментов и их этапов определяет сложность разработки ВСС. Избыточность сегментов приводит к рассеиванию внимания и росту сложности; недостаточность – к сужению ППР. Неправильный набор этапов приводит к росту рисков. Такой подход к проектированию

получил название «принцип разделения интересов» (separate of concerns [34]) и широко используется в различных методологиях и инструментальных средствах, к примеру:

1) Стандарт Essence [39], описывающий язык для работы с методологиями проектирования программного обеспечения. Проект здесь рассматривается с 7 точек зрения, называемых ALPHA.
2) Языки моделирования систем UML [40] и его расширение SysML [41], в состав которых входит множество диаграмм, адресованных разным интересам, xtUML [42].
3) Методологии проектирования и работы с архитектурой ВСС: HLD-методология [27,35] (аспект), RUP [43] (view – представление), ISO 42010 (viewpoint – точка зрения), Capella [44] и другие.
4) Методы моделирования предметной области, основанные на логической парадигме: BORO Method [45], ISO 15926 [46], DoDaF IDEAS [47] и другие высшие онтологии.
5) Методы декомпозиции программного кода по интересам, наиболее ярким из которых является аспектное программирование [31].
6) И многие другие [20,48,49].

Несколько искусственным может показаться разговор об уровневой организации вычислительных систем (см. Рисунок 2), как о проявлении принципа разделения интересов, но по всем признакам это так. Существует множество взаимосвязанных описаний единого объекта (вычислительно процесса), каждое из которых адресовано к своей группе интересов. Впечатление искусственности вызвано наличием строгой взаимосвязи между уровнями, в рамках которой вышележащие уровни реализуются нижележащими.

На Рисунок 8 показано, что совместная разработка программной и аппаратной составляющей позволяет сократить время разработки и снизить интеграционные риски [29,30]. Работы над технологиями совместного проектирования ведутся в рамках методологического направления совместного проектирования (CoDesign [1,50]) и HLD-методологии [27].

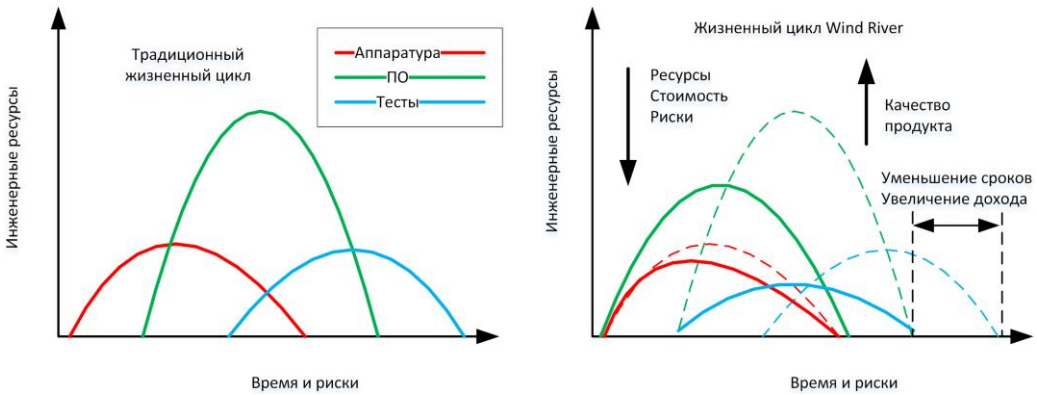


Рисунок 8 – Ранняя интеграция элементов встроенных систем, реализованных различными вычислительными платформами

Данная проблема особенно актуальная для ВСС ввиду требований реального времени и ограниченных ресурсов, приводящих к смешению уровней между собой в сильно связанные конструкции [9], см. Рисунок 9.

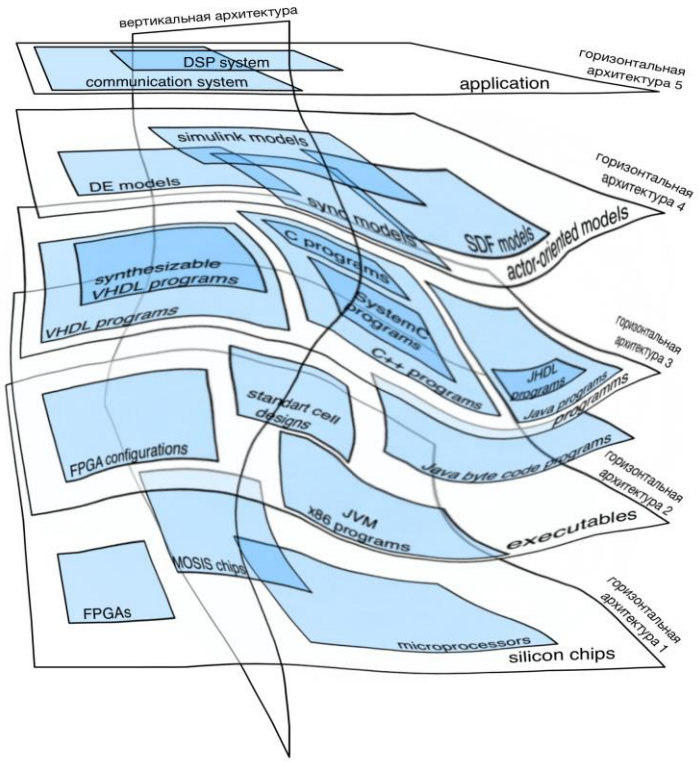


Рисунок 9 – Смешение вычислительных платформ в условиях ограниченных ресурсов и требований реального времени

Современным методологическим базисом, на основе которого возможно разделение интересов при анализе ВСС, является философская логика (иногда именуемая логической парадигмой) [45]. В ней постулируется отсутствие субстанции и рассматриваемых объектов, что позволяет совмещать работу со множеством представлений ВСС (в рамках которых даже границы системы могут быть заданы по-разному). Её применение явно или по косвенным признакам прослеживается во всех приведённых выше примерах.

1.1.2.5 Специализация разработчиков встроенных систем

Как было показано в подразделе 1.1.1, традиционный жизненный цикл ВСС подразумевает разработку программной и аппаратной составляющей вычислительной системы, которые, как правило, производятся различными специалистами. Реальное число специальностей, необходимое для разработки типовой ВСС выше [51] (Рисунок 10), при этом каждая из них может дополнительно специализироваться в зависимости от используемых технологий (к примеру выбор производителей ПЛИС или языка программирования).

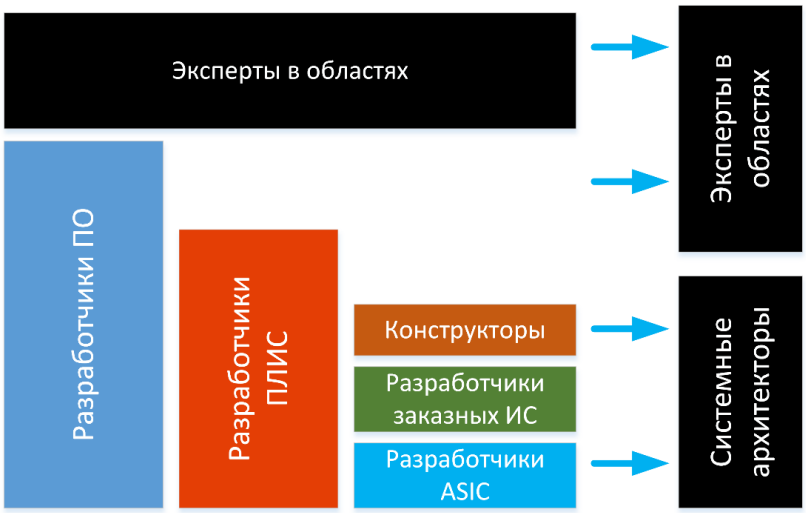


Рисунок 10 – Компетенции, необходимые для разработки встроенных систем

Работа с людьми разных специальностей делает необходимой организацию их совместной работы на всех этапах жизненного цикла системы. Это во многом реализуется за счёт принципа разделения интересов. Основные методы решения этой проблемы лежат в области высокоуровневого проектирования и совместного моделирования.

Высокоуровневое проектирование [52,53] ориентируется на работу с архитектурой ВСС в независимом от способа реализации виде. Это позволяет спроектировать общие принципы работы системы в целом. Далее, становится возможным сформировать набор согласованных частных технических заданий для специалистов конкретных областей. На Рисунок 11 приведено схематическое изображение процесса высокоуровневого проектирования, заимствованное из материалов проекта Capella [44]. К примерам проектов в области высокоуровневого проектирования относится HLD-методология [27], SysML [41], Ptolemy [54] и другие.

Рисунок 11 – Работа с точками зрения в системе Capella

Совместное моделирование относится к более поздним этапам разработки ВСС. На этих этапах работают специалисты конкретных областей. Средства совместного моделирования обеспечивают процессы тестирования, отладки и взаимодействия рабочих групп между собой. Также они позволяют осуществлять раннее прототипирование (rapid prototyping). Примерами систем совместного моделирования являются: Proteus [55], SystemC [56].

1.1.3 Тенденции в области методик проектирования встроенных систем

На сегодняшний день, проектирование ВСС актуально с экономической точки зрения и представляет интерес с научной. Оно требует работы с вычислительной системой в целом, с учётом современных и прогрессивных технологий и методов. Ранее были следующие тенденции:

1) Высокоуровневое проектирование.
2) Совместное проектирование (CoDesign).
3) Применение логической парадигмы при моделировании предметной области.

Далее будут рассмотрены другие важные тенденции.

1.1.3.1 Рост адаптивности встроенных вычислительных систем. Реконфигурируемые вычислители

Большинство современных ВСС содержат как аппаратную, так и программную составляющую. Причём стоимость разработки программной составляющей современных ВСС многократно выше (см. Рисунок 12). Это говорит о том, что большинство ВСС относится к классу систем с преобладающей программной составляющей (Software-Intensive Systems [57]). Их отличительной особенностью является возможность изменения их функциональности после производства, путём замены программного обеспечения. Это позволяет:

1) Радикально сократить стоимость итерации разработки (в пределе – REPL [58,59]).
2) Сократить время вывода продукта на рынок за счёт выпуска обновлений, добавляющих новую функциональность [и исправляющую ошибки].
3) Снизить стоимость поддержки системы, ограничившись выпуском обновлений и заплаток (patch), в противовес полноценной службе гарантийного ремонта, необходимой для поддержки аппаратной составляющей.
4) Снизить затраты на разработку единицы функциональности за счёт большей абстрактности и производительности средств разработки ПО.

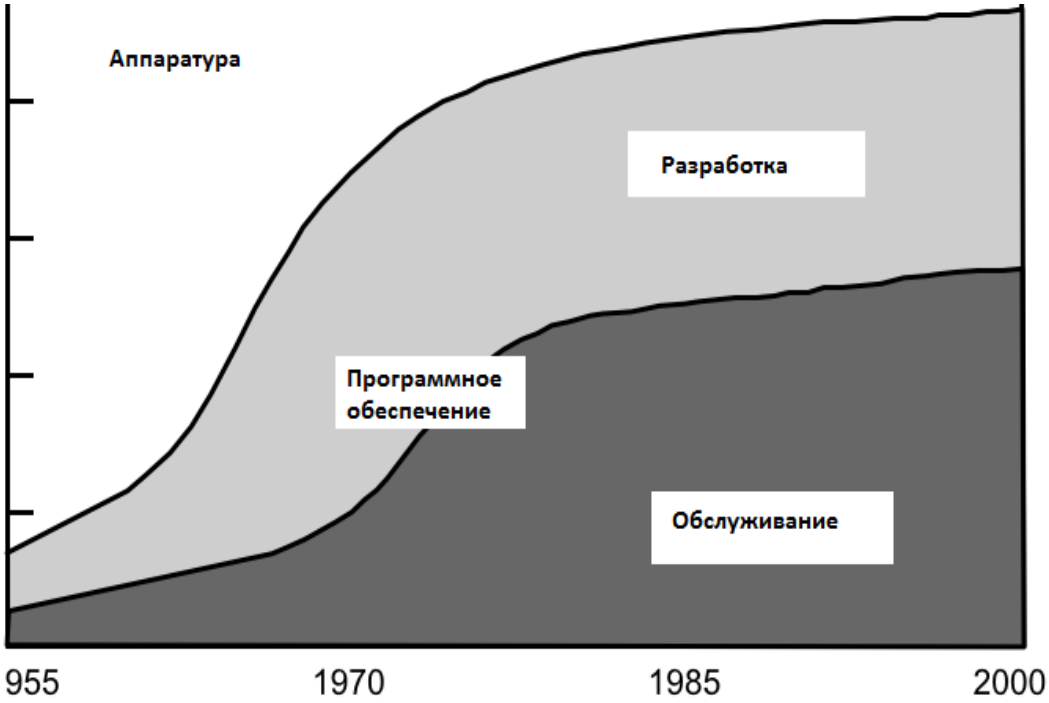


Рисунок 12 – Тенденция изменения стоимости аппаратной и программной составляющей

Создание программируемых логических интегральных схем (ПЛИС) стало прорывом, позволившим значительно расширить область программируемости. Оно предоставило возможность создания интегральных схем для широкого круга разработчиков по низкой цене, пусть и с ограниченными возможностями. Данное явление получило название «*реконфигурируемости*» [1,35,60,61] и изначально подразумевало ПЛИС. Сегодня реконфигурируемые вычислительные системы стали крупным классом, в рамках которого выделяются следующие группы:

1) Системы с мелкой гранулярностью [62], к которым относятся ПЛИС, и крупной гранулярностью [63], в качестве блоков которых выступают устройства АЛУ или самостоятельные процессоры.
2) Однородные и неоднородные вычислители, в рамках которых используются соответственно однотипные или разнотипные блоки обработки данных. К примеру, современные ПЛИС относятся к неоднородным, так как помимо LUT, включат в себя умножители, сумматоры и прочие функциональные элементы.
3) Плоские и иерархические, фиксирующие топологию взаимосвязей блоков обработки данных [64].

Применение реконфигурируемости позволяет не только повысить адаптивность ВСС, но и сократить энергопотребление за счёт более эффективного использования аппаратуры (реконфигурация времени исполнения [1,65,66]). Позволяет снизить стоимость

разработки/производства, создавать узкоспециализированные решения и архитектуры, оптимальные для решения конкретных задач при мелкосерийном производстве.

Другим интересным направлением в области роста адаптивности являются *совместные виртуальные машины* (CoDesign VM [67,68]). С их помощью разработчик может модифицировать/расширить систему команд с целью оптимизации вычислителя под решаемую задачу. При этом вычислитель может быть аппаратным (с микропрограммированием), реконфигурируемым или виртуальной машиной (пример приведён в пункт 4.1.2.2). Это позволяет при помощи локализованных в рамках архитектуры вычислителя изменений, получать новые возможности, реализация которых через систему команд будет менее эффективной. Позиционирование совместных виртуальных машин приведено на Рисунок 13 [67].

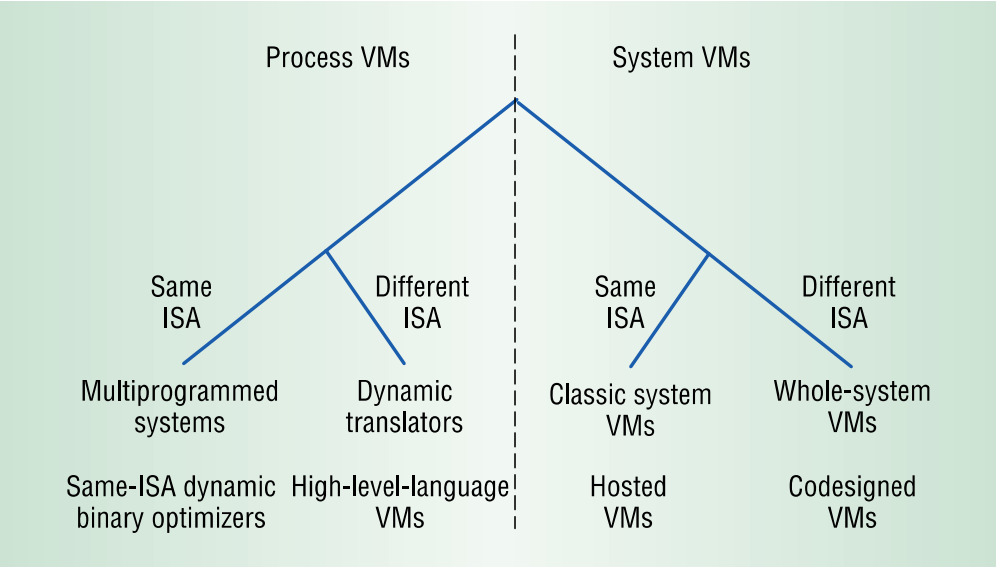


Рисунок 13 – Классификация виртуальных машин [67]

1.1.3.2 Модель-ориентированное проектирование

Модель-ориентированное проектирование [40,69] является одним из вариантов высокоуровневого проектирования. В этом случае разработка вычислительной системы производится в рамках высокоуровневых моделей, транслируемых в исполняемые артефакты. На Рисунок 14 [70] приведена схема жизненного цикла программного обеспечения, разработанного в рамках модель-ориентированного проектирования. Выделяются следующие этапы:

1) Модели, независимые от организации вычислительного процесса (Computation Independent Model). К таким, в частности, относятся модели использования системы, модели предметных областей, модели данных.
2) Модели, независимые от платформы (Platform Independent Model). К ним можно отнести модели, выполненные в рамках основных моделей вычислений: конечный автомат, сети процессов, модель дискретных событий и другие.

3) Платформно-зависимые модели (Platform Specific Model). В них отражается специфика конкретной вычислительной платформы, её организация и использование.
4) Непосредственная реализация производственных спецификаций.

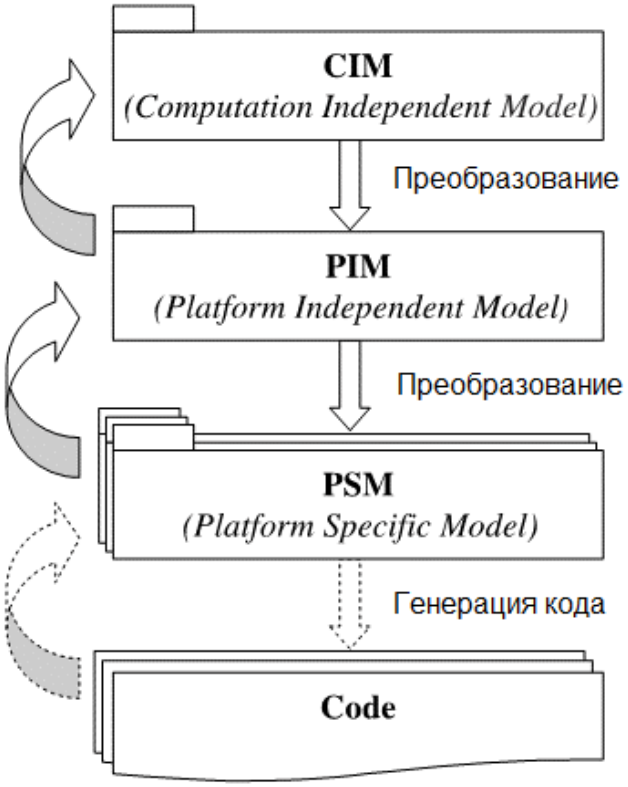


Рисунок 14 – Схема жизненного цикла для модель-ориентированного проектирования

Основная ценность модель-ориентированного проектирования в том, что относительная свобода выбора модели, независимой от платформы, позволяет сформировать инструмент, оптимальный с точки зрения решаемой задачи, а также формально гарантирующую требуемые свойства вычислительного процесса. В некоторых случаях модели позволяют привлечь к процессу разработки специалистов предметной области. Также основная часть разработки производится в платформно-независимом виде, что обеспечивает высокий уровень переносимости результатов между вычислительными платформами.

1.1.3.3 Платформно-ориентированное проектирование

Как было описано выше, процесс модель-ориентированного проектирования строится на создании платформно-независимого описания вычислительной системы с последующим её отображением на вычислительную платформу. Альтернатива –платформно-ориентированное проектирование [11,71,72], процесс разработки построенный на идее «встречи процессов в середине» (meeting-in-the-middle process). Его визуализация приведена на Рисунок 15 [52].

Процесс заключается в параллельной разработке прикладной программы и вычислительной платформы, интерфейс взаимодействия между которыми представлен в абстрактном виде.

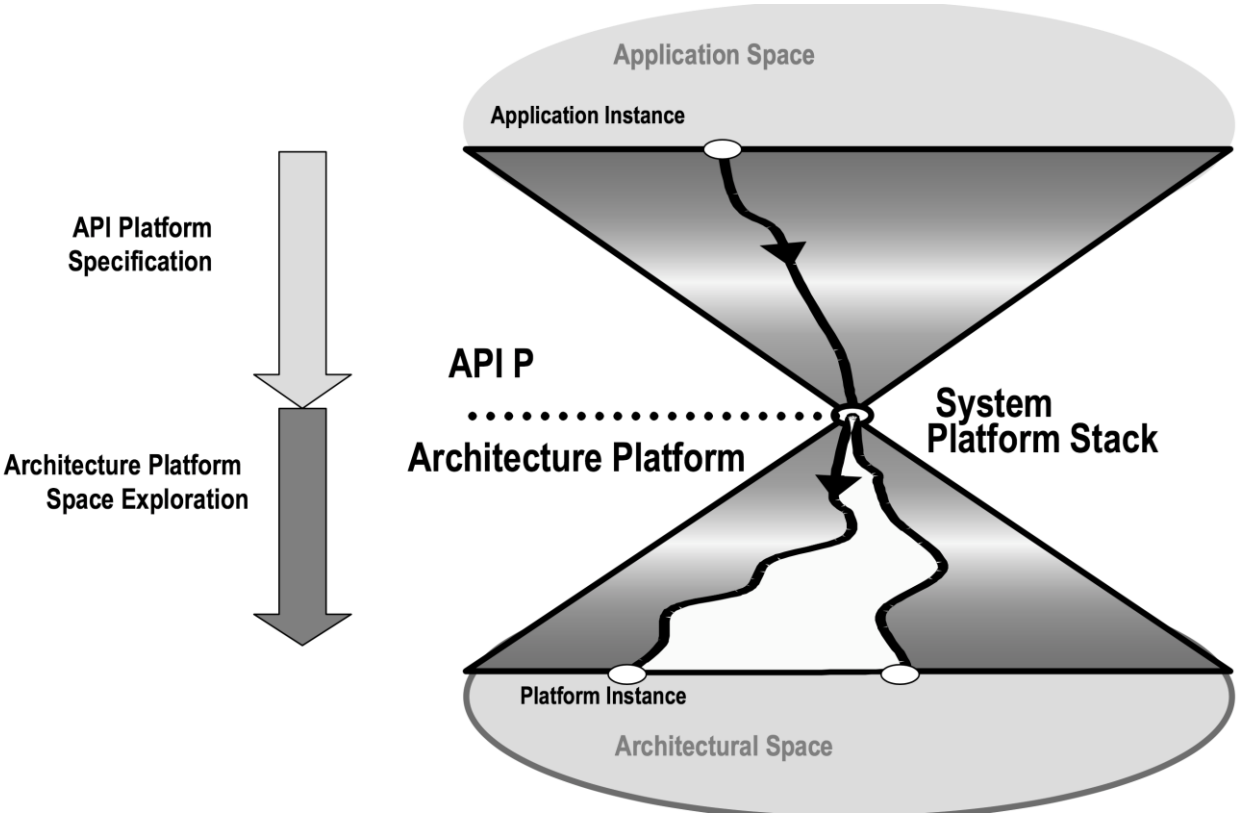


Рисунок 15 – Процесс платформно-оринетированного проектирвоания [52]

Как правило, целевая система включает в себя как элементы программной вычислительной платформы (ВМ, ОС), так и элементы аппаратной (ПЛИС, система команд процессора). Их совокупность называется *системной платформой (СПЛ)*. Явное выделение СПЛ повышает гибкость проекта, позволяя не только решать проблемы устаревания элементной базы, но и изменять технические характеристики системы. Это делается путём модернизации системной платформы, не затрагивая прикладную составляющую (следовательно, сокращая количество ошибок). В случае, если одна СПЛ используется несколькими проектами, то результаты её модернизации становятся доступны для всех проектов сразу, а разработка может быть передана отдельной группе.

1.1.3.4 Совместное проектирование инструментальной и целевой составляющей встроенных систем

Ранее отмечалась актуальность задачи совместного проектирования ВСС на различных уровнях системы (для чего необходимо распределить задачу между ними). Аналогичные вопросы существуют также для распределения задачи между инструментальными средствами и адаптируемыми элементами ВСС (платформы, вычислители, виртуальные машины), и, как отмечают специалисты [1], данная проблема ещё только ожидает своего решения.

В качестве примера можно привести вопросы общего характера, к примеру, насколько эффективнее с точки зрения производительности может быть динамическая компиляция чем статическая, с учётом набранной статистики [73]. И конкретные вопросы, например: какой способ компиляции программы будет эффективнее с точки зрения энергопотребления, с оптимизациями «разворачивания циклов» (Loop unrolling) и «подстановки функции» (Function in lining) или без [74,75].

1.2 Вычислительные платформы

Важное место в проектировании ВСС занимают вычислительные платформы. Это подтверждается структурой типового жизненного цикла ВСС и современными тенденциями в области проектирования (подраздел 1.1.3). Список освоенных разработчиками вычислительных платформ в значительной степени определяет их возможности как коллектива. Популярные платформы определяют вид индустрии в целом (к примеру появление ПЛИС, упоминаемое в пункте 1.1.3.1).

Исторически [24], первые уровневые системы разрабатывались внутри компаний, где решалась, в первую очередь, проблема смешения интересов. С ростом рынка, ростом числа его участников и сложности систем, данный подход стал препятствовать интеграции вычислительных систем и элементной базы. Начался процесс «разукрупнения» (disaggregation [24,72]), схематично приведённый на Рисунок 16. В его рамках уровни вычислительных систем выделялись в отдельные, независимые продукты, используемые разработчиками.

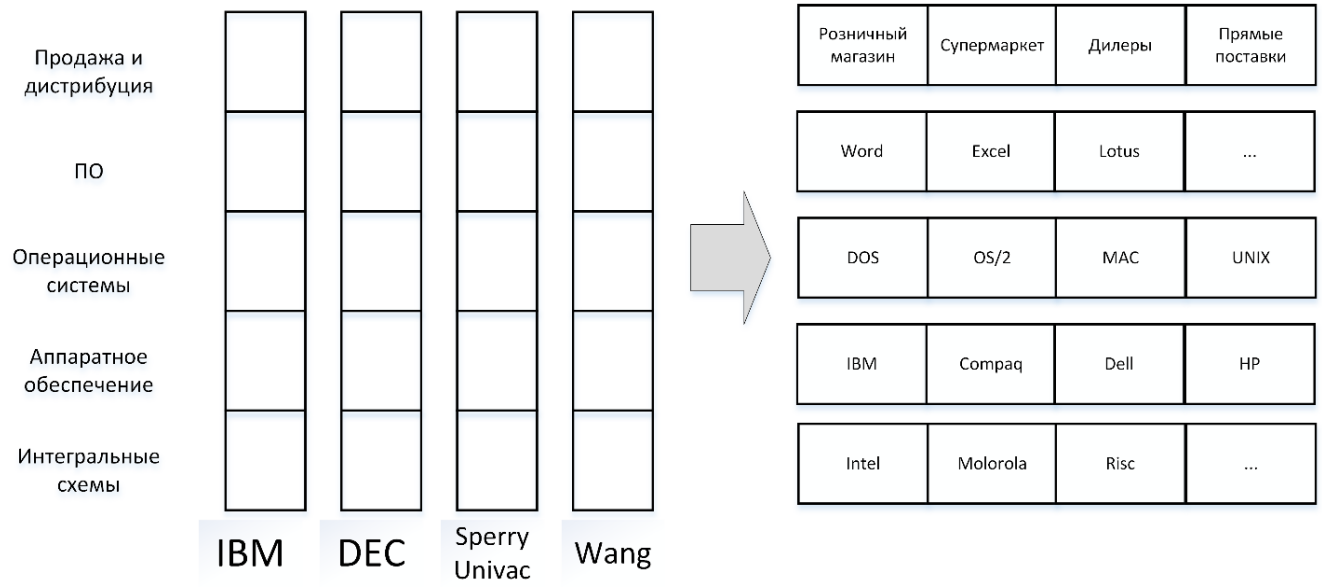


Рисунок 16 – Процесс «разукрупнения»

Данный процесс приводит к формированию вокруг вычислительных платформ замкнутых «экосистем», в рамках которых происходит становление, разработка и эксплуатация ВПЛ.

1.2.1 Понятие вычислительной платформы

Понятие вычислительной платформы является фундаментальным в области вычислительной техники. Оно присутствует во многих методологиях и средствах проектирования [11,12], иногда под другими именами [76]: уровень, виртуальная машина, библиотека, «фреймворк» (framework), вычислитель. Эффективная работа с ВПЛ требует унифицированного взгляда на них, позволяющего абстрагироваться от незначительных деталей в пользу значимых с точки зрения использования вопросов (например, от способа реализации для прикладного программиста).

В работе [6] ВПЛ определяется, как зафиксированный для повторного использования набор спецификаций. Здесь основной акцент делается на причине возникновения ВПЛ, а именно, на повторном использовании, включая методологическую составляющую.

В работе [23] производится анализ ВПЛ с точки зрения модель-ориентированного проектирования, где ВПЛ рассматривается без привязки к способу реализации. Предлагается обобщённая модель платформы, приведённая на Рисунок 17. Она включает в себя: язык (Language), систему типов (Types), элементов (Instances) и шаблоны их использования для решения задач (Patterns). Как можно видеть, акцент делается на функциональном назначении платформы – её применении для решения задач разработки из готового набора компонент по доступным рекомендациям.

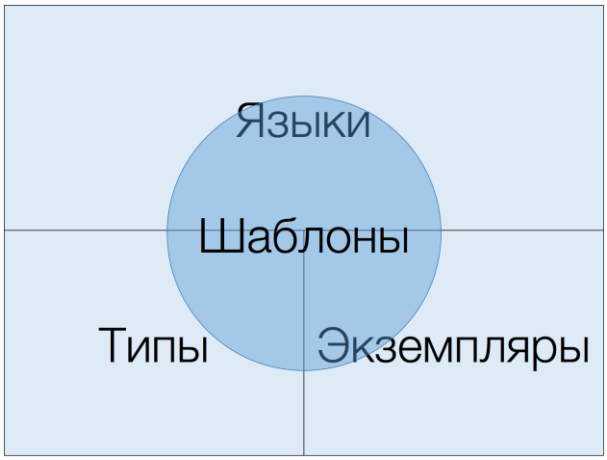


Рисунок 17 – Обощённая точка зрения на платформу

В работе [24] даётся наиболее разносторонний анализ ВПЛ, включающий в себя этимологию слова, экономический анализ, роль ВПЛ в процессе проектирования систем и

индустрии вычислительный техники в целом. Наиболее интересно рассмотрение ВПЛ как шага в развитии вычислительной техники, позволяющего ставить и решать новые задачи. Данная идея коррелирует с понятием парадигмы в области научного знания [77]. Проявление данного процесса можно видеть в области разработки новых языков программирования и моделирования, позволяющим создавать более сложные системы (структурное программирование, язык и платформа Erlang/OTP, модель-ориентированного проектирования). Разработка ВПЛ, в том числе и реконфигурируемых вычислителей, расширяет возможности уже описанных систем, делая их производительнее, надёжнее, доверительнее.

Обобщив рассмотренные представления, было сформировано следующее определение:

Вычислительная платформа (ВПЛ) – особый вид объектов повторного использования для решения заданного класса вычислительных задач, включающего языковые, методологические, инструментальные и технологические средства.

1.2.2 Классификация вычислительных платформ

В данном разделе предложена используемая в работе классификация ВПЛ. Отдельные её элементы можно видеть в работах [11,23,24].

1.2.2.1 Общего назначения и специализированные

С точки зрения спектра решаемых задач, вычислительные платформы можно разделить на ВПЛ общего и специального назначения. Первые должны быть полны по Тьюрингу и, теоретически, должны решать любые задачи. Применимость ВПЛ общего назначения ограничена их нефункциональными характеристиками, такими, как производительность, скорость реакции, предсказуемость, энергопотребление и так далее.

Специализированными являются такие ВПЛ, для которых полнота по Тьюрингу не требуется и, следовательно, они применимы только для определённого класса задач. Это позволяет повысить их эффективность. Также, специализированными можно считать ВПЛ эффективные для определённого класса вычислительных процессов. К примеру, современные графические процессоры являются полными по Тьюрингу, но при этом решения задач, несоответствующих SIMD (Single Instruction Multiple Data, [78]) структуре, является неэффективным. В связи с этим, они относятся к специализированным вычислителям.

1.2.2.2 Широкого и узкого круга пользователей

Данный признак вводится в зависимости от того, на какой круг пользователей ориентирована ВПЛ. Он не находит отражения в организации ВПЛ, а сосредоточен в

маркетинговом аспекте, сопутствующих инструментальных средствах, методических и информационных материалах, вопросах поддержки.

ВПЛ, предназначенные для узкого круга пользователей, имеют более низкие требования к инструментальным средствам и документации, так как зачастую разрабатываются будущими пользователями, или в тесном контакте с ними. За счёт закрытости, они могут являться конкурентным преимуществом. С точки зрения разработки и структуры расходов, отличия ВПЛ для узкого круга пользователей от ВПЛ для широкого колоссальны.

1.2.2.3 Конфигурируемые и неизменные

Неизменяемыми являются такие ВПЛ, в которые пользователь не может вносить изменения. Примерами являются большинство СБИС, процессорных архитектур (ARM, MIPS или x86), а также ПЛИС и закрытые виртуальные машины. Конфигурируемые ВПЛ, как правило, поставляются в виде конфигурируемых IP ядер или виртуальных машин с открытым исходным кодом, которые пользователь может адаптировать под личные нужды.

1.2.2.4 Заказные и стандартные

Данное разделение является условным и производится с целью подчеркнуть причину появления ВПЛ в конкретном проекте. Под стандартными ВПЛ понимаются такие платформы, на организацию которых конкретный проект не оказывает никакого влияния. В заказных – структура платформы в значительной степени определяется нуждами конкретного проекта или серии. Как правило, заказные платформы являются специализированными, предназначенными для узкого круга разработчиков и глубоко конфигурируемыми.

Разделение необходимо для того, чтобы говорить о методах и средствах реализации ВПЛ, так как в случае заказного варианта, количество доступных ресурсов на реализацию ограничено. Это находит своё отражение в используемых методах и средствах разработки – предпочтение отдаётся более экономным с точки зрения реализации вариантам. Ценой является «чистота и аккуратность» ВПЛ (к примеру: встраиваемые языки программирования или Embedded Domain Specific Language (EDSL) [79]).

1.2.2.5 Методы и средства реализации вычислительных платформ

ВПЛ формируются на базе других платформ: для виртуальных машин это операционная система, для операционной системы это процессор, для языка С это машинный код, для СБИС это полупроводники. Способ реализации имеет принципиальное значение для использования ВПЛ . Ниже приведены основные из них.

За пределами анализа оставлены все смешанные варианты ввиду их колоссального многообразия. Примерами их являюстя виртуальные машины с компиляцией времени исполнения [73], процессоры с микропрограммным управлением [78] и многие другие.

Конструктивные вычислительные платформы являются элементной базы, для которой разработчик определяет конфигурацию взаимосвязей. Именно способ сопряжения элементов между собой определяет целевой вычислительный процесс. Конфигурация взаимосвязей может быть аппаратной (печатные узлы и интегральные схемы) или программной (как в случае с ПЛИС и с некоторыми реконфигурируемыми архитектурами), а элементная база может быть, как физической (транзисторы, конденсаторы, контроллеры), так и логической (IP-ядра, программные модули).

Программируемые вычислительные платформы или вычислители, основаны на идее о том, что есть некоторая неизменная часть вычислительной системы, для которой задаётся спецификация её поведения либо единовременно (прошивка для контроллера, конфигурация для ПЛИС), либо в виде потока команд (любого рода интерпретаторы).

Языковые вычислительные платформы основаны на символьных преобразованиях одних спецификаций в другие и позволяют разработчику зафиксировать вычислительный процесс в удобном для себя виде, а затем перевести его в исполняемый. Примерами таких платформ являются компилируемые языки программирования (язык C, преобразуемый в машинный код, Java в объектный код), синтезируемые языки описания аппаратуры (Verilog, Bluespec [80], SystemVerilog), языки моделирования (UML, реализация спецификаций, на котором, «выполняется» разработчиком).

1.3 Уровневая организация встроенных систем

1.3.1 Определение

Как отмечалось выше, современные ВСС разрабатываются с использованием множества ВПЛ. Для каждой ВПЛ разработчиками формируется представление системы, выполненное в рамках соответствующей модели вычислений. Все представления используют специфические наборы модулей, шаблонов проектирования, а также ориентированы на определённые классы задач. Совокупность представлений имеет иерархическую организацию и формирует целевую систему. Представления, лежащие ближе к физическим процессам, обеспечивают работу вышележащих представлений. Их принято именовать уровнями, а их совокупность уровневой организацией (Рисунок 18).

Уровень ВСС – один из вариантов рассмотрения вычислительного процесса целевой системы, адресованный заданной группе интересов и ориентированный на конкретную ВПЛ.

Уровневая организация – совокупность уровней ВСС, организованных в иерархическую структуру, определяющую методику разработки и принципы функционирования системы.

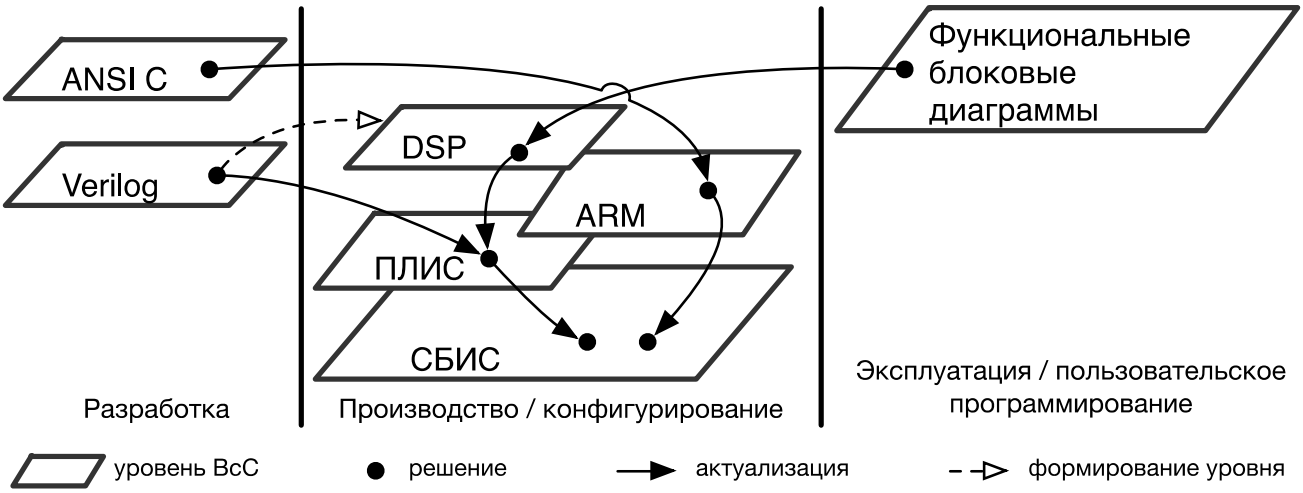


Рисунок 18 – Фрагмент уровневой организации встроенной системы с пользовательским программированием сигнального процессора, реализованного на ПЛИС

Правильный выбор уровневой организации в значительной степени определяет успешность проекта, и его перспективы с точки зрения повторного использования, адаптации и развития. Это можно видеть по косвенным признакам: попытки сравнения доступных ВПЛ; наличие методики платформно-ориентированного проектирования [11,72]; наличие средств и методик проектирования, ориентированных на формирование уровневой организации под проект [58,81,82].

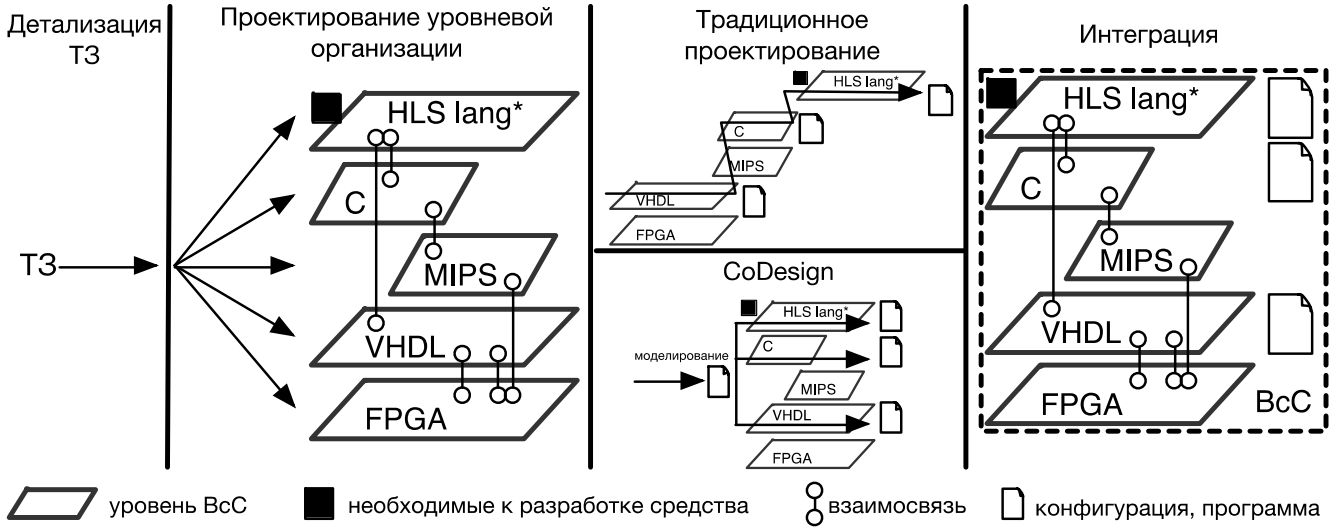


Рисунок 19 – Проектирование встроенной системы с точки зрения уровневой организации

Рисунок 19 демонстрирует основные этапы проектирования ВСС с явным выделением уровневой организации. Важно отметить два процесса:

1) Процесс формирования уровневой организации ВСС, в рамках которого работа с системой должна производиться в представлениях, ортогональных уровневой организации. Здесь решаются задачи описания уровней, их функциональных возможностей, взаимоотношений и методов их использования. Часто на данном шаге фиксируется конкретная уровневая организация или её шаблон, навязанные используемой методикой проектирования, вместо осознанного выбора.
2) Процесс совместной разработки заказных уровней ВСС (включая элементы целевой системы и инструментальные средства) или их конфигураций. Важным свойством процесса является единовременная разработка компонентов максимально независимыми группами разработчиков. Взаимодействие групп должно обеспечиваться совокупностью моделей для снижения рисков при интеграции.

Яркая выраженность этих процессов и является отличительной особенность проектирования многоуровневых ВСС.

1.3.2 Структура пространства проектных решений уровневой организации

Как отмечалось ранее, ППР должно предоставлять разработчику проектные альтернаты, в рамках которых он принимает решения. При этом ППР должно быть связано с конкретным, ограниченным кругом интересов, а альтернативы – сопоставимы между собой. В данном разделе будет предложена структура ППР, адресованная проблемам проектирования уровневой организации. Для этого будет зафиксирован набор осей ППР. Точками данного проектного пространства являются отдельные уровни ВСС, а уровневая организация представляется их множеством.

Как отмечалось выше, уровневая организация включает как программные, так и аппаратные представления ВСС. Это позволяет определить первую ось проектного пространства: *программно-аппаратной реализации*. В случае традиционной организации ВСС, аппаратной платформой является микроконтроллер и его система ввода-вывода. В качестве программной платформы выступает операционная система реального времени [83]. Современные же ВСС, построенные на базе перспективных вычислительных архитектур (совместные виртуальные машины и реконфигурируемые вычислители), не укладываются в подобную двухуровневую модель. Фактически, количество уровней в рамках современных ВСС может быть весьма велико, и распределено по различным вычислительным узлам. В связи с этим, обобщим ось программно-аппаратной реализации до оси *выбора вычислительной платформы*. В ней представлены все доступные ВПЛ без разделения на программные и аппаратные. Это позволяет избежать проблемы классификации реконфигурируемых ВСС [76].

Не претендующий на полноту пример оси выбора ВПЛ приведён на Рисунок 20. Последовательность ВПЛ является условной и может меняться в зависимости от ситуации. К примеру, если последовательность ASM, C, C++, SystemC выглядит естественной, то включение в неё платформ Java, Bluespec [80] или Verilog вызывает определённую неоднозначность.

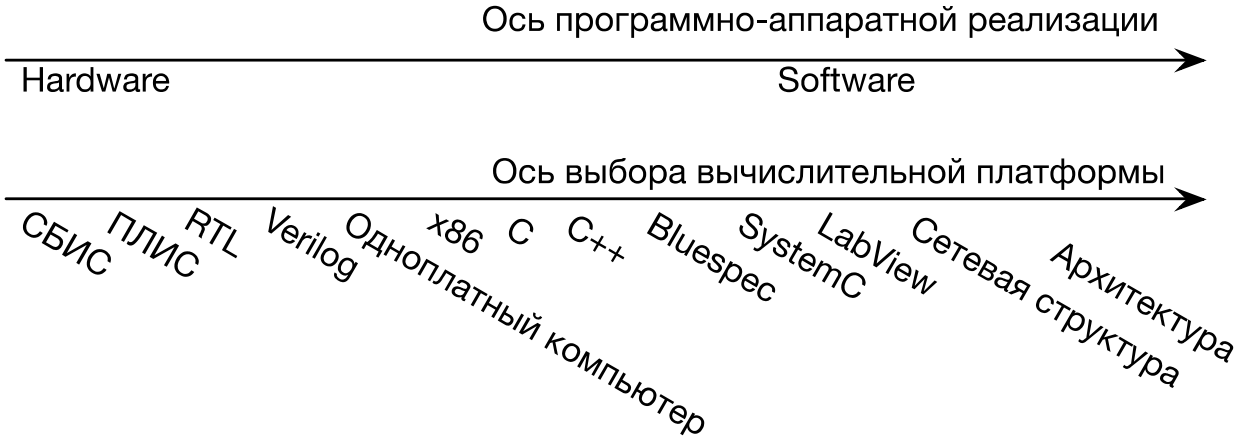


Рисунок 20 – Ось программно-аппаратных альтернатив реализации (сверху) и ось выбора вычислительной платформы (снизу)

Важное значение для организации ВСС имеет время конфигурирования ВПЛ. В HLD-методологии [6], эта альтернатива определяется как *ось Design-time / Run-time (ось выбора времени разработки и времени исполнения)*, на которой есть следующие альтернативы:

1) Design-time – фаза проектирования системы. Средства обеспечения конфигурации доступны проектировщику и входят в состав обеспечивающих систем [28].
2) Run-time – фаза эксплуатации системы. Средства обеспечения конфигурации присутствуют в целевой системе (System-of-interest [28]).

Для практических задач такая детализация является недостаточной, так как не включает часть стадий жизненного цикла системы [28]: архитектурное проектирование, рабочее проектирование, производство, развёртка (иногда – стадия конфигурации, configuration-time [84]), пользовательское программирование, вывод из эксплуатации или модернизация без сохранения основных свойств. Обобщённая версия этой оси именуется *осью стадий конфигурирования* и соответствует тенденциям в области проектирования ВСС.

На Рисунок 21 приведён пример оси стадии конфигурирования. Состав оси зависит от конкретного проекта и требований к нему. Последовательность вариантов определяется структурой жизненного цикла системы. Они располагаются в причинно-следственном порядке (хронологический порядок невозможен ввиду обратных связей и повторяемости процессов конфигурации). Для воплощения большинства вариантов требуется разработка или использование сопутствующего инструментария и обеспечивающих механизмов.

Описанные оси ППР позволяют оперировать ВПЛ, доступными на рынке. Но они являются недостаточными для рассмотрения заказных уровней ВСС, так как не позволяют оперировать способом их реализации, от которого зависит сложность разработки целевой системы, обеспечивающих систем и спектр доступных разработчику возможностей. Для устранения этой проблемы в ППР включается *ось метода обеспечения ВПЛ*, базовые альтернативы которой описаны в подразделе 1.2.2.5. На практике они глубоко ситуативные, и вопрос их взаимного позиционирования не ставится. На Рисунок 22 приведена визуализация данной оси ППР.

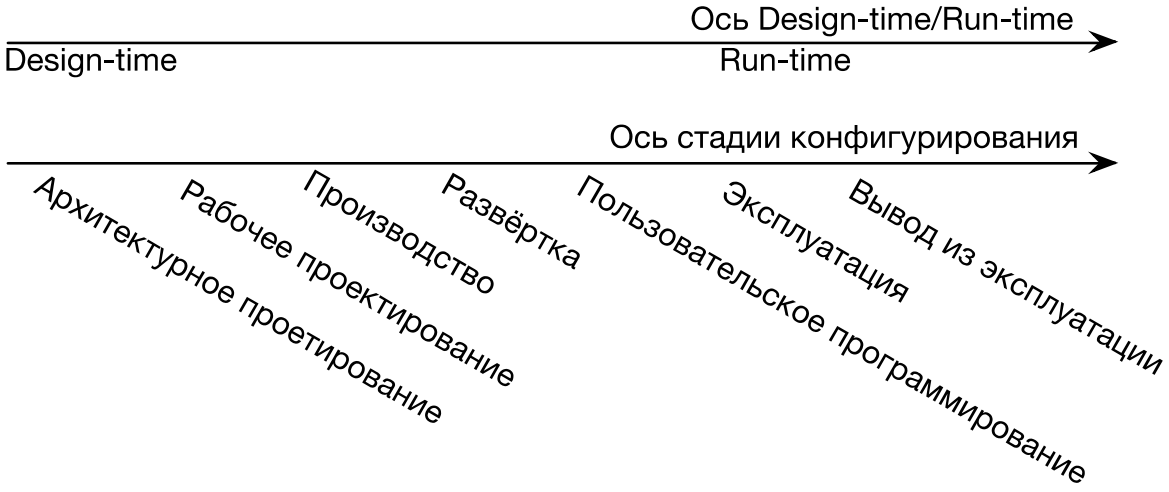


Рисунок 21 – Ось Design-time/Run-time (сверху) и ось стадий конфигурирования (снизу)

Важно отметить характер взаимоотношений между осью метода обеспечения ВПЛ и осью выбора вычислительной платформы для заказных ВПЛ. Возможны следующие варианты:

1) Создаётся альтернативная реализация уже существующей ВПЛ (к примеру, реализация альтернативной версии виртуальной машины Java). В этом случае на оси выбора ВПЛ отмечается платформа Java, а на оси метода обеспечения – характеристика новой реализации.
2) Создаётся новая ВПЛ. В этом случае она отмечается в качестве новой проектной альтернативы на оси выбора ВПЛ.

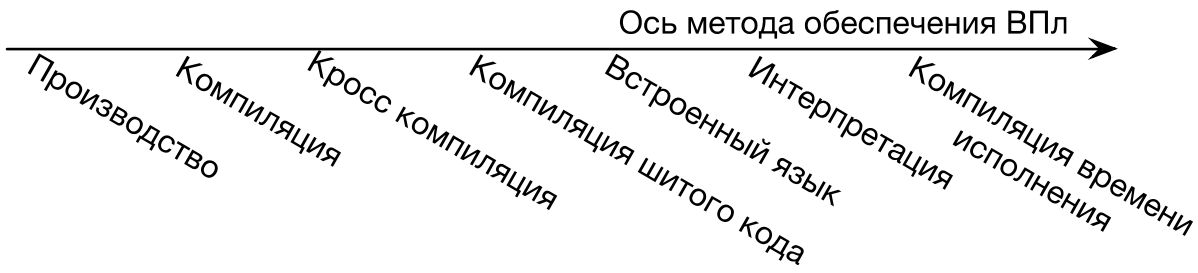


Рисунок 22 – Ось метода обеспечения вычислительной платформы

Список осей ППР является открытым. Пример расширения: классификация ВПЛ, предложенная в подразделе 1.2.2. Приведённая выше структура может рассматриваться как минимально достаточная и должна расширяться в зависимости от деталей технического задания.

Резюмируя полученные результаты, ППР в части уровневой организации содержит по крайней мере три оси: (1) ось выбора ВПЛ, (2) ось выбора стадии конфигурирования, (3) ось выбора метода обеспечения ВПЛ. На Рисунок 23 представлена структура ППР, в которой показана уровневая организация одноплатного компьютера с возможностью конфигурирования портов ввода/вывода при помощи перемычек. Компьютер программируется на языке С и имеет возможность пользовательского программирования на языке Java. Оси выбора ВПЛ и стадии конфигурирования представлены явным образом. Метод обеспечения ВПЛ обозначается использованием точек разного вида.

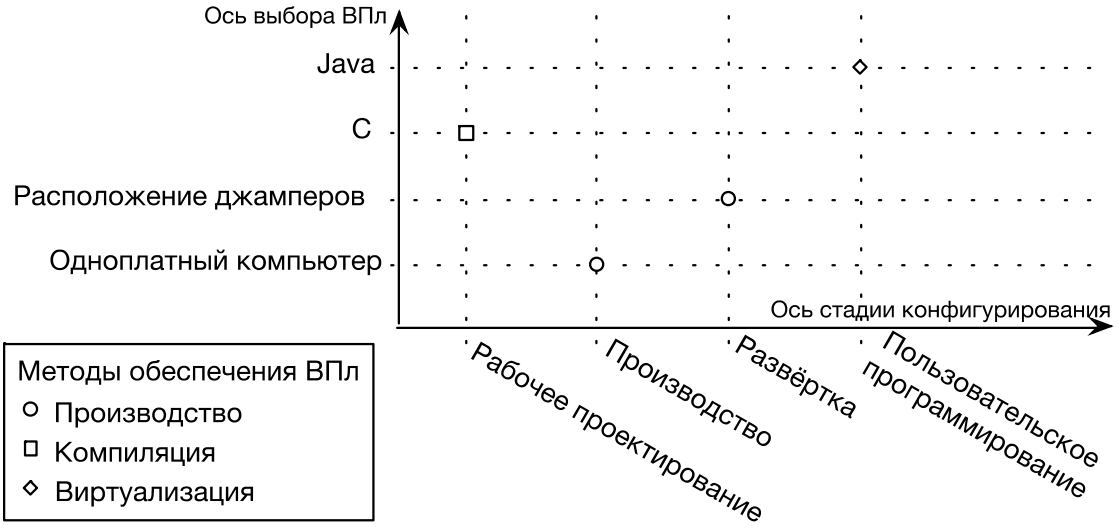


Рисунок 23 – Структура пространства проектных решений уровневой организации

Данная структура ППР позволяет отображать решения, соответствующие актуальным тенденциям в области проектирования ВСС и вычислительных архитектур. Это позволяет говорить о её адекватности текущему состоянию индустрии. Открытость ППР позволяет исследовать в его рамках новые ВПЛ и принципы уровневой организаций.

Несмотря на то, что предложенная структура ППР позволяет описать все уровни ВСС, она не даёт представления о структуре уровневой организации.

Формальная запись структуры ППР может быть представлена как декартово произведение осей. Важно, что часть пересечений будет «выколотыми точками», ввиду практической нецелесообразности или невозможности их реализации (к примеру – модернизация СБИС конечным пользователем). Уровни вычислительной системы должны фиксироваться как

тройки следующего вида: (A, B, C); состав уровневой организации ВСС как их множество: {(A, B, C), (D, E, F)}. Свойства множества не позволяют зафиксировать взаимосвязи между уровнями.

1.4 Средства проектирования уровневой организации встроенных систем

Как отмечалось в разделе 1.3, особенностью проектирования многоуровневых ВСС является необходимость проектирования уровневой организации и совместной разработки уровней ВСС (включая их инструментальную составляющую).

1.4.1 Исследование архитектурных стилей уровневой организации встроенных систем

Проектирование уровневой организации ВСС принято относить к области *архитектурного проектирования* [6,12,18]. Широкое распространение получило следующее определение архитектуры [57]: «Architecture – fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution».

Перевод: *Архитектура* – фундаментальные концепции и свойства системы, воплощённые в её элементах, во взаимосвязях между ними и её окружением, и в принципах проектирования и развития.

Данное определение рекомендуется для использования в рамках программной и системной инженерии. Системная инженерия [38] является дисциплиной, цель которой в создании успешных систем вне зависимости от их класса, области, масштаба или назначения (системы, которые удовлетворяют достаточное количество интересов заинтересованных сторон). ВСС является системой с преобладающей программной реализацией, что говорит о применимости данного определения в рассматриваемой области.

Работа с архитектурой осуществляется через спецификации, выполненных на языках архитектурного описания (Architectural Description Language ADL, [32]). Выбор нотации зависит от разных факторов, например, от задач, решаемых при её составлении или от круга лиц, работающих с ней. В процессе проектирования следует использовать наиболее простые нотаций для сокращения времени на работу с ними. Выбор нотации зависит от объёма спецификации – после достижения спецификацией определённого объёма (в графическом виде), работа с ней становится практически невозможной, ввиду чего необходимо перейти к текстовым представлениям. Семантическую характеристику архитектурной спецификации определяет архитектурный стиль [20,85,86].

Архитектурные стили – инструменты для работы с архитектурной вычислительной системы, включающие методики решения соответствующих задач проектирования и используемые онтологические модели (системы понятий). В качестве параллелей можно указать точки зрения из [57] и аспекты из HLD-методологии. Являются более важным с точки зрения проектирования, чем языки, так как содержат методическую составляющую (язык же может адаптироваться под ситуацию).

Архитектурные языки могут быть как специализированными и универсальными. Универсальность языка противоречит принципу разделения интересов, провоцирует рост сложности описаний и снижение эффективности. Специализация накладывает ограничения на архитектора, создавая ненужные проблемы тогда, когда ему необходимо выйти за границы стиля.

Далее будет приведён обзор архитектурных стилей для проектирования и документирования уровневой организации ВСС или «*уровневых архитектурных стилей*». Под «соответствием» понимается предоставление архитектурным стилем абстракций, для описания уровневой организации: состава уровней, межуровневых отношений, функциональной характеристики уровней, способов их использования.

1.4.1.1 Стиль уровневых диаграмм

Стиль уровневых диаграмм или уровневый стиль (Layered Diagram, см. Рисунок 24) [20] широко распространён и позволяет описывать одностороннее отношение использования между группами модулей (или уровнями, в трактовке данного архитектурного стиля). Стиль предназначен для: структурирования вычислительной системы по уровням и организации повторного использования; декомпозиции задачи; упрощения модификации и миграции системы за счёт выделения доменов [42].

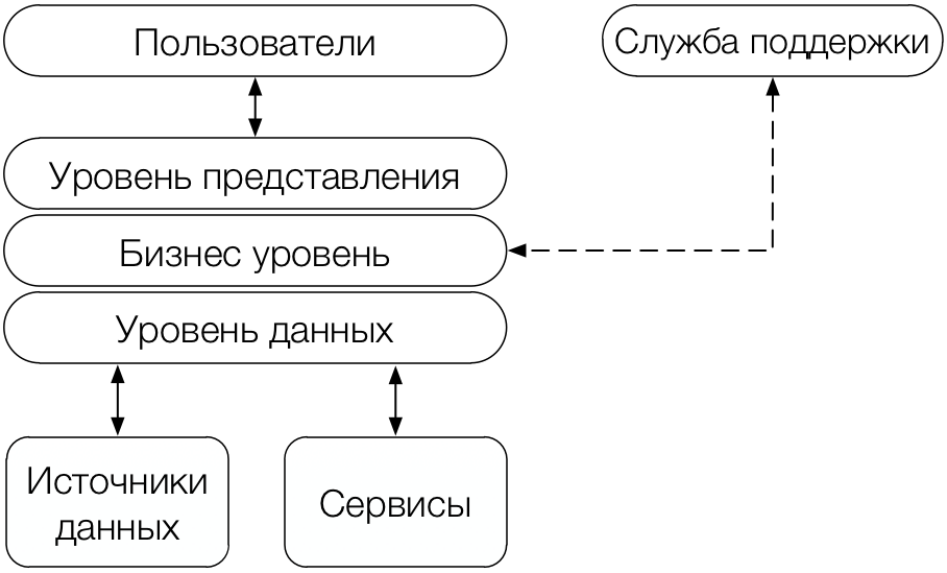


Рисунок 24 – Уровневая диаграмма

Данный стиль решает проблему сужения ППР относительно вариантов ВПЛ. Имеет следующие недостатки:

1) Не предоставляет возможности для описания инструментальной цепочки.
2) Не позволяет различить «уровень ВСС» и «вычислитель», что смещает акценты с вопроса уровневой организации конкретной ВСС.
3) Не позволяет описывать сложные межуровневые взаимосвязи.

1.4.1.2 Диаграмма развёртывания

Диаграмма развёртывания (Deployment style, Рисунок 25) [87] из состава UML, позволяет описать отношения между логическими и физическими элементами ВСС (например, распределение программного обеспечения по вычислительным узлам). Предназначена для оптимизации каналов передачи данных и вычислительных ресурсов; документирования процесса развёртки ВСС. Это соответствует вопросу о распределении функций между ВПЛ.

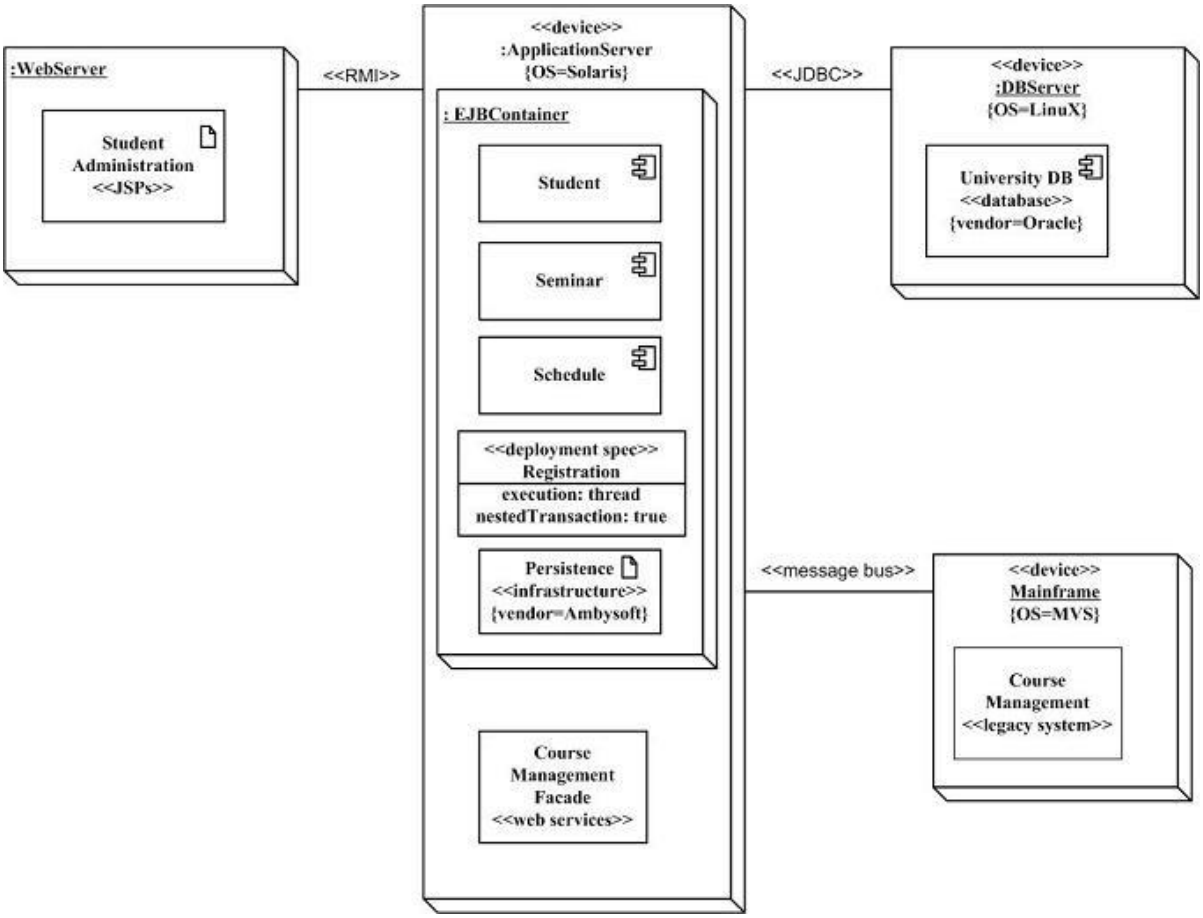


Рисунок 25 – Диаграмма развёртывания

Имеет следующие недостатки:

1) Не позволяет работать с иерархией ВПЛ.

2) Не позволяет работать со спецификациями, отношениями между ними и инструментарием.
3) Смещает акцент с уровневой организации на организацию в рамках отдельных ВПЛ.

1.4.1.3 Граф актуализации вычислительного процесса

Модель актуализации вычислительного процесса (Model of Computational Process Actualization, см. Рисунок 26) [61], позволяет единообразно описать ВСС, инструментарий и процесс проектирования как последовательность трансляторов. Предназначен для решения задачи распределения функций между аппаратной и программной составляющей, между фазами Design-time и Run-time, и, отчасти, для проектирования инструментария.

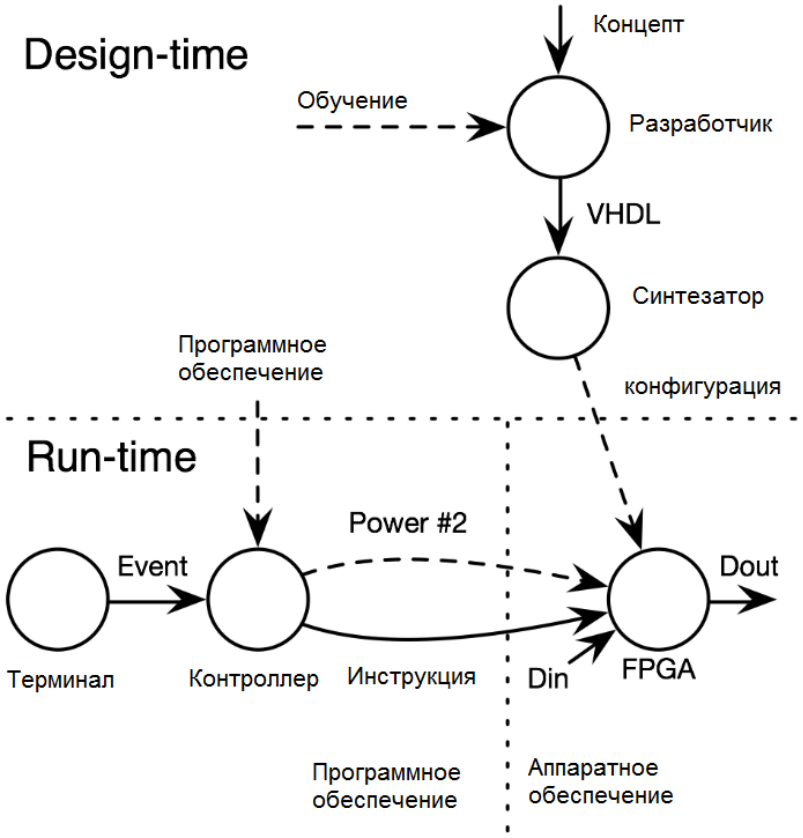


Рисунок 26 – Граф актуализации

Данный архитектурный стиль решает проблему единообразного представления ВСС на разных ВПЛ и разных стадиях жизненного цикла. Это позволяет распределять функции между ВПЛ и определять время конфигурирования. Имеет следующие недостатки:

1) Широкая трактовка понятия «транслятора» в совокупности с его центральным местом в архитектурном стиле затрудняет использование. Также она не позволяет разделить уровневую организацию ВСС и организацию отдельных уровней системы.

2) Данный стиль подразумевает последовательную актуализацию спецификации в физический процесс. На практике, часть спецификаций, используемых для верификации и не участвует в процессе актуализации, что затрудняет работу с ними.

1.4.1.4 «Бургер-диаграмма»

«Бургер-диаграмма» (Burger-diagram) (см. Рисунок 27), вариация которой приведена в [88], позволяет описывать ВСС как иерархию систем (отношение система-подсистема), с обозначением интерфейса их взаимосвязи. Данный стиль предназначен для системного анализа ВСС, где главным является трассировка требований, распределение функций между подсистемами и документирование взаимосвязи компонентами целевой системы.

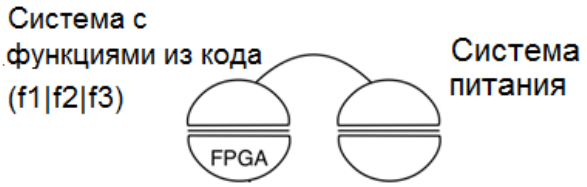


Рисунок 27 – Системно-иерархический стиль

Данный архитектурный стиль может использоваться для анализа функциональной взаимосвязи между уровнями ВСС. Это достигается за счёт возможностей демонстрации свойства многофункциональности. В частности, именно при исследовании данного архитектурного стиля, удалось сформулировать определение [ре]конфигурации, представленное в главе четыре. Имеют следующие недостатки:

1) Понятие «системы» является излишне общим. Это затрудняет разделение системы на уровни.
2) Ориентация на функциональное назначение систем и подсистем делает анализ избыточным.

1.4.1.5 Диаграммы предметной области

Важное значение для построения и развития системы имеет принцип модульности. В его основе лежит разделение ответственности между частями системы. Прекрасным инструментом для того, чтобы отслеживать соблюдение данного принципа, являются диаграммы предметной области (Domain Charts), используемые в объектно-ориентированном системном анализе (Object-Oriented Systems Analysis [89], метод Shlaer-Mellor) и xtUML [42]).

Применение диаграмм предметной области состоит из двух этапов:

1) Выделение предметных областей системы. Предметная область – набор связанных понятий, имеющий отношение к фиксированному кругу вопросов и состоящий из существительных и глаголов. Предметная область должна быть, по возможности,

минимальной. Допустимо повторение слов в разных предметных областях лишь в том случае, если их семантический смысл различен.

2) Отношения между доменами (bridge). Позволяет показать, функциями каких предметных областей текущая предметная область (откуда выходит стрелка) обеспечивает своё функционирование.

На Рисунок 28 приведён пример диаграммы предметной области [42] для системы по резервированию билетов.

Рисунок 28 – Пример диаграммы предметных областей

Данный стиль важен для проектирования уровневой организации, главным образом, с точки зрения формирования уровней относительно их функционального назначения – корректная группировка позволит создать такой уровень системы, который в будущем можно будет повторно использовать в других проектах.

Основной недостаток данного стиля – невозможность применить его к анализу технических решений, так как он не предоставляет необходимых понятий. Главное достоинство – возможность решать задачи определения предметной области с их последующей декомпозицией и отображением на элементы ВСС.

1.4.1.6 Язык для анализа и проектирования архитектуры

Язык для анализа и проектирования архитектуры (Architecture Analysis and Design Language – AADL) [48] – язык для анализа, проектирования и документирования архитектуры

ВСС в компонентном стиле. Изначально разрабатывался для нужд авиации, но в последствии получил более широкое применение для решения задач моделирования программного и аппаратного обеспечения ВСС и систем реального времени. Одной из его отличительных особенностей является наличие как графической, так и текстовой нотации (последняя обеспечивает высокую масштабируемость спецификаций). Высокая мощность данного инструмента позволяет применять его для задач проектирования и документирования уровневой организации, фиксируя детали реализации целевой системы. Основными недостатками данного инструмента, применительно к рассматриваемому классу задач, являются его избыточность, сложность и ориентация на описания целевой системы. Это заставляет описывать уровневую организацию косвенным образом (можно видеть в докладе [90], применительно к документированию реконфигурируемых вычислителей).

1.4.2 Средства разработки многоуровневых встроенных систем

Разработка многоуровневых ВСС требует инструментальной поддержки, в рамках которой должны решаться следующие задачи:

1) Симуляция вычислительного процесса отдельных уровней ВСС.
2) Трассирование взаимосвязей между вычислительным процессом разных уровней.
3) Совместная отладка и профилирование уровневой организации ВСС и средств трансляции.
4) Независимая верификация отдельных уровней ВСС.

Решение этих задач не только обеспечивает независимую разработку отдельных уровней ВСС и соответствующих им описаний целевой системы, но и повысит эффективность целевой системы за счёт более точного анализа показателей, а также снизит интеграционные риски. Сегодня существует множество инструментов, способных решить эти задачи, например: GEM5 [91], Proteus [55], Ptolemy [54], SystemC [56], Jetbrains MPS [81] и многие другие. К сожалению, представленные инструменты ориентированы на применение в рамках конкретных наборов ВПЛ или на работу с определёнными стадиями жизненного цикла системы. Центральное место в них занимают модели вычислительных систем.

Разработка модели многоуровневой ВСС, несмотря на огромное количество инструментария и используемых подходов, является сложной задачей, требующей высокой квалификации от исполнителя. Это связано с повышенными требованиями к качеству результата (повторное использование, сложность отладки) и специфическим характером задачи. Также это связано с ошибками разделения вычислительного процесса на компоненты, с выделением

вычислительных механизмов, с неоднозначностью в вопросах трассирования взаимосвязей между уровнями. Как показал анализ инструментов, позволяющих производить моделирование многоуровневых ВС, модели, лежащие в их основе, разрабатывались для решения конкретной задачи. Это определило конкретный объект моделирования (отличный от уровневой организации ВСС) и ориентировало модели на реализацию определённым набором средств.

В совокупности, это формирует потребность в разработке методики моделирования многоуровневых ВСС независимой от:

- способа реализации (инструментальные средства);
- конкретного вида ВПЛ;
- стадии жизненного цикла системы;

что позволит повысить эффективность разработки и реализации гетерогенных систем с реконфигурируемыми вычислительными архитектурами за счёт совместной разработки и снижения затрат на моделирование ВСС. Методика моделирования многоуровневых ВСС и их элементов должна отвечать следующим требованиям:

1) Возможность моделирования конструктивных, программируемых и языковых ВПЛ.
2) Отсутствие ограничений на количество уровней ВСС и на варианты их взаимосвязей.
3) Разрабатываемые с её помощью модели должны предоставлять возможности сквозной трассировки, включая трассировку через инструментальные средства.

1.5 Постановка задачи

Стремительно растущий рынок ВСС и КФС сталкивается с серьёзными ограничениями существующих методов и инструментов проектирования уровневой организации, препятствующими разработке и внедрению новых вычислительных архитектур. Это формирует следующую *актуальную научную задачу*: развитие методов и инструментов архитектурного уровня для разработки встроенных систем с многоуровневой организацией, что определило направление диссертационного исследования.

Объект исследования – процессы проектирования и разработки встроенных систем, архитектура встроенных систем в части уровневой организации.

Предмет исследования – архитектурные стили для проектирования и документирования уровневой организации встроенных систем, методики проектирования встроенных систем и методики моделирования элементов встроенных систем в САПР.

Цель диссертационной работы – повышение качества архитектурных решений в части уровневой организации встроенных систем и сокращение затрат на разработку заказных элементов уровневой организации за счёт развития научных основ построения САПР, а именно архитектурных стилей и методик моделирования встроенных систем.

В соответствии с целью, в работе ставятся и решаются следующие *задачи*:

1) Исследование архитектурных стилей, предназначенных для работы с уровневой организацией встроенных систем. Определение структуры пространства проектных решений в части уровневой организации и формирование требований к более эффективным архитектурным стилям.
2) Разработка архитектурных стилей и языков архитектурного описания для проектирования и документирования уровневой организации встроенных систем.
3) Формализация разработанных архитектурных стилей с целью создания методики моделирования многоуровневых встроенных систем и их элементов для САПР.
4) Развитие системы архитектурных абстракций для работы с уровневой организацией в соответствии с актуальными тенденциями в проектировании встроенных систем.
5) Модификация традиционной методики проектирования встроенных систем на основе предложенных архитектурных стилей, языков архитектурного описания, методик моделирования и архитектурных абстракций.

Методы исследования. При решении поставленных задач использовались методы системного, аспектного и архитектурного анализа; функционального и объектно-ориентированного программирования; теория категорий, метод структурирования функции качества, методы и приёмы моделирования высших онтологий.

1.6 Выводы

1) Выполнен обзор, демонстрирующий основные проблемы и тенденции в области проектирования встроенных систем. Они выражены в усложнении уровневой организации и в росте популярности реконфигурируемых вычислительных архитектур.
2) Сделан анализ научно-технической литературы по вопросу понятия «вычислительная платформа». Уточнено определение и введена классификация вычислительных платформ.
3) Проведено исследование вопроса «уровневой организации», в рамках которого:

- уточнены понятия уровня и уровневой организации встроенных систем.
- выбраны и обоснованы принципы работы с пространством проектных решений;
- определена структура пространства проектных решений уровневой организации;
- сформулирована задача проектирования уровневой организации встроенных систем.

4) Исследованы доступные архитектурные стили для проектирования и документирования уровневой организации. Выявлены их сильные и слабые стороны.
5) Рассмотрен вопрос разработки многоуровневых встроенных систем. Сформулированы требования к методике моделирования многоуровневых встроенных систем и их элементов.
6) Сформулированы цели и задачи исследования.

2 Разработка архитектурных стилей для уровневой организации

В данном разделе будут приведены результаты исследования и разработки в области архитектурных стилей, а именно:

1) В разделе 2.1 будут подведены итоги проведённого анализа существующих архитектурных стилей. На их основании будут зафиксированы требования.
2) В разделах 2.2 и 2.3 показаны архитектурные стили, разработанные в процессе поисковых работ. Они позволили получить ряд важных теоретических результатов, в том числе унифицированную модель реконфигурации (раздел 4.2).
3) В разделе 2.4 будет представлен архитектурный стиль для проектирования и документирования уровневой организации.
4) В разделе 5.2 будет приведён анализ разработанных архитектурных стилей и их сравнение с аналогами. В рамках данного анализа будут продемонстрированы преимущества предложенного решения.

2.1 Требования к архитектурному стилю уровневой организации

Перед архитектурным стилем для работы с уровневой организацией ВСС ставятся следующие задачи: документирование полученных в процессе проектирования вариантов уровневой организации; сравнение и анализ вариантов уровневой организации; сохранение проектного опыта. Также архитектурный стиль должен способствовать качественному проектированию, в рамках которого архитектор будет решать поставленную задачу, не отвлекаясь на лишние детали (реализация конфигурации уровня) и не прилагая лишних усилий для работы с инструментом [92]. Это формирует базовые требования: полнота представление ППР и абстракция от вопросов, не связанных с уровневой организацией.

Анализ существующих архитектурных стилей (раздел 1.3) позволил не только сформировать дополнительные требования к разрабатываемому архитектурному стилю, но отметить ряд полезных и вредных решений. Ниже приведём их краткий анализ.

Понятие «уровня» из архитектурного стиля уровневых диаграмм (подраздел 1.4.1.1), обозначающее совокупность модулей, зафиксированных для повторного использования, получило широчайшее распространение в индустрии [20]. Рассмотрение уровня как совокупности модулей, а не как целостного элемента вычислительной системы с

самостоятельным представлением, языком и моделью вычислений, порождает искусственные ограничения пространства проектных решений. Отсутствие детализации уровня не позволяет говорить о методических аспектах уровневой организации, выходящих за пределы иерархической организации. Данное понятие должно быть включено в архитектурный стиль в уточнённом виде.

Акцент на процессах, а не на организации системы. Характерен для графа актуализации (пункт 1.4.1.3) и позволяет рассматривать вычислительную систему в динамике. Также, это позволяет визуализировать единство вычислительного процесса.

Единообразное отображение уровней ВСС независимо от стадии жизненного цикла. Характерно для графа актуализации и, отчасти, в уровневых диаграммах. Позволяет работать с уровневой организацией в целом, а не по частям. Необходимо для полноты представления ППР.

Явное отображение *многофункциональности* уровней (как это сделано в «бургер-диаграммах», пункт 1.4.1.4). Необходимо для описания реконфигурируемых систем.

Отображение спецификаций, доступное в архитектурном стиле развёртки (пункт 1.4.1.2). Позволяет описать методологические элементы уровневой организации ВСС.

Отображение организации отдельных уровней. Делает спецификации уровневой организации избыточными. Требует сохранения в ограниченном объёме, достаточном для проектирования функциональных возможностей уровней.

Описание уровневой организации с *использованием общих понятий*, таких как транслятор (пункт 1.4.1.3) или система (пункт 1.4.1.4). Затрудняет процесс архитектурного анализа, размывает границу применимости архитектурного стиля.

Приведённый анализ позволяет зафиксировать дополнительные требования:

1) Должно присутствовать понятие уровня вычислительной системы.
2) Уровни системы должны представляться в унифицированном виде.
3) Архитектурный стиль должен демонстрироваться единство вычислительного процесса и многофункциональность уровней вычислительной системы.
4) Архитектурный стиль должен показывать взаимосвязь вычислительного процесса и конфигурации.
5) Архитектурный стиль не должен позволять специфицировать внутреннее устройство уровней ВСС.
6) Архитектурный стиль должен позволять визуализировать проблемы проектирования многоуровневых ВСС.

2.2 Системно-иерархический

Системно-иерархический стиль построен на базе «Бургер-диаграмм» (пункт 1.4.1.4) с целью расширения границ его применимости, начиная со стадии эксплуатации и заканчивая всем жизненным циклом системы. Работа с жизненным циклом систем основывается на подходе стандарта моделирования данных [46].

Центральное место занимает понятие системы. Уровни вычислительной системы и конфигурации представляются в качестве систем. Работа с жизненным циклом производится через понятия темпоральной части (temporal part) и объекта всего жизненного цикла (whole life individual) [46]. В качестве темпоральных частей рассматриваются временные представления объектов. В случае ВПЛ это конфигурация и сконфигурированный элемент целевой системы.

Под системой понимается совокупность функционального места (точка включения в надсистему, фиксирующая требования) и вычислительного элемента (обеспечивающего выполнение требований), отображаемые верхним и нижним полукругом соответственно. Иерархия уровневой организации описывается через отношение система-подсистема, отображаемое линией между вычислительным элементом и функциональным местом в его составе. Взаимосвязи между системами обозначаются дугами между функциональными местами и могут определяться только в рамках одного уровня иерархии. Цепочки преобразований между темпоральными частями показаны стрелками (см. Рисунок 29).

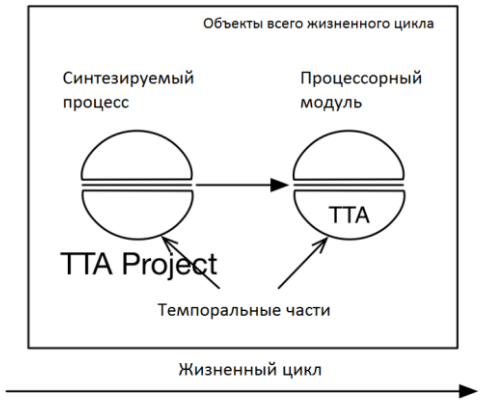


Рисунок 29 – Отображение элементов жизненного цикла компонентов встроенной системы

Построение спецификации рекомендуется начинать с рассмотрения системы на стадии эксплуатации, а лишь затем переходить к описанию их жизненного цикла (Рисунок 30). Показаны: система питания (как пример смешения интересов); подсистема процессорного модуля, определяющая спектр возможных целевых алгоритмов за счёт набора вариантов его

исполнения. Для процессора показана подсистема машинного кода целевого алгоритма, определяющая как функцию, реализуемую процессором, так и функцию системы в целом.

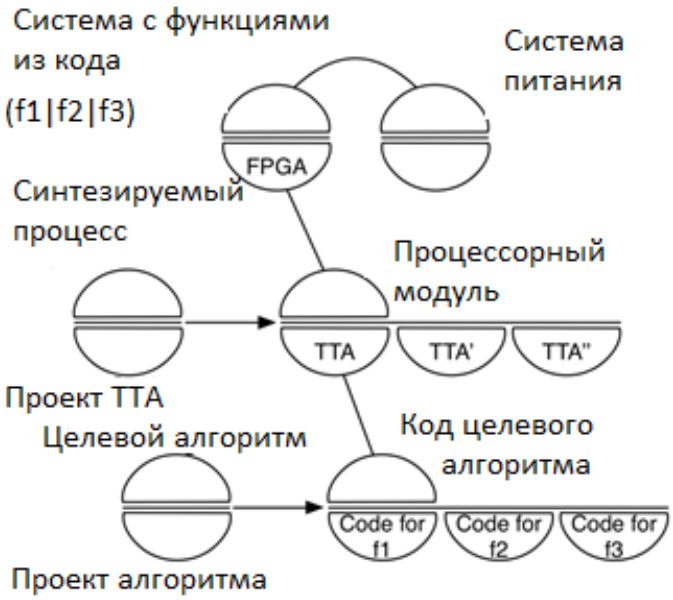


Рисунок 30 – Пример использования системно-иерархической нотации

Данный архитектурный стиль сохранил все свойства «Бургер-диаграмм», расширив его возможности для работы со спецификациями и этапами жизненного цикла. К сожалению, это не решило его основную проблему: использование общих понятий для описания уровневой организации. Описанные свойства делают системно-иерархический архитектурный стиль хорошим инструментом для документирования и анализа существующих систем, а также для теоретических исследований уровневой организации.

2.3 Модифицированный граф актуализации

Архитектурный стиль модифицированного графа актуализации построен на базе графа актуализации (пункт 1.4.1.3). В нём сделана попытка отказа от понятия транслятора, как от центрального элемента стиля с целью устранения проблемы смешения интересов при проектировании. Для этого в качестве вершин графа используются конфигурации уровней ВСС, а в качестве рёбер – преобразования между ними. Пример на Рисунок 31.

Это изменение позволило:

1) Приблизиться по простоте к стилю уровневой диаграммы (пункт 1.4.1.1).
2) Приблизиться к предложенной в подразделе 1.2.1 трактовке ВПЛ.
3) В значительной степени уйти от проблемы смешения интересов.

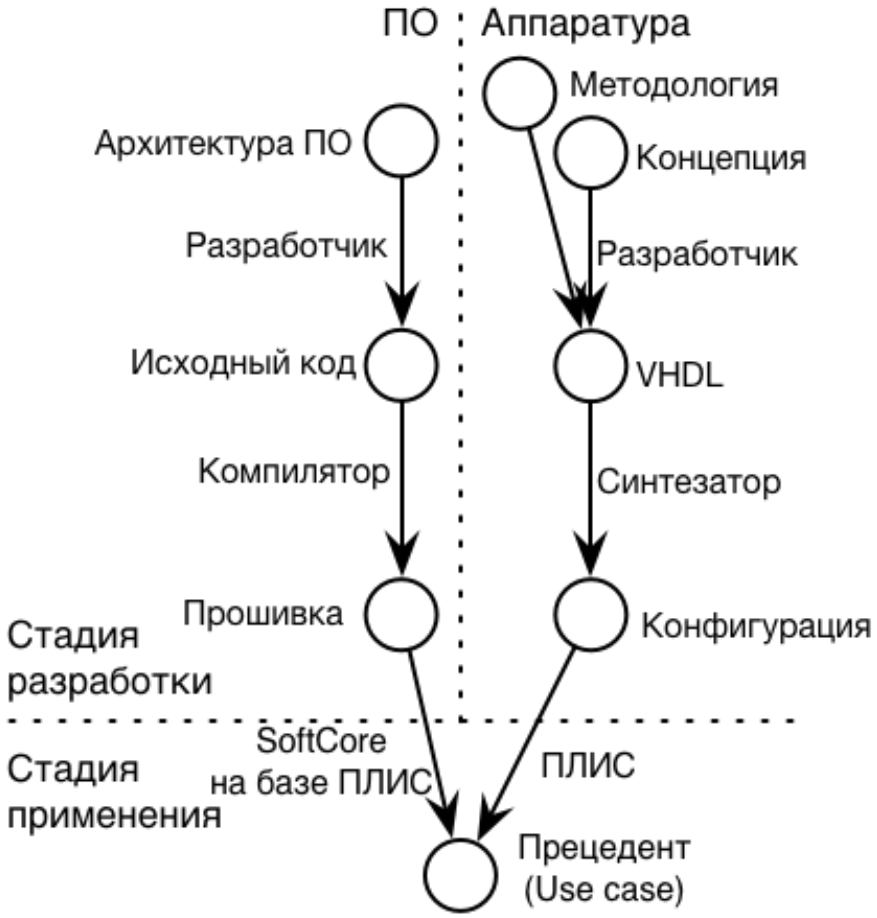


Рисунок 31 – Пример модифицированного графа актуализации

Данный архитектурный стиль позволил продемонстрировать важность моделей вычислений и языковых средств для выделения уровней ВСС. Это позволило уйти от рассмотрения библиотек как отдельных уровней организации ВСС.

К сожалению, архитектурный стиль модифицированного графа актуализации не позволяет описывать уровневую организацию в полном объёме, так как не предоставляет механизмов для описания функциональных возможностей уровней и структуры конфигураций.

2.4 Модель-процесс-вычислитель

Данный архитектурный стиль разрабатывался как инструмент для проектирования и документирования уровневой организации ВСС согласно требованиям раздела 2.1. Название стиля сформировано по элементам, необходимым для описания любого уровня: модели, процесса и вычислителя – «модель-процесс-вычислитель» (МПВ).

Первые результаты работ над МПВ опубликованы в работах [13,61,93] и получены в процессе анализа и документирования уровневой организации существующих систем. Основную сложность представляла фиксация виртуальных вычислителей.

В Таблица 1 приведено краткое описание элементов архитектурного стиля МПВ. Используемые понятия тесно связанны между собой. На Рисунок 32 показана предлагаемая нотация, построена на базе примитивов языка моделирования UML.

Таблица 1 – Элементы архитектурного стиля «модель-процесс-вычислитель»

Название	*Описание*
Уровень	Уровень ВСС – совокупность вычислителя, вычислительных процессов и их моделей. Уровни определяются вне зависимости от конкретного момента времени и могут включать в себя как архитектурные представления, так и модели сформированные в процессе отладки.
Модель	Описание вычислительного процесса или его части на заданном уровне ВСС. Является частичным синонимом понятия «конфигурация».
Вычислительный процесс	Проекция вычислительного процесса на один из уровней ВСС, в рамках которого он имеет структуру и описание, заданное моделью. Целостный вычислительный процесс не отображается ввиду отсутствия средств для работы с ним. Структура вычислительного процесса всегда определена в рамках уровня ВСС.
Частичный вычислительный процесс	Фрагмент вычислительного процесса, в рамках которого происходит абстрагирование от не интересующих частей. Например, вычислительный процесс подпрограммы.
Виртуальный вычислительный процесс	Вычислительный процесс, выполняемый в рамках виртуального вычислителя. Не представлен на стадии эксплуатации системы, но используется при её разработке. Например, вычислительный процесс на языке С или другом компилируемом языке.
Вычислитель	Целостный элемент ВСС, позволяющий актуализировать вычислительные процессы. В отличие от ВПЛ, всегда рассматривается как сконфигурированный для всех необходимых вариантов использования (абстракция от конкретного момента времени).
Вычислительный механизм	Элемент вычислителя, позволяющий актуализировать определённую группу элементов вычислительного процесса. Для вычислительных механизмов не задаются отношения – они могут «пересекаться» между собой. Могут не соответствовать структуре вычислителя. Используются для описания функциональных возможностей вычислителей.
Виртуальный вычислитель	Частный случай вычислителя. Не представлен на стадии эксплуатации системы, но используется при её разработке. На практике соответствует виртуальным машинам синтезируемых или компилируемых языков. Например, С машина, Verilog машина и т.д.

Название	*Описание*
Отношение актуализации вычислительного процесса	Устанавливается между вычислителем и актуализируемым им вычислительным процессом. В нотации отображается при помощи взаимного позиционирования: вычислительный процесс отображается непосредственно над вычислителем.
Отношение соответствия	Устанавливается между моделью и вычислительным процессом. Следует рассматривать как гипотезу, так как может нарушаться в случае сбоев и ошибок.
Отношение трансляции	Формальное соответствие между моделями вычислительного процесса по заданному критерию. Критерий может быть произвольным, но чаще всего это поведение.
Отношение виртуализации	Формальное соответствие между вычислительным процессом и сформированным в рамках него вычислителем.

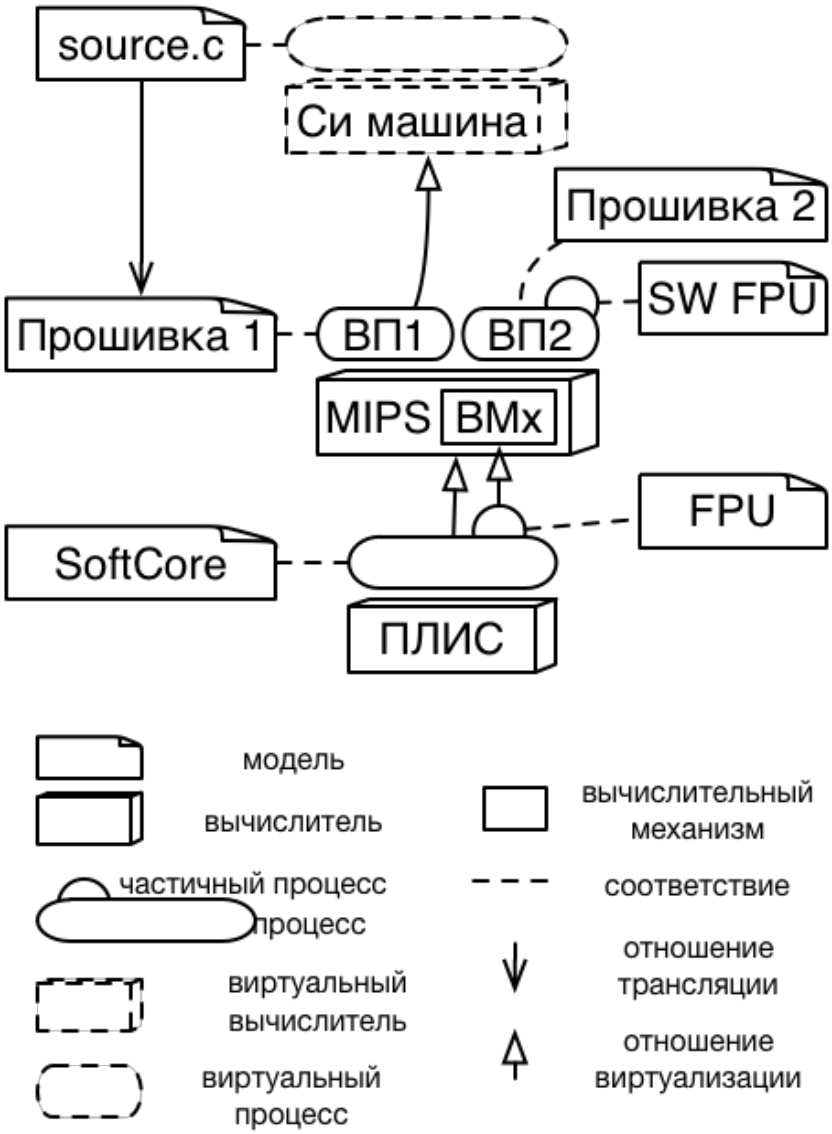


Рисунок 32 – Нотация архитектурного стиля «модель-процесс-вычислитель»

В подразделах будут детально разобраны предлагаемые понятия и причины их появления:

1) В подразделе 2.4.1 будет рассмотрен вопрос представления отдельного уровня ВСС.
2) В подразделе 2.4.2 будет рассмотрен вопрос о взаимодействии между уровнями ВСС, освещены вопросы отношения виртуализации и трансляции; продемонстрированы сложности, возникающий при разработке многоуровневых ВСС.
3) В подразделах 2.4.3, 2.4.4 будут приведены примеры применения МПВ для анализа архитектурных решений в области проектирования уровневой организации.

2.4.1 Представление уровня встроенных систем

Описанные в разделе 2.1 требования, а также выбранная точка зрения на уровневую организацию, требуют рассмотрения уровня ВСС как сложного объекта, включающего в себя разнородные, хотя и тесно связанные между собой объекты: конфигурацию, вычислительный процесс и вычислительную платформу. Связи между ними, обеспечивающие целостность уровня, могут различаться в зависимости от устройства ВПЛ и её инструментальной цепочки. Эта проблема может решаться следующими способами:

1) Игнорирование. Рассмотрение уровня ВСС, как чёрного ящика. Не подходит в виду противоречия требованиям.
2) Детальная классификация и описание. Не подходит, так как препятствует унифицированному рассмотрению уровневой организации.
3) Смена системы абстракций. Замена «естественной» системы понятий на новую, не зависящую от устройства ВПЛ и инструментария.

Из-за того, что первые два варианта противоречат требованиям, был выбран третий. В качестве методологического базиса для новой системы понятий и отношений была выбрана логическая парадигма (пункт 1.1.2.4). В рамках неё система понятий выбирается, основываясь на наших интересах и игнорируя «физический смысл». Аналогично, с точки зрения отношений между объектами – нам не важно ни то, чем обеспечено это отношение, ни то, обеспечено ли оно в принципе, если их определение целесообразно для решения практических задач (далее будут приведены соответствующие примеры).

Как отмечалось выше, «естественными» понятиями для анализа уровней являются: конфигурация, процесс и ВПЛ. Отношение конфигурации связывает уровень в единое целое. Предлагается осуществить их замену на понятия *модели* вычислительного процесса (далее – модель), *процесса* и *вычислителя* соответственно, а отношение конфигурации рассматривать как

свершившийся факт, разбитый на отношения *соответствия* (между моделью и процессом) и *актуализации* (между процессом и вычислителем). Отношение между моделью и вычислителем в явном виде не рассматривается.

Замена понятия конфигурации на понятие модели смещает внимание с вопроса конфигурации (формирования целевого вычислителя) на вопрос соответствия между моделью и вычислительным процессом. Это позволяет рассматривать уровни ВСС с позиции организации вычислительного процесса и работать в условиях частичного соответствия модели и вычислительного процесса (например, с архитектурными описаниями).

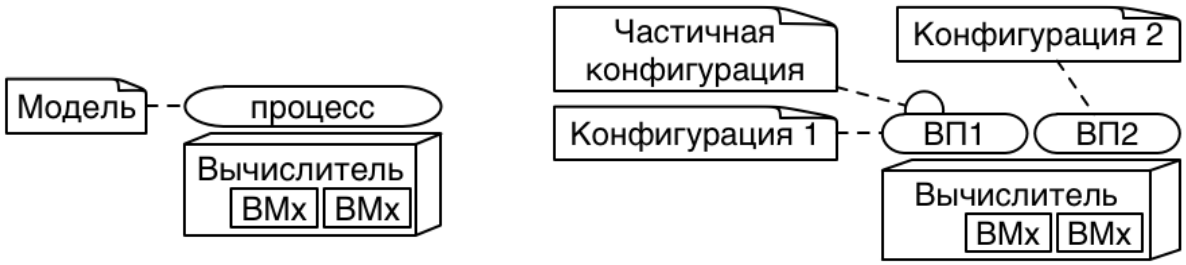
Замена понятия ВПЛ на понятие вычислителя необходимо для обеспечения целостность уровня ВСС. ВПЛ сама по себе может как являться, так и не являться целостным элементом ВСС, например, конструктивные ВПЛ требуют соединительного материала и могут значительно отличаться по составу в каждом конкретном применении. Вычислитель, в свою очередь, всегда конструктивно завершён и обладает конкретной функциональностью. Примечание: альтернативный вариант – использовать термин «виртуальная машина», но предпочтение было отдано «вычислителю», как более общему понятию, не делающем акцент на программном способе реализации.

Цель замены отношения конфигурации на совокупность отношений соответствия и актуализации – абстрагироваться от конкретного момента времени рассмотрения системы. Свойство соответствия определяется человеком, работающим с ВСС и имеющим информацию о её исходных кодах (моделях). При этом данное отношение может быть установлено с ошибкой (версия ПО), либо выступать в роли гипотезы (unit тест). Отношение актуализации фиксирует сам факт выполнения процесса вычислителем, полностью абстрагируясь от того, как и когда это происходит – в результате исполнения модели или потому, что вычислитель произведён так, чтобы соответствовать ей.

На Рисунок 33 а) приведены элементы графической нотации, описывающие отдельный уровень ВСС. Используемых понятий:

1) М*одель вычислительного процесса* (модель) – описание вычислительного процесса на определённом уровне ВСС. К примеру: программа на С, конфигурация ПЛИС, схема электрическая принципиальная, архитектурная спецификация, журнал. В качестве модели может выступать спецификация любого размера, ограничение – единая модель вычислений.

2) *Вычислитель* – целостный элемент ВСС, определяющий возможные варианты развития вычислительного процесса или процессов в рамках отношения актуализации. Является частным случаем вычислительного механизма.
3) *Вычислительный процесс* (процесс, ВП) – последовательность смены состояний вычислителя, соответствующая указанной модели.
4) Вычислительный механизм (далее – ВМХ) [6] – элемент вычислителя, актуализирующий часть его вычислительного процесса. В предельном случае – совпадает с вычислителем.



а) *б)*

Рисунок 33 – (а) Одноуровневая встроенная система; (б) Одноуровневая реконфигурируемая встроенная система с частичной моделью

На Рисунок 33 б) продемонстрирован способ описания частичных моделей, частичных процессов и реконфигурации. Частичные модели и процессы необходимы для документирования методологической составляющей уровневой организации. Реконфигурация отображается через изображение группы вычислительных процессов для одного вычислителя. Взаимное положение процессов произвольно.

Определим основные характеристики элементов уровня ВСС. Для модели – модель вычислений и нотация. Для процесса – структура. Для вычислителя – состав вычислительных механизмов. Структура процесса определяется, частично, моделью вычислений, частично, принципами функционирования вычислителя.

Вычислительный механизм (ВМХ) позволяет обеспечить определённый тип элементов ВП. Отношение между ВМХ и компонентами ВП аналогично отношению вычислителя и ВП. Способ выделения ВМХ не регламентируется и может производиться на основании модели, внутреннего устройства вычислителя или по иным критериям. Наличие и характер взаимосвязей между ВМХ, также не регламентируется, согласно требованиям к архитектурному стилю. Важно отметить, что ВМХ не может реализовываться в рамках текущего уровня ВСС, даже если это возможно (например, функция, реализованная на языке ассемблера, не является ВМХ-мом процессора, в отличие от эквивалентной функции, реализованной в качестве его команды).

Примерами ВМХ могут служить: механизм мгновенной пересылки данных по линии (проводнику), механизм контроля целостности данных в шине, механизм сложения, механизм первой стадии конвейера, механизм выполнения команды `add` в конвейерном процессоре, механизм сторожевого таймера (Watchdog timer).

Модели, процессы и вычислители могут быть виртуальными. Это означает, что данные объекты не существуют, как физические или информационные объекты (подробнее термин «possible individual» стандарта [46]).

2.4.2 Представление уровневой организации многоуровневых встроенных вычислительных систем

Как отмечено в пункте 1.2.2.5, существует три способа обеспечения ВПЛ: языковые, конструктивные и программируемые.

Языковая ВПЛ может быть представлена как функция перевода конфигурации из одного формата в другой. Это может быть описано как две модели, связанные отношением трансляции (достигается за счёт абстракции от конкретного момента времени). Под *отношением трансляции* понимается формальное соответствие двух моделей друг-другу по заданному критерию (как правило, поведенческому). К примеру, компиляция с языка С в исполняемый код, трансляция командой разработчиков архитектурных спецификаций в целевую систему или трассировка схемы электрической принципиальной. Отличия между моделями, состоящими в отношении трансляции, могут заключаться в потере информации (имена переменных, наименование функций) и изменении структуры вычислительного процесса. Хорошим примером последнего являются оптимизирующие компиляторы.

Количество трансляций в системе ограничено практической целесообразностью (в технологиях, ориентированных на разработку специализированных языков [81] их число может достигать десятков). В то же время, добавление любого уровня трансляции требует, как минимум, инструментальных средств, увеличивает семантический разрыв между моделями, с которыми работает разработчик, и моделями, которые фактически исполняются. Это затрудняет отладку и профилирование систем.

Конструктивные и программируемые ВПЛ позволяют разработчику организовать требуемый вычислительный процесс путём программирования или формирования взаимосвязей между элементами ВПЛ. При организации ВП, разработчик абстрагируется от деталей реализации ВПЛ, фокусируясь на вопросах решаемой задачи. В рамках стиля МПВ, формирование ВПЛ данных типов описывается через отношение виртуализации, связывающее

между собой вычислительный процесс нижележащего уровня и сформированный вычислитель. Под *отношением виртуализации* следует понимать абстракцию над вычислительным процессом, формирующую вычислитель вышележащего уровня ВСС, позволяющий актуализировать любой допустимый ВП. Если существует вычислительный процесс невыразимый в рамках сформированного уровня, значит отношения виртуализации определено некорректно. Примеры отношения виртуализации:

- переход от рассмотрения ВП интегральной схемы к рассмотрению ВП, выполняемого процессором (реализованном в виде интегральной схемы);
- переход от рассмотрения ВП программы на С к рассмотрению ВП виртуальной машины (реализованной на языке С).

Количество виртуализаций ограничено падением производительности, сопровождающим большинство переходов, и ростом сложности разработки.

Выбор между виртуализацией и трансляцией, обусловливается сложностью разработки эффективного транслятора, в свою очередь, прямо зависящей от разницы между моделями вычислений. Это можно наблюдать для множества языков программирования, использующих виртуальные машины. В частности, современные высокоуровневые языки программирования предпочитают использовать виртуальную машину или её элементы (так называемый «Run-time» языка программирования) для управления памятью, в то время, как транслируемые решения требуют участия пользователя. Другим интересным примером являются законы роста производительности: для вычислительного ядра – Закон мура (рост в два раза за 18 месяцев) и для компиляторов – «Proebsting’s law» ([94], рост в два раза за 18 лет).

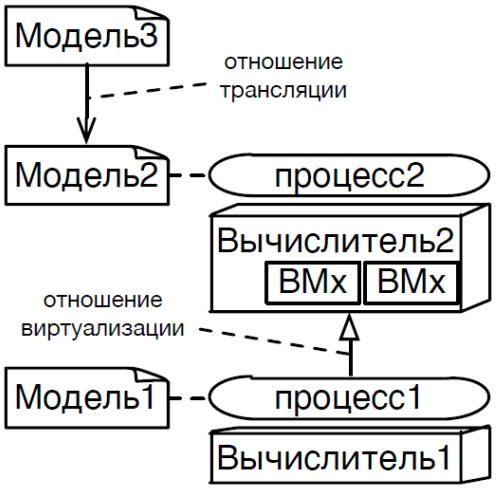
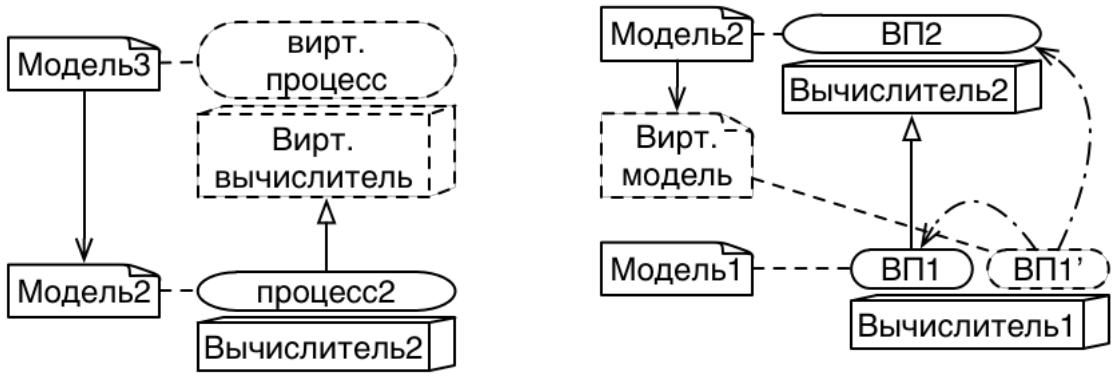


Рисунок 34 – Отношение трансляции и виртуализации

На Рисунок 34 приведена ВСС с тремя уровнями, где уровни 3 и 2 связаны отношением трансляции, а уровни 2 и 1 – отношением виртуализации. Как можно видеть, уровень 3 является

неполным, а уровень 1 не имеет связки с уровнем 2 через трансляцию, что является некорректным, так как рассмотрение уровней перестает быть унифицированным.

Неполнота уровня 3 связана с отсутствием необходимости отображения «несуществующих» элементов вычислительной системы. На практике, работа с ними часто является необходимой, для чего вводятся виртуальные версии элементов уровня ВСС. *Виртуальный вычислитель* – вычислитель, «исполняющий» модель до её трансляции (Рисунок 35, а). К примеру, при работе в отладчике с программой на языке С, реальный вычислительный процесс обеспечивается машинным кодом, исполняемым процессором, хотя создаётся видимость, что код выполняется на языке С. Степень соответствия реального и видимого, зависит от наличия отладочной информации и оптимизации.



а) б)

Рисунок 35 – (а) Виртуальный вычислитель; (б) Виртуальная модель

Отсутствие отношения трансляции между уровнем 1 и 2 решается при помощи *виртуальной модели* (Рисунок 35, б) – модели вычислительного процесса, соответствующие вычислительному процессу, объединяющему в себе свойства *ВП1* и *ВП2* (на схеме показано штрих пунктиром). Виртуальная модель необходима при принятии решения о замене отношения виртуализации отношением трансляции. Практическая работа с виртуальными моделями встречается при низкоуровневой отладке, когда вычислительный процесс перестаёт соответствовать виртуализированному вычислителю, следовательно, необходимо просмотреть вычислительный процесс во всей полноте.

Также виртуальная модель демонстрирует невозможность использования термина «конфигурация» при работе с уровневой организацией. Очевидно, что построение виртуальной модели в понимании «конфигурация» для множества ситуаций является неразрешимой задачей, в то время, как построение модель – относительно тривиальна задача (простейший случай – журнал событий, сложный пример описан в работе [95]).

Также для полноты унификации введено понятие *виртуального процесса,* который является спутником как виртуального вычислителя, так и виртуальной модели.

Необходимо отдельно рассмотреть проблемы, связанные с включением в уровневую организацию новых уровней, участвующих в процессе разработки. Уровни, основанные на отношении трансляции, требуют написания компилятора, позволяющего отобразить любую корректную модель в необходимый вид. При этом соответствие должно быть статическим.

В случае отношения виртуализации, ситуация несколько сложнее, так как при анализе необходимо учитывать переменную составляющую – прикладной код верхнего уровня. Как следствие, необходимо рассматривать развитие вычислительного процесса в динамике на базовом и виртуализированном уровне одновременно, анализируя аномалии ВП. В случае, если уровни имеют значительные отличия с точки зрения моделей вычислений, то, с высокой вероятностью, совместная работа с их инструментальными средствами будет затруднена (Рисунок 36). Отметим, что многие уровни ВСС имеют смешанную организацию.

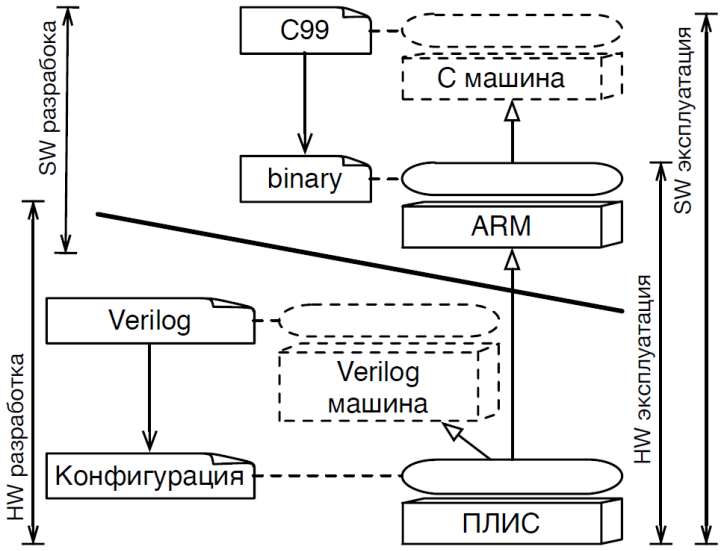


Рисунок 36 – Отношение виртуализации между аппаратным и программным уровнем встроенной системы

2.4.3 Пример: описание проектных альтернатив

Рассмотрим три способа реализации одной и той же функциональности с точки зрения уровневой организации. В качестве примера возьмём механизм мемоизации [96], представляющий из себя встраивание кэша в рекурсивную функцию. Классическим примером, пусть и синтетическим, является наивно реализованная функция Фибоначчи через рекурсивный вызов (пример исходного кода приведён на языке Python):

```
def fib(n):
    if n == 0 or n == 1: return 1
    else: return fib(n - 1) + fib(n - 2)
```

Как можно видеть, данное определение функции является калькой определения из учебника по математике и имеет огромную вычислительную сложность. Применение метода мемоизации будет кешировать каждый вызов функции fib, и в результате вычисления не будут дублироваться. Рассмотрим следующие варианты реализации мемоизации: вручную, при помощи BMX, через трансляцию.

Реализация мемоизации вручную заключается в том, что разработчик должен самостоятельно определить кэш и реализовать работу с ним. Основным недостатком этого решения является то, что его нельзя повторно использовать для других функций. Исходный код данного решения приведён ниже.

```
fib_mem_cache = {}
def fib_mem(n):
    if n in fib_mem_cache:
        return fib_mem_cache[n]
    if n == 0 or n == 1:
        tmp = 1
        fib_mem_cache[n] = tmp
        return tmp
    else:
        tmp = fib_mem(n - 1) + fib_mem(n - 2)
        fib_mem_cache[n] = tmp
        return tmp
```

В данном решении сознательно не сделана очевидная оптимизация (внесение в кэш значений для 0 и 1) с целью сделать модификацию исходного кода наиболее «машинной». Уровневая организация данного решения приведена на Рисунок 37.

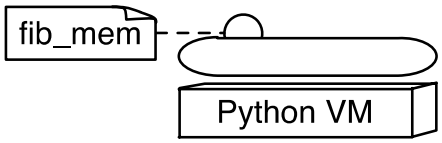


Рисунок 37 – Ручная реализация мемоизации

Реализация при помощи BMX может быть реализована функцией-декоратором (аналогичные функции есть и в стандартных библиотеках Python 3, а именно: `@functools.lru_cache`). Для разработчика будет создана аннотация, позволяющая мемоизировать произвольную функцию. Это решение работает в «run-time» и функционально эквивалентно реализованному вручную.

```
def mem(f):
    cache = {}
    def mem_(n):
        if n in cache:
            return cache[n]
        else:
            tmp = f(n)
            cache[n] = tmp
            return tmp
    return mem_
```

```
@mem
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Как и для большинства решений времени исполнения, имеются накладные расходы. В данном случае это удвоенная глубина стека. Уровневая организация данного решения приведена на Рисунок 38. Важно заметить, что для данного примера выделение механизма мемоизации на отдельный уровень является избыточным, нет смены модели вычислений или языка разработки. Оно сделано с целью демонстрации возможностей архитектурного стиля.

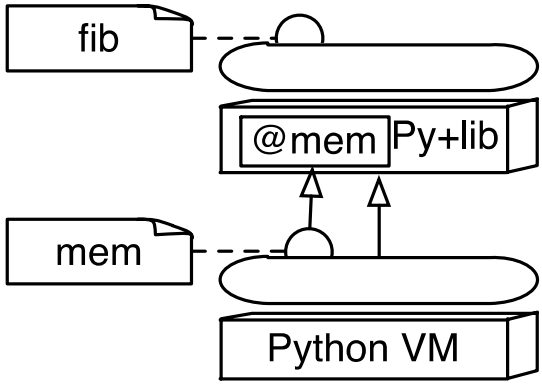


Рисунок 38 – Реализация мемоизации в качестве вычислительного механизма

Решение задачи мемоизации через трансляцию является слишком объёмным, ввиду чего приводиться не будет. Оно заключается в анализе наивной реализации функции Фибоначчи с последующим её приведением к виду, полученному путём реализации вручную. Решение может быть реализовано с использованием любого удобного для разработчика инструментария. Необходимо отметить, что сложность реализации такого транслятора будет значительно превышать сложность описанного выше BMX, но при этом решение не будет иметь накладных расходов времени исполнения. Уровневая организация данного решения приведена на Рисунок 39.

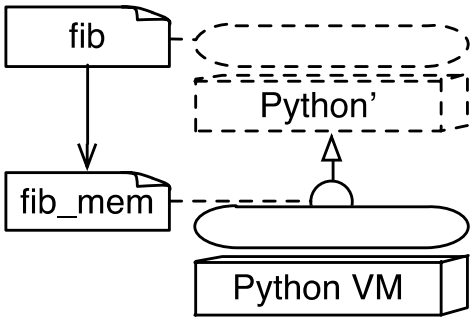
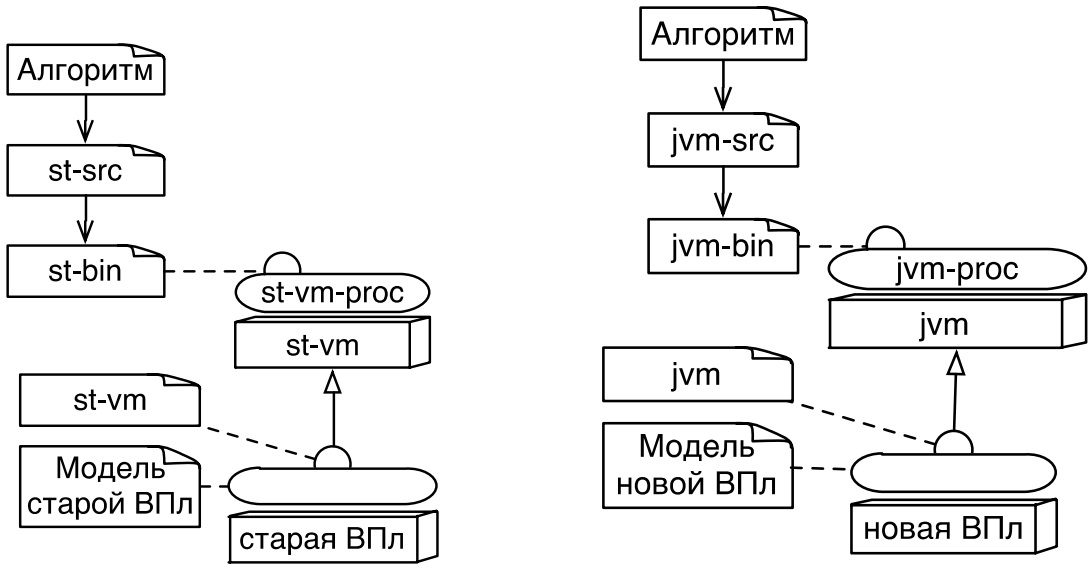


Рисунок 39 – Реализация мемоизации через трансляцию

2.4.4 Пример: проектирование уровневой организации

Важное значение при проектировании занимает исследование ППР, в рамках которого проектировщику необходимо рассмотреть и сравнить между собой доступные варианты уровневой организации. В качестве примера рассмотрим задачу изменения уровневой организации ВСС с переносом скомпилированной пользовательской программы (исходный код отсутствует). Совокупность вычислительных платформ, используемых в ВСС, будем называть системной платформой (СПЛ).



а) б)

Рисунок 40 – а) Старая системная платформа. б) Новая системная платформа

Старая СПЛ приведена на Рисунок 40 а) и включает виртуальную машину языка ST для пользовательского программирования – `st-vm`. Для части используемого заказчиком ПО исходный код недоступен, а повторное его написание затруднено. Бинарные версии программ, полученные в результате компиляции при помощи `st-compiler`, сохранены. Ввиду невозможности дальнейшего производства старой СПЛ (устаревание элементной базы, высокая стоимость миграции системного ПО), стоит задача по миграции на новую СПЛ, в составе которой нет `st-vm`, но присутствует виртуальная машина Java – `jvm`. Её схема приведена на Рисунок 40 б).

На Рисунок 41 представлена спецификация с доступными вариантами уровневой организации. Варианты решения задачи обозначены цифрами, вписанными в окружности. Номера соответствуют приведённому ниже описанию:

1) Миграция `st-vm` на новую ВПЛ. Позволит сохранить инструментальную цепочку для пользователя без изменений, но результат будет избыточным.

2) Разработка транслятора `translator1` из `st-bin` в бинарный код для `jvm` (`jvm-bin`) на новой ВПЛ. Это позволит сохранить инструментальную цепочку пользователя, но усложнит новую СПЛ.
3) Разработка транслятора `translator2` из `st-bin` в бинарный код `jvm-bin` на инструментальной платформе (ПК). Это изменит инструментальную цепочку пользователя, сохранит новую СПЛ неизменной, предоставит широкую свободу в средствах реализации транслятора.
4) Разработка транслятора `translator3` из `st-bin` в бинарный код для `jvm-bin` на `jvm`. Позволит сохранить инструментальную цепочку пользователя, но приведёт к усложнению новой СПЛ. В отличие от варианта 2, реализация на `jvm` будет обладать худшей производительностью, но будет проще в реализации.
5) Трансляция пользовательского кода для `st-src` в бинарный код для `jvm-bin`. Этот вариант аналогичен варианту 2. Недоступен в рассматриваемой ситуации.
6) Воспроизведение прикладного пользовательского кода для `jvm-src`. Аналогичен варианту 2 и 4. Недоступен в рассматриваемой ситуации.

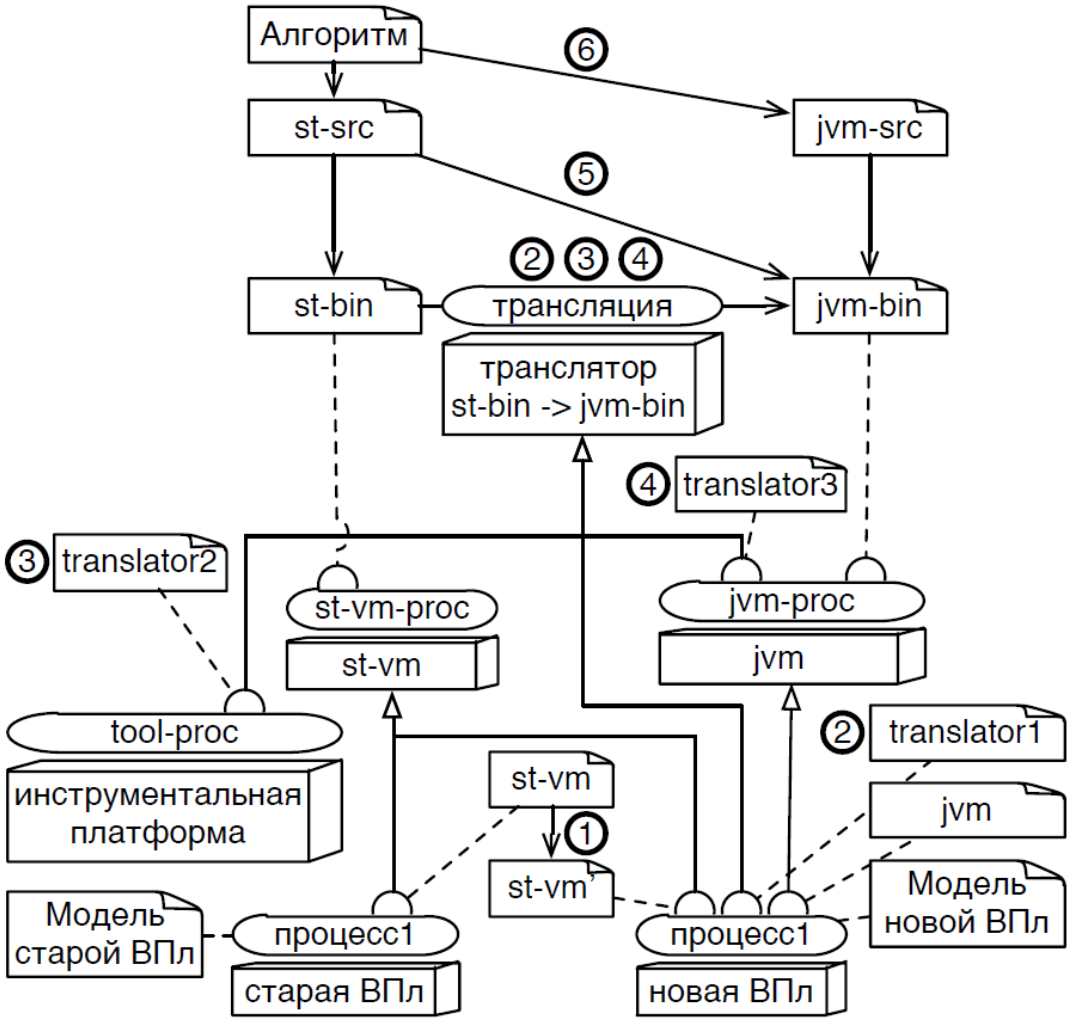


Рисунок 41 – Варианты уровневой организации решения задачи проектирования

Как можно видеть, предлагаемый архитектурный стиль МПВ позволил отобразить все перечисленные варианты уровневой организации. Совместное отображение вариантов значительно упростило поиск вариантов решения задачи и их сравнительный анализа.

2.5 Выводы

1) Выполнен критический анализ особенностей доступных архитектурных стилей для проектирования и документирования уровневой организации. На его основе сформулированы требования к архитектурным стилям уровневой организации.
2) Проведены поисковые работы в области разработки архитектурных стилей для проектирования и документирования уровневой организации, позволившие разработать новые архитектурные стили на основе существующих:
   - системно-иерархический архитектурный стиль, позволяющий фиксировать уровневую организацию с точки зрения системной организации и структуры жизненного цикла;
   - архитектурный стиль модифицированного графа актуализации, позволяющий фиксировать уровневую организацию с точки зрения языковых средств, используемых для описания вычислительного процесса, и их взаимосвязей.
3) Разработан оригинальный архитектурный стиль «модель-процесс-вычислитель», позволяющий фиксировать уровневую организацию в полном объёме и без смешения интересов. Это позволяет повысить качество архитектурной документации и сфокусировать внимание на вопросах уровневой организации. Основан на методах моделирования высших онтологий.
4) Продемонстрирована практическая применимость архитектурного стиля «модель-процесс-вычислитель» для задач проектирования и документирования уровневой организации, включая исследование и анализ пространства проектных решений.
5) Для всех разработанных архитектурных стилей предложены языки архитектурного описания, позволяющие документировать уровневую организацию встроенных систем.

3 Разработка методики моделирования многоуровневых встроенных систем

В данной главе предлагается методика сквозного моделирования многоуровневых ВСС и их элементов на основе архитектурного стиля МПВ. Методика позволяет разрабатывать модели отдельных уровней ВСС, а также их взаимосвязей в терминах архитектурного стиля МПВ. Создаваемые модели применяются в разработке САПР для вычислительных систем с дискретным временем. Они позволяют реализовать:

- сквозную симуляцию, верификацию, отладку и профилирование;
- работу со структурированным представлением конфигураций и ВП;
- возможность трассировки межуровневых взаимосвязей.

Методика включает в себя процедуры для моделирования: конфигураций (модели ВП), ВП (последовательности смены состояний вычислителя), а также отношений актуализации, трансляции и виртуализации.

3.1 Требования к формализации и выбор средств моделирования

Формализация архитектурного стиля «модель-процесс-вычислитель» позволяет применять его для решения задач совместного моделирования уровневой организации ВСС путём их встраивания в САПР. Разработка совместных моделей – комплексная задача, в рамках которой необходимо работать сразу с несколькими, зачастую, различными моделями вычислений. Это само по себе является нетривиальной задачей, требующей реализации отношения актуализации для отдельных уровней ВСС (подраздел 2.4.1). Необходимость согласования уровней ВСС между собой ещё больше увеличивает сложность разработки. САПР для разработки многоуровневых ВСС или их элементов должен отвечать следующим требованиям:

1) Совместная симуляция. Разработчик может видеть процесс на всех уровнях системы и снимать метрики.
2) Совместная отладка. Разработчик может отследить причину каждого шага ВП на любом из уровней системы. Сопоставить её с моделью ВП.
3) Совместное профилирование. Заключается в сборе информации о ходе ВП на всех уровнях системы и в сведении её к единому представлению. Позволяет находить узкие места в устройстве системы, невидимые при анализе отдельных уровней.

Для эффективного использования моделей многоуровневых ВСС и их элементов в задачах проектирования и исследования уровневой организации, необходимо следующее свойство: расширяемость моделей с целью включения новых метрик, ВМХ и уровней представлений ВП. Немаловажное значение играет возможность интеграции моделей с существующими инструментальными средствами. К сожалению, реализация последнего сильно зависит от используемых ВПЛ и не может быть обеспечена в общем случае. Следование актуальным тенденциям в области интеграции данных [45,46] должны обеспечить необходимый уровень адаптивности моделей.

Моделирование уровневой организации ВСС содержит следующие этапы:

1) Определение общих методологических принципов моделирования.
2) Определение независимых от реализации процедур моделирования.
3) Разработка независящих от реализации моделей.
4) Реализация моделей в рамках инструментальной платформы для последующего встраивания в САПР.

Первый этап определяется архитектурным стилем МПВ, (пункт 1.1.2.4). Второй этап имеет принципиальное значение, так как он:

- универсален и может повторно использоваться в различных проектах;
- определяет не только свойства будущих моделей, но и процедуры их реализации.

Обозначенная выше независимость от реализации является условной, так как выбор фиксирует рекомендуемый стиль программирования [86], что косвенно определяет стоимость использования различных ВПЛ при реализации.

В рамках третьего этапа реализуются конкретные модели и их элементы. В качестве элементов могут выступать: представления конфигураций, процедуры их верификации, механизмы симуляции, элементы механизмов трансляции и виртуализации. Отсутствие связанности со способом реализации позволяет повторно использовать модели в других проектах.

Четвёртый этап определяется индивидуально для каждого проекта и коллектива. Выбор основывается на доступных кадрах и специфике решаемой задачи.

Это определяет элементы методики моделирования многоуровневых ВСС и их элементов, представленные на Рисунок 42 в привязке к МПВ. Методика должна включать процедуры для моделирования: моделей ВП (далее конфигураций, чтобы избежать путаницы), вычислительного процесса, отношений актуализации, трансляции и виртуализации.

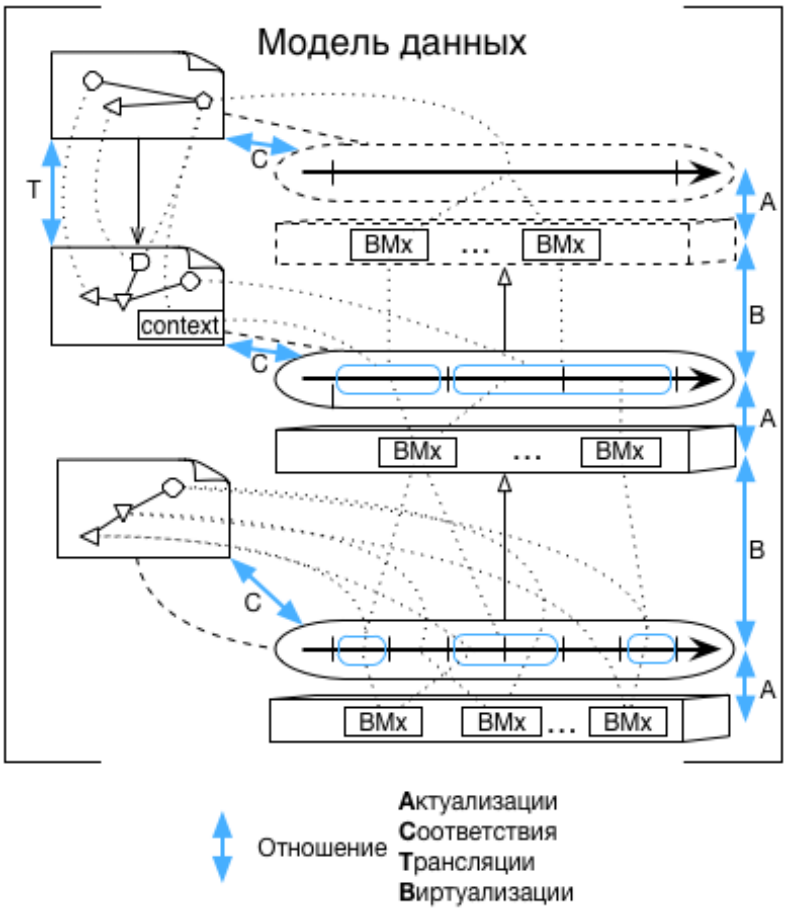


Рисунок 42 – Моделирование уровневой организации

Выбор общих методологических принципов моделирования основывается на двух факторах: (1) простота и скорость последующей реализации моделей и (2) набор свойств, закладываемых в модель по построению. Ниже представлены наиболее перспективные из доступных вариантов:

1) Объектно-ориентированный анализ. Как показано в работе [45], его основа – сущностная парадигма. Она прямо противоречит логической парадигме, что не позволяет сохранить гибкость и адаптивность архитектурного стиля МПВ, и как следствие – не подходит для исследовательских задач.
2) Использование специализированных языков [79]. Разработка стека специализированных языков для описания моделей уровневой организации нашло своё применения в индустрии, в частности, их можно видеть в проектах [81,82]. Данный вариант имеет высокую стоимость, связанную со сложностью задачи и серьёзными проектными рисками.
3) Математический аппарат теории множеств. Позволяет описать уровневую организацию ВСС в статическом виде, демонстрируя состав уровневой организации через множества, а межуровневые взаимосвязи через функциональные отношения.

Описание же динамических процессов в системе, необходимое для реализации САПР затруднено.

4) Математический аппарат теории категорий [97,98]. Позволяет описывать объекты и взаимодействия между ними, не фиксируя внутреннее устройство объектов или природу их отношений. Нашёл широкое применение в областях вычислительной техники [97], системах моделирования, языках программирования [99,100], системной инженерии (SysMoLang [38]), теории проектирования ИУС [101]. Также, математический базис оставляет перспективу для формальной верификации моделей, что полезно для разработки систем ответственного применения [102].

Комбинация второго и четвёртого варианта является оптимальным решением, в рамках которого теория категорий выступает в качестве «каркаса» специализированных языков программирования, а модели реализуются посредством встраиваемых специализированных языков программирования (Embedded Domain Specific Language, EDSL [79]). Такой подход позволяет обеспечить формальный математический базис для многоуровневых моделей (расширяемый «адаптерами» для отображения в другие математические теории, в случае необходимости) и сформировать дружественным к разработчику инструментарий. В рамках апробации предложенного подхода к моделированию, работа с моделями производилась в рамках функционального языка программирования со строгой типизацией (Haskell [100]), объектно-ориентированного языка программирования с динамической типизацией (Python 3 [103]) и низкоуровневого языка программирования (Rust).

3.2 Процедура моделирования конфигурации

В общем случае, модель конфигурации можно считать текстовым документом произвольного формата, либо сводимым к таковому. Это позволяет значительно упростить процесс машинной обработки. К модели конфигурации предъявляются следующие требования:

1) Структурированность. Необходимо для отладки и отслеживания взаимосвязей между ВП и элементами конфигурации.
2) Удобство для машинной обработки (верификации и генерация исполняемого кода).

Моделирование конфигурации производится в рамках двух категорий:

1) $\mathcal{ML}_i$ – моноидальная категория с начальным элементом I, представляющая все модели ВП, независимо от их корректности. Под некорректными моделями понимаются те, в рамках которых используются допустимые синтаксические

конструкции, но нарушена семантика. Например, полнота (отсутствие конца структурного блока) и корректность использования ресурсов (неопределённая переменная). Морфизмы соответствуют операциям по добавлению лексем.

2) $\mathcal{M}_i$ – Категория с начальным элементом I, представляющая все семантически корректные модели ВП. Является подкатегорией $\mathcal{ML}_i$, и определяется функтором вложения $Verif$.

$Obj(\mathcal{ML}_i)$ – множество возможных моделей ВП, в том числе и некорректных.

$Mor(\mathcal{ML}_i)$ – множество морфизмов, соответствующих операциям добавления лексем к моделям.

$$\otimes: \mathcal{ML}_i \times \mathcal{ML}_i \to \mathcal{ML}_i \quad (1)$$

$$\forall A \in Obj(\mathcal{ML}_i): I \otimes A = A, A \otimes I = A \quad (2)$$

$$\forall A \in Obj(\mathcal{ML}_i): \exists m, dom(m) = I, \mathrm{cod(m)} = \mathrm{A} \quad (3)$$

$$Verif: \mathcal{M} \to \mathcal{ML} \quad (4)$$

$$Verif(m) \circ Verif(h) \nleftrightarrow Verif(m \circ h) \quad (5)$$

$$Obj(\mathcal{M}_i) \coloneqq \{A \in Obj(\mathcal{ML}_i) | Verif(A)\} \quad (6)$$

$$Mor(\mathcal{M}_i) \coloneqq \{m \in Mor(\mathcal{ML}_i) | Verif\big(dom(m)\big), Verif\big(cod(m)\big)\} \quad (7)$$

$$\forall A \in Obj(\mathcal{M}_i): \exists m, dom(m) = I, \mathrm{cod(m)} = \mathrm{A} \quad (8)$$

Оператор $Verif$ может использоваться для устранения двусмысленности при использовании композиции протоморфизмов. Важно отметить, что оператор $Verif$ не аддитивен относительно композиции. Примеры возникающих ошибок: переменная не определена и переменная определена повторно.

Объекты данных категорий обозначаются в виде списков: [] – начальный объект (I), [a, b] – модель, в которой первая синтаксическая конструкция это «a», вторая «b». Морфизм обозначается как пара вида ([], a), где первый элемент – объект модели, к которой морфизм применяется, второй – синтаксическая или структурная компонента, добавляемая к модели. Указание исходной модели необходимо для однозначной идентификации морфизма. В виду того, что многие морфизмы похожи между собой (обозначают добавление одной и той же структурной компоненты), используются протоморфизмы, которые указываются без объектов, к которым они привязаны. Изображение диаграммы с протоморфизмами приведено на Рисунок 43. Одним из важнейших свойств морфизмов и протоморфизмов является возможность их композиции (оператор «.»). Это позволяет формировать морфизмы, добавляющие серию синтаксических или структурных конструкций. Пример: $([a], b) \circ ([], a) \Leftrightarrow ([], b \circ a) : [] \to [a, b]$.

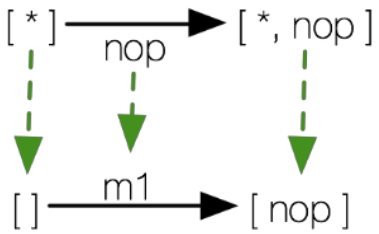


Рисунок 43 – Визуализация протоморфизма и морфизма

3.3 Процедура моделирования вычислительного процесса

Модель вычислительного процесса рассматривает только дискретные вычислительные процессы. Данное ограничение является удовлетворительным компромиссом между областью применения и простотой моделирования, так как большая часть вычислительных систем является дискретными.

Целостный вычислительный процесс представляется моноидальной категорией $\mathcal{P}$ с начальным элементом I, описывающей все возможные состояния вычислителя и варианты развития ВП.

$Obj(\mathcal{P})$ – множество всех возможных состояний вычислителя.

$Mor(\mathcal{P})$ – компоненты ВП. Протоморфизмы соответствуют ВМХ.

Пример диаграммы данной категории приведён на Рисунок 44. Часть объектов представляют состояния вычислителей отдельных уровней (A_i, B_j, C_k) связанных морфизмами (шагами ВП). Другие объекты, обозначенные как $A_i \; x \; B_j$ и $A_i \; x \; B_j \; x \; C_k$, являются произведениями состояний отдельных уровней и описывают целостное состояние ВСС с реальными шагами ВП (относительно одноуровневых представлений). Связанность различных представлений ВП обеспечивается морфизмами-проекциями, извлекающими из произведений составные части (показаны тонкими стрелками).

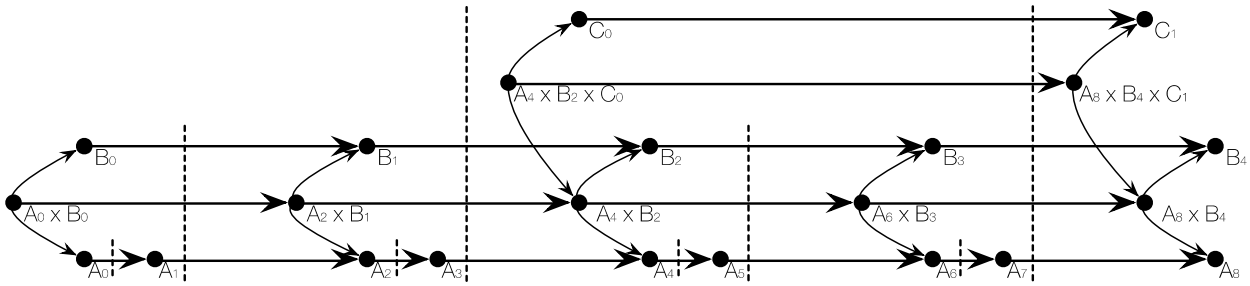


Рисунок 44 – Диаграмма вычислительного процесса

Описание параллельных и последовательных процессов выполняется согласно формулам 9 и 10.

$$g \to h \Leftrightarrow g \circ h \quad (9)$$

$$g \parallel h \Leftrightarrow h \circ g, h \circ g \quad (10)$$

Для упрощения отображения моделей параллельных вычислительных процессов введём нотацию, приведённую на Рисунок 45.

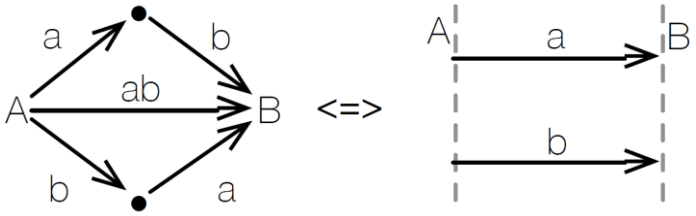


Рисунок 45 – Графическое обозначение параллельных процессов

Протоморфизм однозначно сопоставляется ВМХ – компонентам вычислителя (подраздел 2.4.1). Для описания вычислителя достаточно определить множество ВМХ, соответствующее всем атомарным протоморфизмам. Это позволит описать любой корректный ВП.

3.4 Процедура моделирования отношения актуализации

Отношение актуализации характеризует соответствие между элементами модели вычислительного процесса и ВП. В тоже время, структура ВП однозначно связывается с вычислителем.

Отношение актуализации фиксируется в виде категории $\mathcal{A}$, объектами которой являются пары множества морфизмов категории $\mathcal{M}_i$ и множества морфизмов категории $\mathcal{P}$.

$$Obj(\mathcal{A}) \coloneqq \{(M,H): M \in P(Mor(\mathcal{M}_i)), H \in P(Mor(\mathcal{P}_i)), Actualize(m,h)\} \quad (11)$$

где $Actualize$ есть определённое снаружи отношение соответствия между шагом ВП и фрагментом модели, P – функция взятия множества всех подмножеств. Важно отметить, что морфизмы m и h могут быть как атомарными, так и полученными в результате композиции. Отношение актуализации может реализовываться непосредственно, путём реализации виртуальной машины, или постфактум, восстанавливая взаимосвязь с моделью ВП на основании его структуры. Отношение актуализации использует множества морфизмов в качестве элементов пар, так как одному элементу вычислительного процесса может соответствовать множество элементов модели. Уровень детализации отношения актуализации определяется разработчиком модели и может варьироваться.

Морфизмы категории $\mathcal{A}$, отчасти, совпадают по структуре с морфизмами категории $\mathcal{P}$. Отличия заключаются в том, что данная категория ориентируется на модель вычислений, а не на

работу вычислителя. Это приводит к упрощениям, за счёт которых детали внутреннего устройства и состояния ВП скрываются.

3.5 Процедура моделирования отношения трансляции

Отношение трансляции – отношение между двумя моделями конфигурации, связующие элементы моделей до и после трансляции. Отношение трансляции выражается частичным или полным функтором. Полным называется такой функтор T, который определён для любого объекта категории $\mathcal{M}_i$.

$$T: \mathcal{M}_i \to \mathcal{M}_j \tag{12}$$

$$\forall A.\, A \in Obj(\mathcal{M}_i): A \in dom(T) \tag{13}$$

Частичное представление моделей за счёт трансляции может быть обеспечено функтором $L: \mathcal{M}_j \to \mathcal{M}_i$ (не является обратным для функтора T, как как модель $\mathcal{M}_j$ может быть невыразима в рамках $\mathcal{M}_i$, а трансляция может быть сопряжена с потерей информации):

$$L(g) \circ h \Leftrightarrow g \circ T(h) \tag{14}$$

Введём свойство атомарности морфизма, которое является необходимым для определения полноты функтора T:

$$\begin{aligned} Atom(m) \Leftrightarrow\ & dom(m) = A, cod(m) = B, \\ & \forall h.\, h \in Mor(\mathcal{M}), h \neq m: \nexists h.\, h \neq id_B, cod(m \circ h) = B, \\ & \nexists g.\, g \neq id_A, dom(h \circ m) = A \end{aligned} \tag{15}$$

Теорема. Функтор T является полным, если:

– определён на полном множестве атомарных объектов $\mathcal{M}$:

$$\forall A.\, A \in Obj(\mathcal{M}), Atom(A): A \in dom(T) \tag{16}$$

– снабжён изоморфным преобразованием:

$$\tau(\otimes): T(\mathcal{M}_i) \times T(\mathcal{M}_i) \to T(\mathcal{M}_i) \tag{17}$$

$$\begin{aligned} \tau(\otimes) \coloneqq \{m | A, B \in Obj(\mathcal{M}_i), V(A \otimes B), \\ dom(m) = \big(T(A), T(B)\big), cod(m) = T(A \otimes B))\} \end{aligned} \tag{18}$$

$$\tau(\otimes^{-1}): \mathcal{M}_i \to \mathcal{M}_i \times \mathcal{M}_i \tag{19}$$

$$\begin{aligned} \tau(\otimes^{-1}) \coloneqq \{m | A, B \in Obj(\mathcal{M}_i), V(A \otimes B), \\ dom(m) = \big(T(A \otimes B)\big), cod(m) = (T(A), T(B))\} \end{aligned} \tag{20}$$

Доказательство:

$$T'(O) \coloneqq \begin{cases} (A,B) = \otimes^{-1}(O), \tau(\otimes)(T'(A), T'(B)) \mid O \in dom(\otimes^{-1}) \\ T(O) \mid Atom(O) \end{cases} \quad (21)$$

Модель определяет прямую взаимосвязь между элементами моделей по направлению трансляции. Обратное преобразование описывается в рамках отношения виртуализации, которое позволяет не только восстановить вычислительный процесс транслируемого уровня, но и его отношение актуализации.

Примеры визуализации отношения трансляции на диаграммах приведены на Рисунок 46 и Рисунок 47.

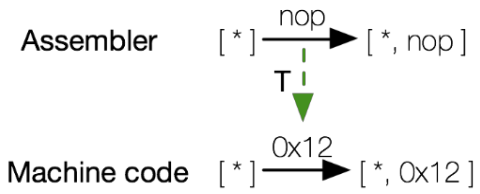


Рисунок 46 – Отношение трансляции для протоморфизма

Фиксация отношения трансляции в виде объектов производится посредством пар, выстроенных по аналогии с отношения актуализации.

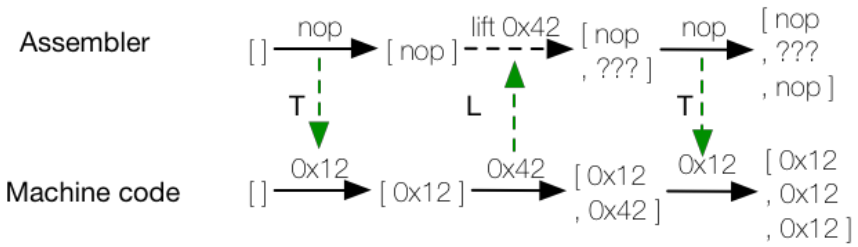


Рисунок 47 – Отношение трансляции для программы с использование функтора

3.6 Процедура моделирования отношения виртуализации

Отношение виртуализации позволяет перейти от рассмотрения ВП к рассмотрению вычислителя вышележащего уровня. Данный переход может быть, как тривиальным – снижение детализации ВП (отказ от структуры составных морфизмов, Рисунок 48), так и сложным – с изменением структуры ВП (включение служебных элементы ВП, изменение последовательности исполнения и т.д.).

Рисунок 48 – Отношение виртуализации

Отношение виртуализации описывается дополнительными морфизмами категории $\mathcal{P}$, заданными следующим выражением:

$$\forall m.\, m \in Mor(\mathcal{P}): \exists A, B.\, dom(m) = A, cod(m) = B, Virtualize(A, B)\} \quad (22)$$

где предикат Virtualize позволяет определить соответствие между состояниями ВП, а A и B – одномоментные состояния ВП разных уровней. Сопоставление фрагментов ВП разных уровней (морфизмов) производится через анализ произведений категории $\mathcal{P}$, как это показано на Рисунок 49. Композиция морфизмов, соответствующая базовому ВП, обозначена пунктиром, соответствующий виртуальному ВП – штрих пунктиром.

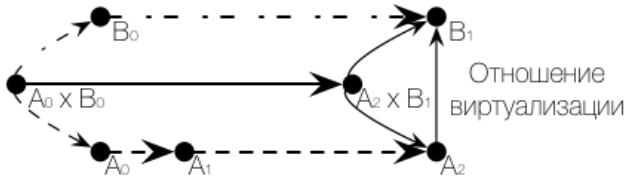


Рисунок 49 – Визуализация морфизма виртуализации

Отношение виртуализации может быть обеспечено следующими способами:

1) В рамках процесса актуализации ВП нижележащего уровня. В таком случае при построении ВП уровня 1, параллельно возникает информация о формировании ВП уровня 2. Этот способ наиболее точный и естественный с точки зрения разработки САПР. Основное ограничение – модель для уровня 1, обеспечивающая виртуальную машину, должна быть реализована в полном объёме.
2) Реализация отношения виртуализации путём анализа ВП нижележащего уровня. Позволяет снизить связанность элементов САПР. Доступны следующие способы анализа ВП:
    a. Анализ состояний. Может быть удобен при работе с программируемыми уровнями, если текущая команда сохраняется в памяти.
    b. Анализ структуры, в рамках которого выполняет поиск паттернов в ВП. Данный подход часто применяется при компиляции времени исполнения (Just In Time, JIT). Гипервизор подменяет фрагменты ВП оптимизированными аналогами, построенными с использованием собранной статистики [73].

3.7 Разработка моделей для САПР

В данном разделе будут приведены примеры моделей элементов многоуровневых ВСС, часть из которых были использованы при построении САПР (подробнее в глава 5).

3.7.1 Моделирование конфигурации: ассемблер

Рассмотрим пример формирования модели для языка ассемблер в рамках категории $\mathcal{ML}$ (Таблица 2). Для ассемблера структурной компонентой является либо мнемоника процессора, либо оператор языка (например, метка).

Таблица 2 – Фрагмент категории $\mathcal{ML}$ для ассемблера

№	Объекты	Морфизмы	Протоморфизмы
1.	[]	$([], jmp\ L)$	$jmp\ L$
2.	$[jmp\ L]$	$([jmp\ L], L:)$	$L:$
3.	$[jmp\ L, L:]$	$([jmp\ L, L:], nop)$	nop
4.	$[jmp\ L, L:, nop]$	$([jmp\ L, L:, nop], jmp\ L)$	$jmp\ L$
5.	$[jmp\ L, L:, nop, jmp\ L]$	…	…

Аналогичная программа для категории $\mathcal{M}$ будет иметь иную структуру (Таблица 3), так как часть морфизмов являются семантически некорректными. Например, морфизм номер один $([], jmp\ L)$, так как присутствует ссылка на неопределённую метку. В то же время, морфизм, соответствующий композиции первого и второго, может быть определён: $([], V(L:.\ jmp\ L))$, так как порождает корректную модель. Таким образом, аналогичная модель в рамках $\mathcal{M}$, будет иметь следующий вид:

Таблица 3 – Фрагмент категории $\mathcal{M}$ для ассемблера

№	Объекты	Морфизмы	Протоморфизмы
1.	[]	$([], V(L:\circ\ jmp\ L))$	$V(L:\circ\ jmp\ L)$
2.	$[jmp\ L, L:]$	$([jmp\ L, L:], V(nop))$	$V(L:)$
3.	$[jmp\ L, L:, nop]$	$([jmp\ L, L:, nop], V(jmp\ L))$	$V(nop)$
4.	$[jmp\ L, L:, nop, jmp\ L]$	…	…

Для разработчика, категории отличаются гранулярностью. $\mathcal{ML}$ представляет гранулярность текстового редактора с подсветкой синтаксиса, в то время как $\mathcal{M}$ – представляет работу с абстрактным синтаксическим деревом, аналогично инструментам класса Language Workbench [81].

Полученные модели обладают следующими свойствами:

1) Соответствие последовательности морфизмов и фактической (не моделируемой) структуры объекта. Позволяет строить EDSL, в которых морфизмы могут соответствовать функциям базового языка, а их композиция – композиции

функций. В функциональных языках программирования это даёт простой способ описания процесса формирования модели.

2) Возможность работы с протоморфизмами, безотносительно контекста их применения. Это позволяет работать с частичными определениями в рамках базового языка программирования, и, следовательно, использовать его как развитый препроцессор и производить частичную трансляцию $\mathcal{M}$ (подробнее в следующих разделах). Отнесение протоморфизмов к одной из представленных категорий позволяет работать как с модулями в обычном понимании (законченный функциональный компонент, зафиксированный для повторного использования), так и с автоматизированным применением шаблонов проектирования. Пример приведён ниже, `mov` – подстановка аргументов в вызываемые операции, `drop_regs` – формирование метки, по переходу в которую будет сбрасываться значение в указанном списке адресов.

```
mov trg src = do
    add src (0 :: Int)
    sacc trg

drop_regs regs = do
    l <- mark LabelMType
    drop regs
    return l
    where
        drop [] = return ()
        drop (reg:regs) = return ()
        add reg (0 :: Int)
        drop regs
```

Практический пример, использующий данную модель приведён в подразделе 5.3.4.

3.7.2 Моделирование конфигурации: структурные схемы

Процедура моделирования применима и к языкам, использующим иные модели вычислений. Например, модель для описания структуры электронных компонент (Таблица 4). Модель, приведённая ниже, использует Verilog-подобный синтаксис и описывает RS триггер в модели вычислений дискретных событий.

Состав протоморфизмов категории $\mathcal{M}$, в данном примере, будет таким же, ввиду строгого соблюдения последовательности объявления и использования идентификаторов.

Можно видеть, что описанная процедура моделирования является универсальной и позволяет в рамках функциональных языков программирования строить выразительные EDSL. Также, описанная модель может строиться автоматически, на базе предоставленных

спецификаций. Важной особенностью является отображение в рамках модели структуры конфигурации, что позволяет осуществлять детальную трассировку взаимосвязей.

Таблица 4 – Фрагмент категории $\mathcal{ML}$ для структурных схем

№	Объекты	Протоморфизмы
1.	[]	$input\ R, S;$
2.	$[input\ R, S;]$	$output\ Q, nQ;$
3.	$[input\ R, S; output\ Q, nQ;]$	$nor(Q, R, nQ);$
4.	$[input\ R, S; output\ Q, nQ; nor(Q, R, nQ);]$	$nor(nQ, S, Q);$
5.	$[input\ R, S; output\ Q, nQ; nor(Q, R, nQ); nor(nQ, S, Q);]$	…

3.7.3 Моделирование вычислительного процесса фон Неймановского процессора

Одной из ключевых особенностей фон Неймановских процессоров является поток команд, изменяющих состояние вычислителя. С точки зрения модели вычислений, команды выполняются последовательно, согласно исходному коду. С точки зрения реального вычислителя, это, как правило, не соответствует действительности, так как большинство современных вычислителей являются конвейерными и/или суперскалярными.

В рамках данного примера произведём моделирование процесса в трёх вариантах:

1) Модель ВП с точки зрения модели вычислений.
2) Модель ВП классического RISC процессора с конвейерной организацией и без параллелизма.
3) Модель ВП классического RISC процессора с конвейерной организацией и параллелизмом.

Схема процессора, для которого строится модель, приведена на Рисунок 50. Данный процессор имеет конвейер с пятью стадиями: выборка инструкций (IF, if – Instruction Fetch), декодирование инструкции (ID, id – Instruction Decode), выполнение операции (EX, ex – Execute), доступ к памяти (MEM, mem – Memory access), запись в регистры (WB, wb – Register write back). В процессоре отсутствуют механизмы предсказания переходов. Переходы обрабатываются следующим образом:

1) При достижении команды перехода стадии EX, производится подмена команд на стадиях IF и ID на пузырёк (o – bubble).
2) При достижении команды перехода стадии MEM, происходит установка требуемого значения в регистр команды, в блок IF загружается пузырёк.

3) На следующем шаге конвейера производится выборка команды по адресу их регистра команды.

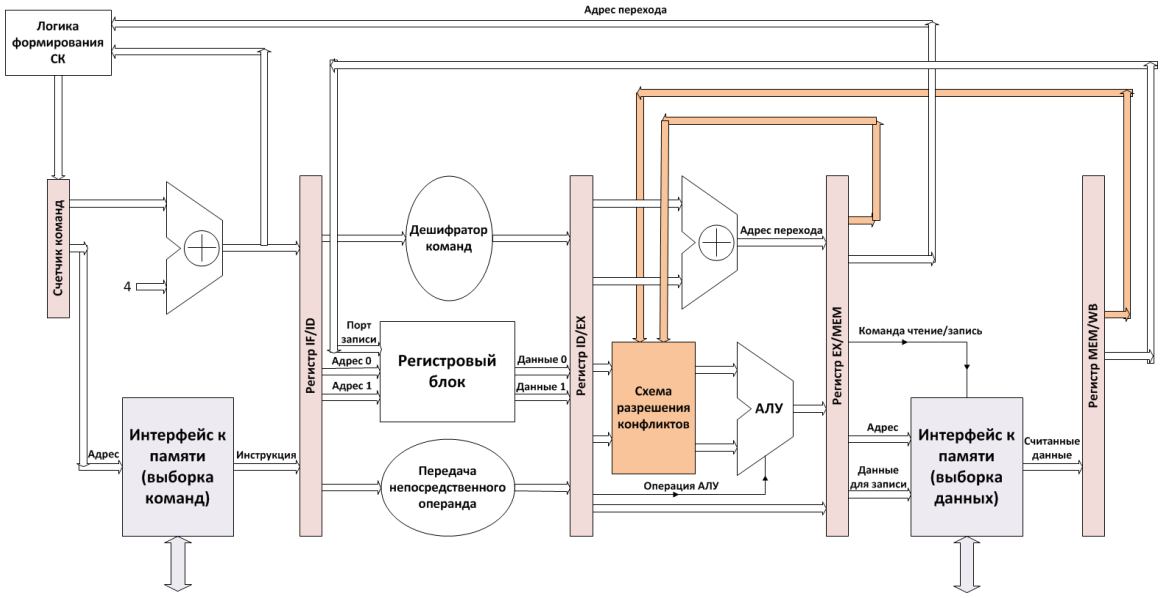


Рисунок 50 – Схема традиционного RISC процессора

Для примера рассматривается следующая модель конфигурации:

```
1: jmp L
2: L:
3: nop
4: jmp L
5: nop
6: nop
```

Как отмечалось ранее, с точки зрения модели вычислений, фон Неймановские процессоры поочерёдно выполняют команды. На Рисунок 51 приведена модель ВП, где объекты (A, B, C…) – состояния вычислителя, а морфизмы – команды, выполняемые процессором. В наименованиях морфизмов указываются номера строк из конфигурации.

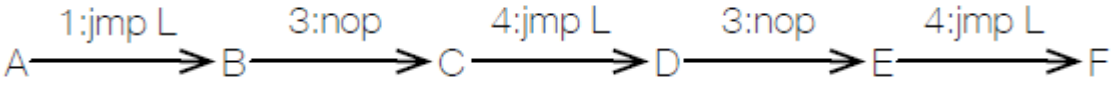


Рисунок 51 – Модель вычислительного процесса с точки зрения модели вычислений

С целью приближения модели ВП к реальности, произведём её детализацию. Для этого возьмём традиционные этапы конвейера RISC процессора и детализируем шаги ВП (без учёта параллелизма). Модель примет вид Рисунок 52. Для неё модель Рисунок 51 является подкатегорией. Состояния ВП, а также морфизмы и композитные морфизмы могут однозначно сопоставляться друг другу. Как можно видеть, отличие этих моделей заключается только в том, насколько детализировано состояние вычислителя.

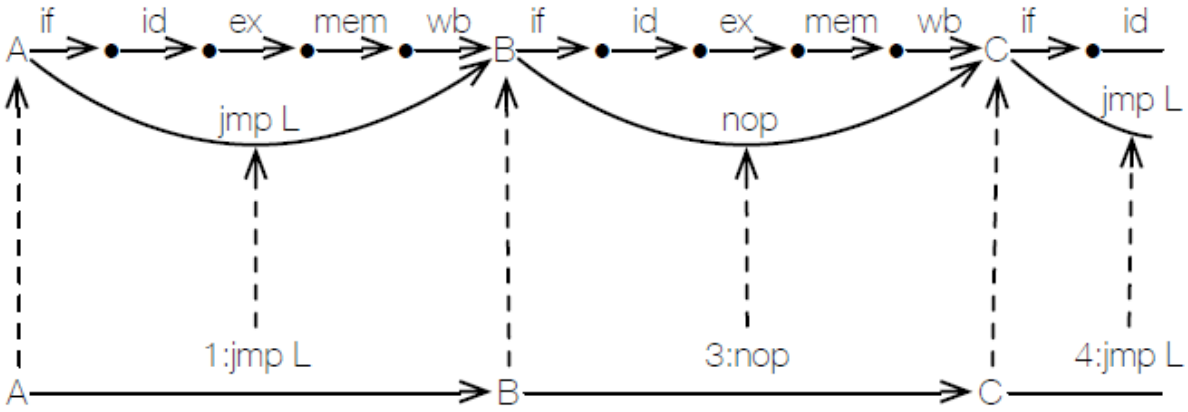


Рисунок 52 – Модель вычислительного процесса классического RISC процессора с конвейерной организацией (без параллелизма)

Реальный ВП складывается из пяти параллельных процессов, в соответствии с числом стадий конвейера. С точки зрения модели вычислений, он сохраняет морфизмы, соответствующие командам вычислителя. Поэтому на схемах будут указываться морфизмы, соответствующие этапам конвейера (чёрным цветом) и морфизмы, соответствующие выполняемым командам (выделены цветом). Морфизмы, соответствующие выполненным командам, обозначены стрелками. Морфизмы, соответствующие прерванным командам – линиями с крестом. Важно, данная модель не включает в себя модели Рисунок 51 и Рисунок 52, так как команды пересекаться во времени, а часть из них прерывается. Способ визуализации модели не является однозначным, что вызвано обозначенной двойственностью. В связи с этим будет приведено две схемы:

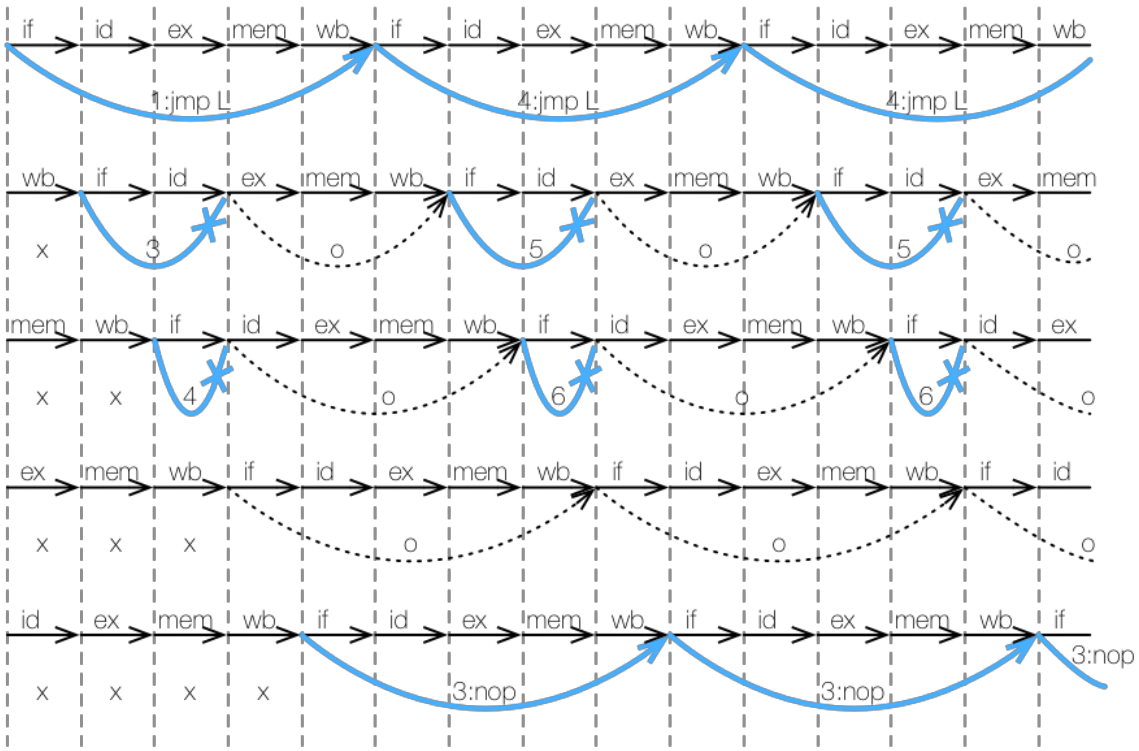


Рисунок 53 – Модель вычислительного процесса классического RISC процессора с конвейерной организацией (с параллелизмом, точка зрения - поток команд)

– схема, ориентированная на «целостное» отображение выполняемых команд (Рисунок 53), но при этом со «смещаемым окном» конвейера;
– схема, ориентированная на отображение этапов конвейера, со смещающимися между ними командами (Рисунок 54).

Данные схемы эквивалентны с точки зрения модели, но визуальные отличия делают первую более удобной для анализа ПО, вторую для анализа процессора.

Как можно видеть, предложенная процедура моделирования подходит для описания ВП со сложной структурой.

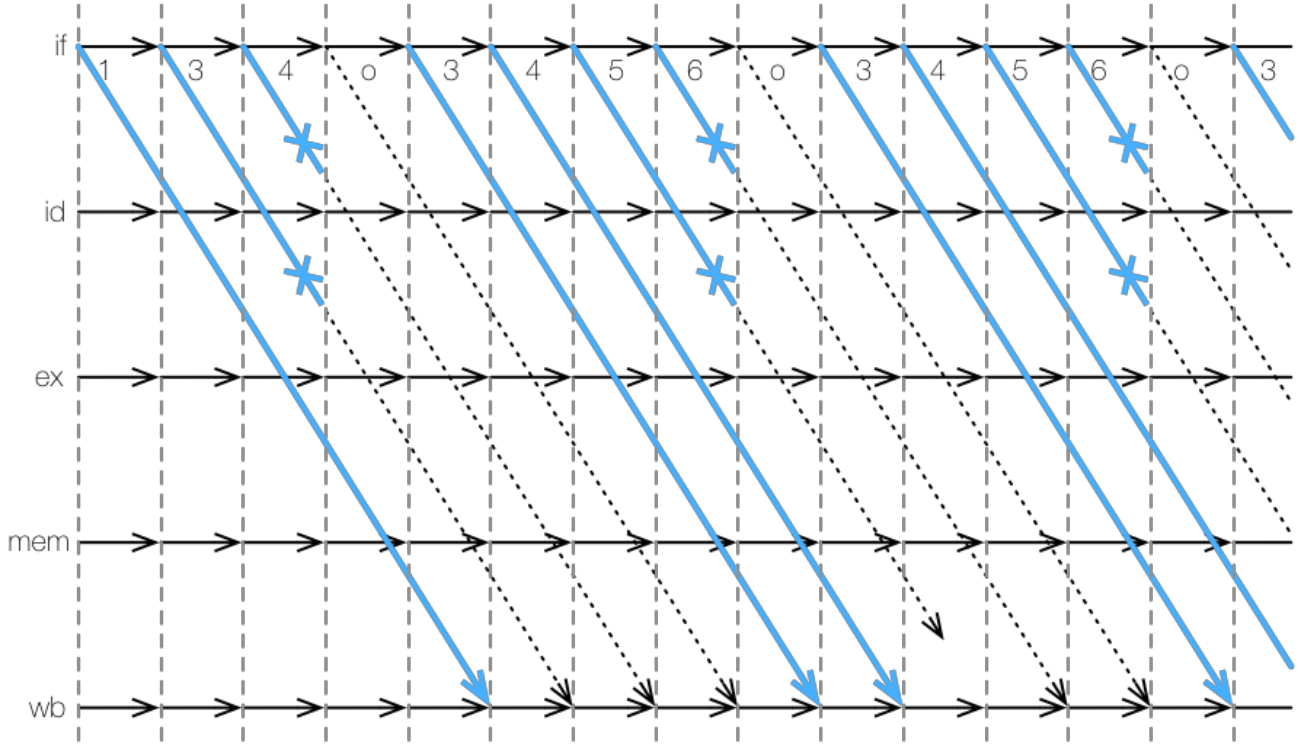


Рисунок 54 – Модель вычислительного процесса классического RISC процессора с конвейерной организацией (с параллелизмом, точка зрения – стадии конвейера)

3.7.4 Моделирование отношения актуализации и виртуализации

В данном примере будет рассмотрено построение отношения актуализации и виртуализации, согласно варианту один, изложенному в разделе 3.6. В примере будет рассмотрена реализация симулятора для фон Неймановского процессора. Для реализации модели выбран язык Haskell.

3.7.4.1 Описание моделируемого процессора

На Рисунок 55 представлена структурная схема моделируемого процессора.

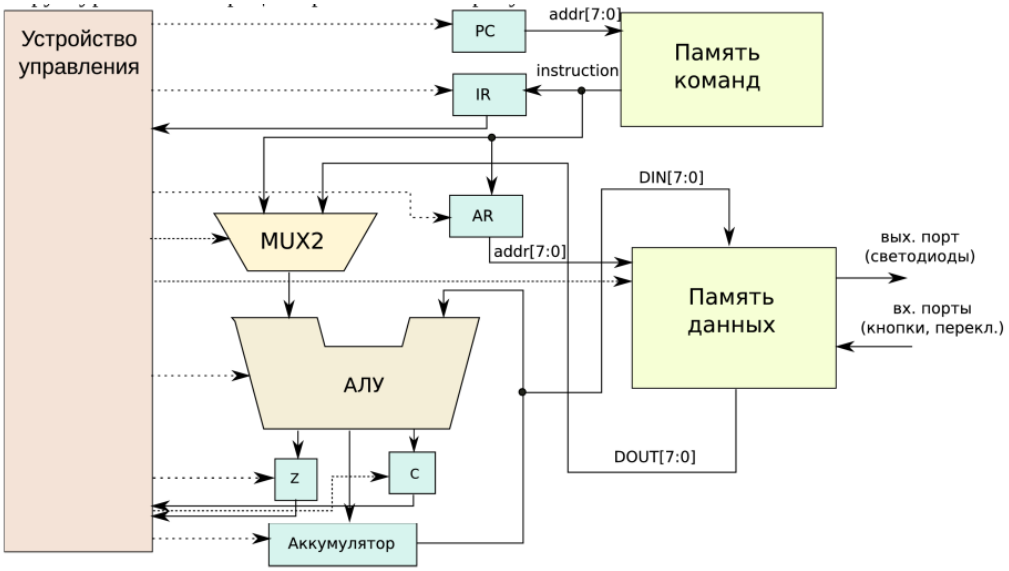


Рисунок 55 – Структурная схема процессора

Сплошной линией обозначены линии данных. Пунктиром обозначены управляющие сигналы. Разрядность процессора: 8 бит. Организация памяти: гарвардская, с отдельными блоками памяти для команд и данных. Подсистема обработки прерываний и команды вызова подпрограмм отсутствует.

Внешние устройства (для данной работы – это светодиоды, двухпозиционные переключатели и кнопки) отображаются в адресное пространство данных. Работа с ними выполняется по опросу.

Команды выполняются за 2 или 3 такта (в зависимости от типа команды):

1) Выборка команды.
2) Выборка операндов.
3) Выполнение команды.

Регистры:

– PC — счетчик команд;
– IR — регистр инструкций;
– AR — регистр адреса операнда;
– C — флаг переноса;
– Z — флаг нуля.

3.7.4.2 Моделирование отношений

Согласно процедуре моделирования ВП, описание отдельных команды можно представить в виде композиции атомарных функции, соответствующих набору ВМХ процессора

(примеры ВМХ: пересылка значения между регистрами, выполнение конкретной операции). Композиция функций с учётом параллелизма позволяет иметь потактовую модель процессора, поведение которой описано в терминах ВМХ. Первый такт любой команды представляет из себя её выборку из памяти (сигнала для инкриминирования регистра PC и чтения команды в регистр IR). Это может быть записано следующим образом:

```
nextCommand = incPC . pmem2IR
```

Модель вычислений языка Haskell сильно отличается от модели синхронных потоков данных (SDF [54]), используемой в языке Verilog, что требует дополнительного внимания к последовательности операций при композиции функций. С использованием данного подхода система команд может быть описана в следующим образом:

```
cmd = [(0x01, Nop,  NoneArg,  [nop]),                       -- nope
       (0x00, Hlt,  NoneArg,  [hlt]),                       -- halt model
       (0x02, Ldm,  DmemArg,  [incPC . pmem2AR, dmem2acc]), -- acc = dmem[arg]; c = const;
z = (dmem[arg] == 0)
       (0x03, Ldi,  ValueArg, [incPC . pmem2acc]),          -- acc = arg; c = const; z =
(arg == 0)
     ...
```

В данном случае важен только первый (код команды) и последний (список функций) элемент кортежа, остальные используются для трансляции. Каждая функция из списка выполняется за один модельный такт. Значительная часть функций задаётся композицией микрокоманд.

Симуляция ВП производится следующей функцией:

```
step (State {on = False}) = return ()
step state = step $ executeCommand state' commandFlow
    where
      state'@(State{ir = hexCommand}) = nextCommand state
      commandFlow = commandFlowFromInfo $ getCommandInfo hexCommand
      executeCommand state [] = state
      executeCommand state (f:fs) = executeCommand (f state) fs
```

Отслеживание отношения виртуализации производится тривиальным образом и сохраняется в рамках значения `hexCommand`. В приведённом примере данное значение никак не экспортируется из функции.

3.8 Выводы

1) Проведён анализ подходов к формализации архитектурного стиля «модель-процесс-вычислитель» для решения задач моделирования многоуровневых встроенных систем и их элементов при разработке САПР с функциями совместной симуляции, верификации, отладки и профилирования.

2) Путём формализации архитектурного стиля «модель-процесс-вычислитель» с использованием математического аппарата теории категорий, предложена методика моделирования многоуровневых встроенных систем и их элементов, включающая следующие процедуры:

   – процедура моделирования конфигураций вычислительного процесса;
   – процедура моделирования целостного вычислительного процесса на всех уровнях встроенный системы;
   – процедура моделирования отношения актуализации;
   – процедура моделирования отношения трансляции;
   – процедура моделирования отношения виртуализации.

3) Даны рекомендации по проектированию компонентов САПР на основе моделей многоуровневых встроенных систем и их элементов, получаемых с использованием предложенной методики моделирования.
4) Разработан набор моделей элементов многоуровневых встроенных систем с использованием предложенной методики моделирования. Часть из разработанных моделей была использована при разработке САПР, что позволило сократить затраты на их создание. Набор включает в себя модели: языка макроассемблера, языка структурных схем, вычислительного процесса фон Неймановского процессора в разных режимах работы, тактовые модели актуализации и виртуализации, модели специализированного сигнального процессора.

4 Разработка методики проектирования многоуровневых встроенных систем

В этой главе будет уточнена используемая система архитектурных абстракций, введён новый тип объектов повторного использования уровневой организации и предложено расширение методики проектирования ВСС, ориентированное на разработку многоуровневых ВСС и их элементов.

4.1 Архитектурные абстракции объектов повторного использования

Для выделения объектов повторного использования уровневой организации ВСС, необходимо разделить прикладную и системную составляющую. К системной части относятся те элементы уровневой организации, в рамках которых задаётся целевая функция системы. Например, интерпретатор языка программирования при разработке «скриптов», САПР и ПЛИС при разработке на языке Verilog. Модели целевого ВП относятся к прикладной составляющей. Важно отметить, что прикладная составляющая:

- необязательно сосредоточена в рамках одного элемента уровневой организации;
- необязательно находится на верхних уровнях организации ВСС (это место, как правило, занимают пользовательские настройки и конфигурации).

Основным элементом повторного использования уровневой организации ВСС являются ВПЛ и СПЛ. Далее будет обобщено понятие СПЛ в соответствие с актуальными тенденциями в области проектирования ВСС.

4.1.1 Обобщённое понятие системной платформы

Системная платформа, согласно платформно-ориентированному проектированию [11,12] – совокупность программной и аппаратной составляющей. Она позволяет разработчику реализовать прикладную функциональность системы. Подразумевается, что системная платформа – гомогенный объект, но, как показано в подразделе 1.1.3, современные ВСС обладают гетерогенной структурой, в которой, конфигурирование производится на нескольких уровнях. В связи с этим, необходимо расширить определение системной платформы для того, чтобы включить в него полную совокупность ВПЛ, используемых при разработке ВСС, а также привести его в соответствие общему определению ВПЛ:

Системная платформа (СПЛ) – совокупность ВПЛ, используемая при разработке и эксплуатации заданного класса ВСС, а также включающая методологические и инструментальные средства разработки.

СПЛ включает в себя целевые и служебные уровни. *Целевыми* называются уровни, с которыми работает разработчик, а *служебными* – те, что необходимы для обеспечения ВП. К примеру, разработка программы на языке С для операционной системы Linux. В качестве целевых уровней выступают операционная система и язык программирования, а служебными являются машинный код и аппаратная платформа, на которой запускается ОС.

Элементы, входящие в состав СПЛ, схематически изображены на Рисунок 56.

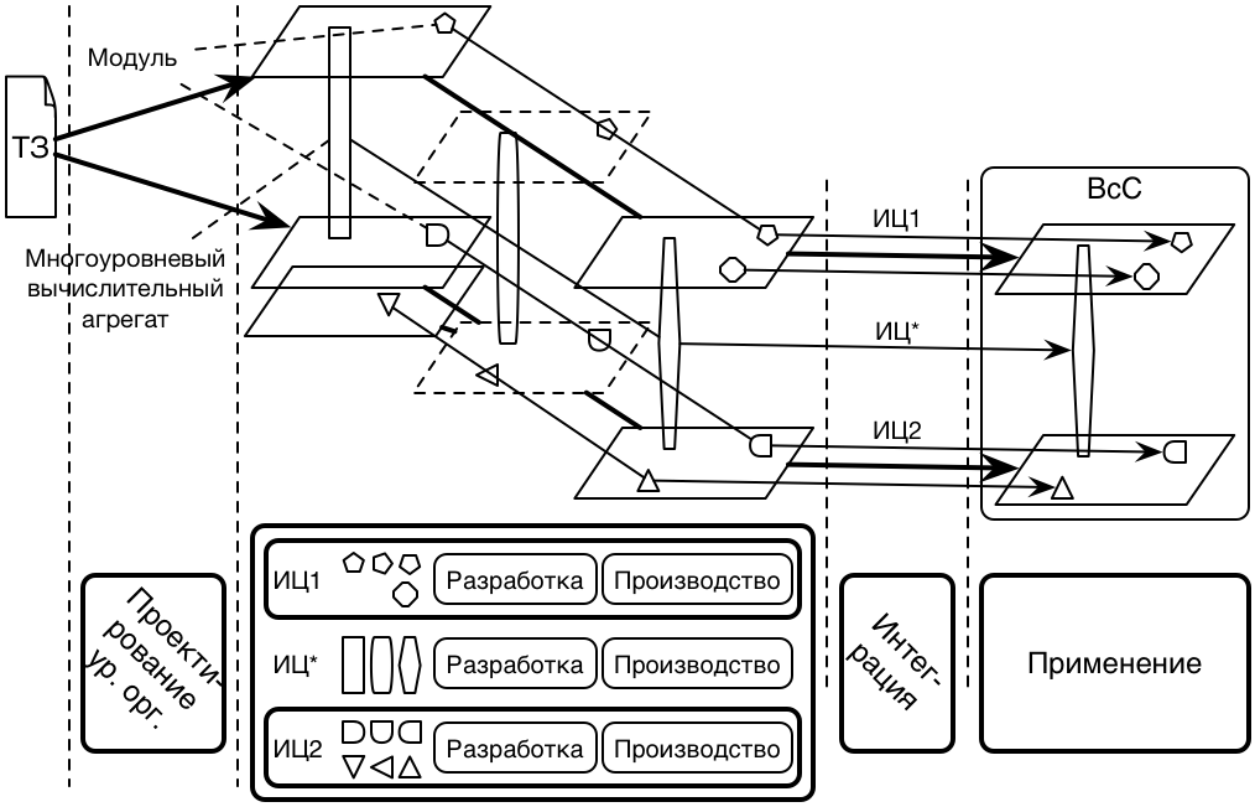


Рисунок 56 – Элементы системной платформы и жизненный цикл встроенной системы

Данная схема соответствует традиционному маршруту проектирования ВСС, за тем исключением, что число уровней не ограничено. Именуемый «многоуровневым вычислительным агрегатом» объект будет определён позднее.

4.1.2 Понятие многоуровневого вычислительного агрегата

Термин «многоуровневый агрегат» был введён с целью адекватного представления современных тенденций в контексте проектирования системных платформ. Он является необходимым для представления многих реконфигурируемых вычислителей и совместных виртуальных машин, использование которых требует работы с группой вычислительных

платформ (минимум – базовая вычислительная платформа), и формируется с целью подчеркнуть сильную связанность конфигураций разных уровней.

Многоуровневый вычислительный агрегат (МВА) – целостный объект повторного использования для решения фиксированного класса вычислительных задач, включаемый в состав СПЛ и требующего конфигурирования нескольких ВПЛ.

Данный термин основывается на понятии «архитектурного агрегата» из области высокоуровневого проектирования (HLD-методология) и используется для обозначения повторно используемых элементов вычислительных систем. Границы МВА определяются с точки зрения функционального назначения, а не с точки зрения внутренней организации.

Многоуровневый агрегат является СПЛ в миниатюре, зафиксированной для повторного использования группой проектов. Основное отличие от СПЛ заключается в том, что он (1) не полон и (2) ориентирован на более узкий класс вычислительных задач.

Зачастую использование МВА без поддержки со стороны САПР значительно затруднена [6]. Выделяются две группы инструментов: средства этапа разработки и средства, интегрируемые в целевую систему.

4.1.2.1 Структура и состав многоуровневого вычислительного агрегата

Многоуровневый агрегат является целостным объектом повторного использования уровневой организации и включает:

1) Совокупность элементов, входящих в состав целевой вычислительной системы и отвечающих за непосредственную актуализацию ВП. Они реализуются с помощью различных ВПЛ, в том числе и заказных. Конфигурации данных элементов задаются разработчиком целевой системы и не входят в состав МВА.
2) САПР. Он обеспечивает эффективное повторное использования МВА и, как правило, включает в себя компиляторы, отладчики, средства профилирования и совместного моделирования.
3) Методологическая составляющая, регламентирующая правила использования многоуровневого агрегата.

Для разработки элементов МВА, входящих в состав целевой системы, актуальны проблемы создания многоуровневых ВСС. Особое место занимает вопрос совместной разработки, решение которого должно быть предоставлено пользователю МВА. Классическим методом обеспечения совместной разработки является использование моделей [1], в которых

детально фиксируется интерфейс взаимодействия между уровнями. Это позволяет организовать независимую разработку отдельных уровней и снизить интеграционные риски.

Разработка САПР обособлена от элементов в составе целевой системы, так как может производится в рамках произвольной инструментальной платформы. Это позволяет утверждать, что разработка инструментальных средств происходит в более благоприятных условиях, чем разработка других элементов МВА. В тоже время, реализация САПР для МВА является сложной задачей, требующей высокой квалификации исполнителя. Это связано с повышенными требованиями к качеству результата (необходимость повторного использования, сложность отладки, повторяемость ошибок) и специфическим характером задачи. САПР должен включать инструменты следующих видов:

1) Средства работы с моделями: редакторы, IDE, «lint» инструменты (pep8, Eclipse).
2) Средства преобразования моделей: компиляторы, синтезаторы, генераторы исполняемого кода (Clang, GCC, Xilinx ISE).
3) Средства анализа вычислительного процесса и построение моделей на его основе: системы профилирования, системы ведения журнала (Muppy, Logger).
4) Средства отладки преобразованных моделей (GDB).
5) Симуляторы и эмуляторы, позволяющие актуализировать вычислительный процесс по модели при отсутствии целевой системы в рамках отдельной ВПЛ (GEM5 [91], Icarus Verilog).
6) Средства совместной симуляции, позволяющие симулировать поведение ВСС, представленных в рамках нескольких ВПЛ (Proteus [55], Ptolemy [54]).

Методологическая составляющая должна разрабатываться при проектирование уровневой организации МВА с помощью стиля МПВ. Она должна фиксировать на архитектурном уровне правила использования МВА.

Разработка моделей МВА при помощи предложенной методики моделирования многоуровневых ВСС и их элементов (глава 3) позволяет:

1) Включить в процесс моделирования МВА инструментальные средства. Необходимо для проработки вопросов совместной разработки.
2) Упростить процесс совместного профилирования и отладки МВА. Актуально как с точки зрения проектирования МВА, так и с точки зрения использования.

4.1.2.2 Пример документирования многоуровневого вычислительного агрегата

Документирование МВА требует цельного рассмотрения, с отображением точек конфигурирования и их функционального назначения. В данном примере будет рассмотрена уровневая организации МВА, являющегося виртуальной машиной `pmvm` (pattern matching virtual machine, исходный текст приведён в приложении Б.2), используемой для управления БПЛА (5.3.3).

Виртуальная машина `pmvm` реализована на языке C в рамках стандарта C99. Она позволяет описывать «ловушки» для событий на языке предикатов, реакцию на них, представленную в виде обновления значений пользовательских регистров и вызова функций обработчиков. Для использования `pmvm` необходимо конфигурирование предикатов и обработчиков на языке C, а также конфигурирование `pmvm-interpretator`, заключающееся в связке реализаций предикатов и обработчиков с их текстовыми именами. Ниже приведена выдержка из пользовательской конфигурации.

```
{
   defaults: {
       gps_step: 0,
       priority: 42,
   },
   rules:
      [
        {
           events: [COMPLETE],
           update: {priority: 0},
        },
        {
           events: [GPS1],
           guard: [&& [=, gps_step, 0], [<, priority, 10]],
           update: {gps_step: 1},
           signal: 1,
        },
        {
           events: [GPS1],
           guard: [=, gps_step, 1],
           update: {gps_step: 0},
           signal: 2,
        },
        {
           events: [GPS1],
           guard: [>=, priority, 1],
        },
      ]
}
```

На Рисунок 57 приведена уровневая организация `pmvm`, отображающая три уровня МВА (снизу-вверх): C, `pmvm-interpretator` *(служебные)* и *экземпляр* `pmvm` *(прикладной)*.

Данная схема позволяет отобразить все уровни модуля, продемонстрировать сквозной характер конфигурации и функциональную зависимость между ВМХ. Также она позволяет описать оптимизацию производительности программы за счёт переноса части кода на более

эффективную ВПЛ, а именно, переноса пользовательского предиката *UP* на платформу C, в качестве *BMXUP*. Показать методологическую составляющую МВА, за счёт описания мест конфигурирования.

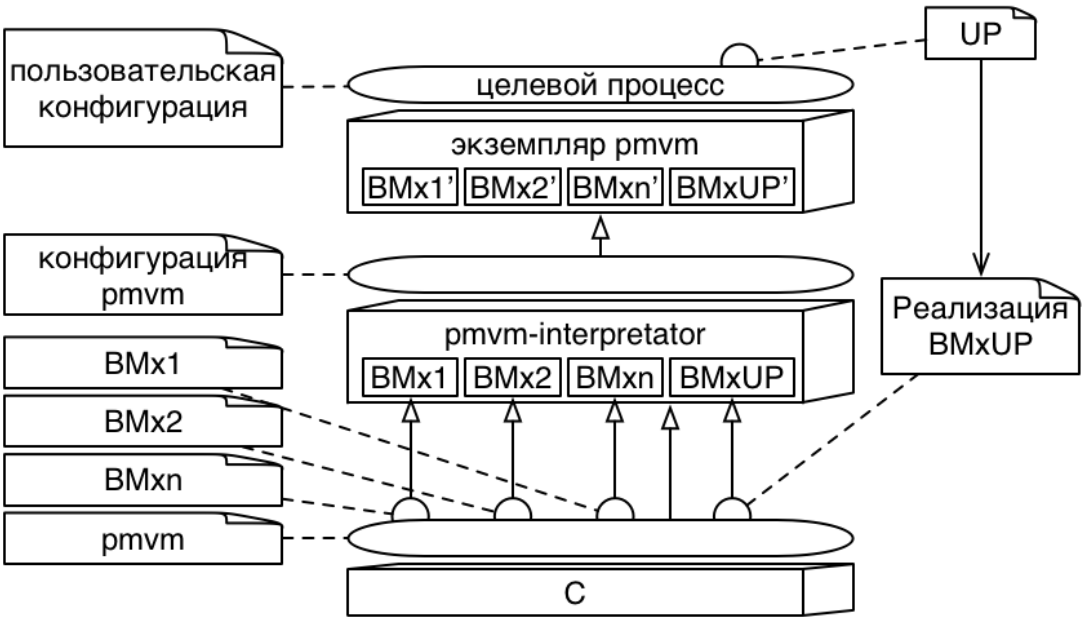


Рисунок 57 – Уровневая организация модуля `pmvm`

4.2 Унифицированная модель [pe]конфигурации

Процесс конфигурирования ВПЛ именуется по-разному: для программного обеспечения используется термин программирование; для ПЛИС и реконфигурируемых вычислителей – синтез, конфигурация или реконфигурация; для СБИС, контроллеров, одно- и много-платных компьютеров — производство или сборка; на прикладном уровне часто используется термин настройка.

Как можно видеть на Рисунок 58 (выполненном средствами системно-иерархического архитектурного стиля, раздел 2.2), процесс конфигурирования заключается в изменении функциональности системы путём замены её модулей. Данная схема справедливой для любого из рассмотренных видов конфигурирования, независимо от момента времени или типа ВПЛ.

Для эффективной работы с ППР необходимо унифицированное представление описанного выше процесса. Будем именовать его процессом [pe]конфигурации или обобщённой реконфигурацией. За основу взято понятие реконфигурации в виду крайне широкого спектра трактовок в литературе [25,26,65,104–106].

Унифицированная модель [pe]конфигурации должна описать все значимые элементы уровня СПЛ и с точки зрения структурны, и с точки зрения ВП. Необходимость обоих точек

зрения обусловлена невозможностью корректного представления процесса [ре]конфигурации без учёта целевой ВСС и её окружения. К структурным элементам относятся:

1) *Базовая ВПЛ* (или совокупность ВПЛ, как в случае с LabView [107], использующей операционную систему реального времени и ПЛИС).
2) *Целевой вычислитель*, сформированный в рамках базовой ВПЛ и отвечающий заданным требованиям. Количество целевых вычислителей на одном уровне определяется числом [ре]конфигураций. Может явным образом присутствовать в целевой системе (для процессора или виртуальной машины), а может присутствовать только на этапе разработки / производства (для транслируемых ВПЛ). Во втором случае целевой вычислитель будет виртуальным.
3) *Конфигурация* базовой ВПЛ, определяющая целевой вычислитель. К примеру, программа на языке высокого уровня, конфигурация ПЛИС или монтажная схема печатного узла.
4) *Механизм [ре]конфигурирования*, осуществлявший процесс [ре]конфигурации базовой ВПЛ. Может входить в состав целевой системы (загрузчик), в рабочее окружение (программатор) или в состав обеспечивающих систем (сервисный центр).

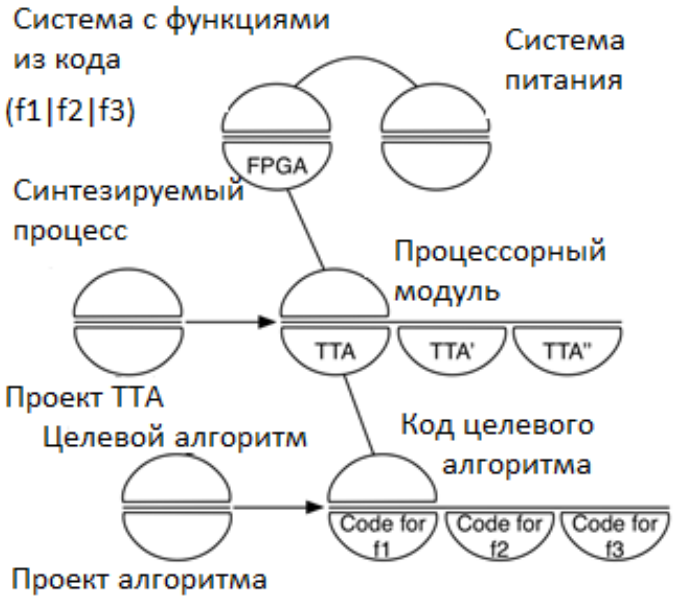


Рисунок 58 – Изменение функциональности путём изменения подсистемы

Структурные элементы обеспечивают элементы процесса [ре]конфигурации:

1) *Процесс [ре]конфигурирования* (ПРК).
2) *Целевой вычислительный процесс* (ЦВП).
3) *Базовый вычислительный процесс* (БВП).

4) Совокупность процессов, связанная с рассматриваемым уровнем или просто – *процесс*.

На Рисунок 59 приведено схематическое изображение унифицированной модели [ре]конфигурации, в рамках которого формируется два целевых вычислителя.

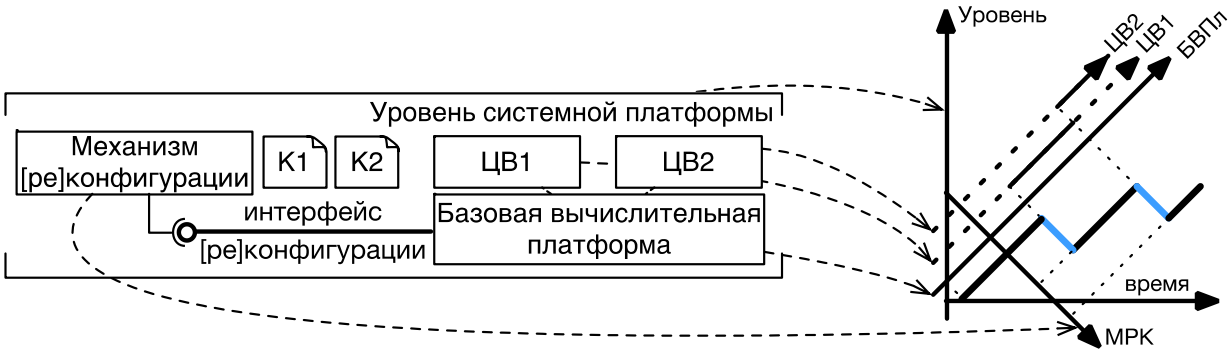


Рисунок 59 – Унифицированная модель [ре]конфигурации для отдельного уровня

Структурная точка зрения показана в нотации, близкой к нотации «Component Diagram» стандарта UML [40]. Используются следующие обозначения: вычислители и механизмы обозначены прямоугольниками; конфигурация обозначена значком документа; интерфейс обозначен в соответствии с «Component Diagram»; границы уровня обозначены параллельными горизонтальными чертами. Конфигурации (К1, К2) соответствуют целевым вычислителям (ЦВ1, ЦВ2).

Важно отметить, что наличие целевого вычислителя определяется не существованием структурного элемента (как в случае одноплатного компьютера), а процессом наблюдения за целевым вычислителем и знанием наблюдателя о его структуре. К примеру – СБИС без маркировки может быть, как аппаратным решением, так и процессором с ПО. Виртуальная машина Java не является физическим объектом, но при этом её вычислительный процесс может быть представлен и проанализирован.

Точка зрения ВП показана графиком, где процесс визуализируется как функция состояния от времени. Ось времени и ось состояний показаны как ось абсцисс и ординат соответственно. Каждое новое состояние считается уникальным, поэтому функция – монотонно возрастающая (что не соблюдается для оси «уровень» с точки зрения изображения, но соответствует сути описываемого явления). Диагональные оси соответствуют состояниям структурных элементов уровня (их проекциям): механизма [ре]конфигурирования (ПРК), базовой вычислительной платформы (БВПЛ) и целевых вычислителей (ЦВ1 и ЦВ2). Тонкие линии – разметка, средние – оси, толстые – вычислительный процесс. Оси, соответствующие проекции на целевые вычислители, частично показаны пунктирной линией, частично –

сплошной, что говорит о невозможности определить состояния до того, как конфигурирования вычислителя. Важно отметить, что ортогональность оси процесса конфигурирования и оси базового вычислительного процесса – это допущение пассивной роли последней в [ре]конфигурации, сделанное для упрощения схемы. В реальной вычислительной системе базовая ВПЛ может принимать активное участие в процессе [ре]конфигурации (например, процесс программирования микроконтроллера). Пунктирными дугами показаны взаимосвязи между точками зрения.

Необходимо отметить следующие свойства унифицированной модели [ре]конфигурации:

1) Ортогональность (взаимная независимость) целевого ВП процессу [ре]конфигурации.
2) Параллельность целевого ВП и процесса базовой ВПЛ. Говорит о том, что ВП целевой системы является целостным процессом, имеющим множество представлений. Это формирует требования к методам и средствам совместного проектирования. Представление определяет не только структуру и гранулярность ВП, но и прикладной смысл операций. Наличие или отсутствие тех или иных представлений определяется, главным образом, информацией о том, как устроена ВСС, а не её структурой, так как последняя не содержит (в общем случае) информации о стадиях проектирования и производства.
3) Разделение целевыми вычислителями общего ресурса (базовой ВПЛ) во времени. Демонстрирует изменчивость уровневой организации ВСС во время эксплуатации. Требует рассмотрения СПЛ без привязки к конкретному моменту времени.
4) Уровень вычислительной системы является объектом, распределённым по всему жизненному циклу системы, и, как следствие, ненаблюдаемым в полном объёме в один момент времени. Это требует рассмотрения его элементов как слабосвязанных структур, где отношения формируются на основании знания и задач наблюдателя, а не по «физическим причинам». Это, в свою очередь, позволяет вводить в систему уровни по необходимости, без оснований с точки зрения её устройства. Примерами могут являться архитектурные спецификации и модели для отладки и верификации.

Помимо вышесказанного, целевой вычислитель может являться как функционально завершённым элементом, так и ВПЛ. Это позволяет нам перейти от рассмотрения отдельных уровней к ВПЛ в целом.

4.3 Модель иерархической организации системной платформы

Унифицированная модель [ре]конфигурации позволяет перейти к целостному рассмотрению СПЛ. При этом вышележащие уровни обеспечиваются нижними. Целевой вычислитель нижнего уровня выступает в роли базовой вычислительной платформы верхнего. Такие элементы далее будут называться *целевыми вычислительными платформами* (ЦВПЛ).

На Рисунок 60 приведена схема многоуровневой СПЛ. Нотация незначительно изменена по сравнению с Рисунок 59: опущено отображение конфигураций, добавлена ось уровня иерархии. *Уровень иерархии* обозначается латинскими буквами и характеризует глубину иерархии в указанный момент времени. Механизмы [ре]конфигурации могут быть как в рамках уровневой иерархии, так и вне её – это необходимо для соблюдения независимости вычислительного процесса от процесса конфигурации. Используются следующие обозначения для структурных элементов:

- для механизмов [ре]конфигурирования – МРК_L;
- для целевых вычислителей – ЦВ_CN;
- для базовых и целевых вычислительных платформ – БВПЛ_CN и ЦВПЛ_CN соответственно.

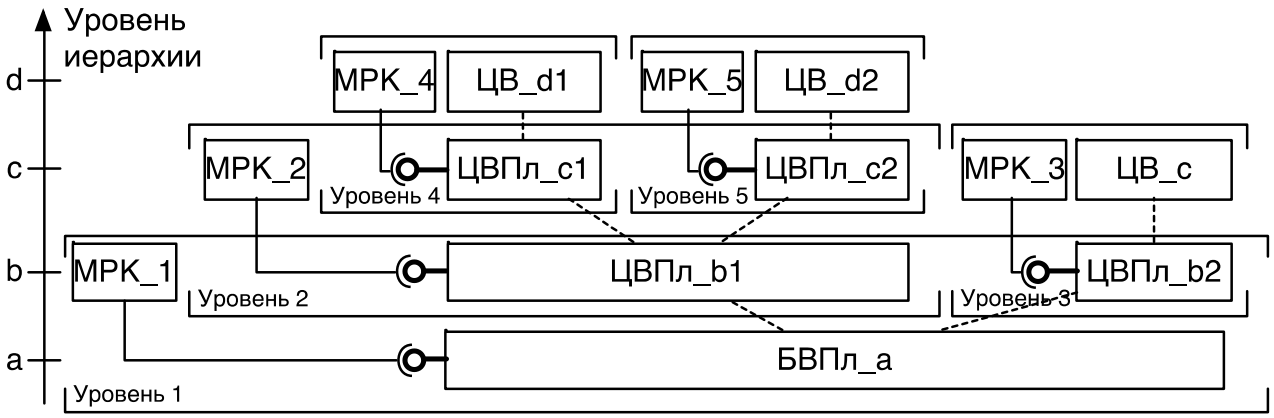


Рисунок 60 – Модель иерархической организации системной платформы

Здесь L – цифровой идентификатор уровня системной платформы, C – буквенный идентификатор уровня иерархии, N – цифровой идентификатор, используемый в случае, если элементов данного типа на уровне C более чем один.

В упрощённом виде данная схема может быть отображена в виде связанного ориентированного ациклического графа, называемого *иерархической организацией СПЛ*. Вершинами являются уровни СПЛ, рёбрами – отношение обеспечения, направленное от обеспечиваемого уровня к обеспечивающему (сверху – вниз). Это обусловлено как прикладным

причинами (трансляция высокоуровневых операций в низкоуровневые), так и практическими (избежать рассмотрения графа как ориентированного дерева). Абстракция от целевых вычислителей обусловлена необходимостью разделения целевой ВСС и СПЛ, которая может быть повторно использована без изменений в других проектах. На Рисунок 61 приведена иерархическая организация СПЛ.

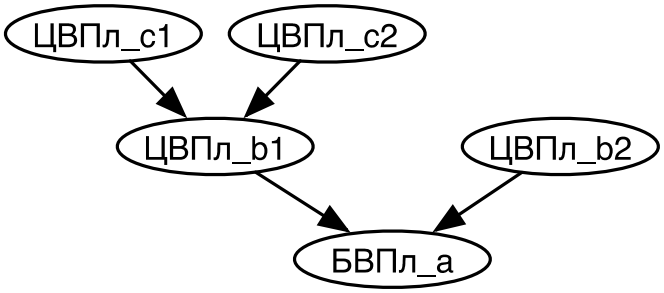


Рисунок 61 – Иерархическая организация системной платформы

Такой способ отображения позволяет значительно упростить описание структуры уровневой организации, достаточное для определения необходимых компетенций.

На Рисунок 62 показана структура ВП с рассмотренной СПЛ. Способ визуализации значительно изменён. Горизонтальная ось демонстрирует развитие процессов ВСС во времени; вертикальная ось демонстрирует структурные элементы СПЛ; горизонтальные пунктирные стрелки показывают место процессов в рамках отдельных структурных элементов (сплошными линиями показаны сами процессы); прямоугольниками со скруглёнными углами показаны границы уровней иерархии; границы уровней СПЛ обозначены также, как на структурном представлении; вертикальные пунктирные стрелки показывают формирование и разрушение целевых вычислителей и ВПЛ.

Рассмотрим подробнее особенности представленного примера:

1) Процесс [ре]конфигурации может производиться только в тот момент, когда конфигурируемая (базовая) ВПЛ наблюдаема (присутствует её ВП).
2) Процесс повторной [ре]конфигурации всегда сопряжён с уничтожением целевого вычислителя. Данный процесс на приведённой схеме не показан, хотя может требоваться в явном виде (очистка памяти). Как правило, реализуется механизмом [ре]конфигурации.
3) Все приведённые выше процессы, идут параллельно процессу [ре]конфигурации или процессу на более высоком уровне иерархии. Это является особенностью конкретного примера, так как ВПЛ может разворачивать самостоятельный процесс точно также, как и целевые вычислители.

4) На схеме все структурные элементы СПЛ показаны в единичном экземпляре. Это является особенностью конкретного примера. Количество [ре]конфигураций не ограничено.

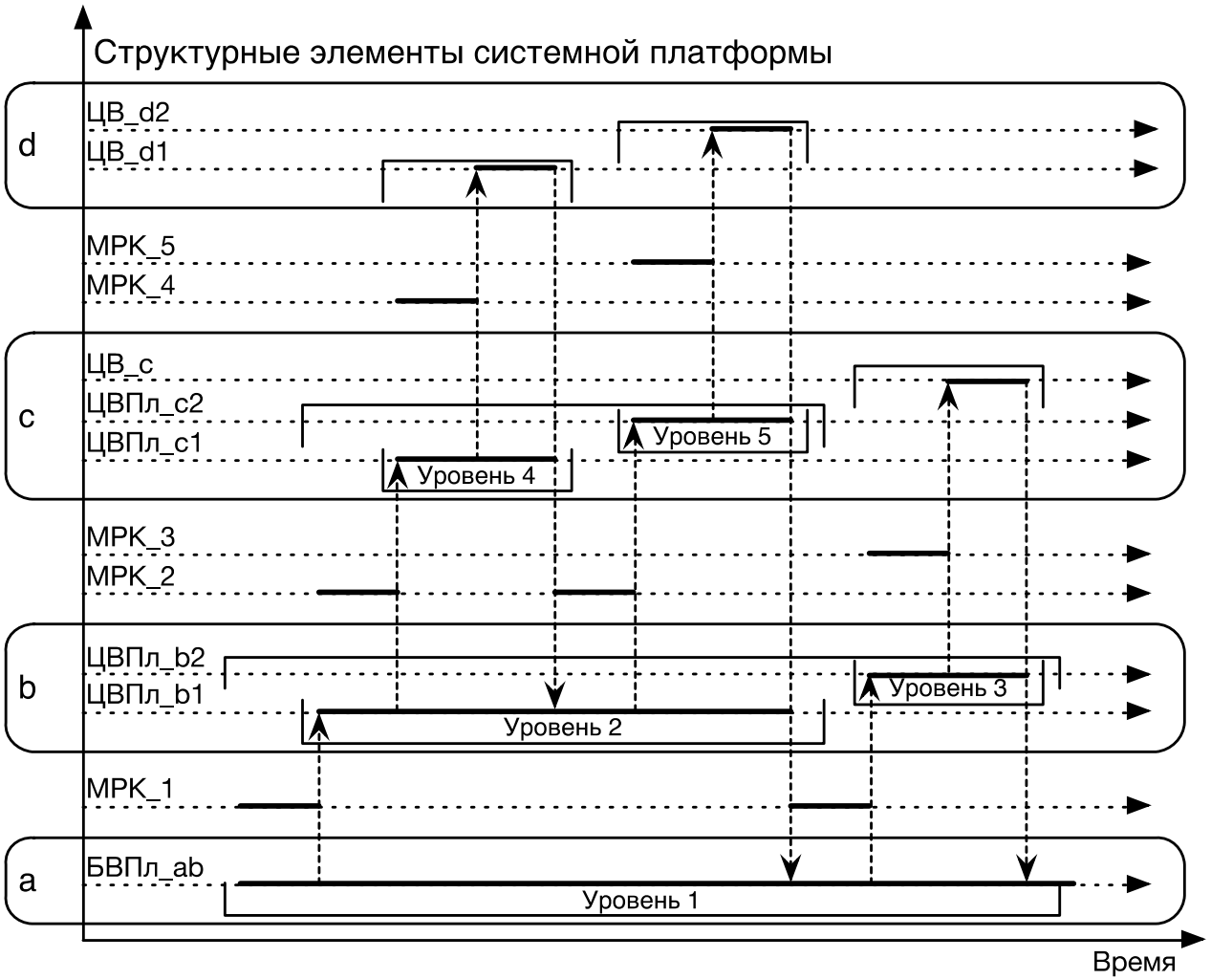


Рисунок 62 – Модель вычислительного процесса многоуровневой встроенной системы

Как можно видеть, полное представление ВП требует оперирования всеми уровнями иерархии. При этом, интерпретация процесса производится через известные неэквивалентные конфигурации, в рамках которых задана его прикладная функция. Глубина погружения в иерархию определяется особенностями конкретного проекта.

Исследование механизма формирования уровней СПЛ позволило выявить следующие свойства:

1) Количество уровней иерархии в [ре]конфигурируемых ВСС переменно. Это должно учитываться как при архитектурном проектировании СПЛ (абстракция от конкретного момента времени), так и при реализации МВА (изменение состава представлений ВП в разные моменты времени).

2) Не все элементы СПЛ равнозначны для проектирования, как это продемонстрировано для целевых вычислителей.
3) Архитектурный стиль для полноценного документирования СПЛ должен включать элементы иерархической организации, используемые конфигурации и целевой вычислительный процесс.

Приведённые онтологические модели соответствуют опытным наблюдениям, что подтверждает корректность.

4.4 Расширение методики проектирования встроенных систем

В данном разделе предлагается *методика проектирования ВСС,* обобщающая результаты диссертационного исследования (Рисунок 63). Она является расширением традиционного маршрута проектирования ВСС (раздел 1.1). Внесены следующие изменения:

1) Выделен этап проектирования СПЛ при помощи архитектурного стиля МПВ.
2) Делается акцент на выделении и разработке МВА с применением предложенной методики сквозного моделирования при создании сопутствующих САПР.
3) Предложен новый метод приближённой оценки вариантов ВПЛ, основанный на методе структурирования функции качества (Quality Function Deployment).
4) Предложен метод грубой оценки накладных расходов, вызванных иерархической организацией СПЛ.

Рисунок 63 – Расширенная методика проектирования встроенных систем

4.4.1 Этапы расширенной методики проектирования

Предлагаемое расширение построено, основываясь на концепциях HLD-методологии и совместного проектирования (как HW/SW, так и Compiler / Architecture). Оно ориентировано на разработку многоуровневых ВСС, включающих МВА.

Отличительной особенностью расширенной методики является выделение этапа проектирования уровневой организации (СПЛ). Это позволяет:

1) Расширить ППР, что повышает качество проектирования.
2) Повысить уровень формализации принятых технических решений за счёт использования специализированных архитектурных стилей.
3) Повысить качество документирования СПЛ и объектов повторного использования в её составе.
4) Сократить время на разработку заказных ВПЛ и МВА за счёт использования предложенной методики моделирования многоуровневых ВСС и их элементов.

На Рисунок 62 показана предлагаемая методика проектирования ВСС. Способ визуализации аналогичен использованному в подразделе 1.1.1. Внесены следующие изменения:

1) Этап разделения на программную и аппаратную составляющую заменён на этап проектирования уровневой организации (СПЛ). Это позволило расширить ППР. Этап формирует новые требования к принятию решений и документированию результатов.
2) Этап рабочего проектирования аппаратной и программной составляющей заменён на проектирование и разработку отдельных уровней ВСС, определённых на втором этапе.

В Таблица 5 приведена сводная характеристика этапов проектирования предлагаемой методики. Этап (2) является главной отличительной особенностью предлагаемой методики и подробно рассмотрен в подразделе 4.4.2. Его основным инструментом является архитектурный стиль МПВ (раздел 2.4). Этапы (1), (4), (5), (6) и (7) являются стандартными для жизненного цикла ВСС (раздел 1.1) и рассматриваться не будут. Этап (3) определяется решаемой задачей и результатами этапа (2), в связи с чем не может быть детализирован сверх общего случая, рассмотренного в главе 3. Конкретные примеры работ по данному этапу приведены в главе 5.

Применение предложенной методики проектирования в полном объёме целесообразно только для ВСС с нетривиальной СПЛ, где необходима [ре]конфигурация минимум двух ВПЛ.

Таблица 5 – Этапы методики проектирования встроенных систем

Этап	*Описание*	*Артефакты*
1. Спецификация продукта	Анализ интересов и ограничений к системе с позиции пользователя и других заинтересованных сторон. Запись значимых функциональных и нефункциональных требований, а также процедуры приёмочных испытаний в виде документа. Может осуществляться в свободной форме согласно нормативным документам или методологиям (ГОСТ, RUP, SysML и т.д.).	Спецификация функциональных и нефункциональных требований к ВСС. Описание процедуры приёмочных испытаний.
2. Проектирование уровневой организации (СПЛ)	Разработка и запись архитектуры СПЛ. Задача данного этапа – проанализировать варианты, чтобы найти оптимальное решение. Особенность этапа – абстракция от деталей организации отдельных уровней. Инструмент: архитектурный стиль МПВ. Включает следующие подзадачи: – определение состава СПЛ; – определение иерархической организации СПЛ; – определение архитектуры СПЛ.	Архитектурная спецификация системной платформы, фиксирующая структуру и методику использования СПЛ.
3. Декомпозиция	Проработка взаимодействия между уровнями. Разработка интерфейсов, эталонных моделей, симуляторов и других инструментальных средств для обеспечения независимой разработки. Инструментальные средства данного этапа зависят от используемых ВПЛ и их сочетаний. Для моделирования следует использовать методику моделирования многоуровневых ВСС и их элементов.	Модели ВСС, описания интерфейсов, список необходимого инструментария.
4. Проектирование и разработка уровней ВСС	Данный этап построен на параллельной (в соответствии с методикой совместного проектирования) разработке элементов ВСС на разных уровнях. Применяемые инструменты, методы и средства для проектирования зависят от используемых ВПЛ и их взаимосвязей.	Специфичные для используемых ВПЛ артефакты, включая: архитектурные спецификации.
5. Интеграция	Объединение артефактов, полученных в результате проектирования отдельных уровней системы согласно архитектурным спецификациям. Полученная система должна пройти внутренние испытания на проверку соответствия общесистемным требованиям и требованию по взаимодействию уровней между собой.	Целевая система.
6. Приёмочные испытания	Выполняются приёмочные испытания, определённые на этапе 1. В случае их успешного прохождения, проект может считаться завершённым.	Акт о проведении испытаний.
7. Поддержка и обновление	Разрабатываемая система находится в эксплуатации.	Документация и обновления целевой системы.

4.4.2 Проектирование уровневой организации

Высокая сложность сравнения ВПЛ и их разнообразие (что особенно хорошо видно для языков программирования [86]), а также сложность в определении подхода к реализации целевой функциональности вынуждает применять принцип разделения интересов (раздел 1.1.2.4). В связи с этим необходимо абстрагироваться от вопросов организации ВСС в рамках отдельных уровней. Также данный принцип является основанием для выделения следующих этапов проектирования уровневой организации:

1) Этап определения используемых ВПЛ и МВА.
2) Этап определения иерархической организации СПЛ.
3) Этап определения архитектуры уровневой организации.

Далее будет дано описание этих этапов.

4.4.2.1 Определение списка вычислительных платформ

Этап определения списка ВПЛ базируется на предложенной в разделе 1.3 структуре ППР. Проектировщику необходимо сделать выбор из доступных, или потенциально реализуемых ВПЛ. Принятые решения состоят из следующих альтернатив (осей проектного пространства):

1) Тип ВПЛ.
2) Стадия конфигурирования.
3) Метод обеспечения ВПЛ.

На данном этапе внимание обращается на выбор ВПЛ, непосредственно необходимых при реализации системы. Служебные ВПЛ будут определяться на следующем шаге. К сожалению, методы сравнительного анализа слабо проработаны. Это видно на примере «священных войн» между ВПЛ (Verilog и VHDL, Java и .Net и т.д.) и общими принципами построения ПО (функциональное и объектно-ориентированное проектирование). В рамках последних, большая часть аргументов основывается на текущей моде или личных предпочтениях [108]. Помимо того, что подобный анализ неэффективен, он ещё и препятствует созданию заказных элементов уровневой организации, так как не позволяет содержательно фиксировать потребности и нужды разработчиков. В связи с этим, в данной работе используется метод структурирования функции качества (Quality Function Deployment [109]) или дом качества.

Пользовательские характеристики вычислительных платформ:

1) *Гибкость иерархической организации* системной платформы, выраженная в количестве вариантов обеспечения ВПЛ с принципиально различными свойствами. Например, высокоуровневый синтез, интерпретация, компиляция и интерпретация с компиляцией времени исполнения следует рассматривать как различные, в то время как два однотипных компилятора одного языка следует рассматривать как эквивалентные.
2) *Популярность* или доступность специалистов с соответствующей квалификацией.
3) *Порог вхождения* – характеристики сложности освоения ВПЛ новыми специалистами. Чем более высокий порог – тем больше усилий необходимо затратить на подготовку специалиста.
4) *Доступность* – характеризует возможность работы с инструментом для неподготовленных специалистов. К подобным работам относится: чтение исходного кода с целью анализа логики работы, отладка и поддержка системы. Не является напрямую связанным с порогом обучения (например, языки программирования

семейства lisp имеют низкий порога вхождения, но не являются доступными). Как правило, высокая доступность инструментария даёт большую гибкость с точки зрения организации работ, но провоцирует ошибки, связанные с интуитивным пониманием инструментария.

5) *Скорость разработки* – характеризуется способностью языка к быстрой реализации и прототипированию систем или их элементов без детального проектирования архитектуры и структур данных. Например, языки с динамической типизацией и минимизированным синтаксисом, такие как Lisp и Verilog (в меньшей степени), позволяют быстрее реализовать систему, чем их более формальные аналоги, такие как Haskell и VHDL.
6) *Предсказуемость* – характеризуется формальными свойствами конфигураций, выполненных для ВПЛ. Как пример: системы типов, гарантии времени исполнения, темпоральные логики [36], формально доказанные свойства (от проверки программ на наличие блокировок (deadlock) до формального доказательства соответствия спецификациям [37,102]) и т. д.
7) *Масштабируемость, адаптивность и повторное использование* конфигураций или их элементов, определяющая максимальный размер спецификаций. Данная характеристика охватывает широкий диапазон вариантов, начиная с программирования в машинных кодах, имеющего минимальные возможности в данном вопросе, и заканчивая системами по построению языков, в которых можно внести практически любые изменения.
8) *Производительность* – общая характеристика скорости обработки информации в рамках ВПЛ.
9) *Стоимость реализации* (для заказных системных платформ).

Технические решения в ВПЛ:

1) Выбор метода обеспечения реализации вычислительной платформы:
    a. *Обеспечение ВПЛ аппаратными вычислительными платформами, включая:* конструктивные и программируемые ВПЛ, а также ВПЛ, обеспеченные высокоуровневым синтезом. Под аппаратными вычислительными платформами понимаются ВПЛ, обладающие высоким уровнем параллелизма и соответствующими моделями вычислений (DE, SDF и другие [54]).
    b. *Обеспечение ВПЛ языковыми средствами*.
    c. Обеспечение ВПЛ программно-реализованными вычислительными платформами.

2) *Применение высокоуровневых языковых средств*, выражается в несоответствии модели вычислителя языку вычислительной платформы.
3) *Искусственные ограничения* и элементы формального специфицирования конфигураций. Ярким примером ВПЛ, в которой присутствуют ограничения, является платформа Erlang. Отсутствие изменяемых данных позволило сделать её инструментом для написания массивно-параллельных приложений. Другие примеры ограничений, упрощающих разработку: системы типов, статические анализаторы (Lint tools) и д.р.
4) *Уровень специализации* на классе задач использования.
5) Наличие *механизмов расширения / адаптации* ВПЛ для новых задач и применений. Примерами является расширяемость системы команд, блоков обработки данных или микропрограммировании.
6) *Механизмы расширения языковых средств*, характеризующие потенциал роста языка ВПЛ для применения в новых задачах. В большинстве случаев, данную возможность следует рассматривать как механизм построения более специализированных ВПЛ без существенных затрат.

Приведённые выше характеристики и технические решения позволяют сопоставить вычислительные платформы, а также обнаружить незаполненные ниши для разработки заказных решений. Расстановка весовых коэффициентов для каждого конкретного случая должна производиться архитектором самостоятельно, в зависимости от ситуации.

Необходимо отметить, что результаты данного этапа имеют высокую ценность с точки зрения сохранения проектного опыта, но при этом низкую с точки зрения взаимодействия архитектор-разработчик, так как они содержат мало применимой в разработке информации.

4.4.2.2 Определение иерархической организации системной платформы

Понятие иерархической организации СПЛ было введено в разделе 4.3. Целью этапа является определение иерархической организации СПЛ, в которой одни уровни будут обеспечивать другие. Для этого могут устанавливаться отношения между уже выбранными ВПЛ, или добавляться новые служебные ВПЛ.

Общей рекомендацией для формирования иерархической организации СПЛ является минимизация числа уровней, так как большинство межуровневых переходов снижает производительность и повышает сложность ВСС. Необходимо разделять сложность разработки и сложность ВСС. Например, использование языков высокого уровня упрощает процесс разработки, но в тоже время привносит во ВСС сложные инструменты, вероятность наличия

дефектов в которых весьма высока. Также, введение новых уровней в СПЛ может быть оправдано в случае серьёзного выигрыша в скорости разработки или, если это необходимо для оптимизации (например, ассемблерные вставки).

4.4.2.3 Определение архитектуры уровневой организации

Задача превращения структуры уровневой организации в архитектуру ВСС решается на данном этапе. Он включает:

1) Определение состава конфигураций, необходимых для разворачивания системной платформы, включая конфигурации её служебных и целевых уровней.
2) Определение состава и функциональных возможностей заказных элементов уровневой организации.
3) Подготовка рекомендаций для распределения функций по уровням системы и вариантам оптимизаций. Иначе, определение правил использования.

Основным инструментом, с помощью которого производится документирование и исследование архитектуры уровневой организации, является стиль МПВ (раздел 2.4). Он позволяет не только отражать варианты уровневой организации, но и документировать обозначенные выше вопросы. Кроме того, данный инструмент может использоваться для постановки частных технических заданий для заказных элементов. Важно отметить степень его интеграции в методику проектирования, которая позволяет плавно перейти от вопросов архитектурного специфицирования многоуровневых ВСС, их элементов и МВА к реализации сопутствующих САПР.

4.5 Выводы

1) Предложено расширение системы архитектурных абстракций для работы с уровневой организацией встроенных систем. Для этого:
   - уточнено понятие системной платформы в соответствии с современным состоянием области проектирования встроенных систем;
   - предложено понятие многоуровневого вычислительного агрегата, как основного объекта повторного использования уровневой организации, определена его структура и состав;
   - предложены онтологические модели для [ре]конфигурации и иерархической организации, унифицирующие и формализующие рассмотрение уровневой организации встроенных систем.

2) Предложена *методика проектирования встроенных систем*, явно выделяющая этап проектирования уровневой организации. Свойства и эффект предлагаемой методики обеспечиваются унифицированным рассмотрением уровневой организации, применением архитектурного стиля «модель-процесс-вычислитель» и методикой моделирования многоуровневых встроенных систем и их элементов. Эффект заключается в повышении качества принимаемых архитектурных решений в части уровневой организации встроенных систем и сокращении затрат на разработку заказных элементов уровневой организации.

5 Анализ результатов исследования

В данной главе анализируются результаты исследования и эффект от их использования, включая повышение качества архитектурных решений в части уровневой организации встроенных систем и сокращение затрат на разработку заказных элементов уровневой организации. Оценка архитектурных стилей производится аналитическим способом, разработанным на основе представленных в литературе примеров анализа из близких областей (языки программирования, стили программирования, модели вычислений).

Практическая значимость и эффективность результатов исследования подтверждается использованием оного в решении практических задач.

5.1 Методы оценки эффективности архитектурных стилей

Есть большое количество инструментальных средств методологического уровня для разработки и проектирования ВСС. Примеры:

- методики разработки, направленные на организационные вопросы (каскадная модель разработки, итеративная модель разработки, гибкие методологии разработки, экстремальное программирование, Scrum, Kanban…);
- стили программирования (сентенциальное программирование, функциональное программирование, автоматное программирование, событийное программирование, структурное программирование, объектное программирование…);
- языки программирования (C, C++, Pascal, List, Python, Java, ML, Haskell, Smalltalk, Ruby, Go, Rust, Nim, Julia…);
- механизмы синхронизации / обеспечения взаимодействия параллельных процессов (семафор, рандеву, транзакционная память, сети процессов Кана);
- архитектурные стили;
- библиотеки, фреймворки, паттерны проектирования (Design Patterns), методики моделирования данных и т.д.

Из-за такого разнообразия вариантов появляется проблема выбора инструмента, а это одна из наиболее насущных проблем в области вычислительной техники. Её актуальность подтверждается в том числе, большим числом публикаций на тему в научно-технической литературе [37,86,108,110–115]. Разработка общего решения позволит значительно повысить эффективность работы индустрии в целом, так как:

1) Оптимальный выбор инструментальных средств позволит сократить длительность и стоимость проектов, повысить качество результата.
2) Точная оценка необходимых инструментальных средств позволит оценивать эффект от их модификации, следовательно, повысит эффективность их разработки.

На сегодня нет универсального решения для оценки инструментария методологического уровня, получившего широкое распространение и при этом с необходимым уровнем детализации. Для существующих способов оценки характерно: ориентация на легко поддающиеся оценке метрики (объём исходного кода; уровень интереса к языку, выраженный в статистике поисковых запросов и т.д.); высокая сложность интерпретации результатов, требующая активного участия человека; преобладание понятийных оценок над численными. Эти недостатки обусловлены следующим:

1) Высокая размерность задачи, выраженная в количестве сопоставляемых объектов и критериев сопоставления. Множество рассматриваемых объектов постоянно расширяется новыми решениями. Высокая размерность обусловлена следующим:
    a. Описание объектов является научной проблемой (Например, для методик проектирования [111]), так как именно способ описания позволяет выявить и подчеркнуть сходства и различия рассматриваемых инструментов, а также, определяет уровень абстракции. Например, при сравнении языков программирования нет однозначного решения, должны ли быть включены в рассмотрение инструментальные средства. В первом случае, это приводит к резкому росту размерности задачи, во втором – к рассмотрению задачи «в вакууме».
    b. Инструментальные средства методологического уровня могут рассматриваться с разных точек зрения: производительность, энергопотребление, скорость реакции, эффективность разработчика, доверительность результата, обслуживаемость, расширяемость, доступность специалистов нужного профиля, опыт команды разработчиков, сложившиеся традиции и т.д. Точки зрения могут противоречить друг-другу. Для каждой из них определены свои требования. Это делает множество критериев потенциально бесконечным.
2) Сложность определения зависимости между прикладными свойствами и измеряемыми метриками (например, наличие абстракции или функции X, наличие оператора `goto`, количество используемых абстракций в архитектурном стиле или

операторов в языке программирования). Это приводит к необходимости использования косвенных критериев. Примечания:

   a. Под прикладными свойствами рассматривается интегральный эффект от применения инструмента в проекте, выраженный в сокращении затрат на разработку и поддержку системы, приобретение системой новых свойств.
   b. Взаимосвязь метрик исследуемого инструмента и прикладных свойств зависит от разработчика, решаемой задачи и сочетания инструментов. Это делает задачу определения взаимосвязи затруднительной.

3) Высокая сложность и стоимость сбора экспериментальных данных. Субъективность анализа. Примечания:

   a. Лишь часть характеристик проекта может быть измерена непосредственно (производительность, бюджет, срок разработки). Интересны такие характеристики, как: стоимость адаптации системы под изменившиеся требования, расширяемость, возможность повторного использования, стоимость расширения команды разработчиков - измерить их значительно сложнее.
   b. Высокое влияние человеческого фактора на экспериментальные данные не позволяет однозначно оценивать влияние инструмента. Как отмечают специалисты, эффективность разработчиков может отличаться на порядок при выполнении одинаковых задач [116].
   c. Множество систем являются закрытыми проектами, а принятые в процессе их разработки решения – коммерческой тайной. Это приводит к сокращению количества примеров, которые можно рассмотреть. Информация о плохих решениях часто скрывается, а об удовлетворительных – раздувается, с целью улучшения образа (имиджа) разработчиков.
   d. Большинство ВСС – коммерческие продукты. Это препятствует постановке «чистого» эксперимента. Экономически нецелесообразно делать несколько одинаковых систем разным инструментарием.

Отсутствие готового метода оценки архитектурных стилей в совокупности с отмеченными обстоятельствами стало основанием для разработки специального способа оценки. В нём выделяются следующие этапы:

1) Формирование критериев оценки архитектурных стилей для задач проектирования и документирования уровневой организации. Критерии должны быть сформированы на основе обзора подраздела 1.4.1 и требований, сформированных в разделе 2.1. Для

предложенных критериев должны быть сформулированы методы оценки архитектурных стилей с минимальным влиянием человеческого фактора. Критерии выбираются исходя из их значимости для задач проектирования и документирования уровневой организации, а также, из возможности получения доверительного результата при сравнении.

2) Оценка архитектурных стилей по сформированным критериям. Представление результатов оценки в наглядном виде, демонстрирующем отличия между рассматриваемыми инструментами.
3) Интерпретация полученных оценок по заданным критериям и качественная оценка прикладным возможностям рассмотренных архитектурных стилей. При проведении оценки должны учитываться: (1) отсутствие нормализации оценок как по отдельным критериям, так и между ними; (2) зависимость влияния критерия от особенностей конкретного проекта и решаемой задачи.

Итак, доверительность первого этапа основывается на полноте рассмотрения вопроса в главе 1. Доверительность второго этапа основывается на сужении задачи оценки за счёт отказа от нормализации и численных оценок в пользу относительных оценок по простым критериям. Доверительность третьего этапа обеспечивается опытом применения результатов в реальных проектах (раздел 5.3).

5.2 Аналитическая оценка архитектурных стилей

Далее будут приведены критерии оценки архитектурных стилей для проектирования и документирования уровневой организации ВСС. Для каждого критерия будут приведены: наименование, краткое описание, влияние критерия на прикладные свойства, способ оценки.

В качестве оценки будут использоваться натуральные числа от 1 до 10, что сделано для удобства визуализации. Оценка 10 будет выставляться лучшему архитектурному стилю из рассматриваемых, 1 – худшему. Промежуточные значения будут расставляться с равным шагом значений. Если оценка одинакова, значит по рассматриваемому критерию стили эквивалентны или их отличия незначительны с практической точки зрения. Оценки не нормированы, следовательно, оценки 4 и 8 не говорит о том, что первый стиль в два раза хуже второго по заданному критерию. Оценки по разным критериям также не связаны между собой.

Степень покрытия ППР. Характеризуется уровнем покрытия ППР, определённого в подразделе 1.3.2. Ограниченное представление ППР приводит к сокращению числа рассматриваемых вариантов и повышает вероятность пропустить лучший вариант. Обратной

стороной является рост размерности задачи проектирования. Ограничения могут быть вызваны как «игнорированием» части проектного пространства, так и целенаправленным абстрагированием. Пример оценки: полное покрытие пространства проектных решений – 10, покрытие произвольно числа уровней целевой системы без учёта способа реализации ВПЛ – 6, покрытие ограниченного числа уровней – 4.

Соответствие абстракций архитектурного стиля задаче проектирования и документирования уровневой организации ВСС (далее – соответствие абстракций). Характеризуется соответствием набора абстракций архитектурного стиля уровневой организации. Влияет на сложность применения стиля для решения практических задач. Например, понятие «системы» позволяет описывать уровневую организацию, но также позволяет описать вопросы, не связанные с ней, а это «размывает» решаемую задачу и повышает сложность процесса проектирования. Оценка 10 ставится только тогда, когда архитектурный стиль имеет необходимый набор абстракций для описания уровневой, и при этом не позволяет их использовать для других объектов. Полнота системы понятий не учитывается.

Избыточность архитектурного стиля. Характеризуется объёмом возможностей архитектурного стиля, не относящимся к работе с уровневой организацией. Избыточные возможности повышают порог вхождения при использовании архитектурного стиля, а также провоцируют на решение посторонних задач (смешение интересов). 10 баллов ставится для стилей, не предоставляющих избыточных возможностей. Например, наличие в архитектурном стиле возможности описания межмодульных интерфейсов является избыточной, в отличие от возможности фиксация типа межуровневой взаимосвязи.

Возможность *отображения [ре]конфигурации*. Характеризуется возможностью архитектурного стиля визуализировать возможности [ре]*конфигурации*. Косвенным образом связана со степенью покрытия ППР. Её явное отображение позволяет говорить о лучшем соответствии актуальным тенденциям в области проектирования ВСС. 10 балов ставится в случае, если стиль позволяет описать замену функциональности в явном виде и при этом различает одновременный и поочерёдный способ использования конфигураций.

Возможность описания *методологического аспекта* уровневой организации. Характеризуется возможностями для описания способа использования СПЛ, ВСС и многоуровневых вычислительных агрегатов. 10 баллов ставится в случае, если стиль позволяет описать способ конфигурирования СПЛ с необходимым уровнем детализации. Данный критерий вступает в противоречие с избыточностью, так как расширение возможностей архитектурного

стиля может привести к выполнению данного свойства, но при этом значительно усложнив архитектурный стиль.

Инструменты для описания *возможностей ВПЛ*, МВА и уровней ВСС. Влияет на применимость стиля в проектировании новых ВПЛ и их функциональных возможностей. 10 баллов ставится в случае, если архитектурный стиль позволяет описать возможности ВПЛ с необходимым уровнем детализации, при этом не касаясь функциональности, реализованной в её рамках (прикладной функциональности). Пример: уровневые диаграммы не позволяют описать детали функциональных возможностей ВПЛ; диаграммы развёртки позволяют описать функциональность ВПЛ лишь в привязке к структуре ВСС.

Сложность архитектурного стиля. Характеризуется количеством понятий, нужных для использования архитектурного стиля. Влияет на порог вхождения и трудоёмкость использования. Оценка 10 ставится для наиболее простого архитектурного стиля. Пример: сложность архитектурного стиля графа актуализации сильно ниже по сравнению с МПВ.

Анализ архитектурных стилей уровневой организации производится для тех, что рассмотрены в главе 1 и предложены в главе 2 , а именно:

1) Диаграммы предметной области.
2) Диаграммы развёртки.
3) AADL.
4) Уровневые диаграммы.
5) «Бургер-диаграмма».
6) Системно-иерархический стиль.
7) Граф актуализации.
8) Модифицированный граф актуализации.
9) Модель-процесс-вычислитель (МПВ).

Полученные оценки приведены в Таблица 6. Для наглядности, данные из таблицы приведены на круговых диаграммах, подходящих для описания многомерных пространств. Для лучшего восприятия информации, диаграммы разбиты по группам:

1) Архитектурные стили, основанные на взаимосвязанных множествах (Рисунок 64).
2) Архитектурные стили, основанные на понятии компонента (Рисунок 65).
3) Архитектурные стили, основанные на понятии системы (Рисунок 66).
4) Архитектурные стили, основанные на понятии трансляции (Рисунок 67).
5) Предложенные архитектурные стили (Рисунок 68).

6) Отобранные в процессе анализа архитектурные стили (Рисунок 69).

Таблица 6 – Сравнительная оценка архитектурных стилей

	Диаграммы предметной области	**Диаграммы развёртки**	**AADL**	**Уровневая диаграмма**	**«Бургер-диаграмма»**	**Системно-иерархический стиль**	**Граф актуализации**	**Модифицированный граф актуализации**	**МПВ**
Покрытие ППР	5	2	10	6	4	8	8	8	10
Соответствие абстракций	4	2	8	8	4	4	4	8	10
Избыточность	10	4	2	10	5	5	6	8	10
Сложность	10	4	2	10	6	4	8	10	4
[Ре]конфигурация	2	4	8	2	10	10	6	4	10
Методологический аспект	2	6	10	2	2	8	4	4	8
Документирование ВПЛ	2	6	10	2	4	4	6	2	8

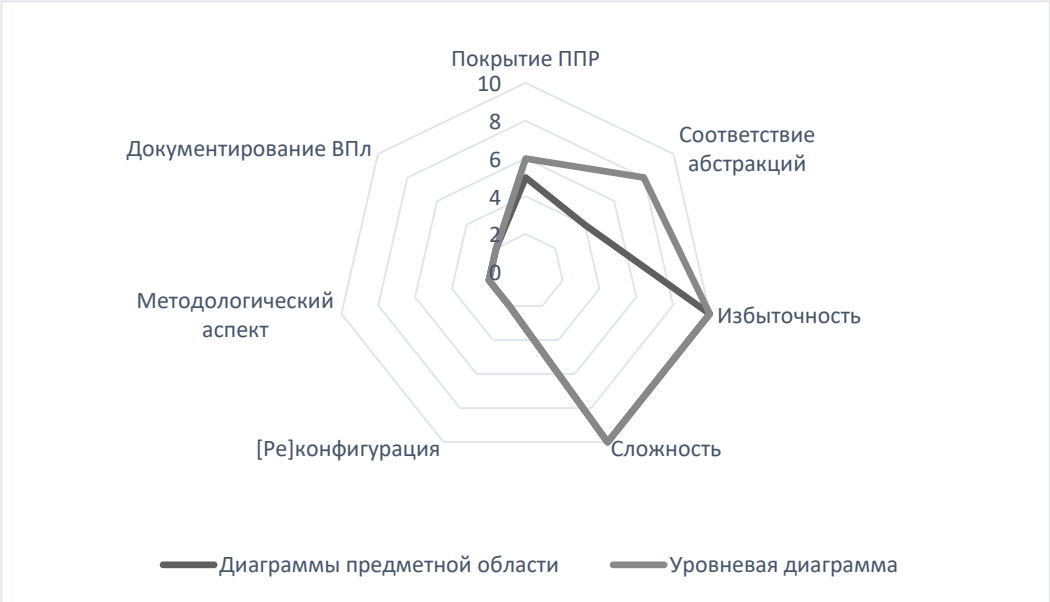


Рисунок 64 – Архитектурные стили, основанные на взаимосвязанных множествах

На Рисунок 64 можно видеть, что архитектурный стиль уровневой диаграммы по всем критериям превосходит, либо эквивалентен диаграмме предметной области. Такой результат ожидаем, так как диаграммы предметной области являются инструментом более ранних этапов проектирования. Это отражается в несоответствии абстракций и худшем покрытия ППР.

Рисунок 65 – Архитектурные стили, основанные на понятие компонента

На Рисунок 65 язык архитектурного описания AADL за исключением сложности и избыточности превосходит диаграммы развёртки по всем параметрам. Так вышло потому, что данные архитектурные стили относятся к одному классу, но в AADL была сделана попытка сделать язык описания универсальным, а в диаграммах развёртки была сохранена его специализация (пусть и частично).

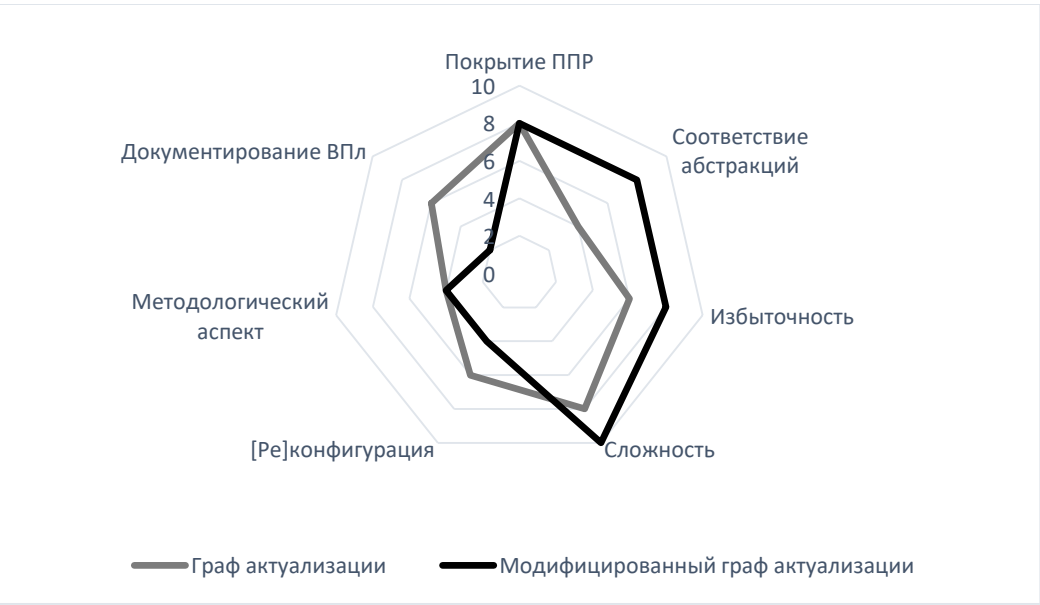


Рисунок 66 – Архитектурные стили на основе понятия системы

На Рисунок 66 показан архитектурный стиль графа актуализации и его модифицированный вариант. Для модифицированного варианта удалось сократить сложность и избыточность; повысить уровень соответствия абстракций за счёт невозможности документирования внутреннего устройства. Превосходство оригинального стиля наблюдается лишь по двум критериям (методологический аспект и [ре]конфигурация), и обеспечено излишне

абстрактным базовым понятием (транслятор). Это позволяет говорить о слабой применимости к практическим задачам.

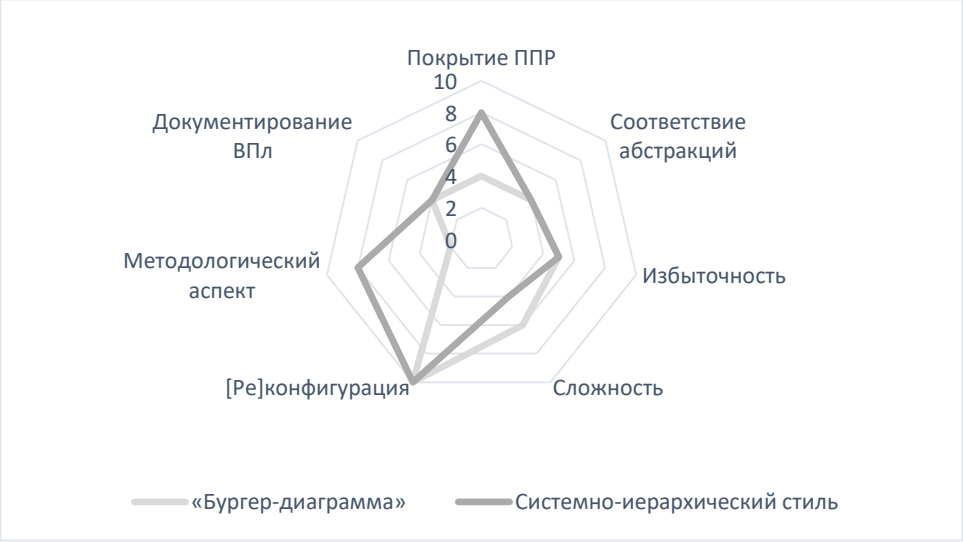


Рисунок 67 – Архитектурные стили на основе понятия трансляции

На Рисунок 67 показано сравнение архитектурных стилей, основанных на понятии системы. Системно-иерархический стиль получен путём расширения стиля «Бургер-диаграммы». Это позволило получить лучшие характеристики по всем критериям кроме сложности. Также, это означает, что применительно к задачам проектирования и документирования уровневой организации, предложенный архитектурный стиль работает лучше.

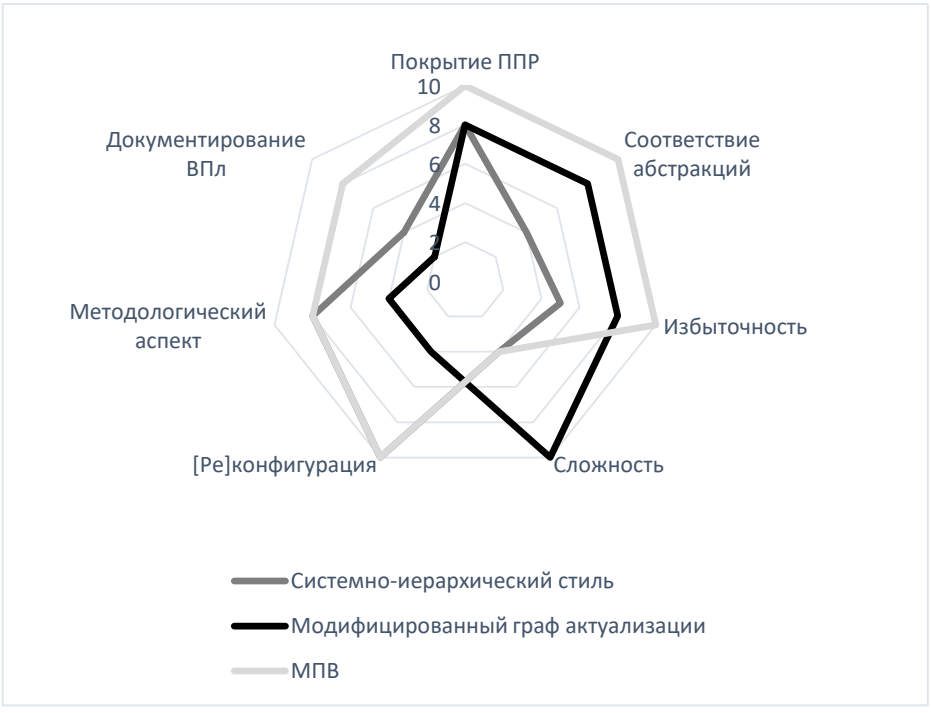


Рисунок 68 – Предложенные архитектурные стили

На Рисунок 68 показана оценка разработанных архитектурных стилей. Архитектурный стиль МПВ превосходит аналоги по большинству характеристик. Исключением является сложность в сравнении с модифицированным графом актуализации, что делает последний лёгким в освоении инструментом, но пригодным только для описания общих вопросов уровневой организации.

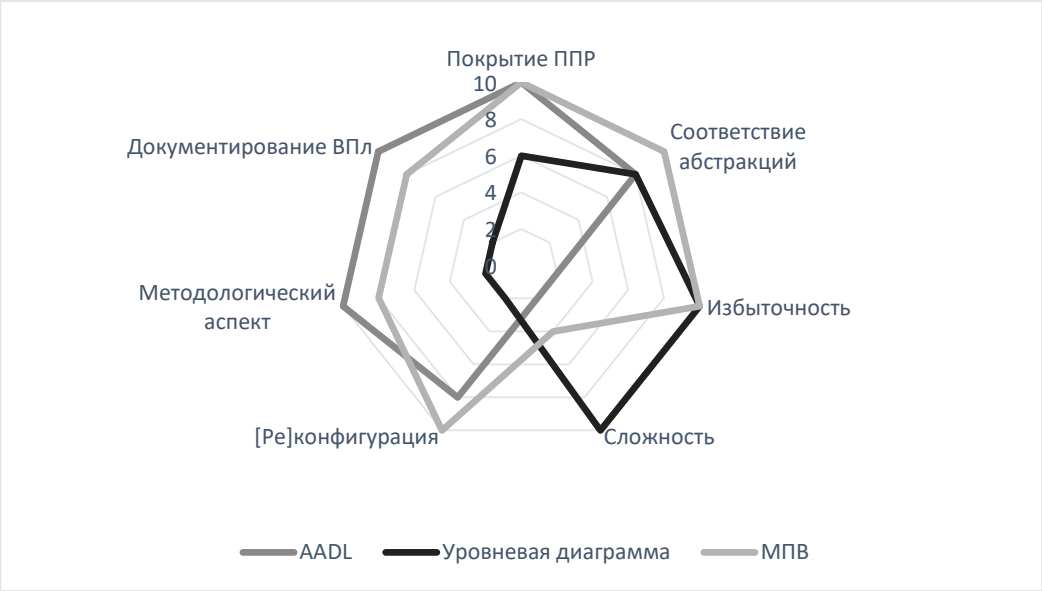


Рисунок 69 – Отобранные в процессе анализа архитектурные стили

Из приведённых выше диаграмм видно, что по совокупности характеристик, наиболее интересными архитектурными стилями являются AADL, уровневые диаграммы и МПВ. Их совместное рассмотрение приведено на Рисунок 69.

Представленные архитектурные стили сильно различаются: AADL – обладает самыми мощными возможностями с точки зрения документирования уровневой организации, но при этом сложен и избыточен; уровневые диаграммы – самый простой архитектурный стиль, но позволяет описывать ограниченный объём ППР с минимальной детализацией; МПВ – сильно проще и компактнее AADL, но уступает ему по уровню детализации. Для формирования общего вывода из анализа, необходимо зафиксировать основные области применения (точки зрения):

1) Задачи документирования уровневой организации ВСС. Наиболее важным свойством здесь является возможность однозначно и точно зафиксировать результат проектирования. Избыточность описаний может рассматриваться как преимущество, так как позволяет сократить количество документов и снизить затраты на актуализацию документации.
2) Задачи проектирования уровневой организации ВСС. В данном случае важна полнота ППР, минимальная избыточность и соответствие абстракций. Сложность

инструментария для проектирования не приоритетна, так как предполагается, что проектировщик высоко квалифицирован.

Сравнительный анализ показал:

1) Архитектурный стиль языка *AADL* эффективен для задач документирования уровневой организации если: необходимы описания деталей реализации отдельных уровней ВСС; необходима высокая детализации архитектурных спецификаций; необходима организация работы крупной команды разработчиков. Это обеспечивается большим количеством специализированных абстракций и наличием текстовых спецификаций. В то же время, его избыточность препятствует решению задач проектирования, отвлекая разработчика на вторичные для уровневой организации вопросы. Ярким пример можно видеть в [90].
2) Архитектурный стиль «*уровневых диаграмм*» обладает ограниченным представлением ППР и низким уровнем детализации, что не позволяет использовать его в проектах с нетривиальной уровневой организацией, многоуровневыми вычислительными агрегатами и заказными уровнями ВСС. При этом, простота делает его лучшим для ВСС с простой уровневой организацией, что подтверждается его популярностью в индустрии. Возможность его использования в конкретном проекте должна определяться на ранних этапах проектирования с использованием других стилей или постановкой задачи.
3) Архитектурные стили *графа актуализации* и *модифицированного графа актуализации* позволяют описывать инструментальные средства и последовательность преобразований описаний ВП. *Системно-иерархический стиль* позволяет описать структуру проекта и её изменения на всех этапах жизненного цикла. Эти стили пригодны для работы с уровневой организацией, но их практическое использование осложнено общностью используемых абстракций, приводящей к смешению интересов и усложнению процесса проектирования.
4) *Архитектурный стиль МПВ* предназначен для проектирования и документирования уровневой организации ВСС. Его недостатком является довольно высокая сложность, связанная с нетрадиционным методологическим базисом и большим числом используемых понятий. Но это компенсируется полнотой представления ППР, минимальной избыточностью, возможностью описания методологического аспекта и функциональности отдельных уровней.
5) Архитектурные стили *«Бургер-диаграммы»*, *системно-иерархический стиль*, *граф актуализации*, *диаграммы развёртки* не рекомендуются для применения в рамках

проектирования и документирования системной платформы в качестве основного инструмента.

6) Архитектурный стиль «*диаграммы предметной области*» можно применять в качестве вспомогательного инструмента для формирования ВПЛ и уровней ВСС на раннем этапе проектирования.

В следующем разделе будет приведён анализ применения результатов исследования при решении практических задач, подтверждающий полученные оценки.

5.3 Применение результатов исследования в практических задачах

В данном разделе производится анализ использования результатов исследования в практических задачах (разработка ВСС, программного обеспечения ВСС и их модулей). Репрезентативность обосновывается следующим:

1) Рассматриваемые проекты выполнялись малыми коллективами.
2) Сопоставимая квалификация исполнителей (образование – высшее в области вычислительной техники, опыт работы от 1 до трёх лет).
3) Решения, принятые в уровневой организации, оказали значительное влияние на успех проектов. Часть решаемых задач заключается в перепроектировании существующих систем с использованием предложенного инструментария.
4) Для части проектов разрабатывались прототипы, на основании которых сделаны оценки архитектурных решений.

В подразделах 5.3.1 – 5.3.6 приведены описания проектов. Для каждого из них: (1) дано краткое описание; (2) описаны возможности использования результатов исследования; (3) приведено описание принятых архитектурных решений в уровневой организации и оказанный ими эффект. В разделе 5.4 приведена сводная характеристика по рассматриваемым проектам, а также сопоставление практических и аналитических данных.

Справки об использовании результатов исследования в проектах приведены в приложении А.

5.3.1 ИИС Луч-ТС М

Информационно-измерительная система «Луч-ТС М» (в дальнейшем – ИИС Луч-ТС М) предназначена для измерения активной и реактивной электрической энергии, мощности. Свидетельство об утверждении типа: RU.C.34.033.A № 57631.

5.3.1.1 Описание системы

ИИС Луч-ТС М представляет собой многоуровневую информационно-измерительную систему с централизованным управлением, компонуемую на объекте эксплуатации в соответствии с проектной документацией из технических средств. ИИС Луч-ТС М применяется как законченная система непосредственно на объекте эксплуатации.

ИИС Луч-ТС М выполняет следующие основные функции:

- измерение приращений активной и реактивной электрической энергии на заданных интервалах времени, поддерживаемых применяемыми счетчиками электрической энергии и не противоречащие техническим характеристикам измерительных каналов;
- измерение средних значений активной и реактивной электрической мощности на заданных интервалах времени, поддерживаемых применяемыми счетчиками электрической энергии и не противоречащие техническим характеристикам измерительных каналов;
- синхронизация времени в автоматическом режиме с помощью СОЕВ, соподчинённой национальной шкале времени безотносительно к интервалу времени с погрешностью не более ± 5 с;
- периодический и (или) по запросу автоматический сбор результатов измерений приращений электроэнергии и средних значений электрической мощности с заданной дискретностью учета, синхронизированных со шкалой UTC;
- хранение результатов измерений, информации о состоянии объектов и средств измерений в базе данных (допустимая глубина хранения не менее 3,5 лет).

Механическая защита от несанкционированного доступа обеспечивается пломбированием:

- конфигурационных портов каналообразующего оборудования ИВКУУ;
- крышек отсеков с клеммами счетчиков электрической энергии.

ИИС Луч-ТС М, состоит из трёх уровней:

1) Уровень измерительно-информационного комплекса (ИИК), автоматически выполняющий измерения в точке учёта. В состав ИИК входят: измерительные трансформаторы тока, измерительные трансформаторы напряжения, счетчики электрической энергии.
2) Уровень информационно-вычислительного комплекса узла учёта (ИВКУУ), принимающий и передающий данные в узле учёта между ИИК и информационно-

вычислительным комплексом (ИВК), включающий каналообразующую аппаратуру и соответствующее встроенное ПО.

3) Уровень ИВК, выполняющий автоматизированный сбор и хранение результатов измерений с уровня ИИК, подготовку различных обобщенных форм отчетов, передачу их всем заинтересованным сторонам, и включающий в себя:
   - центральные устройства сбора и передачи данных;
   - технические средства приёма-передачи данных;
   - ПО верхнего уровня;
   - Система обеспечения единого времени (СОЕВ).

Схема организации ИИС Луч-ТС М приведена на Рисунок 70.

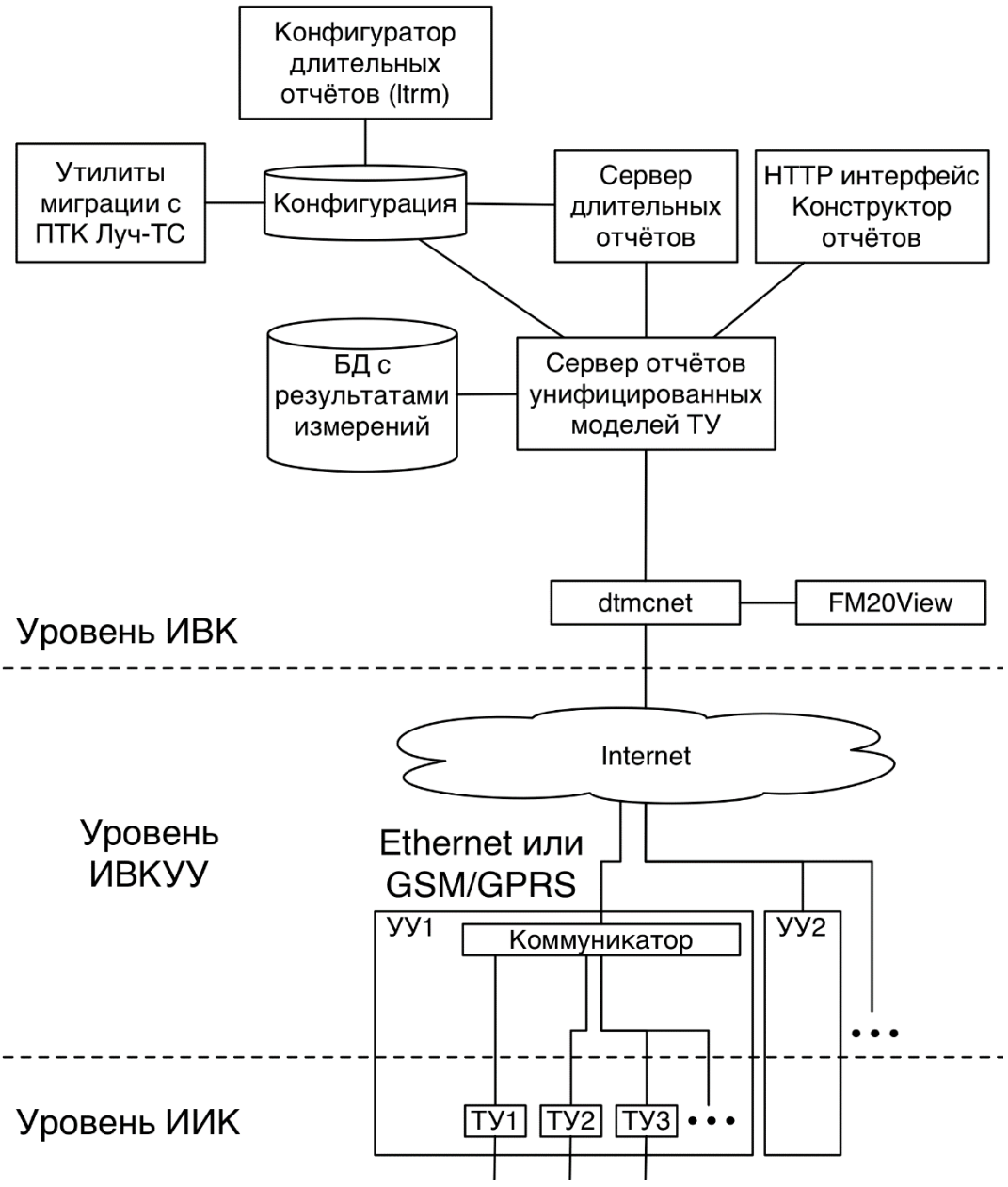


Рисунок 70 – Структура ИИС Луч-ТС М

Точка учёта (ТУ). Непосредственное измерение показаний потребления электроэнергии.

Узел учёта (УУ). Элемент ИИС Луч-ТС М, включающий в себя одну или более точек учёта, расположенных на относительно малом удалении друг от друга и подключённых к одному коммуникатору.

Коммуникатор. Элемент ИИС Луч-ТС М, организующий канал передачи данных между уровнем ИВК и конкретными точками учёта.

Сетевая подсистема ИИС Луч-ТС М (dtmcnet). Отвечает за обеспечение каналов передачи данных между УУ (включая ТУ) и ПО уровня ИВК.

БД с результатами измерений. СУБД, поддерживающая язык SQL. Предназначена для длительного хранения показаний потребления электроэнергии, тепла и объёма жидкости, проходящей по трубопроводу.

Конфигуратор длительных отчётов. Приложение с интерфейсом командной строки, позволяющее сконфигурировать длительные отчёты в системе ИИС Луч-ТС М.

Сервер длительных отчётов. Обеспечивает выполнение длительных отчётов в ИИС Луч-ТС М путём активации по событию или по расписанию с последующим построением отчётов.

HTTP интерфейс. Конструктор отчётов. Обеспечивает интерфейс для создания, получения и удаления отчётов для унифицированных моделей точек учёта. Конструктор отчётов является веб приложением.

Сервер отчётов унифицированных моделей ТУ. Обеспечивает формирование и сбор данных для отчётов к унифицированным моделям точек учёта.

FM20View. Утилита для работы с сетью dtmcnet, позволяющая управлять конфигурацией ПО уровня ИВКУУ.

5.3.1.2 Решаемая задача и способ оценки

В рамках проекта по разработке ИИС Луч-ТС М решалась задача перепроектирования системы ПТК Луч-ТС (свидетельство об утверждение типа № 40993-09). Ставились следующие задачи:

1) Снижение стоимости подключения к системе нового оборудования.
2) Снижение стоимости расширения функциональности.
3) Повышение общей надёжности и стабильности системы.

4) Повышение коммерческой привлекательности системы.

Поставленные задачи были решены в ходе перепроектирование ПО уровня ИВК. При использовании МПВ и расширенной методики проектирования, были приняты следующие решения, связанные с уровневой организацией:

1) Предложена концепция модели ТУ, абстрагирующая реализацию ИВК от специфики конкретного вида оборудования. Частично реализована в ПТК Луч-ТС, но из-за ограничений интерфейса не позволила решить поставленные задачи.
2) Механизм обеспечения надёжности взаимодействия с удалённым оборудованием. Является заказным уровнем системы, в котором задаётся дисциплина восстановления после сбоев каналов передачи данных или стороннего оборудования. Обеспечен специализированной виртуальной машиной.
3) Механизм верификации иерархических конфигураций. Для решения данной задачи был предложен уровень системы, в котором разработчик аннотирует существующий код, чем формирует сквозные механизмы верификации.
4) Механизм удалённого конфигурирования пользовательского интерфейса. Реализован при помощи интерпретатора на клиентской стороне, строящего пользовательский интерфейс согласно полученным с сервера конфигурациям.

Оценка процесса разработки системы производится на основе сравнения данных о процессе разработки ПТК Луч-ТС и ИИС Луч-ТС М: затраты на реализацию ядра системы; затраты на подключение к системе нового вида оборудования; количество дефектов, обнаруженных при тестировании нового оборудования; затраты на расширение функциональности системы.

Для механизма верификации иерархических конфигураций делается оценка объёма исходного кода с использованием предложенных механизмов и без них («ручная» реализация). Для простоты оценки предполагается, что на каждом уровне иерархической организации может возникнуть только один тип ошибок (на практике больше, а с учётом использования внешних библиотек, список типов ошибок следует считать открытым). Для обработки исключений (exception) необходимо минимум 4 строки исходного кода: две – на блок обработки исключений, одна – на формирование информации об ошибке, и ещё одна – это проброс исключения на вышележащий уровень иерархии. Кроме того, при реализации должна обеспечиваться передача и хранение данных из контекста. При использовании разработанных механизмов на один элемент структуры конфигурации затрачивается одна строка кода. Оценка эффективности предложенного решения определяется следующим образом:

1) Определяется объём исходного кода для задачи верификации конфигурации с использованием предложенного решения.
2) Оценивается объём исходного кода без использования предложенного решения.
3) Оценивается соотношения объёмов исходного кода с учётом реализации предлагаемых механизмов и без неё.

Необходимо отметить, что «ручная» реализации имеет следующие недостатки, не связанные с объёмом исходного кода: (1) необходимость передачи и хранения данных контекста; (2) возникновение исключительной ситуации, не учтённой разработчиком; (3) затраты на поддержку программного кода; (4) влияние методических указаний по структурированию исходного кода для применения механизма.

5.3.1.3 Полученные результаты

В Таблица 7 приведены оценки процесса разработки системы.

Таблица 7 – Сравнение затрат на элементы ПТК Луч-ТС и ИИС Луч-ТС М

Элемент системы	Трудоёмкость (ч./м.)			Сроки разработки (м)		
	ПТК Луч-ТС	ИИС Луч-ТС М	%	ПТК Луч-ТС	ИИС Луч-ТС М	%
Ядро системы	25	30	120	10	13	130
Модуль профиля мощности	5	2	40	3	0,75	25
Модуль тарифных расписаний	-	1,5	-	-	0,5	-
	5	*1,75*	*35*	*3*	*0,625*	*21*
Подключение семейства счётчиков CE102	3	1,5	50	2	0,76	38
Подключение семейства счётчиков CE300, CE303, CE6850M	6	2	33	4,5	1,5	33
Подключение семейства счётчиков CE102M	-	0,5	-	-	0,5	-
Подключение семейства счётчиков СЭБ2А	-	1,25	-	-	0,75	-
Подключение семейства счётчиков Маяк	-	1	-	-	0,75	-
Подключение семейства счётчиков Миртек	-	1,5	-	-	1,25	-
	4,5	*1,3*	*29*	*3,2*	*0,9*	*28*

Из приведённых данных делаются следующие выводы:

1) Разработка ядра системы ИИС Луч-ТС М требует большее количество ресурсов. Это обусловлено расширением функциональности системы и необходимостью реализации элементов, направленных на расширяемость и адаптивность.
2) Среднее время добавления новой функциональности сокращено. Это стало возможным за счёт улучшения модульности системы. Основной проблемой расширения ПТК Луч-ТС была высокая связанность элементов, приводящая к длительному процессу отладки.
3) Среднее время подключения нового оборудования к системе ИИС Луч-ТС М было сокращено на ~30%, что было основной целью модернизации. Это стало возможным за счёт улучшения модульности системы и внесения декларативных элементов конфигурации. Основная проблема ПТК Луч-ТС заключалась в высокой сложности отладки системы, связанной с отсутствием централизации исходного кода, отвечающего за поддержку одного типа оборудования.

В Таблица 8 приведена оценка механизма верификации конфигураций.

Таблица 8 – Оценка объёма исходного кода в случае использования механизма верификации конфигураций

Метрика	С использованием механизма	Ручная реализация
Объём прикладного исходного кода		173
Объём исходного кода для верификации	15	60
Реализация механизма верификации	54	-
Отношение прикладного исходного кода и исходного кода верификации	9%	35%
Отношение прикладного исходного кода и исходного кода верификации с учётом реализации механизма	40%	-

Объём исходного кода для верификации конфигурации с использованием механизма верификации составляет менее 10% от прикладного алгоритма, а в случае ручной реализации – 35%. На практике такое соотношение означает, что реализация верификации будет «замусоривать» прикладной алгоритм. Если смоделировать расширение конфигурации (с линейным ростом объёма прикладного кода на элемент), то будет видно, что относительный объём прикладного исходного кода будет сокращаться, как это показано на Рисунок 71. Горизонтальная ось – количество структурных элементов конфигурации, вертикальная ось – отношение прикладного исходного кода и исходного кода верификации с учётом реализации механизма.

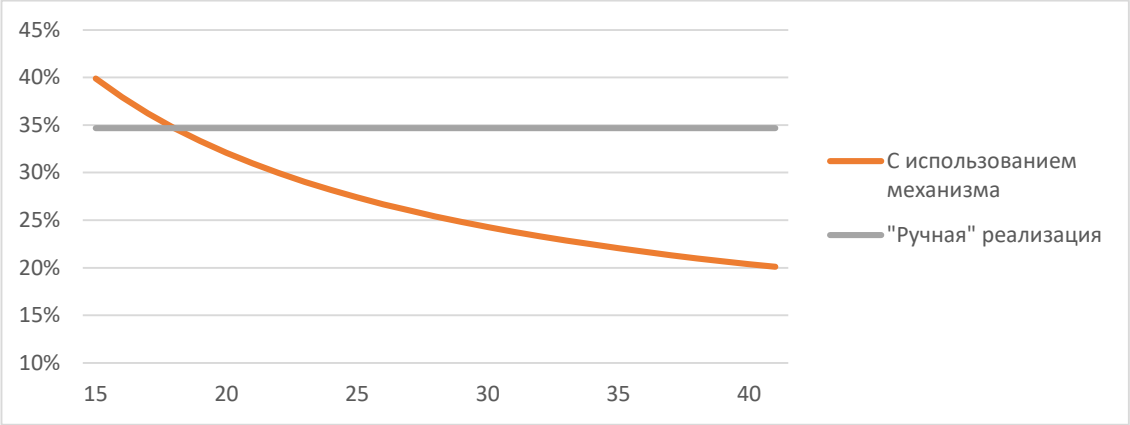


Рисунок 71 – Отношения прикладного исходного кода и исходного кода для верификации

Как видно из приведённых данных – механизм верификации конфигурации позволяет значительно сократить объём исходного кода, что снижает стоимость его поддержки и разработки. Выделение механизмов в системное ПО позволяет повторно использовать его для работы с иными конфигурациями, а это дополнительно сокращает издержки.

Ниже представлен список объектов повторного использования (разработанных при создании ИИС Луч-ТС М), модифицирующих уровневую организацию:

1) Механизм верификации конфигураций.
2) Библиотека для анализа слабо формализованных текстовых данных / Ключев А.О., *Пенской А.В.*, Пинкевич В.Ю. – Свидетельство о государственной регистрации программы для ЭВМ № 2015615882 от 26.05.2015.

Приведённые данные позволяют говорить о том, что изменения уровневой организации позволяют повысить качество архитектурных решений в части уровневой организации и архитектуры систем в целом (расширяемость и адаптивность) и упростить реализацию отдельных элементов системы.

5.3.2 САПР сигнального процессора реального времени NL3

В основе комплекса «Nanoeducator LE» [6], предназначенного для обучения основам нано технологий и проведения научных работ, лежит зондовый микроскоп. Одна из особенностей зондовой микроскопии - необходимость высокоскоростной цифровой обработки сигналов (ЦОС) в реальном времени. Требования к ЦОС: разрядность 32 бита, частота дискретизации сигнала 1 МГц. Типовые алгоритмы: фильтрация; ПИД-регулирование; синхронное детектирование. Дополнительным требованием является необходимость пользовательского программирования.

Для решения задачи ЦОС был предложен и разработан сигнальный процессор реального времени NL3 и сопутствующая САПР [117] (в совокупности – система NL3). Автор данного

диссертационного исследования не принимал участия в разработке архитектуры вычислителя и первой версии САПР, но руководил рабочей группой независимого проекта со сходными концепциями – «стрелочный вычислитель» [118]. Опыт эксплуатации первой версии САПР NL3 показал ряд недостатков: (1) искусственное снижение производительности; (2) непредсказуемый и длительный процесс компиляции; (3) серьёзные ограничения на модификацию вычислителя; (4) ограниченные возможности средств разработки. В совокупности, это не позволило раскрыть потенциал технологии.

Далее будет дано описана система NL3 в текущем состоянии, показаны результаты архитектурного анализа с позиции уровневой организации и требований к модернизации САПР NL3, а также описан разработанный прототипа САПР NL3.

5.3.2.1 Архитектура процессора NL3

Для ЦОС доступны следующие варианты:

1) *Процессор общего назначения.* Не позволяет обеспечить необходимую производительность в сочетании с требованиями реального времени. Избыточен по габариту, энергопотреблению, тепловыделению.
2) *Специализированный процессор для ЦОС.* Для эффективного использования требуется глубокое понимание разработчиком. Уровень параллелизма варьируется в зависимости от архитектуры.
3) *Графический процессор.* Ориентирован на параллельную обработку нескольких наборов данных с использованием одной программы. Имеет ограничения на алгоритмы ЦОС. В общем случае не применим.
4) *Программируемая логика (ПЛИС)*. Обладает неприемлемой сложностью с точки зрения программирования. В остальном, удовлетворяет требованиям.
5) *Высокоуровневый синтез.* Как правило, это генерация конфигураций для ПЛИС из программ на языках программирования высокого уровня. Отличается высокой сложностью (а зачастую и стоимостью) инструментальных средств. Требует глубокого понимания инструментальной цепочки, в противном случае неэффективен.

Так как ни один из вариантов не отвечает запросам, был предложен промежуточный вариант между специализированным процессором для ЦОС и высокоуровневым синтезом. От первого взята идея использования специализированного программируемого вычислителя, от второго – идеология высокоуровневого программирования, сосредоточение сложности в САПР

и гибкость аппаратной составляющей. Данный вариант характеризуется следующими параметрами:

1) Использование специализированного процессора для ЦОС с минимальной инфраструктурой и гибким набором блоков обработки данных (БОД), допускающего автоматический подбор состава БОД и генерацию вычислителя под алгоритм.
2) Использование ПЛИС в качестве ВПЛ. Обеспечивает низкую стоимость при мелкосерийном производстве и гибкость аппаратной составляющей.
3) Диспетчеризация вычислительного процесса выполняется статически для минимизации аппаратной составляющей и обеспечения реального времени.
4) Сложность системы ЦОС сосредоточена в БОД и САПР.
5) Прикладное программирование осуществляется с использованием графического языка программирования, ориентированного на неспециалистов в области вычислительной техники.

В качестве архитектуры была выбрана гибридная *TTA* / *NISC* архитектура (Transport Triggered Architecture [119], архитектура, ориентированная на пересылку данных / Not Instruction Set Controller [120], контроллер без системы команд). На Рисунок 72 схематично представлены архитектуры: фон Неймановского и суперскалярного процессора, TTA и NL3 вычислителей. Используемые обозначения:

– арифметически-логическое устройство (АЛУ);
– блок обработки данных (БОД). Является общим случаем АЛУ, которым может быть ячейка памяти, регистр, контроллер ввода/вывода и др.;
– регистр аккумулятора (АКК).

Элементы TTA архитектуры обеспечили: высокий уровень параллелизма; адаптивность под условия конкретной задачи за счёт изменяемого состава БОД; отсутствие «бутылочного горлышка» фон Неймановской архитектуры – памяти [86]. Уровень параллелизма ограничен прикладным алгоритмом и общей шиной. Количество шин и топология связи БОД могут меняться.

Элементы NISC архитектуры снизили сложность вычислителя (вырождение модуля управления до сдвигового регистра); убрали унификацию БОД, присущую TTA архитектуре и накладывающую ограничения на их разработчика. Основным недостатком стала низкая плотность исходного кода.

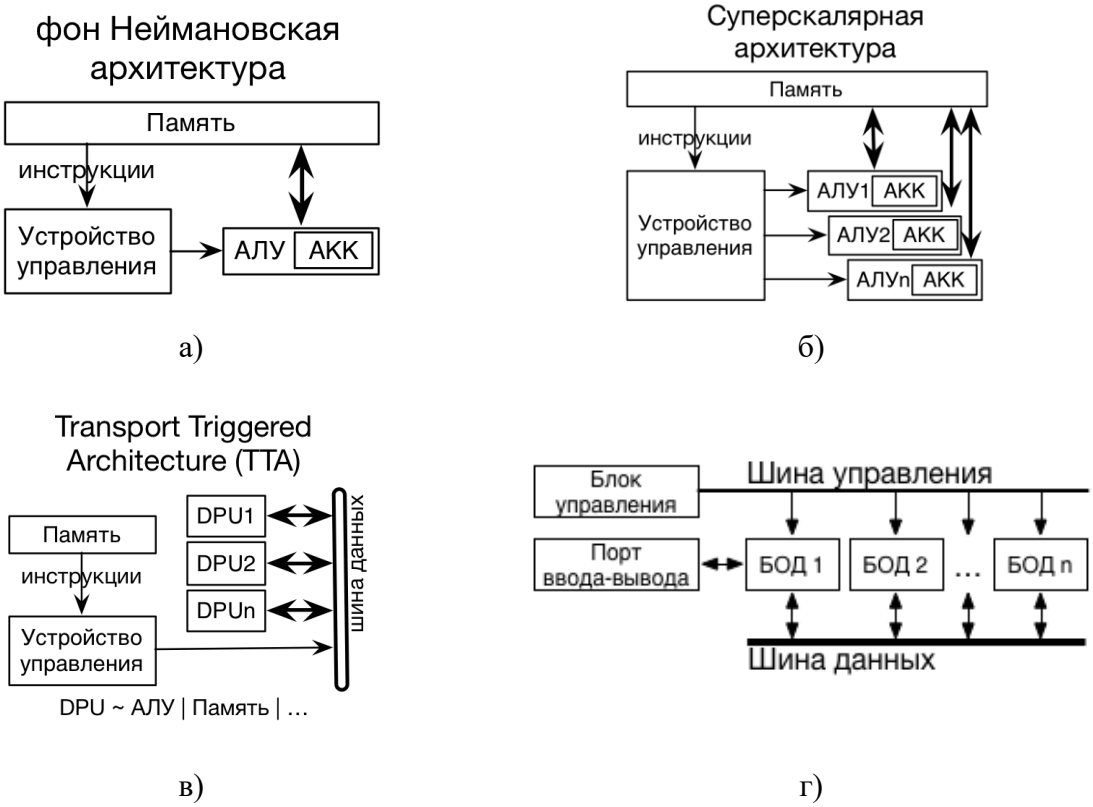


Рисунок 72 – Вычислители с: а) фон Неймановской архитектурой; б) суперскалярной архитектурой в) TTA архитектурой; г) архитектурой NL3

Главным (ожидаемым) недостатком гибридной архитектуры стала высокая сложность САПР. Сложность обусловлена следующими факторами:

1) Язык высокого уровня должен выражать параллелизм вычислительного процесса без ограничений.
2) Отсутствие унификации интерфейса БОД должно быть поддержано САПР без искусственных ограничений. Другими словами, представление БОД должно быть полно по Тьюрингу.
3) САПР не должен быть завязан на конкретную топологию среды передачи данных между БОД.
4) САПР должна отображать прикладной алгоритм на БОД с учётом:
   - дублирования БОД;
   - многофункциональности БОД;
   - возможности сочетания множества функций на одном БОД;
   - прикладного алгоритма;
   - возможных оптимизаций прикладного алгоритма или способа организации вычислительного процесса.

Ключевыми элементами САПР NL3 являются: (1) модель БОД; (2) язык описания прикладных алгоритмов; (3) компилятор.

Модель БОД текущей версии САПР описывает вычислительные возможности блока обработки данных. БОД представляется как функциональный блок без внутреннего состояния, где вход однозначно определяет выход. Длительность вычислительного процесса не зависит от данных. Модель БОД включает:

1) Описание шины управления. Количество сигнальных линий и их разрядность не нормируются.
2) Описание входных и выходных портов (абстракций, характеризующих загрузку или выгрузку операндов вычислительного процесса). Для входных данных фиксируется длительность загрузки, для выходных – задержка до выставления значения на шину и длительность хранения значения.
3) Описание входного и выходного терминала, объединяющих порты в последовательности операций БОД. Использование терминалов определяет вычислительный процесс.

Такая модель позволяет описать вычислительный процесс как систему неравенств:

$$\begin{cases} 3 \le n \le 14 \\ 4 \le n_1 \le 15 \\ 12 \le n_2 \le 23 \\ n_1 - n_0 \ge 1 \\ n_2 - n_1 \ge 8 \end{cases}$$

где n_i – момент времени, в который активен один из портов БОД, а целые числа – такты вычислительного процесса. Любое решение данного уравнения позволяет механически сформировать программу для БОД.

Описание прикладных алгоритмов осуществляется с использованием графического языка, в основе которого лежит понятие *логического БОД*. Логический БОД идентичен физическому и отображаться компилятором на него. Количество логических БОД ограничено вычислительными ресурсами и доступным временем. Пример алгоритма приведён на Рисунок 73.

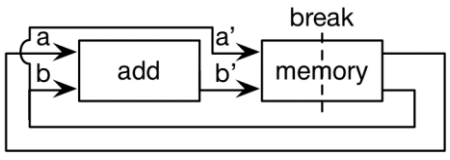


Рисунок 73 – Расчёт последовательности Фибоначчи

Семантика описания прикладного алгоритма соответствует модели вычислений синхронных потоков данных (SDF [54]), за исключением свойства «разрыва» (на рисунке – «break»). Это свойство необходимо для переноса значений на следующий вычислительный цикл. Визуализация вычислительного процесса и его циклов проведена на Рисунок 74 (заимствовано из работы [6]), где «SOS» - начало, «EOS» - окончание вычислительного цикла. По горизонтали отмечены такты вычислительного процесса. По вертикали показаны физические БОД. Прямоугольниками обозначен ВП логических БОД.

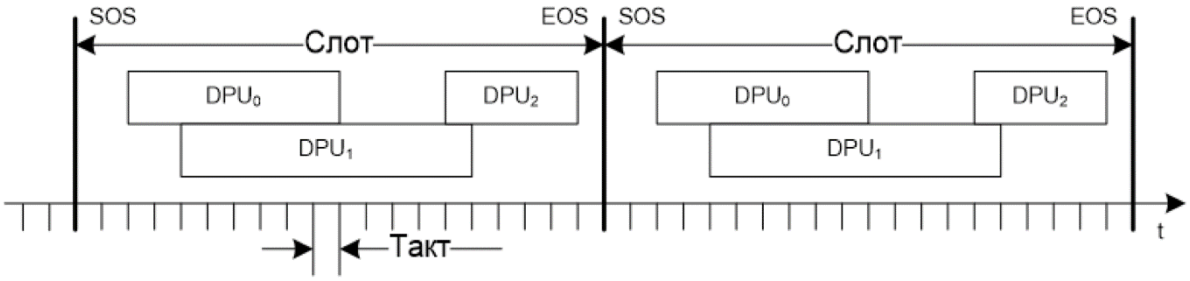


Рисунок 74 – Структура вычислительного процесса NL3

Компилятор вычислителя NL3 работает следующим образом: основываясь на описании прикладного алгоритма, генерируется множество систем уравнений, описывающих выполнение логических БОД. Затем производится их объединение в единую систему уравнений с добавлением выражений, описывающих передачи данных. Решение полученной системы уравнений механически преобразуется в исполняемый код.

Процесс решения системы уравнений основан на переборе. Вычислительная сложность $O(n,t) = t^n$, где n – количество работающих портов, t – длительность вычислительного цикла. Алгоритм имеет экспоненциальную сложность, а применение эвристических методов затруднено ввиду утери концептуальной информации об устройстве вычислителя и прикладном алгоритме.

5.3.2.2 Анализ уровневой организации

Система NL3 является МВА. Чем больше уровней доступно пользователю для конфигурирования, тем выше эффективность создаваемых им решений:

1) Уровень прикладного алгоритма. Пользователь решает задачу ЦОС с помощью языка высокого уровня.
2) Уровень модели БОД. Модель описывает вычислительные возможности БОД, а также способ генерации исполняемого кода для компилятора.
3) Уровень конфигурации процессора. Он характеризует состав БОД и их подключение к шине управления.

4) Уровень программы процессора NL3. Машинный код, организующий вычислительный процесс в соответствии с прикладным алгоритмом. Выделен в отдельный уровень, так как:
   – является необходимым для работы системы;
   – не накладывает ограничений на вычислительный процесс.
5) Уровень БОД. Он описывает реализацию БОД с использованием языка описания аппаратуры.

Нижележащие уровни не рассматриваются, так как они недоступны пользователю. Данная уровневая организация имеет следующие недостатки:

1) Независимое описание и трансляция логических БОД для одного физического БОД препятствует генерации эффективного исполняемого кода.
2) Модель БОД не является полной по Тьюрингу. Это накладывает искусственные ограничения на организацию вычислительного процесса.
3) Использование системы уравнений для процесса компиляции имеет высокую вычислительную сложность и препятствует применению эвристических оптимизаций.
4) Одномоментная трансляция пользовательской программы в систему неравенств препятствует итеративной оптимизации исходной программы.

Также необходимо отметить высокую сложность отслеживания межуровневых взаимосвязей, что препятствует отладке и развитию технологии.

Для решения этих проблем были использованы результаты данного диссертационного исследования (архитектурный стиль МПВ, методика проектирования и методика моделирования). Они позволили не только модернизировать уровневую организацию, но и разработать прототип нового САПР NL3.

5.3.2.3 Модернизация уровневой организации

Архитектура САПР NL3, в значительной степени, определяется моделью БОД. К ней предъявляются следующие требования:

1) Полнота по Тьюрингу. Необходима для оптимального использования БОД без искусственных ограничений.
2) Ориентация на эвристические алгоритмы. Необходима для сокращения вычислительной сложности задачи диспетчеризации.

3) Высокоуровневое пользовательское программирование без сопоставления функции и БОД.
4) Трассировка и визуализация межуровневых взаимосвязей.

Для генерации исполняемого кода выбран подход, ориентированный на передачу данных. Результатом компиляции является последовательность пересылок данных между разными БОД в конкретные моменты времени, механически транслируемая в исполняемый код. Особенностью подхода является отказ от рассмотрения процесса трансляции, как преобразования представлений программы. Вместо этого осуществляется процесс итеративного моделирования целевого вычислительного процесса (или его вариантов). Без применения эвристических оптимизаций, его вычислительная сложность $O(n) = n!$, что несущественно лучше текущей реализации. Реальная же вычислительная сложность значительно ниже, так как: (1) значительная часть вариантов отсеивается из-за причинно-следственных связей прикладного алгоритма; (2) эвристические оптимизации позволяют производить перебор вариантов интеллектуально, увеличивая вычислительную сложность просчёта одного варианта, но значительно сокращая их число. Примеры эвристик:

– наиболее длинный путь прикладного алгоритма;
– уровень загруженности БОД;
– время ожидания БОД.

Изменяется язык описания прикладного алгоритма. Язык на основе логических БОД заменяется на язык функциональных блоков ISO 61131. Это позволяет:

1) Использовать средства разработки из области систем автоматизации.
2) Отказаться от описания вычислительного цикла через аннотирование логических БОД в пользу явного выделения начала и конца (Рисунок 75). Осуществлять выбор БОД в процессе компиляции.
3) То, что функциональный блок (ФБ) не привязан к физическому БОД, позволяет одновременно исполнять несколько ФБ на одном БОД (если это технически возможно) и автоматизировать распределение ФБ.

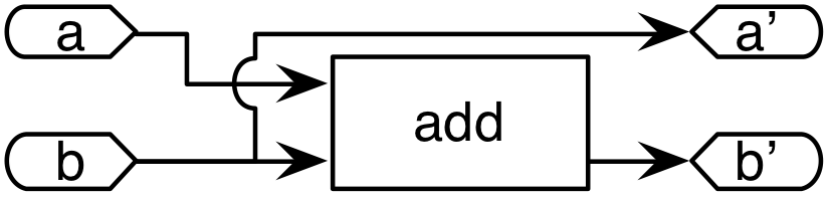


Рисунок 75 – Расчёт последовательности Фибоначчи

Описанный подход к трансляции позволяет использовать модели БОД, полные по Тьюрингу (не имеющие искусственных поведенческих ограничений). Он фиксирует минимум два уровня БОД:

1) Низкоуровневое представление, описывающее БОД как совокупность состояний входов и выходов, соответствующих физическим интерфейсам. Определены следующие состояния: на вход подаётся константа; передаётся значение переменной (используется для шины данных); значение не определено. Данное представление используется для генерации исполняемого кода.
2) Высокоуровневое представление, описывающее реализуемые БОД функциональные блоки.

Для вспомогательных нужд, как правило, разрабатывается дополнительное промежуточное представление. Также модель должна реализовать программный интерфейс для взаимодействия с компилятором (описание интерфейса приведено на языке программирования Rust):

```
trait DPU {
    fn is_over(&self) -> bool;
    fn tick(&self) -> usize;
    fn io_variants(&self) -> Vec<IO>;
    fn time_constrain(&self, io: &IO) -> TimeConstrain;
    fn process_step(&mut self, time_offset: usize, duration: usize, io: &IO);
    fn dependency(&self) -> Vec<(&String, &String)>;
    fn add(&mut self, cmd: &Any);
}
```

Реализация данного интерфейса превращает модель БОД из спецификации в узкоспециализированный компилятор. Интерфейс позволяет:

1) Получить список возможных пересылок на следующем шаге вычислительного процесса. Список определяется внутренним устройством БОД и прикладным алгоритмом.
2) Получить временные ограничения для заданной пересылки, включая: интервал времени до возможного начала пересылки; длительность; момент времени, после которого данное действие становится невозможным.
3) Выполнить шаг вычислительного процесса с указанной пересылкой, моментом времени и длительностью. Пересылка может быть из БОД в самого себя. Пересылка значения из БОД может быть деструктивной (если более данное значение не требуется) и сохраняющей (значение может быть вычитано повторно).

Процесс компиляции строится на итеративном поиске лучшей пересылки переменной между разными БОД, с учётом прикладного алгоритма, эвристических оптимизаций и текущего

состояния. В случае, если последовательность отбраковывается, производится откат. Генерация исполняемого кода осуществляется путём слияния низкоуровневых программ всех БОД.

Уровневая организация модернизированной системы NL3 представлена на Рисунок 76, где HDL – инструментарий для языка описания аппаратуры (Hardware Description Language). Здесь показан параллелизм уровневой организации процессора в целом и уровневой организации БОД. Необходимо отметить: (1) аппаратная реализация БОД выступает в роли BMX процессора NL3; (2) модель БОД содержит описание функций, используемых при описании алгоритма. Модель БОД является самостоятельным объектом, непосредственно не включаемым в систему NL3 с точки зрения уровневой организации.

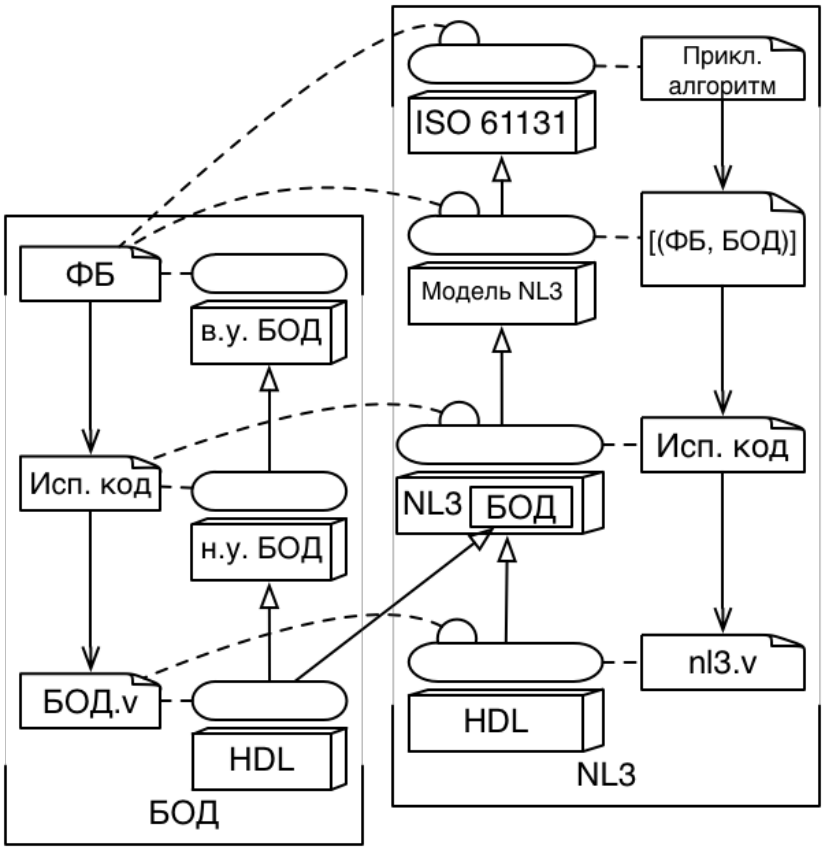


Рисунок 76 – Уровневая организация модернизированной системы NL3

С применением предложенной методики моделирования многоуровневых ВСС и их элементов, реализован прототип новой версии инструментальных средств системы NL3. Элементы исходного кода приведены в приложении Б.1. Итак, применение результатов диссертационного исследования в работе по модернизации САПР NL3 позволило:

1) Провести анализ уровневой организации САПР NL3 при помощи МПВ и поставить задачи по модернизации.
2) Разработать итеративный подход к компиляции, основанный на моделировании вычислительного процесса и эвристических оптимизациях, который значительно

снижает вычислительную сложность при помощи предложенной методики проектирования.

3) Сформировать архитектуру, процедуры и структуры данных САПР NL3 при помощи методики моделирования многоуровневых ВСС, а также снять искусственные ограничения трансляции за счёт использования полной по Тьюрингу модели БОД. Обеспечить трассировку межуровневых взаимосвязей.

5.3.3 Виртуальная машина для беспилотного летательного аппарата

Виртуальная машина (ВМ) для беспилотного летательного аппарата (БПЛА). Обеспечивает пользовательское программирование с детерминированным временем исполнения при фиксированных вычислительных ресурсах. Является МВА. Оценка эффективности производилась путём сопоставления двух вариантов реализации: с проектированием уровневой организации и без проектирования уровневой организации.

БПЛА является автономной системой реального времени. Управляющая программа БПЛА должна описывать миссию, следовательно, регулярно меняться пользователем. Описание миссии не содержит сложных поведенческих алгоритмов и представляет из себя таблицу реакций на те или иные события (например, прибытие в заданные координаты, набор определённой высоты, получение команды и т.д.). Реакция –запуск требуемой подпрограммы. В связи с этим, использование языка программирования общего назначения не подходит, так как приводит к повышению сложности программ; уменьшению безопасности, надёжности и предсказуемости; росту стоимости эксплуатации БПЛА. В то же время, реализация управляющей программы в виде таблицы «событие-подпрограмма» создаёт слишком много ограничений.

В результате был разработан специализированный язык программирования, основанный на модели вычислений конечного автомата. Программа данного языка состоит из конечного числа активируемых по событиям правил и множества пользовательских регистров. Для каждого правила определены следующие опции:

1) Событие или группа событий для активации правила.
2) Условное выражение, проверяющее состояние регистров и, в случае успеха, активирующее правило (опционально).
3) Выражение, определяющее новые значения пользовательских регистров (опционально).
4) Процедура, вызываемая после обновления пользовательских регистров.

Уровневая организация виртуальной машины рассмотрена в пункте 4.1.2.2. Необходимость конфигурирования в рамках нескольких ВПЛ позволяет отнести её к классу МВА. Методика моделирования многоуровневых ВСС и их элементов не применялась. Исходный код модуля приведён в приложении Б.2.

Альтернативная реализация ВМ для управления БПЛА без проектирования уровневой организации, при сопоставим объёме исходного кода и ресурсах на разработку, обладала меньшей производительностью. Также, её использование столкнулось с ограничениями пользовательского конфигурирования, отсутствующими в описанной выше реализации.

5.3.4 САПР для реконфигурируемого фон Неймановского процессора

САПР для работы со специализированным реконфигурируемым фон Неймановским процессором включает: макроассемблер с развитой системой типов и механизмами генерации исполняемого кода; компилятор для специализированного процессора «SysBoot» [53].

Основным назначением специализированного реконфигурируемого процессора SysBoot является задача синхронизации и передачи данных между вычислительными узлами системы на кристалле. Его архитектура отличается возможностью подключения/отключения блоков специального назначения, взаимодействие с которыми производится через внутренние регистры. Для эффективного использования процессора необходим САПР, позволяющий работать с исходным кодом, конфигурировать процессор, компилировать исполняемый код. Для этого были разработаны:

1) Макроассемблер с развитой системой типов, системой контроля корректности исходного кода. Это позволило обеспечить развитые средства мета-программирования.
2) Система генерации исполняемого кода, решающая задачи компиляции и позволяющая трассировать межуровневые взаимосвязи.

Разработанный САПР имеет три уровня:

1) Уровень макроассемблера, в рамках которого реализуются высокоуровневые элементы программирования (выделение регистров, меток, констант, структурные блоки и т.д.).
2) Уровень ассемблера, однозначно отображаемый в машинный код, в рамках которого определены адреса, сдвиги, выделены необходимые ресурсы.
3) Уровень машинного кода.

Реализация выполнена в соответствии с моделями, описанными в разделе 3.7.1. Детали реализации являются специфичными для используемых инструментальных средств.

Отношение трансляции представлено в соответствии с разделом 3.5. Исключение составляет то, что делается специализация под конкретную конфигурацию вычислителя (`opt`).

```
class Translate opt l1 l2 where
    translate :: opt -> l1 -> l2
```

Реализация трансляции между уровнями макроассемблера и ассемблера построена с учётом принципа полиморфизма, в рамках которого разработчик может управлять наличием или отсутствием BMX в составе процессора. Реализация трансляции в машинный код реализована с использованием теоремы 1. Конкретно: реализована трансляция всех атомарных морфизмов модели $\mathcal{M}$ и способ композиции морфизмов. Описание платформо-зависимой части транслятора выглядит, как показано ниже (фрагмент):

```
instance Translate () (SysBootIS IAddr PAddr PData) SysBootCode where
    translate () (SACC_i  (IAddr i)          ) = SysBootCode 09 i  00000000
    translate ()  SACCI                        = SysBootCode 33 00 00000000
    translate ()  LACCI                        = SysBootCode 32 00 00000000
    translate () (STR_id  (IAddr i) (PData d)) = SysBootCode 01 i  d
```

Использование методики моделирования многоуровневых ВСС и их элементов позволило создать прототип САПР в сжатые сроки. В совокупности с использованием языка программирования «Haskell», это позволило обеспечить высокий уровень доверительности разработанного ПО. Наиболее интересные фрагменты исходного кода прототипа САПР приведены в приложении Б.3.

5.3.5 Редактор для документирования архитектуры информационно-управляющих систем

Редактор спецификаций технических и программных средств информационно управляющих систем (ИУС) предназначен для разработки документации на технические решения, используемые при проектировании в рамках аспектного проектирования [6,121]. Программный продукт представляет псевдо-текстовый редактор структурированных данных, реализованный на базе технологической платформы MPS [81].

Программа позволяет:

- создавать, редактировать псевдо-текстовые архитектурные описания ИУС;
- определять аспекты проектирования с произвольной структурой полей;
- определять прикладные и не прикладные аспекты проектирования;
- автоматически контролировать корректность спецификаций;

– использовать аспектные описания со специализированным синтаксисом.

Пример фрагмента архитектурной спецификации приведён на Рисунок 77.

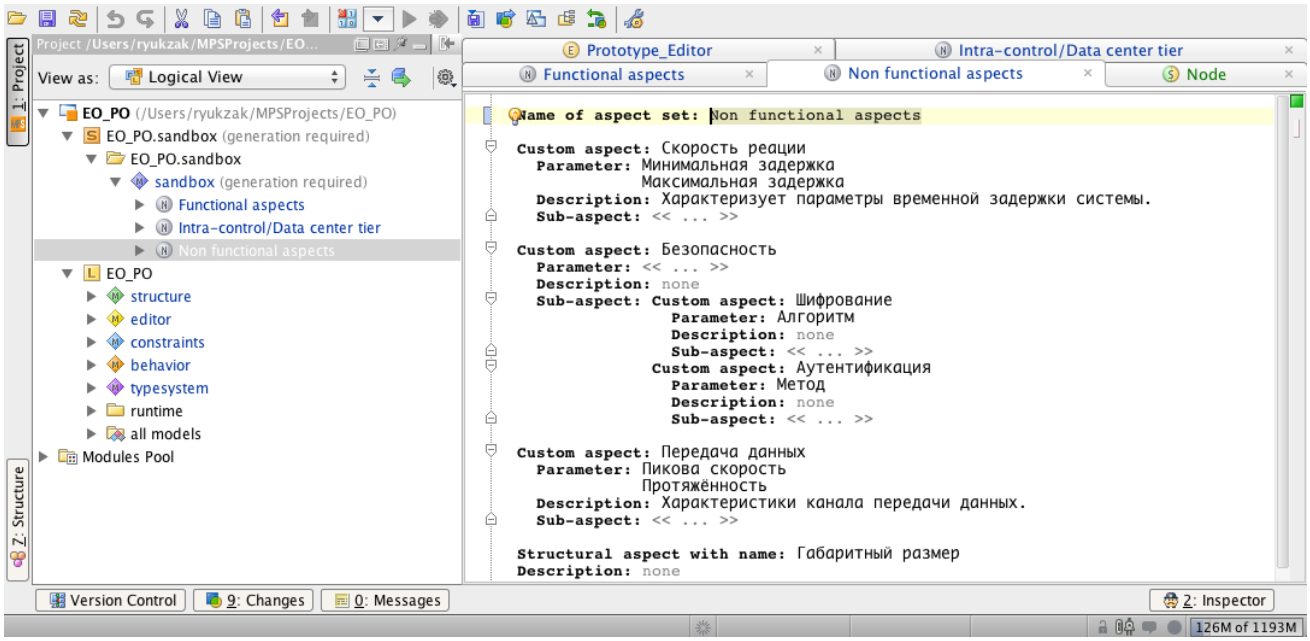


Рисунок 77 – Язык описания аспектов в среде MPS

Следующие механизмы обеспечивают повышение производительности проектировщика:

– автоматическая подсветка и форматирование;

– автоматический контроль синтаксической и семантической корректности;

– автоматическое дополнение (Рисунок 78);

– автоматическая проверка ограничений.

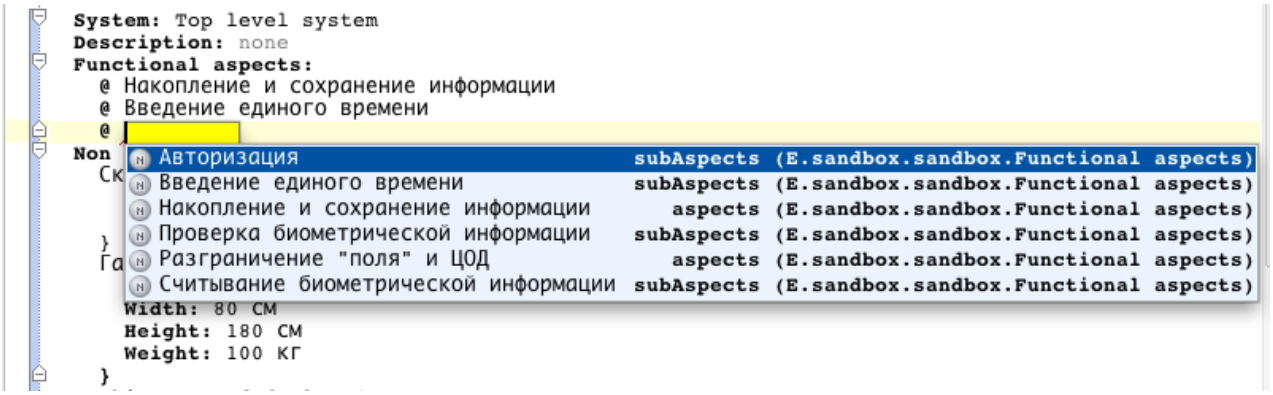


Рисунок 78 – Пример механизма авто-дополнения

Данный инструментарий использует принципы, лежащие в основе предложенной методики проектирования. Элементы в составе редактора являются зарегистрированными программными продуктами:

- Редактор архитектурных спецификаций информационно-управляющих систем. Ключев А.О., Кустарев П.В., Николаенков А.В., *Пенской А.В.* – Свидетельство о государственной регистрации программы для ЭВМ № 2012618605 от 21.09.2012.
- Редактор спецификаций технических и программных средств информационно-управляющих систем / Николаенков А.В., *Пенской А.В.*, Платунов А.Е. – Свидетельство о государственной регистрации программы для ЭВМ № 2012618606 от 21.09.2012.

5.3.6 Библиотека для ввода слабо формализованных данных

При работе с САПР, разработчику необходимо вводить текстовые данные. В виду разнообразия способов записи, удобным является ввод в слабо формализованном виде. Это формирует потребность в реализации анализаторов слабо формализованных данных.

Для решения этой задачи была разработана библиотека, позволяющая в высокоуровневом стиле описывать формат входных данных с высоким уровнем вариативности. Функции и отличительные особенности данной библиотеки:

1) Наглядность синтаксиса позволяет быстро разрабатывать и модифицировать анализаторы. Модульность упрощает их поддержку и возможность повторного использования.
2) Ориентация на работу со структурированными данными упрощает проектирование.
3) Библиотека позволяет совмещать анализ текстовых данных с предварительной обработкой. Это сокращает объём исходного кода, улучшая его структуру.
4) Доступны механизмы для реконфигурации анализатора во время эксплуатации.
5) Возможно расширение стандартной библиотеки анализа на языках программирования Clojure и Java.

В основе библиотеки лежит разработанный уровень организации вычислительного процесса, позволяющий комбинировать между собой анализаторы. Вычислительный процесс анализатора может закончиться успехом (шаблон совпал, данные извлечены) или ошибкой. Комбинация анализаторов осуществляется тремя операторами:

1) «<>>>» Аргумент: список анализаторов. Результат: анализатор, последовательно выполняющий анализаторы-аргументы. В случае если один из анализаторов-аргументов завершится ошибкой, результирующий анализатор также завершится с ошибкой.

2) «&&&» Аргумент: список анализаторов. Результат: анализатор, применяющий анализаторы-аргументы параллельно к входным данным. Результаты применения анализаторов аргументов объединяются в кортеж. В случае если один из анализаторов-аргументов завершится ошибкой, результирующий анализатор также завершится с ошибкой.
3) «<+>» Аргумент: список анализаторов. Результат: анализатор, последовательно применяющий анализаторы-аргументы к входным данным, пока один из них не завершится успешно. Результат первого успешно завершившегося анализатора-аргумента станет результатом результирующего анализатора.

В качестве примера использования работы библиотеки приведено описание разбора телефонного номера.

```
(def tel-p (parse
  (char-p "(")
  (code <- (take-p digit-p 3))
  (char-p ")")
  space-p
  (number <- (take-p (ignore-p (char-p "-") digit-p) 7))
  end-p
  (output {:code code, :phone number}))
```

Данный анализатор может быть применён к строке следующим образом:

```
(run-arrow tel-p "(812) 123-45-67")
```

Результатом данного вычисления будет:

```
{:code "812" :phone "1234567"}
```

Данная библиотека является зарегистрированными программным продуктом: Библиотека для анализа слабо формализованных текстовых данных / Ключев А.О., *Пенской А.В.*, Пинкевич В.Ю. – Свидетельство о государственной регистрации программы для ЭВМ № 2015615882 от 26.05.2015.

5.4 Выводы

1) Проанализирована проблема оценки архитектурных стилей. Предложен подход к их анализу, основанный на попарной сравнительной характеристике по частным критериям.
2) Проведён сравнительный анализ уже известных и предложенных в рамках данной диссертации архитектурных стилей для проектирования и документирования уровневой организации. По результатам анализа даны практические рекомендации по использованию архитектурных стилей.

3) Показано, что архитектурный стиль «модель-процесс-вычислитель» отличается полнотой покрытия пространства проектных решений, высокой детализацией уровневой организации, отсутствием избыточности и однозначностью используемой системы понятий. Показано, что применение данного архитектурного стиля позволяет повысить качество архитектурных решений в части уровневой организации.
4) Проанализирован опыт использования результатов диссертационного исследования в практических задачах. В анализ были включены работы над следующими проектами:

   – система коммерческого учёта электроэнергии «ИИС Луч-ТС М»;
   – САПР сигнального процессора реального времени NL3;
   – САПР для реконфигурируемого фон Неймановского процессора;
   – система управления беспилотным летательным аппаратом;
   – редактор для документирования архитектуры информационно-управляющих систем;
   – библиотека для ввода слабо формализованных данных в компонентах САПР.

Проведенный анализ опыта использования результатов диссертационного исследования подтвердил повышение качества архитектурных решений в части уровневой организации встроенных систем и сокращение затрат на разработку заказных элементов уровневой организации.

Заключение

Центральным результатом диссертационной работы является *архитектурный стиль «модель-процесс-вычислитель»*, основанный на принципах моделирования высших онтологий. Он характеризуется специализацией на вопросах проектирования и документирования уровневой организации встроенных систем, полнотой представления пространства проектных решений и высоким уровнем детализации при минимальной избыточности. В совокупности с предложенной системой архитектурных абстракций и методикой проектирования, он позволяет повысить качество принимаемых архитектурных решений, эффективность и скорость проектирования, степень формализации проектного опыта. Его формализация в рамках *методики моделирования* многоуровневых встроенных систем и их элементов позволяет сократить затраты на разработку САПР в составе заказных элементов уровневой организации за счёт наличия готовых процедур моделирования, снижения рисков архитектурного проектирования САПР и повторного использования.

В процессе разработки архитектурного стиля «модель-процесс-вычислитель» и на его основе были получены следующие выводы и результаты:

1) *Определена структура пространства проектных решений* в части уровневой организации встроенных систем. *Сформированы требования* к архитектурному стилю для проектирования и документирования уровневой организации.
2) *Предложены архитектурные стили* для проектирования и документирования уровневой организации встроенных систем: «*модифицированный граф актуализации*», «*системно-иерархический*» и «*модель-процесс-вычислитель*».
3) Разработана *методика моделирования* многоуровневых встроенных систем и их элементов.
4) Расширена *система архитектурных абстракций*. Введено понятие *многоуровневого вычислительного агрегата*; уточнены понятия *вычислительной* и *системной платформы, уровня* и *уровневой организации*; разработаны *онтологические модели [ре]конфигурации* и *иерархической организации*.
5) Предложена *методика проектирования встроенных систем*, выделяющая этап проектирования уровневой организации и обобщающая результаты диссертационного исследования.
6) Показано превосходство архитектурного стиля «модель-процесс-вычислитель» над аналогами в задачах проектировании встроенных систем с нетривиальной уровневой

организацией малыми и средними коллективами, а также влияние архитектурного стиля на качество принимаемых архитектурных решений.

Проведена успешная апробация результатов работы в следующих проектах: автоматизированная система коммерческого учёта электроэнергии ИИС «Луч-ТС М», САПР сигнального процессора реального времени NL3, виртуальная машина для управления беспилотным летательным аппаратом, САПР для реконфигурируемого фон Неймановского процессора, редактор для документирования архитектуры информационно-управляющих систем, библиотека для ввода слабо формализованных данных в компонентах САПР. Это подтвердило теоретические выводы работы.

Перспективным направлением для развития работ по теме диссертации является создание полномасштабной САПР для моделирования и разработки многоуровневых встроенных систем и их элементов на основе предложенных архитектурных стилей и методики моделирования.

Список работ, опубликованных по теме диссертации в изданиях из перечня ВАК или индексируемых SCOPUS:

1) Platunov A. Expanding Design Space for Complex Embedded Systems with HLD-methodology / A. Platunov, A. Kluchev, *A. Penskoi* // International Congress on Ultra Modern Telecommunications and Control Systems and Workshops – 2015. – P. 157-164.
2) *Пенской А.В.* Архитектурное документирование встроенных систем с многоуровневой конфигурацией / А.В. Пенской // Изв. вузов. Приборостроение. – 2015. – Т 58, № 7. – С. 527-532.
3) Kluchev A. HLD methodology in embedded systems design with a multilevel reconfiguration / A. Kluchev, A. Platunov, *A. Penskoi* // Proceedings - 2014 3rd Mediterranean Conference on Embedded Computing, MECO 2014 - Including ECyPS 2014. – 2014. – P. 36-39.
4) Platunov A. The Architectural Specification of Embedded Systems / A. Platunov, *A. Penskoi*, A. Kluchev // Proceedings - 2014 3rd Mediterranean Conference on Embedded Computing, MECO 2014 - Including ECyPS 2014. – 2014. – P. 48-51.
5) Platunov A. HLD Methodology: The Role of Architectural Abstractions in Embedded Systems Design / A. Platunov, A. Kluchev, *A. Penskoi* // International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM – 2014. – P. 209-218.

6) Platunov A. Architectural representation of embedded systems / A. Platunov, A. Nickolaenkov, *A*. *Penskoy* // 2012 Mediterranean Conference on Embedded Computing, MECO 2012 – 2012. – P. 80-83.

Публикации в прочих издания:

7) Ключев А.О. Использование HLD-методологии для проектирования сенсорных сетей / А.О. Ключев, П.В. Кустарев, *А.В. Пенской*, А.Е. Платунов // Сборник трудов II Международной научно-практической конференции «Sensorica-2014». – СПб: Университет ИТМО. – 2014. – С. 35-36.
8) *Пенской А.В.* Поведенческое описание аппаратных блоков обработки данных в не фон Неймановских процессорах / Пенской А.В. // Сборник трудов молодых ученых и сотрудников кафедры ВТ. – вып. 3. – 2012. – С. 30-33.

Учебно-методические публикации:

9) Быковский С.В. Сопряжённое проектирование встраиваемых систем (Hardware/Software Co-Design). Часть 1. / Быковский С.В., Горбачев Я.Г., Ключев А.О., *Пенской А.В.*, Платунов А. Е. // Учебное пособие – СПб: Университет ИТМО. – 2016. – 108 с.
10) Быковский С.В. Сопряжённое проектирование встраиваемых систем (Hardware/Software Co-Design). Часть 2. / Быковский С.В., Горбачев Я.Г., Ключев А.О., *Пенской А.В.*, Платунов А.Е. // Учебное пособие – СПб: Университет ИТМО. – 2016. – 105 с.

Объекты интеллектуальной собственности:

11) Библиотека для анализа слабо формализованных текстовых данных / Ключев А.О., *Пенской А.В.*, Пинкевич В.Ю. – Свидетельство о государственной регистрации программы для ЭВМ № 2015615882 от 26.05.2015.
12) Библиотека для разработки программных интерфейсов управления мобильным оборудованием / Ключев А.О., *Пенской А.В.*, Пинкевич В.Ю. – Свидетельство о государственной регистрации программы для ЭВМ № 2015614306 от 14.04.2015.
13) Редактор архитектурных спецификаций информационно-управляющих систем. Ключев А.О., Кустарев П.В., Николаенков А.В., *Пенской А.В.* – Свидетельство о государственной регистрации программы для ЭВМ № 2012618605 от 21.09.2012.

14) Редактор спецификаций технических и программных средств информационно-управляющих систем / Николаенков А.В., *Пенской А.В.*, Платунов А.Е. – Свидетельство о государственной регистрации программы для ЭВМ № 2012618606 от 21.09.2012.

Список сокращений и условных обозначений

HW аппаратное обеспечение, Hardware

ISA архитектура системы команд, Instruction Set Architecture

NISC контроллер, без системы команд, Not Instruction Set Controller

TTA архитектура, ориентированная на пересылку данных, Transport Triggered Architecture

БПЛА беспилотный летательный аппарат

ВМХ вычислительный механизм

ВП вычислительный процесс

ВПЛ вычислительная платформа

ВСС встроенная система

ИВК информационно-вычислительный комплекс

ИВКУУ информационно-вычислительный комплекс узла учёта

ИИК измерительно-информационный комплекс

ИИС информационно-измерительная система

ИУС информационно-управляющие системы

КФС кибер-физические системы

МВА многоуровневый вычислительный агрегат

МПВ архитектурный стиль «модель-процесс-вычислитель»

НИОКР научно-исследовательская опытно-конструкторская работа

НИР научно-исследовательская работа

ОС операционная система

ПК персональный компьютер

ПЛИС программируемая логическая интегральная схема

ПО, SW программное обеспечение, Software

ППР пространство проектных решений

ПТК программно-технический комплекс

САПР система автоматизированного проектирования

СПЛ системная платформа

ТЗ техническое задание

Список литературы

1. Teich J. Hardware/Software Codesign: The Past, the Present, and Predicting the Future / J. Teich // Proc. IEEE – 2012. – Т. 100– № Special Centennial Issue– 1411–1430с.
2. Ebert C. Embedded software: Facts, figures, and future / C. Ebert, C. Jones // Computer (Long. Beach. Calif). – 2009. – Т. 42 – № 4– 42–52с.
3. Embedded Systems Outlook 2013 – Nuremberg: NIGlobal, 2013.
4. Ключев А.О. Программное обеспечение встроенных вычислительных систем / А. О. Ключев, П. В. Кустарев, Д. Р. Ковязина, Е. В. Петров – СПб ГУ ИТМО, 2009.– 212с.
5. Ключев А.О. Аппаратные и программные средства встраиваемых систем / А. О. Ключев, Д. Р. Ковязина, П. В. Кустарев, А. Е. Платунов – СПб ГУ ИТМО, 2010.– 286с.
6. Платунов А.Е. Теоретические и методологические основы высокоуровневого проектирования встраиваемых вычислительных систем : диссертация ... доктора технических наук : 05.13.12 / Платунов А.Е. - Санкт-Петербург, 2010.- 477 с.: ил. РГБ ОД, 71 11-5/47.
7. Lee E.A. Cyber physical systems: Design challenges / E. A. Lee // Object Oriented Real-Time Distrib. Comput. (ISORC), 11th IEEE Int. Symp. – 2008. – 363–369с.
8. Lee E.A. Introduction to Embedded Systems - A Cyber-Physical System Approach / E. A. Lee, S. A. Seshia – UC Berkeley, 2011.– 491с.
9. Broman D. Viewpoints, formalisms, languages, and tools for cyber-physical systems / D. Broman, E. A. Lee, S. Tripakis, M. Törngren // Proc. 6th Int. Work. Multi-Paradigm Model. - MPM ’12 – 2012. – Т. 931843– 49–54с.
10. Sheth A. Physical-Cyber-Social Computing: An Early 21st Century Approach / A. Sheth, P. Anantharam, C. Henson // IEEE Intell. Syst. – 2013. – Т. 28 – № 1– 78–82с.
11. Sangiovanni-Vincentelli A. Platform-based design and software design methodology for embedded systems / A. Sangiovanni-Vincentelli, G. Martin // IEEE Des. Test Comput. – 2001. – Т. 18 – № 6– 23–33с.
12. Lee E.A. Actor-Oriented Design of Embedded Hardware and Software Systems / E. A. Lee, S. Neuendorffer, M. J. Wirthlin // J. Circuits, Syst. Comput. – 2003. – Т. 12– 231–260с.

13. Platunov A. Expanding Design Space for Complex Embedded Systems with HLD-methodology / A. Platunov, A. Kluchev, A. Penskoi // 2014 6th Int. Congr. Ultra Mod. Telecommun. Control Syst. Work. - Telecommun. – 2015. – 157–164c.

14. Jia Z.J. A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC / Z. J. Jia, A. Núñez, T. Bautista, A. D. Pimentel // Microprocess. Microsyst. – 2014. – Т. 38 – № 1– 9–21c.

15. Pomante L. System-level design space exploration for application-specific HW/SW systems / L. Pomante, L. Imbriglio, F. Graziosi // TIC-STH'09 2009 IEEE Toronto Int. Conf. - Sci. Technol. Humanit. – 2009. – 569–574c.

16. Pimentel A.D. A systematic approach to exploring embedded system architectures at multiple abstraction levels / A. D. Pimentel, C. Erbas, S. Polstra // IEEE Trans. Comput. – 2006. – Т. 55 – № 2– 99–111c.

17. Мизес Л. Человеческая деятельность. Трактат по экономической теории / Л. Мизес – Социум, 2005.

18. Berger A.S. Embedded Systems Design: An Introduction to Processes, Tools and Techniques / A. S. Berger – CMP Books, 2001.– 237c.

19. Baldwin C.Y. Design Rules: The Power of Modularity Volume 1 / C. Y. Baldwin, K. B. Clark – MIT Press, 2000.– 471c.

20. Clements P. Documenting Software Architectures / P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford – Carnegie Mellon, 2010. Вып. 2.

21. Freeman E. Head First Design Patterns / E. Freeman, E. Freeman – Carcinogenesis, 2013.– 619c.

22. Gamma E. Design patterns: elements of reusable object-oriented software / E. Gamma, R. Helm, R. E. Johnson, J. Vlissides – Addison-Wesley Professional, 1995.– 395c.

23. Atkinson C. A Generalized Notion of Platforms for Model-Driven Development Springer Berlin Heidelberg, 2005. – 119–136c.

24. Baldwin C.Y. The Architecture of Platforms: A Unified View. / C. Y. Baldwin, C. J. Woodard // Work. Pap. -- Harvard Bus. Sch. Div. Res. – 2008. – 31c.

25. Jóźwiak L. Modern architectures for embedded reconfigurable systems — A survey / L. Jóźwiak, N. Nedjah // J. Circuits, Syst. Comput. – 2009. – Т. 18 – № 2– 209–254c.

26. Hartenstein R. The Relevance of Reconfigurable Computing / R. Hartenstein // Reconfigurable Comput. – 2011. – 7–34с.

27. Platunov A. HLD Methodology: The Role of Architectural Abstractions in Embedded Systems Design / A. Platunov, A. Kluchev, A. Penskoi // 14th GeoConference Informatics, Geoinformatics Remote Sens. – 2014. – 209–218с.

28. ГОСТ Р ИСО/МЭК 15288-2005 Информационная технология. Системная инженерия. Процессы жизненного цикла систем. – 54с.

29. Boehm B. The ROI of systems engineering: Some quantitative results for software-intensive systems / B. Boehm, R. Valerdi, E. Honour // Syst. Eng. – 2008. – Т. 11 – № 3– 221–234с.

30. Tassey G. The Economic Impacts of Inadequate Infrastructure for Software Testing / G. Tassey – 2002.

31. Kiczales G. Aspect-oriented programming / G. Kiczales, E. Hilsdale // ACM SIGSOFT Softw. Eng. Notes – 2001. – Т. 26 – № 5– 313с.

32. Clements P.C. A survey of architecture description languages / P. C. Clements // Proc. 8th Int. Work. Softw. Specif. Des. – 1996. – 16–25с.

33. Teich J. Invasive computing: An overview / J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, G. Snelting // Multiprocessor Syst. Hardw. Des. Tool Integr. – 2011. – 241–268с.

34. Dijkstra E.E.W. EWD 447: On the role of scientific thought / E. E. W. Dijkstra // Sel. Writings Comput. A Pers. Perspect. – 1982. – 60–66с.

35. Platunov A. Architectural representation of embedded systems / A. Platunov, A. Nickolaenkov, A. Penskoy // 2012 Mediterr. Conf. Embed. Comput. – 2012. – 80–83с.

36. Das S. Synthesis of system verilog assertions / S. Das, R. Mohanty, P. Dasgupta, P. P. Chakrabarti // Proc. -Design, Autom. Test Eur. DATE – 2006. – Т. 2.

37. Lee E.A. The problem with threads / E. A. Lee // Computer (Long. Beach. Calif). – 2006. – Т. 39 – № 5– 33–42с.

38. Левенчук А. Системноинженерное мышление в управлении жизненным циклом / А. Левенчук – TechInvestLab.ru, 2015.– 306с.

39. Essence – Kernel and Language for Software Engineering Methods – Foundation for the Agile Creation and Enactment of Software Engineering Methods (FACESEM) RFP, 2012.– 207с.

40. OMG Unified Modeling Language TM (OMG UML), Superstructure v.2.3 // InformatikSpektrum – 2010. – T. 21– 758c.

41. Hause M. The SysML Modelling Language / M. Hause // Fifteenth Eur. Syst. Eng. Conf. – 2006. – № September – 12c.

42. Mellor S.J. Executable UML: A Foundation for Model-driven Architecture / S. J. Mellor, M. J. Balcer – Addison-Wesley Professional, 2002.

43. Kroll P. Rational Unified Process Made Easy: A Practitioner's Guide to the RUP / P. Kroll, P. Kruchten – , 2003.– 1-386c.

44. Capella | PolarSys - Open Source Tools for Embedded Systems [Электронный ресурс]. URL: https://www.polarsys.org/proposals/capella.

45. Partridge C. Business Objects: Re-engineering for Re-use / C. Partridge – Butterworth-Heinemann, 1996.– 442c.

46. ISO 15926-2, Industrial automation systems and integration Integration of life-cycle data for process plants including oil and gas production facilities Part 2: Data model – 2003.

47. DoDAF Architecture Framework Version 2.2 // Deputy Chief Information Officer [Электронный ресурс]. URL: http://dodcio.defense.gov/dodaf20.aspx.

48. Feiler P.H. The Architecture Analysis & Design Language (AADL): An Introduction / P. H. Feiler, D. P. Gluch, J. J. Hudak – Software Engineering Institute, 2006.

49. Lankhorst M.M. The architecture of the ArchiMate language / M. M. Lankhorst, H. A. Proper, H. Jonkers // Lect. Notes Bus. Inf. Process. – 2009. – T. 29 LNBIP – 367–380c.

50. Wolf W. A decade of hardware/software codesign / W. Wolf // Computer (Long. Beach. Calif). – 2003. – T. 36 – № 4– 38–41+4c.

51. Lockwood A.J. Understanding the Total Cost of Embedded Design / A. J. Lockwood // Deskt. Eng. – 2012.

52. Keutzer K. System-level design: Orthogonalization of concerns and platform-based design / K. Keutzer // IEEE Trans. Comput. Des. Integr. Circuits Syst. – 2000. – T. 19 – № 12– 1523–1543c.

53. Pinkevich V. Using architectural abstractions in embedded system design / V. Pinkevich, A. Platunov // Proc. 4th Mediterr. Conf. Embed. Comput. (MECO 2015). Work. Prog. Embed. Comput. (CD ROM) – 2015. – T. 1– 3–6c.

54. Bhattacharyya S. Vol. 1 - Ptolemy 0.7 User's Manual / S. Bhattacharyya, J. T. Buck, W.-T. Chang, M. J. Chen, B. L. Evans, et al. – California: College of engineering department of electrical engineering and computer sciences Berkeley, 1997.– 532c.

55. Baldin D. Proteus, a hybrid virtualization platform for embedded systems / D. Baldin, T. Kerstan // Archit. Model. Embed. Syst. – 2009. – 185–194c.

56. Grötker T. System design with SystemC / T. Grötker, S. Liao, G. Martin, S. Swan – Springer, 2002.– 236c.

57. ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description – 2011.

58. Seibel P. Practical common lisp / P. Seibel – Apress, 2005.– 499c.

59. Goldberg A. Smalltalk-80 : the interactive programming environment / A. Goldberg – Addison-Wesley Professional, 1984.– xi, 516 c.

60. Platunov A. Problems of Abstract Representation of Embedded Systems at High-level Stages Design f / A. Platunov, P. Kustarev // Proc. Int. Work. Networked Embed. Control Syst. Technol. Eur. Russ. R&D Coop. – 2009. – 100–107c.

61. Platunov A. The Architectural Specification of Embedded Systems / A. Platunov, A. Penskoi, A. Kluchev // 3rd Mediterr. Conf. Embed. Compuning – 2014.

62. Ahmed E. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density / E. Ahmed, J. Rose // IEEE Trans. Very Large Scale Integr. Syst. – 2004. – T. 12– 288–298c.

63. Brunelli C. A coarse-grain reconfigurable architecture for multimedia applications featuring subword computation capabilities / C. Brunelli, F. Garzia, J. Nurmi // J. Real-Time Image Process. – 2008. – T. 3– 21–32c.

64. Burleson W. VLSI Interconnects: A Design Perspective / W. Burleson, A. Maheshwari – San Francisco: Elsevier/Morgan Kaufman, 2008.

65. Chattopadhyay A. Ingredients of Adaptability: A Survey of Reconfigurable Processors / A. Chattopadhyay // VLSI Des. – 2013. – T. 2013– 18c.

66. Qu Y. System-Level Design for Partially Reconfigurable Hardware / Y. Qu, K. Tiensyrja, J.-P. Soininen, J. Nurmi // 2007 IEEE Int. Symp. Circuits Syst. – 2007.

67. Smith J.E. The architecture of virtual machines / J. E. Smith, R. Nair // Computer (Long. Beach. Calif). – 2005. – T. 38 – № 5– 32–38c.

68. Würthinger T. Multi-level virtual machine debugging using the Java Platform debugger architecture / T. Würthinger, M. L. Van De Vanter, D. Simon // Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) – 2010. – T. 5947 LNCS – 401–412c.

69. Poole J.D. Model-Driven Architecture : Vision , Standards And Emerging Technologies / J. D. Poole // Work. Metamodeling Adapt. Object Model. ECOOP01 – 2001. – № April – 15c.

70. Model Driven Architecture / OMG – 2000.– 12c.

71. Davare A. A Next-Generation Design Framework for Platform-Based Design / A. Davare, D. Densmore, T. Meyerowitz, A. Pinto // DVCon 2007 – 1959. – 927–954c.

72. Densmore D. A platform-based taxonomy for ESL design / D. Densmore, R. Passerone, A. Sangiovanni-Vincentelli // IEEE Des. Test Comput. – 2006. – T. 23 – № 5– 359–373c.

73. Danese P. JIT production, JIT supply and performance: investigating the moderating effects / P. Danese, P. Romano, T. Bortolotti // Ind. Manag. Data Syst. – 2012. – T. 112 – № 3– 441–465c.

74. Wahlen O. Application specific compiler/architecture codesign / O. Wahlen, T. Glökler, A. Nohl, A. Hoffmann, R. Leupers, H. Meyr // ACM SIGPLAN Not. – 2002. – T. 37 – № 7– 185c.

75. Dubach C. Exploring and predicting the architecture/optimising compiler co-design space / C. Dubach, T. M. Jones, M. F. P. O'Boyle // Proc. 2008 Int. Conf. Compil. Archit. Synth. Embed. Syst. - CASES '08 – 2008. – 31c.

76. Kluchev A. HLD - methodology in embedded systems design with a multilevel reconfiguration / A. Kluchev, A. Platunov, A. Penskoi // 3rd Mediterr. Conf. Embed. Compuning – 2014.

77. Кун Т. Структура научных революций / Т. Кун – Прогресс, 1977.– 300с.

78. Таненбаум Э. Архитектура компьютера / Э. Таненбаум, Т. Остин – Питер, 2015.

79. Hudak P. Building domain-specific embedded languages / P. Hudak // ACM Comput. Surv. – 1996. – T. 28 – № 4– 196c.

80. Nikhil R.S. What is Bluespec? / R. S. Nikhil, Arvind // SIGDA Newsl – 2008. – T. 38– 1c.

81. Voelter M. Language modularity with the MPS language workbench , 2012. – 1449–1450c.

82. Amelang D. Steps Toward Expressive Programming Systems / D. Amelang, B. Freudenberg, T. Kaehler, A. Kay, et al. // Report – 2010.

83. Hambarde P. The survey of real time operating system: RTOS / P. Hambarde, R. Varma, S. Jha // Proc. - Int. Conf. Electron. Syst. Signal Process. Comput. Technol. ICESC 2014 – 2014. – 34–39c.

84. Becker J. Configware and morphware going mainstream / J. Becker, R. Hartenstein // J. Syst. Archit. – 2003. – Т. 49 – № 4–6– 127–142c.

85. Shaw M. Software Architecture: Perspectives on an Emerging Discipline / M. Shaw, D. Garlan – Pearson, 1996.

86. Непейвода Н.Н. Стили и методы программирования. Курс лекций. Учебное пособие / Н. Н. Непейвода – Интернет-университет информационных технологий, 2005.– 320c.

87. Booch G. Unified Modeling Language User Guide / G. Booch, J. Rumbaugh, I. Jacobson – Addison Wesley, 1998.– 512c.

88. Gielingh W. A Theory For The Modelling Of Complex And Dynamic Systems / W. Gielingh // J. IT Constr. – 2008. – Т. 13.

89. Booch G. Object-oriented Analysis and Design with Applications / G. Booch – Addison Wesley Longman (Singapore) Pte. Limited, Indian Branch, 1994.

90. Noll T. Analyzing Reconfigurable Component-Based Systems Using Attribute Grammars / T. Noll // 8th Int. Symp. Form. Asp. Compon. Softw. – 2011. – 245–263c.

91. Binkert N. The Gem5 Simulator / N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, et al. // Comput. Arch. News – 2011. – Т. 39 – № 2– 7c.

92. Спольски Д.Х. Джоэл о программировании / Д. Х. Спольски – Символ-Плюс, 2006.

93. Пенской А.В. Архитектурное документирование встроенных систем с многоуровневой конфигурацией / А. В. Пенской // Изв. вузов. Приборостроение – 2015. – Т. 7– 527—532c.

94. Scott K. On Proebsting’s Law / K. Scott – University of Virginia Dept. of Computer Science Tech Report, 2012 – 3c.

95. Быковский С.В. Метод встроенной динамической актуализации функциональных моделей систем на кристалле / С. В. Быковский // Изв. вузов. Приборостроение – 2015. – Т. 3– 197–203c.

96. Астапов Д. Рекурсия + мемоизация = динамическое программирование / Д. Астапов // Практика функционального программирования – 2009. – Т. 3– 17–33c.

97. Fiadeiro J.L. Categories for software engineering / J. L. Fiadeiro – Springer, 2005.– 1-250c.

98. Aluffi P. Algebra: Chapter 0 (Graduate Studies in Mathematics) / P. Aluffi – American Mathematical Society, 2009.– 728c.

99. Hutton G. Programming in Haskell / G. Hutton – Cambridge University Press, 2007.– 184c.

100. O'Sullivan B. Real World Haskell / B. O'Sullivan, J. Goerzen, D. Steward – O'Reilly Media, 2008.– 714c.

101. Ковалёв С. П. Теоретико-категорные модели и методы проектирования больших информационно-управляющих систем: диссертация ... доктора физико-математических наук: 05.13.17 / Ковалёв С.П. - Москва, 2014.- 281с.

102. Chlipala A. Certified Programming with Dependent Types / A. Chlipala // Lect. Notes Comput. Sci. – 2009. – Т. 3622– 130–170с.

103. Pilgrim M. Dive Into Python 3 / M. Pilgrim – Apress, 2009.– 360с.

104. Jóźwiak L. Modern development methods and tools for embedded reconfigurable systems: A survey / L. Jóźwiak, N. Nedjah, M. Figueroa // Integr. VLSI J. – 2010. – Т. 43 – № 1– 33с.

105. Hauck S. Reconfigurable computing: the theory and practice of FPGA-based computation / S. Hauck, A. DeHon – , 2010.– 908с.

106. Compton K. Reconfigurable computing: a survey of systems and software / K. Compton, S. Hauck // ACM Comput. Surv. – 2002. – Т. 34 – № 2– 171–210с.

107. Bitter R. Labview Advanced Programming Techiniques / R. Bitter, T. Mohiuddin, M. Nawrocki // Amercia CRC Press LLC – 2001.

108. Jacobson I. Discover the Essence of Software Engineering / I. Jacobson // CSI Commun. – 2012. – 12–14с.

109. Chan L.-K. Quality function deployment: A literature review / L.-K. Chan, M.-L. Wu // Eur. J. Oper. Res. – 2002. – Т. 143 – № 3– 463–497с.

110. Jacobson I. The Essence of Software Engineering: Applying the SEMAT Kernel / I. Jacobson, N. Pan-Wei, E. M. Paul, I. Spence, S. Lidman – Addison-Wesley Professional, 2013.– 352с.

111. Jacobson I. Discover the Essence of Software Engineering / I. Jacobson // CSI Commun. – 2012. – 12–14с.

112. Непейвода Н.Н. Стили программирования как общий подход к системе понятий информатики / Н. Н. Непейвода – 2003.

113. Brooks F. No Silver Bullet: Essence and Accident of Software Engineering / F. Brooks // IEEE Softw. – 1987. – Т. 20– 12с.

114. Shaw M. Comparing architectural design styles / M. Shaw // IEEE Softw. – 1995. – Т. 12 – № 6– 27–41с.

115. Angeline S.J. Comparison of Software Architecture Styles Using Quality Attributes / S. J. Angeline, N. Snehalatha, P. Rodrigues – 2010.

116. Brooks F. The mythical man-month / F. Brooks // Proc. Int. Conf. Reliab. Softw. – 1975. – Т. 10 – № 6– 293–294с.

117. Болгаров И.С. Проектирование приборных контроллеров / И. С. Болгаров, Н. А. Маковецкая, А. Е. Платунов, Н. . Постников // звестия высших учебных заведений. Приборостроение – 2012. – Т. 55 – № 10– 73–78с.

118. Пенской А.В. Инструментальные средства и методы стрелочных вычислителей : выпускная квалификационная работа / А. В. Пенской – 2012. – Университет ИТМО. - Санкт-Петербург, 2012.

119. Corporaal H. Design of transport triggered architectures / H. Corporaal // VLSI, 1994. Des. Autom. High Perform. VLSI Syst. – 1994.

120. Reshadi M. A cycle-accurate compilation algorithm for custom pipelined datapaths / M. Reshadi, D. Gajski // Proc. 3rd IEEE/ACM/IFIP Int. Conf. Hardware/software codesign Syst. Synth. - CODES+ISSS ’05 – 2005. – 21с.

121. Platunov A. Aspects in the design of software-intensive systems / A. Platunov, A. Nickolaenkov // 2012 Mediterr. Conf. Embed. Comput. – 2012. – 80–83с.

Приложение А. Акты о внедрении результатов работы

А.1 Справка об использовании в НИР Университета ИТМО

УТВЕРЖДАЮ
Проректор по научной работе
Университета ИТМО
д.т.н., профессор
_____ В.О. Никифоров
«13» сентября 2016 г.

СПРАВКА

об использовании научных и практических результатов кандидатской диссертации Пенского А. В. на тему «Разработка и исследование архитектурных стилей проектирования уровневой организации встроенных систем»

Настоящая справка подтверждает использование результатов, полученных Пенским А. В. в ходе исследований по теме диссертационной работы, при выполнении на кафедре вычислительной техники Университета ИТМО следующих НИР:

1. Разработка и исследование аспектно-ориентированных технологий проектирования на базе унифицированных элементов информационно-коммуникационной инфраструктуры активно-адаптивных энергосетей ГК № 07.514.11.4073 от «13» октября 2011 г., Шифр: 2011-1.4-514-120-048.
2. Исследование механизмов обеспечения надежности аппаратно-резервированных информационно-измерительных систем на базе ПЛИС № 214434.
3. Создание бесшовных технологий проектирования встраиваемых систем и систем на кристалле на основе реконфигурируемых архитектур № 713564.
4. Разработка методов и средств системотехнического проектирования информационных и управляющих вычислительных систем с распределенной архитектурой № 610481.
5. Нелинейное и адаптивное управление сложными системами № 713546.

В перечисленных НИР были использованы следующие результаты, полученные Пенским А. В.: архитектурные стили, методики моделирования многоуровневых встроенных систем и методика проектирования встроенных систем.

Кроме того, в НИР «ГК № 07.514.11.4073 от «13» октября 2011 г., Шифр: 2011-1.4-514-120-048» Пенским А. В. были апробированы основные методы исследования, используемые в диссертационной работе. В НИР № 713564 была разработана унифицированная онтологическая модель [ре]конфигурации. В НИР № 610481 был предложен архитектурный стиль системно-иерархической организации.

Заведующий кафедры вычислительной техники,
д.т.н., профессор Т. И. Алиев

А.2 Справка об использовании в учебном процессе Университета ИТМО

УТВЕРЖДАЮ
Проректор по научной работе
Университета ИТМО
д.т.н., профессор

_____ В.О. Никифоров

«_16_» _сентября_ 2016 г.

СПРАВКА

об использовании научных и практических результатов кандидатской диссертации Пенского А.В. в учебном процессе в Санкт-Петербургском национальном исследовательском университете информационных технологий, механики и оптики

Настоящая справка подтверждает использование в учебном процессе на кафедре вычислительной техники следующих результатов диссертационной работы Пенского А.В.:

- архитектурных стилей для проектирования и документирования уровневой организации;
- формальных моделей языка макроассемблера, языка структурных схем, вычислительного процесса фон Неймановского процессора, тактовых моделей актуализации и виртуализации, моделей специализированного сигнального процессора и д.р.;
- понятий вычислительной платформы, системной платформы, многоуровневого вычислительного агрегата и структуры пространства проектных решений;
- онтологических модели [ре]конфигурации и иерархической организации.

Указанные результаты использованы при подготовке учебно-методических материалов (конспектов лекций, презентаций, методических указаний к лабораторным работам) для проведения лекционных и лабораторных занятий в рамках:

1) Магистерской образовательной программы 09.04.01 «Проектирование встроенных вычислительных систем» по дисциплинам:
   - программное обеспечение встраиваемых вычислительных систем;
   - управление программными проектами.
2) Бакалаврской образовательной программы 09.03.01 «Информатика и вычислительная техника» по дисциплинам:
   - информационно-управляющие системы;
   - организация ЭВМ и систем;
   - интерфейсы периферийных устройств.

Заведующий кафедрой вычислительной техники
д.т.н., профессор _____ Т.И. Алиев

А.3 Справка об использовании в ООО «ЛМТ»

ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ "ЛМТ"

СПРАВКА

об использовании научных и практических результатов кандидатской диссертационной работы Пенского А.В. на тему «Разработка и исследование архитектурных стилей проектирования уровневой организации встроенных систем»

Данная справка подтверждает использование результатов, полученных Пенским А.В. в ходе исследований по теме диссертационной работы, в следующих разработках предприятия ООО «ЛМТ»:

- система коммерческого учёта электроэнергии «ИИС Луч-ТС М» (используется для обслуживания субъектов оптовых и розничных рынков электроэнергии, таких как жилые здания, коттеджные посёлки, промышленные предприятия, объекты энергетической инфраструктуры. Система «ИИС Луч-ТС М» внедрена в городах Великий Устюг, Вологда и Череповец);
- САПР сигнального процессора реального времени NL3 (используется для проведения исследований в области зондовой микроскопии);
- система управления беспилотным летательным аппаратом;
- САПР для реконфигурируемого фон Неймановского процессора.

В перечисленных проектах были использованы следующие научные результаты Пенского А.В.:

- архитектурные стили и языки архитектурного описания;
- методика моделирования многоуровневых встроенных систем и их элементов;
- модифицированная методика проектирования встроенных систем, ориентированная на системы с нетривиальной уровневой организацией.

Использование данных результатов позволило значительно сократить затраты на разработку, поддержку и развитие; повысить коэффициент повторного использования; снизить число дефектов.

Генеральный директор ООО «ЛМТ»
д.т.н., профессор А.Е. Платунов

«02» 09 2016 г.

ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ "ЛМТ" "LMT" Ltd. САНКТ-ПЕТЕРБУРГ

199034, г.Санкт-петербург, Биржевая линия, д.16
тел./факс: 331-88-46, http://lmt.ifmo.ru, email: info@lmt.ifmo.ru

Приложение Б. Фрагменты исходных кодов программного обеспечения

Б.1 САПР сигнального процессора реального времени NL3

Б.1.1 Моделирование блоков обработки данных (общая часть, фрагмент)

```
#[derive(Debug, Clone, PartialEq, Eq)]
enum IO {
    Const(usize),
    In(String),
    Out(String),
    Undef,
}
impl IO {
    fn v(&self) -> &String {
        match *self {
            IO::In(ref s) | IO::Out(ref s) => s,
            _ => panic!(),
        }
    }
}

trait GetIO {
    fn io(&self) -> IO {
        self.io_ref().clone()
    }
    fn io_ref(&self) -> &IO {
        panic!("io_ref not impl!")
    }
}
impl<T> GetIO for Vec<T> where T: GetIO
{
    fn io_ref(&self) -> &IO {
        self.first().expect("GetIO from empty vec!").io_ref()
    }
}
#[derive(Debug)]
struct State<'a, ST, M: 'a, L, Lnk> {
    tick: usize,
    t2l: Vec<(usize, usize)>,
    process: Vec<(usize, M, Vec<L>)>,
    program: Vec<&'a BHigh<M>>,
    state: ST,
    links: Option<Lnk>,
}
type HighId = usize;
#[derive(Debug, Clone)]
struct TimeConstrain {
    duration: usize,
    available: (usize, usize),
}
impl TimeConstrain {
    fn new(dur: usize, a: usize, b: usize) -> TimeConstrain {
        TimeConstrain {
            duration: dur,
            available: (a, b),
        }
    }
}
```

```
trait DPU {
    fn allow_self_transfer(&self) -> bool {
        false
    }
    fn is_over(&self) -> bool;
    fn tick(&self) -> usize;
    fn info(&self, usize) -> String;
    fn io_variants(&self) -> Vec<IO>;
    fn time_constrain(&self, io: &IO) -> TimeConstrain;
    fn process_step(&mut self,
                    time_offset: usize,
                    duration: usize,
                    io: Vec<&IO>);
    fn dependency(&self) -> Vec<(&String, &String)> {
        vec![]
    }
}

fn naive<'a, ST, M, L, Lnk>(st: &mut State<'a, ST, M, L, Lnk>)
    where State<'a, ST, M, L, Lnk>: DPU,
          M: std::fmt::Debug,
          L: std::fmt::Debug
{
    while !st.is_over() {
        let io_variants = st.io_variants();
        println!("naive::io_variants: {:?}", io_variants);
        let io = io_variants.first().unwrap();
        let tc = st.time_constrain(&io);
        st.process_step(tc.available.0, tc.duration, vec![&io]);
    }
}
```

Б.1.2 Моделирование функциональных блоков (фрагмент)

```
trait High<M> {
    fn high2midle(&self) -> Vec<M>;
}

trait DHigh<M>: High<M> + Dependency + std::fmt::Debug {}
impl<T, M> DHigh<M> for T where T: High<M> + Dependency + std::fmt::Debug{}

type BHigh<M> = Box<DHigh<M>>;
type HighRef<M> = Box<DHigh<M>>;
trait Dependency {
    fn dependency(&self) -> Vec<(&String, &String)> {
        vec![]
    }
    fn transfers(&self) -> Vec<&String>;
}

#[derive(Debug, Clone)]
struct Add {
    a: String,
    b: String,
    c: String,
}
impl Add {
    fn new(a: &str, b: &str, c: &str) -> Add {
        Add {
            a: a.to_string(),
            b: b.to_string(),
            c: c.to_string(),
        }
    }
}
```

```
impl Dependency for Add {
    fn dependency(&self) -> Vec<(&String, &String)> {
        vec![
            ( &self.c, &self.a ),
            ( &self.c, &self.b ),
            ]
    }
    fn transfers(&self) -> Vec<&String> {
       vec![&self.a, &self.b, &self.c]
    }
}

#[derive(Debug, Clone)]
struct Neg {
    a: String,
    b: String,
}
impl Neg {
    fn new(a: &str, b: &str) -> Neg {
        Neg {
            a: a.to_string(),
            b: b.to_string(),
        }
    }
}
impl Dependency for Neg {
    fn dependency(&self) -> Vec<(&String, &String)> {
        vec![
            ( &self.b, &self.a ),
            ]
    }
    fn transfers(&self) -> Vec<&String> {
       vec![&self.a, &self.b]
    }
}

#[derive(Debug, Clone)]
struct InMemoryReg {
    a: String,
    b: String,
    addr: usize,
}
impl InMemoryReg {
    fn new(addr: usize, a: &str, b: &str) -> InMemoryReg {
        InMemoryReg {
            a: a.to_string(),
            b: b.to_string(),
            addr: addr,
        }
    }
}
impl Dependency for InMemoryReg {
    fn dependency(&self) -> Vec<(&String, &String)> {
        vec![
            ( &self.b, &self.a ),
            ]
    }
    fn transfers(&self) -> Vec<&String> {
       vec![&self.a, &self.b]
    }
}
#[derive(Debug, Clone)]
struct Output {
    a: String,
    addr: usize,
}
```

```
impl Output {
    fn new(addr: usize, a: &str) -> Output {
        Output {
            a: a.to_string(),
            addr: addr,
        }
    }
}

impl Dependency for Output {
    fn transfers(&self) -> Vec<&String> {
       vec![&self.a]
    }
}

#[derive(Debug, Clone)]
struct Input {
    a: String,
    addr: usize,
}

impl Input {
    fn new(addr: usize, a: &str) -> Input {
        Input {
            a: a.to_string(),
            addr: addr,
        }
    }
}

impl Dependency for Input {
    fn transfers(&self) -> Vec<&String> {
       vec![&self.a]
    }
}
```

Б.1.3 Моделирование блока обработки данных сумматора (фрагмент)

```
impl High<AccumM> for Add {
    fn high2midle(&self) -> Vec<AccumM> {
        vec![
            AccumM::In(false, IO::In(self.a.to_string())),
            AccumM::Ln(false, IO::In(self.b.to_string())),
            AccumM::Gn(IO::Out(self.c.to_string())),
            ]
    }
}

impl High<AccumM> for Neg {
    fn high2midle(&self) -> Vec<AccumM> {
        vec![
            AccumM::In(true, IO::In(self.a.to_string())),
            AccumM::Gn(IO::Out(self.b.to_string())),
            ]
    }
}

#[derive(Debug)]
struct AccumState {
    selected: Option<(usize, Vec<AccumM>)>,
    remain: Vec<usize>,
    midles: Vec<Vec<AccumM>>,
}
```

```
type AccumDpu<'a> = State<'a, AccumState, AccumM, AccumL<IO>, AccumL<Link>>;

impl<'a> AccumDpu<'a> {
    fn new() -> AccumDpu<'a> {
        State {
            tick: 0,
            t2l: vec![],
            process: vec![],
            program: vec![],
            state: AccumState {
                selected: None,
                remain: vec![],
                midles: vec![],
            },
            links: None,
        }
    }
    fn add(&mut self, high: &'a BHigh<AccumM>) {
        self.state.midles.push(high.high2midle());
        self.program.push(high);
        self.state.remain.push(self.program.len() - 1);
    }
}

impl<'a> DPU for AccumDpu<'a> {
    fn tick(&self) -> usize {
        self.tick
    }
    fn info(&self, tick: usize) -> String {
        if self.t2l.len() <= tick {
            return "empty!".to_string();
        }
        let (high_i, low_i) = self.t2l[tick];
        format!("{:?} {:?} {:?}",
                self.program[self.process[high_i].0],
                self.process[high_i].1,
                self.process[high_i].2[low_i])
    }
    fn is_over(&self) -> bool {
        self.state.selected.is_none() && self.state.remain.is_empty()
    }
    fn dependency(&self) -> Vec<(&String, &String)> {
        let mut result: Vec<(&String, &String)> = Vec::new();
        for midle in &self.state.midles {
            println!(">>> {:?}", midle);
            for i in 0..(midle.len() - 1) {
                result.push(( midle[i + 1].io_ref().v(), midle[i].io_ref().v() ));
            }
        }
        result
    }
    fn io_variants(&self) -> Vec<IO> {
        match self.state.selected {
            Some((_, ref midle)) => {
                // TODO: Экспорт с оптимизацией.
                assert!(!midle.is_empty());
                vec![midle.io_ref().clone()]
            }
            None => {
                self.state
                    .remain
                    .iter()
                    .map(|i| self.program[*i].high2midle().io_ref().clone())
                    .collect()
            }
        }
    }
```

```
    fn time_constrain(&self, io: &IO) -> TimeConstrain {
        let in_time_constrain = TimeConstrain::new(2, 0, usize::max_value());
        match self.state.selected {
            Some((_, ref midles)) => {
                assert_eq!(midles.io_ref().clone(), *io);
                match *midles.first().unwrap() {
                    AccumM::In(_, _) => in_time_constrain,
                    AccumM::Ln(_, _) => TimeConstrain::new(1, 0, usize::max_value()),
                    AccumM::Gn(_) => TimeConstrain::new(1, 2, usize::max_value()),
                }
            }
            None => in_time_constrain,
        }
    }

    fn process_step(&mut self,
                    time_offset: usize,
                    duration: usize,
                    io: Vec<&IO>) {
        assert!(io.len() == 1);
        let io = io.first().unwrap();
        let step = process_step_inner(time_offset, duration, io, self);
        if match self.state.selected {
            Some((_, ref midles)) => midles.is_empty(),
            None => false,
        } {
            self.state.selected = None;
        }
        for i in 0..step.2.len() {
            self.t2l.push((self.process.len(), i));
        }
        self.tick += step.2.len();
        self.process.push(step);
        let mut k = 0;
        for (i, p) in self.process.iter().enumerate() {
            for (j, low) in p.2.iter().enumerate() {
                k += 1;
            }
        }
    }

}

fn process_step_inner(time_offset: usize,
                      duration: usize,
                      io: &IO,
                      st: &mut State<AccumState, AccumM, AccumL<IO>, AccumL<Link>>)
                      -> (usize, AccumM, Vec<AccumL<IO>>) {
    let time_constrain = st.time_constrain(io);
    assert!(time_constrain.available.0 <= time_offset + max(duration,
time_constrain.duration));
    match st.state.selected {
        Some((high_i, ref mut midles)) => {
            let mut microcode = vec![ Default::default(); time_offset ];
            let midle = midles.remove(0);
            assert_eq!(midle.io_ref(), io);
            match midle {
                AccumM::In(neg, _) => {
                    microcode.append(&mut vec![AccumL {
                        hint: "init".to_string(),
                        init: IO::Const(1), load: IO::Const(1),
                        neg: IO::Const(if neg { 1 } else { 0 }),
                        bus: io.clone(),
                        ..Default::default()
                    }; time_constrain.duration])
                }
```

```
                AccumM::Ln(neg, _) => {
                    microcode.append(&mut vec![AccumL {
                        hint: "load".to_string(),
                        load: IO::Const(1),
                        neg: IO::Const(if neg { 1 } else { 0 }),
                        bus: io.clone(),
                        ..Default::default()
                    }; time_constrain.duration])
                }
                AccumM::Gn(_) => {
                    microcode.append(&mut vec![AccumL {
                        hint: "get".to_string(),
                        oe: IO::Const(1),
                        bus: io.clone(),
                        ..Default::default()
                }; duration])
                }
            };
            (high_i, midle, microcode)
        }
        None => {
            let mut selected: Option<(usize, Vec<AccumM>)> = None;
            let mut take_from_remain: Option<usize> = None;
            assert!(st.state.remain.len() > 0);
            for (i, high_i) in st.state.remain.iter().enumerate() {
                let high_i = *high_i;
                let midles = st.program[high_i].high2midle();
                if midles.io_ref() == io {
                    selected = Some((high_i, midles));
                    take_from_remain = Some(i);
                    break;
                }
            }
            assert!(selected.is_some() && take_from_remain.is_some());
            st.state.selected = selected;
            st.state.remain.swap_remove(take_from_remain.unwrap());
            process_step_inner(time_offset, duration, io, st)
        }
    }
}

#[derive(Debug, Clone, PartialEq, Eq)]
enum AccumM {
    In(bool, IO),
    Ln(bool, IO),
    Gn(IO),
}
impl GetIO for AccumM {
    fn io(&self) -> IO {
        self.io_ref().clone()
    }
    fn io_ref(&self) -> &IO {
        match *self {
            AccumM::In(_, ref v) => v,
            AccumM::Ln(_, ref v) => v,
            AccumM::Gn(ref v) => v,
        }
    }
}
#[derive(Debug, Clone)]
struct AccumL<T> {
    hint: String,
    init: T,
    load: T,
    neg: T,
    oe: T,
    bus: T,
}
```

```
impl Default for AccumL<IO> {
    fn default() -> AccumL<IO> {
        AccumL {
            hint: "nop".to_string(),
            init: IO::Const(0),
            load: IO::Const(0),
            neg: IO::Undef,
            oe: IO::Const(0),
            bus: IO::Undef,
        }
    }
}
```

Б.2 Виртуальная машина для беспилотного летательного аппарата

Б.2.1 Заголовок виртуальной машины (фрагмент)

```
#define MAX_NAME_SIZE 20  // for variables and functions
#define JSON_TOKENS 128   // in input configuration, more detail in jsmn documentation
#define PMVM_HEAP_SIZE 1024  // memory for mpvm internal configuration
//////////////////////////
// #define DEBUG
#ifdef DEBUG
#include <stdio.h>
#define dprintf(fmt, ...) printf((fmt), ##__VA_ARGS__)
#endif
#ifndef DEBUG
#define dprintf(fmt, ...)
#endif

// exports
struct State {
    int events;
    int *regs;
};

struct Conf {
    struct Pattern *patterns;
    int patterns_len;
    struct Reg *regs;
    int regs_len;
    struct State state;
    void *vm_heap;
    int vm_heap_offset;
   int vm_heap_size;
};
int set_event(char *tag, struct Conf *conf);
int set_regs(char *tag, int value, struct Conf *conf);
int get_regs(char *tag, int *value, struct Conf *conf);
int unsafe_get_regs(char *tag, struct Conf *conf);
int mk_state(struct Conf * conf);
int mk_dispatcher(struct Conf *conf, const char *dump);
int dispatch(struct Pattern * patterns, int count, struct State *st);
typedef int(* FunctionPtr)(int, int);
struct FunctionTag {
    FunctionPtr fun;
    char tag[MAX_NAME_SIZE];
};
struct EventTag {
    int code;
    char tag[MAX_NAME_SIZE];
};
```

Б.2.2 Экземпляр виртуальной машины

```
int g_eq(int a, int b) { return a == b; }
int g_l(int a, int b) { return a < b; }
int g_me(int a, int b) { return a >= b; }
int g_and(int a, int b) { return a && b; }
int g_or(int a, int b) { return a || b; }

const struct FunctionTag functions[] = {
    { g_eq,  "=="},
    { g_l,   "<" },
    { g_me,   ">=" },
    { g_and, "&&"},
    { g_or,  "||"},
};

const struct EventTag events[] = {
    { 0x00000001, "GPS1"},
    { 0x00000010, "BUTTON"},
    { 0x00000100, "COMPLETE"},
};
```

Б.3 САПР для специализированного фон Неймановского процессора

Б.3.1 Основные структуры данных

```
-- Generic
class To p where from :: (Integral i) => i -> p
class (To t) => To' f t where from' :: f -> t


-- MachineASM lang

data MachineAsmF i next
    = InstructionMAF i next
    | MemoryDumpMAF ByteString next

instance Functor (MachineAsmF i) where
    fmap f (InstructionMAF i next) = InstructionMAF i (f next)
    fmap f (MemoryDumpMAF bs next) = MemoryDumpMAF bs (f next)

class MachineAsmC l where
    instruction :: i -> Free (l i) ()
    memoryDump :: [Word8] -> Free (l i) ()

instance MachineAsmC MachineAsmF where
    instruction i = liftF $ InstructionMAF i ()
    memoryDump bs = liftF $ MemoryDumpMAF (pack bs) ()


-- MachineASM builder

data MachineAsmSt i
    = InstrMASt i
    | MemoryDumpMASt ByteString
    deriving (Show)

data MachineAsm i = MachineAsm
    { sourceMA :: [MachineAsmSt i]
    } deriving (Show)
```

```
machineAsm (Pure x) = return x
machineAsm (Free (InstructionMAF mnem next)) = do
    x@MachineAsm{..} <- get
    put x{sourceMA=InstrMASt mnem : sourceMA}
    machineAsm next
machineAsm (Free (MemoryDumpMAF bs next)) = do
    x@MachineAsm{..} <- get
    put x{sourceMA=MemoryDumpMASt bs : sourceMA}
    machineAsm next

-- ASM lang

data AsmF i next
    = InstructionAF i                  next
    | MemoryDumpAF   ByteString        next
    | MarkDeclAF     (Mark i)         (Mark i -> next)
    | MarkDefineAF   (Mark i)         (Mark i -> next)
    | SubstitutionAF (Substitution i)  next

instance Functor (AsmF i) where
    fmap f (InstructionAF i next ) = InstructionAF i (f next)
    fmap f (MemoryDumpAF bs next ) = MemoryDumpAF bs (f next)
    fmap f (MarkDeclAF m next    ) = MarkDeclAF m (f . next)
    fmap f (MarkDefineAF m next  ) = MarkDefineAF m (f . next)
    fmap f (SubstitutionAF s next) = SubstitutionAF s (f next)

instance MachineAsmC AsmF where
    instruction i = liftF $ InstructionAF i ()
    memoryDump bs = liftF $ MemoryDumpAF (pack bs) ()

class (MachineAsmC l) => AsmC l where
    declare :: Mark i -> Free (l i) (Mark i)
    mark :: Mark i -> Free (l i) (Mark i)
    substitution :: i -> Mark i -> (Mark i -> Either (Substitution i) i) -> Free (l i) ()

instance AsmC AsmF where
    declare m = liftF $ MarkDeclAF m id
    mark m = liftF $ MarkDefineAF m id
    substitution i m m2i = liftF $ SubstitutionAF (Substitution i m m2i) ()

data Substitution i
    = Substitution i (Mark i) (Mark i -> Either (Substitution i) i)

instance (Show i, MarkBuilderC i) => Show (Substitution i) where
    show (Substitution i m _m2i) = "\\" ++ show m ++ " -> " ++ show i

-- ASM builder

data AsmSt i
    = MarkASt (Mark i)
    | InstrASt i
    | MemoryDumpASt ByteString
    | SubstitutionASt (Substitution i)

deriving instance (Show i, MarkBuilderC i) => Show (AsmSt i)

data Asm i = Asm
    { sourceA :: [AsmSt i]
    , markStA :: BuilderMarkSt i
    }

deriving instance (Show i, MarkBuilderC i) => Show (Asm i)

class ( Show isa
      , Show (Mark isa)
      , Show (BuilderMarkSt isa)
      , Default (BuilderMarkSt isa)
      ) => MarkBuilderC isa where
```

```
    data Mark isa :: *
    data BuilderMarkSt isa :: *

    declareMark :: Mark isa -> Asm isa -> (Mark isa, Asm isa)
    defineMark  :: Mark isa -> Asm isa -> (Mark isa, Asm isa)

asm (Pure x) = return x
asm (Free (InstructionAF mnem next)) = do
    x@Asm{..} <- get
    put x{ sourceA = InstrASt mnem : sourceA }
    asm next
asm (Free (SubstitutionAF s next)) = do
    x@Asm{..} <- get
    put x{sourceA=SubstitutionASt s : sourceA}
    asm next
asm (Free (MemoryDumpAF bs next)) = do
    x@Asm{..} <- get
    put x
        { sourceA = MemoryDumpASt bs : sourceA
        }
    asm next
asm (Free (MarkDeclAF m next)) = do
    (m', st) <- get >>= return . declareMark m
    put st
    asm . next $ m'
asm (Free (MarkDefineAF m next)) = do
    (m', st) <- get >>= return . defineMark m
    put st
    asm . next $ m'

---------- machine code ----------
data MachineCodeF cmd next
    = DumpF ByteString next
    | CmdF cmd next
    | EndF Int Word8
    deriving Functor

dump :: [Word8] -> Free (MachineCodeF cmd) ()
dump dt = liftF $ DumpF (pack dt) ()

cmd :: cmd -> Free (MachineCodeF cmd) ()
cmd c = liftF $ CmdF c ()

end :: Int -> Word8 -> Free (MachineCodeF cmd) ()
end size value = liftF $ EndF size value

class Dump a where
    pack'   :: a -> ByteString
    length' :: a -> Int

data MachineCodeSt cmd
    = DumpSt ByteString
    | CmdSt cmd
    deriving (Show)

instance (Dump cmd) => Dump (MachineCodeSt cmd) where
    pack'   (DumpSt dt) = dt
    pack'   (CmdSt cmd) = pack' cmd
    length' (DumpSt dt) = undefined
    length' (CmdSt cmd) = length' cmd

data MachineCode cmd = MachineCode
    { code :: [MachineCodeSt cmd]
    } deriving (Show)
instance (Dump cmd) => Dump (MachineCode cmd) where
    pack' MachineCode{..} = BS.concat $ map pack' code
    length' MachineCode{..} = sum $ map length' code
```

```
machine (Pure x) = return x
machine (Free (DumpF dt next)) = do
    x@MachineCode{..} <- get
    put x{code=DumpSt dt:code}
    machine next
machine (Free (CmdF cmd next)) = do
    x@MachineCode{..} <- get
    put x{code=CmdSt cmd:code}
    machine next
machine (Free (EndF size value)) = do
    x@MachineCode{..} <- get
    let size' = fromIntegral $ BS.length $ pack' x
    let code' = if size' <= size
        then (DumpSt $ pack $ Prelude.replicate (size - size') value):code
        else error "Machine code too big."
    put x{code=code'}
    return ()

class Translate opt l1 l2 where
    translate :: opt -> l1 -> l2

class ( Show (MarkSt isa code)
      , Default (MarkSt isa code)
      , MarkBuilderC isa
      , Dump code
      ) => MarkTranslatorC isa code where
    data MarkSt isa code :: *
    registerToken :: MachineAsmSt isa -> MarkSt isa code -> MarkSt isa code
    mkMarkState :: Mark isa -> MarkSt isa code -> MarkSt isa code
    findMark :: Mark isa -> MarkSt isa code -> Mark isa

instance ( MarkBuilderC isa
         , MarkTranslatorC isa code
         ) => Translate code (Asm isa) (MachineAsm isa) where
    translate _ Asm{..} = MachineAsm sourceMA
        where
            (sourceMA0, markSt) = foldr (\n (src, mSt) -> case n of
                InstrASt i -> ((Right $ InstrMASt i) : src, registerToken (InstrMASt i)
mSt)
                MemoryDumpASt bs -> ((Right $ MemoryDumpMASt bs) : src, registerToken
(MemoryDumpMASt bs) mSt)
                MarkASt m -> (src, mkMarkState m mSt)
                SubstitutionASt (Substitution i m cons) -> (Left (m, cons) : src,
registerToken (InstrMASt i) mSt)
                ) ([], def :: MarkSt isa code) sourceA
            sourceMA = map (either sub id) sourceMA0
            sub (m, cons) = case cons $ findMark m markSt of
                Left (Substitution _i m' cons') -> sub (m', cons')
                Right i -> InstrMASt i

instance (Translate () isa code, Dump code) => Translate () (MachineAsm isa) (MachineCode
code) where
    translate () MachineAsm{..} = MachineCode $ map f sourceMA
        where
            f (InstrMASt i) = CmdSt $ translate () i
            f (MemoryDumpMASt bs) = DumpSt bs

instance ( MarkBuilderC isa
         , MarkTranslatorC isa code
         , Translate () isa code
         , Default code
         ) => Translate () (Asm isa) (MachineCode code) where
    translate () asm = machineCode
        where
            machineAsm = translate (def :: code) asm :: MachineAsm isa
            machineCode = translate () machineAsm :: MachineCode code
```