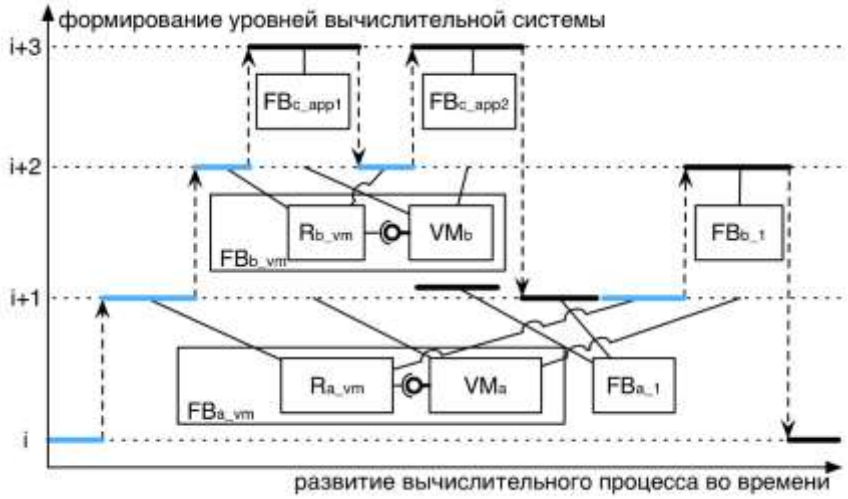
УНИВЕРСИТЕТ ИТМО

С.В. Быковский, Я.Г. Горбачев, А.О. Ключев, А.В. Пенской, А.Е. Платунов

СОПРЯЖЁННОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ (HARDWARE/SOFTWARE CO-DESIGN)

Часть 2



Санкт-Петербург

2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

С.В. Быковский, Я.Г. Горбачев, А.О. Ключев,
А.В. Пенской, А.Е. Платунов

СОПРЯЖЁННОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ (HARDWARE/SOFTWARE CO-DESIGN)

Часть 2

Учебное пособие

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург

2016

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

В учебном пособии представлены обзорные и оригинальные материалы по прогрессивным методологиям высокоуровневого проектирования встраиваемых вычислительных систем.

Для подготовки магистров по направлению 09.04.01 «Информатика и вычислительная техника» по программе «Проектирование встраиваемых вычислительных систем и систем на кристалле» и магистров по направлению 09.04.04 «Программная инженерия» по программе «Программное обеспечение мобильных и встраиваемых систем».

Рекомендовано к печати ученым советом мегафакультета КТиУ Университета ИТМО, протокол № 6 от 21.06.2016.

УНИВЕРСИТЕТ ИТМО

Университет ИТМО — ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО — участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО — становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Учебное пособие является введением в высокоуровневое проектирование специализированных, в частности – встраиваемых, вычислительных систем, и содержит обзорные материалы по направлению, в англоязычной литературе носящему название Hardware/Software Co-Design. В отечественной литературе нет общепринятого названия данного направления, для его обозначения используют такие термины, как «аппаратно-программный кодизайн», «кодизайн», «сопряжённое проектирование аппаратуры и программного обеспечения», «совместное проектирование аппаратуры и программного обеспечения».

Необходимость высокоуровневого проектирования подобных систем и применения новых методологий обусловлены рядом факторов. Вычислительная техника со времени создания первых программируемых ЭВМ уже проделала огромный путь и не прекращает развиваться. Современные компьютеры далеко обогнали своих предшественников, с каждым годом предоставляемые ими возможности растут. Увеличиваются вычислительные мощности отдельных процессоров, создаются всё более сложные гетерогенные вычислительные системы, объединяющие различные вычислительные ядра. Всё большее распространение получают постоянно усложняющиеся распределённые системы. Программное обеспечение также становится всё более сложным, современные продукты насчитывают миллионы строк кода – и эти цифры растут.

Одновременно, возрастают потребности в специализированных вычислительных системах. Сегодня сложно представить нашу жизнь без электроники во всех сферах человеческой деятельности. Бытовые приборы, блоки управления автомобилями, кораблями и самолётами, охранные системы, электронные ключи, прогрессивные системы вооружений, связь – везде используются специализированные компьютеры, так называемые встраиваемые системы. Фактически, в составе парка вычислительных систем они составляют абсолютное большинство.

Требования, предъявляемые к таким системам, по определению жёстче, чем к ЭВМ общего назначения. В общем случае встраиваемые системы должны быть надёжными, малогабаритными, дешёвыми, с низким энергопотреблением и с предсказуемым временем реакции на внешние воздействия (должны удовлетворять требованиям реального времени). Ввиду этого, зачастую использование существующих аппаратных решений оказывается неэффективным, и помимо написания программного обеспечения – как делается обычно в системах общего назначения – встаёт задача создания также и специализированной аппаратуры.

Таким образом, при проектировании встраиваемых систем разработчик вынужден проектировать аппаратуру и программное обеспечение с учётом множества разноплановых аспектов, и зачастую вынужден работать одновременно в нескольких областях инженерной деятельности. Условия рыночной экономики добавляют к вышеперечисленному жёсткие временные ограничения, накладываемые на процесс разработки, требования к качеству конечного продукта и необходимость учёта рисков. Немалое внимание приходится уделять также человеческим ресурсам и правильной организации команды разработчиков. Кроме того, как говорилось ранее, элементная база и потенциальные возможности постоянно совершенствуются, предоставляя всё более совершенные и эффективные средства для создания специализированных вычислительных систем.

Исходя из всего вышеперечисленного, проектирование встраиваемых систем представляет собой очень трудоёмкую и многоплановую задачу. Самым простым и очевидным подходом для борьбы со сложностью является применение шаблонных решений и подходов, заимствование готовых элементов. При этом довольно часто это производится без рассмотрения всех возможных способов реализации, а некоторые важные аспекты просто не учитываются. Подавляющее большинство встраиваемых систем сегодня создаётся именно таким путём, что очень редко позволяет создавать действительно эффективные и оптимальные решения.

Направление кодизайна объединяет многочисленные исследования в области параллельного и скоординированного проектирования программного обеспечения и аппаратуры. Рассмотрение абстрактной задачи и алгоритма её решения сначала на системном уровне, безотносительно к реализации, и последующее распределение подзадач между программными и аппаратными компонентами даёт большую свободу в поиске компромиссов, позволяет лучше настраивать систему под конкретную задачу. В контексте проектирования специализированных вычислителей термин «кодизайн» фактически используется как синоним прогрессивных направлений системного, высокоуровневого или архитектурного проектирования, и представляет собой набор подходов и инструментальных средств для решения актуальных проблем проектирования специализированных вычислительных систем.

Во второй части пособия обсуждается сущность понятия «кодизайн» и приводится обзор достижений данного направления. Описывается подход, основанный на методологии ESL (Electronic System Level). На примере типового маршрута ESL-проектирования объясняются три важных составляющих кодизайна: создание спецификации на проектируемое изделие, исследование пространства проектных решений, разделение на

программные и аппаратные части. Даётся обзор инструментальных средств автоматизации проектирования, позволяющих решать возникающие при описанном маршруте проектирования задачи.

1 СОПРЯЖЁННОЕ ПРОЕКТИРОВАНИЕ АППАРАТУРЫ И ПО

1.1 Направление «кодизайн»

Сопряжённое, или совместное проектирование аппаратуры и ПО, называемое в англоязычной литературе «hardware-software co-design», «software-hardware co-design», или просто «co-design» – представляет собой процесс параллельного и скоординированного проектирования электронных программно-аппаратных систем, который основывается на не зависящем от конечной реализации описании, и использует средства автоматизации проектирования [1].

Кодизайн помогает находить компромиссы между гибкостью вычислительной системы и её производительностью, комбинируя при проектировании два радикально отличающихся подхода: последовательную декомпозицию во времени, создаваемую при помощи ПО, и параллельную декомпозицию в пространстве, реализуемую на основе аппаратного обеспечения [2]. Исследование компромиссов при распределении вычислений между ПО и аппаратурой помогает создавать достаточно эффективные программно-аппаратные решения, оптимизированные для решаемых задач. Благодаря одновременному анализу, исследованию и проектированию аппаратного и программного обеспечения сокращаются сроки разработки и уменьшаются риски задержек выхода на рынок.

Основные цели и смысл кодизайна могут быть выведены из различных трактовок слога «ко» в слове кодизайн [1]:

- Координация: методы кодизайна используются для координации этапов проектирования, выполняемых междисциплинарными группами. В эти группы со стороны ПО входят разработчики встроенного микропрограммного обеспечения, системные и прикладные программисты, а со стороны аппаратного обеспечения – разработчики аппаратных средств и микросхем. Это является исходной интерпретацией слога «ко» в слове кодизайн. Другие, описанные далее интерпретации, являются скорее дополнениями.
- Параллелизм (Concurrency): Сжатые рамки времени выхода на рынок вынуждают разработчиков ВС работать одновременно, вместо того, чтобы начать работу с ПО лишь после создания аппаратной платформы. Кодизайн как-раз достиг значительного прогресса в обходе данного узкого места процесса разработки, избавляя от необходимости иметь готовую аппаратуру. Это достигается благодаря использованию исполняемых спецификаций и/или благодаря применению концепции виртуальных платформ и виртуального прототипирования – для того, чтобы запускать

разрабатываемое ПО на симуляторе целевой платформы, на очень раннем этапе.

- Корректность: Требования к отсутствию ошибок во встраиваемых системах требуют, чтобы не только проверялись корректность ПО и аппаратуры по отдельности, но и ко-верифицировалась правильность их взаимодействия после интеграции.
- Сложность (Complexity): Методы ко-дизайна решают проблему роста сложности проектирования сегодняшних электронных систем, они являются средством (или, по крайней мере, пытаются быть таким средством) для решения известных проблем проектирования, для создания корректно работающих и оптимизированных системных реализаций.

Сокращение сроков разработки является одним из основных преимуществ, которые достигаются благодаря кодизайну [1]. Традиционно, как правило, основываясь на общей спецификации и, возможно, на некоторых начальных данных, полученных при симуляции высокоуровневых моделей, сначала принимаются аппаратные решения, и, в худшем случае, команды разработчиков встроенного и прикладного ПО не начинают разработку и тестирование своих продуктов до появления доступных рабочих прототипов аппаратуры – т.е., на очень позднем этапе разработки.

У такого подхода огромный риск возникновения задержки всей цепочки проектирования продукта – в случае позднего обнаружения концептуальных аппаратных или проектных ошибок. Кроме того, при таком подходе нет способа исследовать потенциальные возможности для других, более оптимальных по каким-нибудь критериям (таким как стоимость, производительность, расширяемость), вариантов реализаций. Обычно аппаратные проектные решения принимаются на основе опыта команды разработчиков, а успех и эффективность создания программного обеспечения зависит от команды разработчиков программного обеспечения.

Недостатки классической цепочки проектирования многочисленны [1]:

- длинный критический путь и обычно непредсказуемое время выхода на рынок:
- риски очень позднего обнаружения потенциальных ошибок в каждой части проектной цепочки;
- риск избыточности или неполноты конечного продукта из-за отсутствия ранней оценки проектных альтернатив.

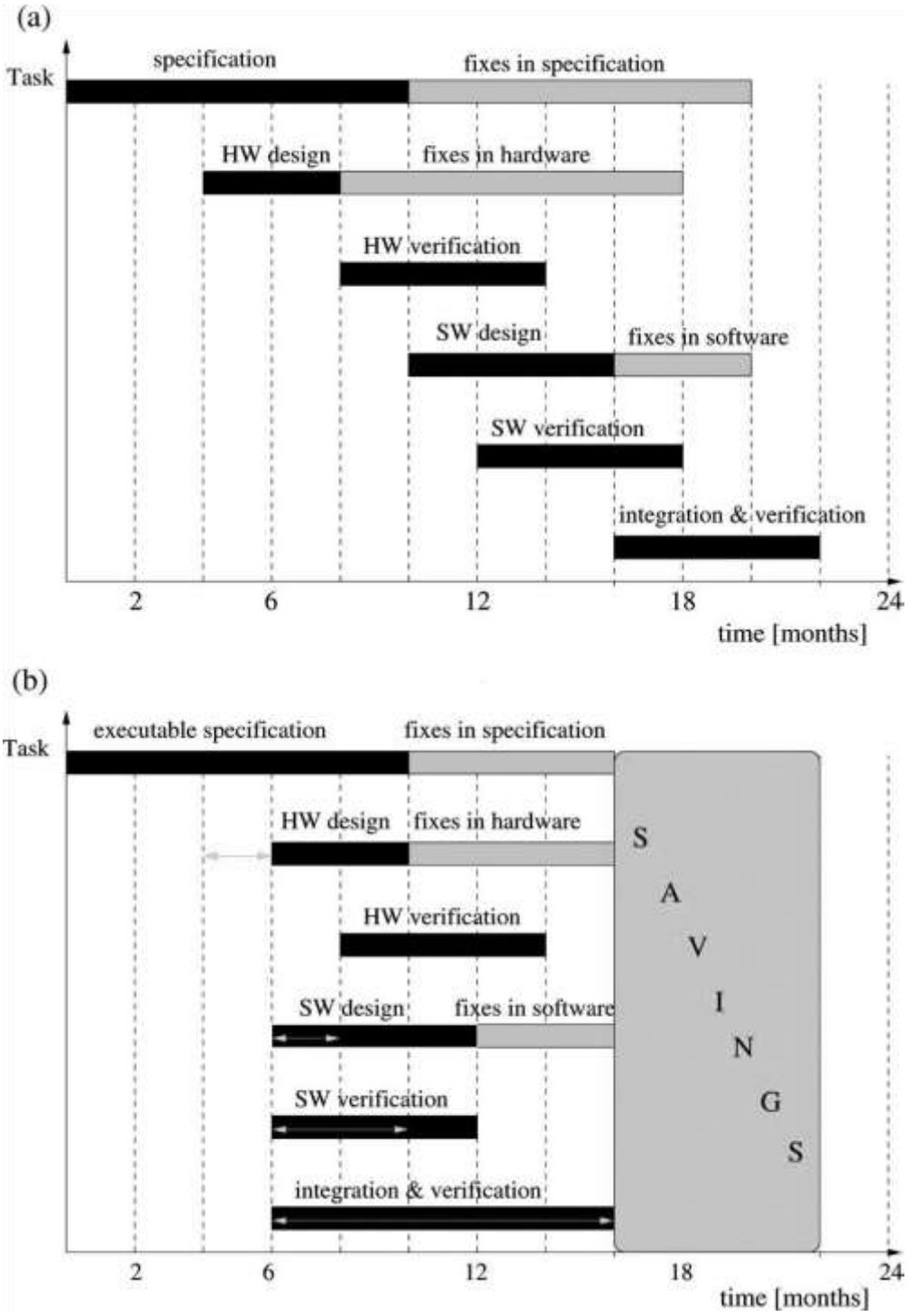


Рис. 1. Классический (a) маршрут проектирования и маршрут проектирования ESL (b) [1].

Использование современных, основанных на ESL, инструментов кодизайна, позволяет получить значительный выигрыш по времени выхода на рынок. Процесс разработки с использованием методов кодизайна начинается с создания на ранних этапах проектирования функциональной, но уже исполняемой, спецификации системы (например, на языках C, C++ или SystemC). Кроме того, в зависимости от существующей платформы или аппаратных IP, создаётся модель платформы, включающая в себя

модели стоимости. Благодаря этим начальным накладным расходам на моделирование, становится возможным раннее исследование пространства потенциальных решений при выборе системных компонентов, их соединений, компоновки памяти, распределения программных функций. При этом, конечно, моделирование архитектуры системы, параметризация моделей для исследования пространства проектных решений, калибровка моделей соответствующим образом может занять значительное количество времени.

По окончании исследования пространства проектных решений и нахождения некоторого оптимального варианта производится разделение на программные и аппаратные части, создаются спецификации на их разработку, и запускается скоординированное проектирование этих частей в параллели, с последующей взаимной интеграцией. Одновременная разработка ПО и аппаратуры как-раз и является ключевой, отличительной особенностью кодизайна.

Бок о бок с самим понятием кодизайна идут родственные ему – косимуляция (Co-симуляция) и коверификация (Co-верификация). Косимуляция позволяет производить запуск проектируемого ПО на модели проектируемой, но ещё не существующей аппаратуры, благодаря чему возможны как ранняя отладка ПО на целевой аппаратуре до её физической реализации, так и оценка параметров будущей системы. Коверификация позволяет осуществлять раннюю верификацию ПО (формальную оценку корректности его выполнения), также используя некоторую работающую модель аппаратуры, что позволяет наряду с проверкой ПО производить обнаружение и исправление возможных аппаратных ошибок.

Кроме того, помимо необходимости в создании спецификаций, формальном анализе и инструментах ко-симуляции для анализа характеристик производительности разрабатываемого продукта и затрат на его создание, одной из значимых составляющих процесса проектирования с применением кодизайна является синтез. Важно понимать, что синтез на всех уровнях абстракции является очень схожим процессом – по определённым правилам, элементам более высокого уровня абстракции в соответствие ставятся реализующие их элементы более низкого уровня. Выбор элементов, правила их компоновки могут зависеть от выбранной оптимизации. Например, простейший случай – выбор для аппаратной реализации операции сложения сумматора – может отображать одну и ту же операцию на сумматоре с низким энергопотреблением или с высоким быстродействием, и выбор будет происходить в зависимости от требований к проекту. Однако, к сожалению, подобные техники автоматического синтеза доступны далеко не для всех уровней представления ВС.

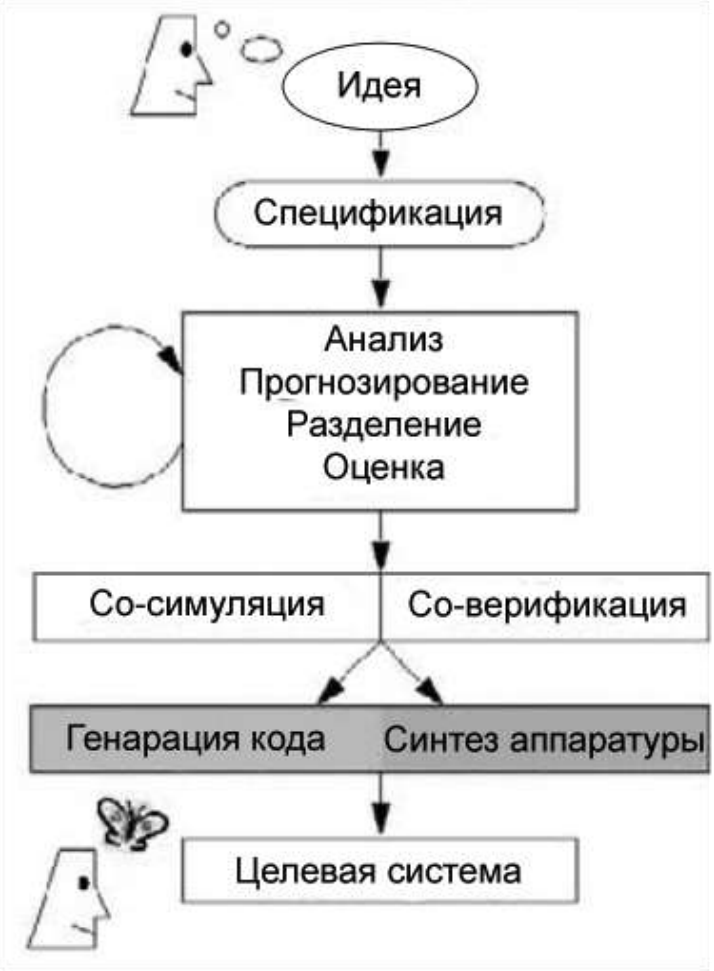


Рис. 2. Современный маршрут проектирования на основе кодизайна.

Кодизайн не только предоставляет вспомогательные средства для проектирования на системном уровне. Одновременно, он должен позволять объединять существующие полу- и автоматизированные этапы проектирования и предоставлять интерфейсы для соединения различных уровней абстракции. Идеальным является случай, когда кодизайн выполняет всю необходимую детализацию проекта автоматически, позволяя экономить время и обеспечивая быструю верификацию этапов проектирования.

1.1.1 Ретроспектива развития кодизайна

Jurgen Teich в своей статье “Hardware/Software Codesign: The Past, the Present, and Predicting the Future” [1] приводит краткий обзор истории развития направления «кодизайн». Кодизайн появился в начале 90-х годов XX века как новая дисциплина для проектирования сложных интегральных схем (ИС), включающих программируемые процессоры. Тогда параллельное проектирование аппаратного и программного обеспечения уже было достаточно часто встречающейся практикой, по крайней мере, в микропроцессорных компаниях, где решались вопросы разработки

интерфейсов между аппаратными средствами микропроцессоров и их программным обеспечением. Эта задача, включавшая определение и реализацию архитектуры системы команд, не рассматривалась как задача ко-дизайна, но она была направлена на достижение тех же целей и решение тех же проблем, что и сегодняшние методологии ко-дизайна (эти цели включают автоматизацию проектирования на системном уровне (SLD), разработку корректных электронных систем, включающих миллиарды транзисторов, выполнение программ с миллионами строк кода, и наконец, интеграцию множества микропроцессоров на однокристальной схеме [системе на кристалле, СнК], в пределах типичного 18-24-месячного времени выхода на рынок).

Проблема ко-синтеза была впервые сформулирована в работе Пракаша и Паркера [3] в виде смешанной целочисленной линейной программы, которая одновременно определяет топологию многопроцессорной системы и распределение конкретных задач по времени и между архитектурами. В первые годы много внимания было уделено проблеме разделения заданной функциональной спецификации на две части – на аппаратное и программное обеспечение. Спецификация определялась на С подобном языке и отображалась на заданную аппаратную платформу одного центрального процессора (ЦП) и определяемый специализированный аппаратный блок или интегральную схему (ASIC). Обе части связывались по шине и использовали разделяемую память или регистры для реализации буфера – т. е., за исключением вопроса, какие части реализовать на ASIC, целевая платформа была уже фиксированной.

Среди представителей первого поколения ко-дизайна можно выделить два довольно комплементарных подхода. В проекте Vulcan, разработанном Gupta и др. в Стэнфорде, Калифорния [4], идея состояла в том, чтобы с целью снизить затраты на проектирование начать с чисто аппаратного решения и переместить насколько возможно больше задач в программное обеспечение, контролируя при этом ограничения по времени выполнения. Система проектирования Cosyma, разработанная в то же время в Техническом университете Braunschweig [5], подошла к проблеме с другого конца – для них отправной точкой было разделение блоков программного обеспечения, с последующей миграцией задач в аппаратные средства для удовлетворения ограничений производительности.

Несмотря на то что эти два подхода включали высокоуровневый аппаратный синтез в пределах цикла разделения для оценки производительности различных аппаратных реализаций алгоритмов, они были достаточно ограничены – как простой архитектурной моделью, так и моделью выполнения: предполагалась реализация в одном потоке, ЦП и ASIC функционировали поочерёдно, и ЦП должен был ожидать в

нерабочем режиме, пока аппаратура завершит выполнение своей функции. Однако, эти проекты стали фундаментом для большого количества последующих работ по исследованию методов разделения и создания архитектурных расширений.

Уже до начала 2000-х была хорошо проработана проблема разделения аппаратных средств и программного обеспечения, значительно расширившись для более сложных типов архитектур. Ограничение на однопоточное выполнение программ было убрано, появилась возможность использования мультипрограммирования и многопроцессорной обработки. Кроме того, ко-симуляция [6,7] стала всё более важной областью исследования, предоставляя возможность ранней проверки допустимости проектных решений.

При ко-симуляции выполнение программного обеспечения моделируется на ЦП с использованием виртуальной модели аппаратных средств процессора или синтезируемой аппаратной спецификации разрабатываемой системы. На вентильном уровне или RTL уровне такие системы не моделируются из-за большой сложности и, следовательно, слишком медленной симуляции. Согласно [8], проблема ко-симуляции заключается в связывании различных моделей, необходимом для того, чтобы осуществлять моделирование аппаратуры с приемлемой точностью. Одним из первых решений данной проблемы того времени является Seamless CVS от Mentor Graphics, в котором использовалась модель процессора – симулятор системы команд и модели шин, для абстрагирования взаимодействия процессора с памятью. Вообще, может использоваться еще более высокоуровневое поведенческое моделирование ПО, абстрагирующееся полностью от аппаратных средств и только моделирующее временные характеристики интерфейсов, что позволяет производить симуляцию на любом хосте, на котором имеется инструментарий для моделирования аппаратных частей.

Развивались также и формальные методы для временного анализа, особенно алгоритмы Li и Malik для анализа наихудшего времени исполнения (worst case execution time, WCET) [9] и анализа методов планирования, позволяющих оценить синхронизацию смешанных программно-аппаратных систем в лучшем, худшем и среднем случаях.

Поскольку производительность – один из наиболее критичных для оптимизации факторов или используется в качестве ограничения, также заслуживают внимания вычисления в реальном времени, разработанные Thiele и др. [10], и работы по композиционному временному анализу в исполнении Ernst и др. [11].

Сегодняшние сложные архитектуры далеко ушли от допущений, принятых в этих ранних работах, современные СнК могут содержать

несколько ЦП различных типов. В 1998 был опубликован основанный на (графе) подход [12]. При помощи него можно формально моделировать неоднородную целевую архитектуру, включая соединения между процессорами. Определяется граф спецификации, состоящий из графа задачи с зависимостями от данных, и граф архитектуры, описывающий разнообразие доступных аппаратных компонентов и их средств связи. В этой основанной на графе модели используются дуги между задачами и ресурсами, чтобы описать ограниченные возможности отображения. Каждая дуга отображения может быть аннотирована любым количеством атрибутов стоимости, таких как размер кода, электропитание и другие, связанные с отображением качественные атрибуты. Задержки на транспортировку данных по коммуникационным каналам рассматриваются и отображаются в виде так называемых коммуникационных задач между информационно-зависимыми задачами. Они отображаются на коммуникационные ресурсы как неотъемлемая часть проблемы разделения. Эти идеи быстро распространились и были значительно усовершенствованы под названием платформно-ориентированного проектирования.

Таким образом, ранние подходы деления на две части, основанные на допущении, что архитектура задана и фиксирована, сегодня не применимы. Кроме того, методы оптимизации, такие как минимизация стоимости аппаратуры при ограничениях производительности или максимизация производительности при ограничениях стоимости, были признаны сильно ограничивающими пространство решений и слишком специализированными – ведь у каждого изделия могут быть абсолютно различные цели, включая стоимость, потребление энергии, надежность. Чтобы иметь возможность исследовать пространство проектирования различных аппаратных решений и вариантов разделений, важно не только позволить разработчику реализовывать его собственные функции оценки среды автоматизированного проектирования (CAD) для ко-дизайна, но также и позволить многим целям быть рассмотренными одновременно и без любого априорного смещения или взвешивания.

Наконец, проблема синтеза интерфейса между аппаратными и программными частями СнК также была признана важным полем исследований; см., например, [13]. В этой области были исследованы методы автоматической генерации схемы сопряжения из временных диаграмм или сетей Петри. Кроме того, важная проблема автоматической детализации абстрактных протоколов связи, таких, как в сетях процессов для протоколов шин, попадает в эту область. Одна из первых ранних попыток не только предлагает синтез для одного только аппаратного и программного обеспечения, но также и методы создания интерфейсов – это система CoWare [14].

1.2 Современный кодизайн на основе ESL

Считается, что сегодня мы уже живём в третьем поколении технологии ко-дизайна, который обусловлен постепенным появлением инструментов для сквозного синтеза сложных электронных систем. В течение первого десятилетия XXI века имел место быть значительный прогресс, в основном вызванный решением следующих проблем [1]:

- Технология гетерогенных СнК стала реальностью благодаря достижениям в микро- и нано-электронике. Сложные системы, сочетающие несколько микропроцессоров различных типов (DSP, ASIP, микроконтроллеров), могут быть интегрированы в одну СнК. вместе с IP-блоками аппаратных ускорителей, аналоговых устройств и блоками памяти. Сегодня востребованы методы её эффективного использования.
- Сложность аппаратуры и ПО постоянно растут, и необходимы средства для борьбы с нею. Сложность аппаратуры проявляется не только в технологии СнК, но и на уровне распределенных систем, состоящих из взаимодействующих электронных устройств – таких, как электронные блоки управления (ЭБУ) современных автомобилей. В них десятки взаимосвязанных ЭБУ, предоставляющих специальные функции.
- Требуется панацея интеграции. С точки зрения разработчика ВсС важнейшим препятствием было и до сих пор остаётся отсутствие стандартов на интеграцию подсистем, разработанных даже другими компаниями, для обеспечения приемлемых сроков выведения продуктов на рынок. Это, опять же, в особенности верно для области автомобилестроения. В ней интеграция блоков начинается на тестовом стенде, где соединяются различные подсистемы, и трудоёмкие сценарии тестирования применяются для анализа функциональной корректности и выявления потенциальных временных ошибок до того, как происходит интеграция в автомобиль. Причём эти тесты производятся очень часто без знания и наличия ПО каждой подсистемы, так как они разрабатываются поставщиками.

Вышеупомянутое привело к пониманию того, что:

- необходимо повышать уровень абстракции, на котором разработчики представляют свои системы при проектировании,
- необходим способ комбинирования и повторного использования проектов сквозь различные уровни абстракции.

Это дало рождение идее проектирования электроники на системном уровне – ESL.

Фундаментальный принцип ESL-проектирования [15] – детализация и усложнение абстракций при некоторой фиксированной проектной концепции. В течение процесса проектирования проект представляется на различных уровнях абстракции, отличающихся степенью детализации.

В качестве уровней абстракции выделяются:

- Требования рынка товаров. Диктуют функционал, предоставляемый конечному пользователю, и физические требования к продукту.
- Функциональная спецификация. Описывает функциональные требования к изделию как к чёрному ящику.
- Архитектурная модель. Показывает разделение аппаратной и программной частей.
- Спецификации проектных решений аппаратной и программной частей. Описывают требования к реализации аппаратной и программной частей.
- Функциональные и поведенческие модели аппаратной и программной частей. Реализуют алгоритмы, требуемые последующими моделями с приблизительной точностью моделирования времени выполнения или вообще без учёта временного аспекта.
- Модели RTL и программного обеспечения. Синхронизированные поведенческие модели аппаратуры и ПО.
- Модель на уровне вентилей и встраиваемое ПО. Модель на уровне вентилей синтезируется из RTL, встраиваемое ПО транслируется машинным образом из модели ПО, или, что чаще встречается сейчас – пишется вручную.
- Базы данных связей аппаратных компонентов. Задаёт структурную геометрию и соединения всех аппаратных элементов.

Концепция проекта ("design intent") возникает как результат коллективного мыслительного труда маркетологов, системных архитекторов и разработчиков. Содержание процесса проектирования можно представить, как информацию, передаваемую через некий воображаемый канал передачи данных – канал проектирования. С каждой итерацией эта информация, представленная в виде модели, детализируется и дополняется, становится более точной, в конце концов превращаясь в

базу данных связей аппаратных компонентов и низкоуровневое ПО. Другими словами, эти трансформации можно представить через удаления нескольких степеней свободы на каждом шаге повышения детализации, после принятия определённых архитектурных решений, которые, в конце концов, приводят к конкретной реализации заданной спецификации. Начальные этапы, вплоть до создания моделей RTL и ПО, производятся вручную, остальные – частично автоматизированы.

Таким образом, разработка на основе модели заменяет традиционное проектирование, заключающееся в последовательности «требования – анализ – проектирование – реализация – верификация» одной последовательной доработкой начальной модели с её итеративной детализацией. На каждом следующем уровне она всё больше уточняется, становится менее абстрактной, и происходит верификация некоего очередного, дополнительного уровня детализации.

Принципы ESL наглядно демонстрирует так называемая модель «двойной крыши» (Рис. 3, [1]). Данная модель отображает проблему синтеза электронных систем, которая включает в себя три больших задачи так называемого отображения [1].

- ***Выделение***: выделение и отображение системных ресурсов, включая процессоры, аппаратные IP-блоки (интерфейсы, запоминающие устройства и т.д.), и их сети соединений, и таким образом формирование архитектуры системы с точки зрения ресурсов. Эти ресурсы могут существовать как библиотечные шаблоны, но, с другой стороны, маршрут проектирования должен иметь возможность синтезировать их.
- ***Привязка***: отображение: функциональности (т.е. задач, процессов, функций или базовых блоков) – на конкретные вычислительные ресурсы, переменных и структур данных – на запоминающие устройства, коммуникаций – на маршруты между соответствующими ресурсами. В современных системах привязка не только рассматривает взаимно разделяемую память и простые шинные коммуникации, но также должна определять маршруты от источников до места назначения, отражая очень сложные сети коммуникационных соединений, включая также и сети на кристаллах.
- ***Планирование***: определение временных аспектов выполнения конкретных функций на выделенных им ресурсах, включая само выполнение функции, обращения к памяти и коммуникацию с другими блоками. Может включить определение частичного порядка выполнения или

спецификацию планировщиков для каждого ЦП, ресурсов коммуникации и памяти, наряду с приоритетами задач и т.д.

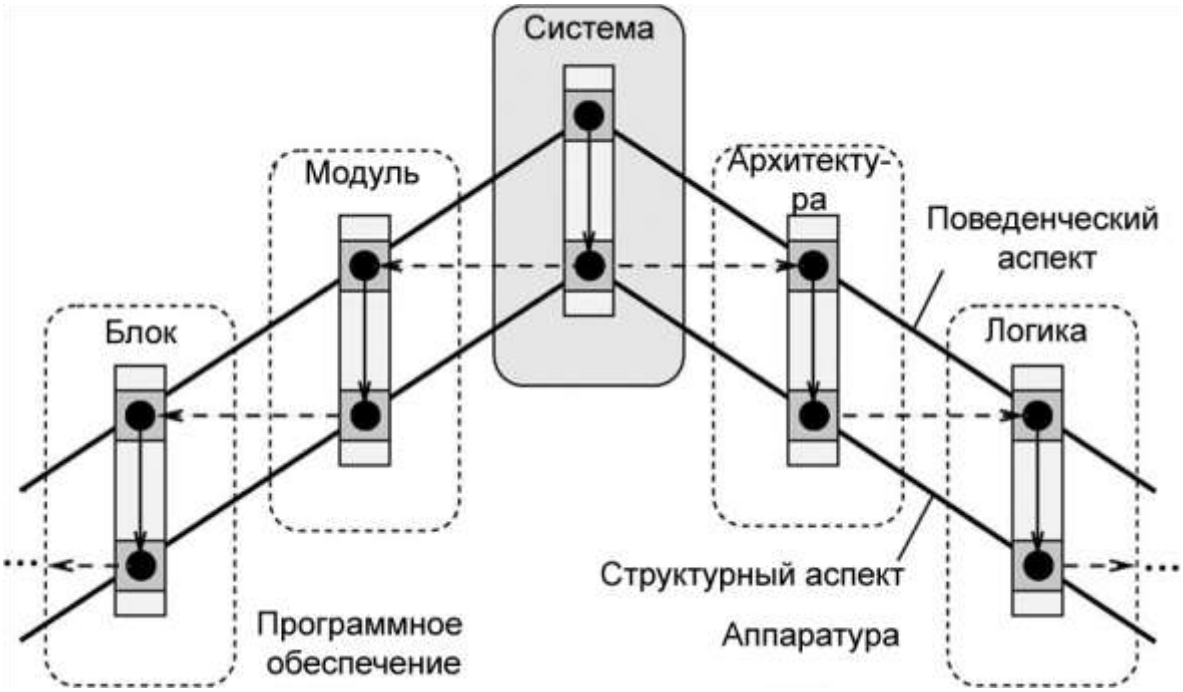


Рис. 3. Модель «двойной крыши» [1].

В модели с двойной крышей системный уровень представления ВС, на котором еще нельзя различить аппаратуру и ПО, изображён коньком крыши. Системный уровень соединяет процессы разработки программного обеспечения (скат слева) и аппаратуры (скат справа) путем последовательного детализирующего синтеза (вертикальные стрелки, идущие вниз). Верхний свод описывает функциональный вид или вид спецификации системы на соответствующем уровне абстракции, тогда как нижний свод описывает свою структурную реализацию, включая назначенные ресурсы, решения планирования, привязки и соответствующий код. Каждый шаг синтеза отображает функциональную спецификацию на структурную реализацию на следующем, более низком уровне абстракции.

Посредством каждого этапа синтеза спецификация преобразуется в реализацию на следующем, более низком уровне абстракции. Горизонтальные стрелки указывают шаги передачи информации о реализации на определенном уровне к следующему, более низкому уровню абстракции. Каждый шаг синтеза добавляет дополнительную информацию о спецификации или ограничениях. Например, на уровне архитектуры на аппаратной стороне назначение включило бы определение количества функциональных блоков, таких как множители и сумматоры каждого типа, для составления канала передачи данных следующего аппаратного компонента. Здесь информация о выборе функционального блока, такого

как сумматор с переносом с определенной точностью, служила бы дополнительным входным сигналом на функциональном уровне для последующего логического синтеза этого сумматора.

Конечно, сегодня нет инструментария для полностью автоматизированного маршрута проектирования на всех показанных уровнях абстракции. Кроме того, в разных компаниях используются разные инструменты, и каждый проект может потребовать различных специализированных шагов детализации и средств для их реализации. Чисто нисходящее проектирование является невозможным или нежелательным для многих компаний и продуктов – некоторые компоненты, такие как типы и количество ЦП, существующие блоки IP, имеющиеся в компании, могут задаваться заранее, ещё в задании. Использование существующих компонентов может быть достигнуто ограничением задачи синтеза, использованием предварительно выделенных компонентов, сужением пространства проектирования.

Типичный маршрут проектирования, представленный моделью двойной крыши, обычно начинается со спецификации ESL. Это функциональная (поведенческая) спецификация всей системы, которая либо представляет собой модель, либо языковое описание. Дополнительно, задается ряд ограничений (максимальная занимаемая на кристалле площадь, минимальная пропускная способность и т.д.). Модель платформы в ESL обычно – структурная модель, состоящая из архитектурных компонентов, таких как процессоры, шины, сети на кристалле (NoC), запоминающие устройства и аппаратные IP-блоки, использующиеся в качестве ускорителей или внешних коммуникационных интерфейсов. Задача синтеза ESL – процесс выбора соответствующей архитектуры платформы из этого разнообразия, определение отображения поведенческой модели на эту архитектуру и генерация соответствующей реализации поведения для данной платформы. Результатом является детализированная модель, содержащая все проектные решения и обычно множество нефункциональных метрик качества, таких как пропускная способность, задержка на обработку данных, количество элементарных ячеек и энергопотребление. Компоненты этой уточнённой модели, если принимается решение их выбора, затем используются в качестве условий для процесса проектирования на более низких уровнях абстракции. Как правило, каждый аппаратный компонент и программный процессор в архитектуре системы может проектироваться в дальнейшем отдельно.

Синтез на более низких уровнях абстракции – подобный процесс. Здесь также детализируются поведенческие или функциональные спецификации в структурные реализации. В зависимости от уровня абстракции гранулярность объектов, с которыми ведётся работа во время синтеза, отличается, и некоторые решения могут быть более важными, чем

другие. Три главные задачи проектирования, которые должны быть обеспечены любым инструментальным средством синтеза (проблемы назначения ресурсов, привязки к ним конкретных вычислительных задач и их планирования), существуют на каждом из представленных уровней абстракции. Отличия в них – это, главным образом, различие типов ресурсов, целей и методов оптимизации. Но суть, абстрактное содержание – одно для всех.

Модель двойной крыши пытается не только рассмотреть в перспективе системный уровень как новый и важный уровень абстракции для проектирования электронных встроенных систем, но также и связывает существующие уровни абстракций проектирования и синтеза для их объединения и взаимодействия. Интеграция определенных средств проектирования между уровнями абстракции не всегда происходит свободно и легко достижима, это все еще нуждается в значительной «доработке напильником» и адаптируемости. Настраиваемость инструментария – важное условие при ESL-проектировании, поскольку каждая компания может иметь желание настроить свой собственный комплект инструментальных средств. Даже для каждого отдельного изделия такая цепочка должна быть настраиваемой, в соответствии с различными целями проектирования.

1.3 Типовой маршрут ESL-проектирования

Идеализированный процесс ESL-проектирования можно разбить на шесть шагов, которые параллельны вышеупомянутым уровням абстракций [15]:

- создание спецификации и моделирование;
- предварительный анализ перед разделением на аппаратные и программные части;
- разделение на аппаратные и программные части;
- анализ, отладка и устранение обнаруженных ошибок после разделения;
- верификация после разделения;
- реализация аппаратуры;
- реализация ПО;
- верификация реализации.

В реальности проектирование не всегда идёт чётко сверху вниз – например, отправной точкой процесса могут быть существующие платформы, библиотеки, разработки и т.д. В то же время проектирование может быть представлено как сходящаяся спираль – требуется множество

повторений всех итераций для получения окончательного готового продукта.

Первый шаг – ***написание спецификаций и моделирование***. Он представляет собой разработку документов, описывающих систему или концепцию проекта и его ограничения, а также их трансляцию в различные исполняемые или декларативные модели. Спецификации и ограничения на этом этапе пишутся на обычном языке, что обеспечивает некоторую свободу трактовки и не ограничивает какой-то конкретной реализацией модели. Контроль сложности проекта начинается уже на уровне спецификации и заключается в разделении системы на подсистемы и компоненты, даже если они не предполагаются для конкретных реализаций.

Для ESL-проектирования неприемлем традиционный подход, когда для разработки ПО требуется достаточно стабильная аппаратная платформа – аппаратура и ПО проектируются параллельно для того, чтобы избежать сюрпризов на финальном этапе интеграции. Для сохранения требований к конечному продукту в процессе проектирования используется система отслеживания требований. В идеале модель ПО должна быть создана первой, чтобы анализ до разделения на аппаратную и программную части выявил требования со стороны ПО к аппаратуре.

В дополнение к спецификации на обычном языке, существует класс исполняемых спецификаций, которые представляют собой функциональные или поведенческие описания системы, являются моделью будущего изделия как чёрного ящика. В этот класс попадают исполняемые архитектурные и проектные спецификации. Первые являются средством для демонстрации альтернатив реализаций проекта и концепции выбранного решения. Вторые – отражают ключевые структурные решения реализации микроархитектуры, представляют систему прозрачной. Типичные языки для создания ESL-спецификаций: MATLAB M-Code, SystemC, SystemVerilog, и т.д.

Анализ перед разделением на аппаратные и программные части (pre-partitioning analysis) – процесс исследования спектра алгоритмических компромиссов. Его целью является нахождение некоторой золотой середины между производительностью, занимаемым на кристалле объёмом, энергопотреблением, сложностью реализации и временем, необходимым на доведение разработки до некоторого окончательного вида, приемлемого для конечного пользователя. В ходе анализа сравниваются затраты и предполагаемый положительный эффект при выборе различных вариантов реализации системы. Для их оценки можно использовать различные сценарии и методы – статический анализ,

платформно-ориентированное проектирование, динамический анализ, алгоритмический анализ и т.д.

Статический анализ может использоваться при отсутствии исполняемой спецификации. Существует некоторое количество техник подобного анализа, например – анализ сложности системы (system complexity analysis) и "ility" анализ. Название последнего происходит от английского суффикса "-ility", а суть заключается в анализе таких атрибутов, как надёжность, ремонтопригодность, удобство, критичность и т.д. (в английском языке эти слова оканчиваются на "-ility"). Анализ сложности системы базируется на методе из программной инженерии – анализе функциональных точек (function point analysis), который позволяет предсказывать характеристики кода на ранних стадиях проектирования и позволяет рассчитывать с некоторой погрешностью системные характеристики – энергопотребление, производительность, затраты на разработку, а также выявить предпочтительные для заданных ограничений варианты разделения на аппаратные и программные модули.

Платформно-ориентированное проектирование – интенсивное повторное использование существующих элементов, обычно представляемых в виде чёрных ящиков, которые без ущерба функциональным требованиям предоставляют лучшую производительность, энергопотребление или другие преимущества по сравнению с некоторыми стандартными аналогами. Правда, у данного подхода есть недостаток – при его применении команда разработчиков невольно начинает скатываться в традиционное проектирование. Другими словами, наличие существующей платформы непреднамеренно влияет на решения по разбиению системы.

Динамический анализ даёт возможность получить и более точную информацию, которую можно извлечь только из исполняемых моделей. Симуляция этих моделей позволяет оценивать задержки и пропускные способности вычислительных трактов, влияние политик планирования, вычислительные и коммуникационные накладные расходы на вычислительные операции.

В дополнение к анализу информации, получаемой из статического и динамического анализа, требуется также рассмотрение моделей вычислений при помощи алгоритмического анализа. Его цель – оценка потребностей в ресурсах, а также оценка таких эксплуатационных параметров системы, как загрузка узлов (computational load), требования к каналам передачи данных, вероятности возникновения ошибочных битов в потоках данных (bit error rates).

Важно помнить об области применения исполняемых моделей. Так как моделирование по определению содержит некоторые артефакты

реализации, которые могут искажать получаемые данные, исполняемые модели следует использовать для доступа только к высокоуровневым характеристикам проекта, которые относительно независимы от подобных искажений. Под артефактами понимается некоторое поведение моделей, которое является лишь побочным результатом их реализации.

Разделение – процесс, в ходе которого производится выбор, какие алгоритмы (или их части), заданные в спецификации, реализовывать в виде запускаемого на процессорах программного обеспечения (также, на этом этапе выбираются типы процессоров), какие – в виде чисто аппаратных компонентов. Происходит разделение алгоритмов между аппаратными и программными частями. Процесс разделения можно рассматривать в следующем порядке: *функциональная декомпозиция*, *архитектурная декомпозиция, разделение, разделение аппаратной части*, *разделение программной части*, *определение реконфигурируемых частей*, и *реализация коммуникаций*.

В *функциональной декомпозиции* за отправную точку берётся спецификация, настолько свободная от артефактов реализации, насколько это возможно, и позволяет получить представление о возможностях распараллеливания на уровне приложений. Эти требования можно удовлетворять двумя способами: использованием функционального параллельного языка для создания исполняемых спецификаций или же последовательного, но имеющего поддержку инструментов автоматического извлечения параллелизма. Пример параллельного языка – Simulink от MathWorks. Применение же второго подхода, несмотря на его большую привлекательность, является достаточно сложной задачей и не имеет универсальных решений.

В *архитектурном описании* принято использовать элементарные блоки, каждый из которых реализует какую-то функцию: хранение данных, вычисления, связи с другими блоками. Некоторые из этих блоков – фиксированные или конфигурируемые процессоры, произвольные или стандартные аппаратные модули, шины, блоки памяти, операционные системы и сервисы, API, стеки протоколов. Они используются для отображения функционального описания на общую архитектуру, часто – платформу. В простых случаях начальное отображение функций на конкретные архитектурные элементы чаще всего отражает конечное выбранное аппаратное разделение. Однако, в более сложных проектах не все функции отображаются непосредственно на какие-то конкретные аппаратные компоненты, а могут соответствовать, например, объектам объектно-ориентированного программного обеспечения. Для компенсации затрат на разработку таких сложных архитектур они реализуются как платформы – параметризированные аппаратно-программные топологии,

конфигурируемые для некоторого диапазона применений (примеры – Nexperia и OMAP).

Параллельное приложение может быть *разделено* либо через последовательную детализацию, либо используя явно заданное указание отображения, понятное моделями и инструментами синтеза. При явном задании отображения начальная функциональная модель остаётся неизменной, меняются только коммуникационные примитивы. Это ключевое отличие от процесса последовательной детализации, каждый шаг которой добавляет некоторые детали, достаточные для трансформации функциональной модели в отображённую на конкретные элементы (при этом используется один мульти уровневый механизм моделирования). Временная точность двух ортогональных аспектов модели – коммуникационного и вычислительного – увеличивается после каждого этапа. Д. Гайски (D. Gajski) выделяет три степени временной детализации: без учёта временного аспекта (untimed), с апроксимированными временными параметрами (approximate-timed) и с тактовой точностью (cycle-timed).

После уточнения или отображения абстрактных моделей на архитектуру требуется *разделить аппаратные и программные компоненты*. Аппаратное разделение может быть реализовано в виде одного или нескольких процессоров распределённой системы. Однако, системное проектирование аппаратуры всё больше становится ничем иным, как просто конфигурированием платформы, когда основная масса работы в проектировании снизу-вверх перемещается в область программного обеспечения.

Программные модули состоят не только из операционной системы, библиотек и промежуточного ПО (middleware), но и из приложений, которые зачастую берут на себя основную функциональную нагрузку. Приложения, в свою очередь, могут быть разделены различными способами: между процессорами, между задачами, между разными блоками памяти. При разделении функциональности между разными процессорами они могут быть гомо- или гетерогенными, выбор зависит от параметров, которые оптимизируются. Типичная конфигурация представляет собой процессор общего назначения для реализации сетевых функций и обработки пользовательских интерфейсов и DSP для обработки радиочастотных и медийных потоков. На каждом конкретном процессоре программные функции, в свою очередь, могут быть разбиты между задачами, процессами и нитями. После такого разделения необходимо выбрать алгоритм планирования. Подобные разделения накладывают некоторые ограничения, которые следует учитывать при оптимизации производительности всей системы в целом. Также при разделении кода или данных по разным подсистемам памяти требуется учитывать

виртуализацию больших, медленных, недорогих технологий хранения небольшими, быстрыми и более дорогими. Подобные технологии кэширования хорошо изучены, но всё равно учёт их влияния – серьёзная задача при проектировании ПО на системном уровне.

Реконфигурируемые вычислители призваны заполнить пропасть между аппаратурой и программным обеспечением, позволяя добиться компромиссов в производительности, стоимости и энергопотреблении. Аппаратные вычислительные элементы могут быть реконфигурированы однажды, когда система выпускается, раз в какое-то количество лет, для устранения багов и улучшений, или же раз в несколько часов, чтобы адаптироваться под текущую задачу. Проблемой реконфигурируемых вычислителей является сложность их программирования. Подобные вычислители представлены двумя основными категориями: реконфигурируемый массив, действующий как функциональный модуль управляющего процессора, или реконфигурируемый массив, работающий как сопроцессор, подключённый к основному процессору.

Одной из самых ответственных задач при разделении является *реализация коммуникаций*, так как от выбранных протоколов зависит множество аспектов системы, таких, например, как производительность и надёжность. Существуют два подхода к реализации коммуникационных архитектур: использование шаблонных решений (template instantiation) и синтез интерфейсов. Первый способ состоит в подборе одного или нескольких существующих вариантов решений под конкретные требования. Второй возможен только при выполнении следующих требований: средняя пропускная способность передающих и принимающих интерфейсов примерно одинаковы, т.е. не используются глубокие буферы, и на сигнальном уровне протоколы должны иметь семантику конечного автомата для возможности автоматического или основанного на языковом описании синтеза. При удовлетворении этих условий могут использоваться некоторые стратегии синтеза.

После разделения следует обязательный и важный этап *анализа и верификации*. Он состоит из двух шагов. Первый заключается в реализации алгоритмов, которые предназначаются для выполнения на процессорах и написаны на одном из языков программирования, т.е. в их компиляции для конкретных процессорных архитектур. Второй – в моделировании на поведенческом уровне предназначенных для аппаратной реализации алгоритмов, при помощи одного из языков описания аппаратуры. После этих двух шагов идёт проверка соответствия полученной модели требованиям спецификации, в ходе чего может выясниться, что необходимо снова вернуться к этапу разделения для оптимизации разрабатываемой системы.

Моделирование аппаратуры и ПО всей системы ограничивается целями моделирования (исследование параметров проекта, валидация, верификация), которые должны быть не слишком всеобъемлющими, для возможности создания модели. Кроме того, при комплексном и всеобъемлющем удовлетворении требованиям одной из этих целей, как правило, другие страдают, поэтому необходимо искать компромиссы и расставлять приоритеты для информации, которую требуется получить из модели. Модели для аппаратуры и ПО должны иметь возможность исполняться параллельно. Существует три варианта моделирования программных элементов:

- общая модель аппаратуры и ПО,
- модель только ПО с адаптерами для аппаратных коммуникаций (hardware communication adapters),
- модели аппаратуры и ПО отдельно.

После получения системы с начальным разделением – хотя бы представленным отмеченными компонентами – можно начинать получение более детальных оценок производительности, энергопотребления и стоимости. В подобном раннем разделении обычно используется сеть взаимодействующих процессов (network of communicating processes), когда варьируется сама сеть для "what-if" анализа. При определении пределов пропускной способности для каждой конфигурации сети используются неограниченные очереди, буферизирующие различные скорости передачи между элементами. Для более детального анализа и предварительной оценки производительности и потребляемой мощности могут использоваться веса и временные метки данных. Также может понадобиться реализация самих интерфейсов для получения уверенности, что они вообще могут быть реализованы. После подобной проверки обычно происходит возврат к более абстрактным интерфейсам для дальнейшего анализа.

Способ моделирования компонентов моделей до разделения и после должны быть совместимы. Смысл разделения состоит не только в распределении алгоритма на аппаратные и программные компоненты, но и в разделении его на составные части, что обусловлено особенностью человеческого мозга обрабатывать только конечное число информации в один момент времени. Так как детали добавляются на каждом этапе детализации абстракции, растёт сложность и наполненность информацией модели, понимать и контролировать её всю целиком становится сложно – требуется разбиение. Разделение на отдельные части и разделение каждой части аппаратной или программной реализации – отдельные задачи. Это позволяет смешивать разные элементы в процессе анализа и даже использовать вместе модели до и после разделения, если их

функциональность явно не пересекается с коммуникационными интерфейсами. Благодаря этому можно исследовать разные направления реализации отдельных частей системы без необходимости расписывать всю систему на более глубоких уровнях абстракции, что существенно экономит затраты.

Абстракции используются для ограничения количества информации и "сложностей", которые воспринимаются в каждый момент времени. Без принятых договорённостей, определяющих уровни абстракций, каждый инженер, как правило, имеет собственное представление об их границах. Наличие же подобных договорённостей приводят к согласованным стандартам уровней абстракции, что позволяет измерять вдоль каждой оси абстракции (временной, данных, функциональной и структурной) то, что определяет данный уровень. Также задаётся стандартный набор полезных уровней абстракции: вид с точки зрения программиста – PV (Programmers View), вид с точки зрения программиста с учётом времени – PV+T (Programmers View+timing), с тактовой точностью – cycle callable, на уровне языка описания аппаратуры – RTL. В добавление к стандартным уровням абстракции, необходимы стандартные интерфейсы (API) для облегчения связей между моделями. Перемещения вдоль оси данных достаточно прямолинейны, перемещения вдоль временной оси гораздо сложнее. Информация добавляется при перемещении в сторону менее абстрактных представлений (более точных) и удаляется при перемещении в сторону более абстрактных.

В дополнение к управлению уровнями абстракции и управлению структурой моделей, необходимо задавать коммуникационные требования к каждой модели. Коммуникационный интерфейс модели диктует абстракции информации для трансляции в модель и из неё, а также временные задержки, параллелизм и конфигурации модели. Разделение функциональных и коммуникационных требований приводит к многим вытекающим из него удобствам для проектирования, верификации, конфигурации и повторного использования. Спецификация интерфейса должна быть первой документированной частью модели, так как требуется и для моделей, и для существующих IP-элементов. Она должна писаться теми, кто занимается разделением проекта, так как охватывает начальные требования к коммуникации внутри модулей.

Как можно заметить, существует большое количество моделей, необходимых для анализа и поиска ошибок после этапа разделения. Это ещё дополняется необходимостью обеспечения совместной симуляции с моделями, предшествующими этапу разделения и идущими после него. Обычно применяются следующие анализы: функциональный, производительности, интерфейсов, энергопотребления, занимаемого места, стоимости и пригодности для отладки.

Функциональный анализ, обычно относимый к функциональной верификации, требуется на этом шаге проектирования для понимания требований к размеру различных элементов хранения. Начальный *анализ производительности* производится для валидации разделения и определения размеров аппаратных и программных элементов. *Анализ интерфейсов* позволяет обнаружить рано, задолго до реализации, подходит или нет конкретный вариант коммуникационных интерфейсов модуля. *Анализ энергопотребления* выгоден на этом этапе в случае, когда модель имеет свойства скорее физической реализации, чем программной. *Анализ занимаемого места* использует метрики сложности для оценки требований к аппаратуре по занимаемому на кристалле месту. *Анализ стоимости* при помощи эвристик рассчитывает стоимость проектирования, производства, поддержки и полного жизненного цикла изделия. Наконец, *анализ пригодности для отладки* проверяет сложность и риск функциональных ошибок предполагаемой реализации аппаратуры, чтобы выяснить, какие возможности контролируемости и наблюдаемости требуются от внешних интерфейсов физической реализации чипа.

Верификация после этапа разделения должна выявить, сохранилось ли ожидаемое поведение компонентов системы разрабатываемого изделия в уточнённой модели, полученной после разделения. Это первый шаг верификации, следующий – это уже верификация реализации. В обоих случаях поведение модели на высшем уровне абстракции должно быть сравнено с поведением модели на нижнем. При этом имеются некоторые эффекты, которые отсутствуют в более грубых моделях и появляются только после детализации. Для решения этой и других проблем предпринимается три шага: планирование верификации, реализация окружения верификации и анализ результатов верификации.

Планирование верификации – перечисление в документе, возможно, в формате, пригодном для чтения какими-либо инструментами автоматизации, проблем, подлежащих верификации и их решений. Список составляется в иерархическом виде, начиная с анализа спецификации, продолжая одновременно функциями системы как чёрного ящика и как прозрачного и заканчивая перечислением моделей для верификации. Как в самом процессе ESL-проектирования, изначально мягкие требования к проектируемому изделию итеративно ужесточаются вплоть до этапа реализации, так и процесс верификации итеративно ограничивает пространство допустимых поведений системы, приходя к описанию её операционного пространства. Однажды описанное, это пространство может быть исследовано статически через формальный анализ или динамически, через симуляцию. Решение проблемы верификации с использованием статических и динамических методов описывается во второй половине плана верификации, служащей в качестве

функциональной спецификации для компонентов, для которых план написан.

Когда область интереса для верификации определяется, а план верификации и функциональная спецификация для верификационного окружения написана, создаётся само *верификационное окружение*, обычно с использованием высокоуровневых языков верификации (High-Level Verification Languages), возможно даже – аспектно-ориентированных.

После создания окружения запускается "bring-up" тест, который проверяет набор базовых операций с целью показать баги, которые могут не позволить производить верификацию проекта вообще. Результаты этих запусков должны быть проанализированы и со стороны корректности, и со стороны полноты. *Анализ корректности результатов* должен выявлять общие источники ошибок для прогонов тестов. *Поиск первопричин* – второй из трёх шагов отладки, заключающийся в нахождении багов, их диагностике и устранении. Работа по нахождению багов выполняется генерацией тестов и проверкой аспектов динамического окружения верификации или проверкой модели статического окружения. *Анализ полноты покрытия* призван выявить, почему существуют конкретные дыры в покрытии целевой области тестами.

Реализация аппаратуры – процесс создания моделей, которые могут быть синтезированы в модели вентильного уровня. Аппаратные модели обычно описываются на уровне передач сигналов между регистрами (RTL-модели), или даже на более высоком поведенческом уровне. До сих пор существует множество выборов, производящихся на этом этапе перехода от поведенческого описания к RTL-уровню, таких как разделение ресурсов и добавление конвейеров, которые могут повлиять на производительность системы, поэтому некоторый анализ всё равно всегда будет требоваться даже для проверки сгенернированных RTL-решений. Если используется синтез, также важно убедиться, что связь с оригинальным описанием сохраняется. Существует пять основных вариантов реализации аппаратуры, которые доступны на данном этапе: расширяемые (extensible) процессоры, DSP сопроцессоры, специализированные VLIW сопроцессоры, ASIP и ASIC/FPGA.

Расширяемый процессор – ЦП общего назначения, который может быть адаптирован под конкретное приложение добавлением SIMD инструкций, блоков умножения с накоплением, блоков аппаратной реализации контроля циклов ("zero-overhead looping"), дуальных элементов хранения и записи ("dual load-stores"), элементов DSP и мультиоперационных инструкций. Расширяемый процессор обычно не требует аппаратной разработки, потому что необходимая трансляция, от объявления инструкции к трансляции RTL-реализации функциональности

этой инструкции, производится автоматически инструментами, предоставляемыми многими поставщиками. Подобный процесс обычно параметризуется приоритетами оптимизации для выбора решений по реализации.

DSP – процессор, оптимизированный для выполнения алгоритмов цифровой обработки сигналов. Типичными чертами DSP являются одно тактовое умножение с накоплением, каналы параллельного доступа к памяти ("parallel memory access data paths") и аппаратная реализация контроля циклов ("zero-overhead loops"). DSP сопроцессоры подключаются к основному ЦП через шину и отображаются в его память или пространство ввода-вывода.

Специализированный *VLIW* сопроцессор оптимизирован в количестве и типе функциональных блоков для выполнения конкретного алгоритма. Их число выбирается, чтобы соответствовать количеству операций алгоритма, которые могут выполняться параллельно, а тип – на основе типов данных и операций. Некоторые производители САПР для электронных изделий предоставляют инструменты, которые генерируют VLIW сопроцессоры из описаний алгоритмов, с подключением к шине ЦПУ.

ASIP используют специализированные тракты обработки данных и конечные автоматы для реализации конкретного алгоритма. Коммерческие инструменты, как правило, для их генерации анализируют написанные на C или C++ алгоритмы.

Процесс ESL-проектирования строится поверх традиционного RTL-проектирования, состоящего из следующих шагов: создания RTL, верификации RTL, синтеза RTL в вентили (элементарные логические элементы), верификации временных задержек, размещения и трассировки вентилей, проверки правил проектирования, генерации битового потока. RTL-описание, как правило, создаётся с компонентами тракта обработки данных ("data path") и тракта управления ("control flow"), где последний представляет собой реализацию взаимодействующих конечных автоматов, которые управляют прохождением потока информации через тракт передачи данных. ESL-проектирование, состоящее из создания системной спецификации, разделения на аппаратные и программные компоненты, создания виртуального прототипа, проектирования на уровне транзакций, верификации на уровне транзакций – заканчивается, в том числе, и синтезом в RTL для аппаратных компонентов. Таким образом, RTL-описание находится на стыке, используется для обмена данными между RTL и ESL процессами проектирования.

При создании аппаратуры могут использоваться полностью специфицированные IP блоки, настраиваемые (обычно за счёт параметров)

RTL компоненты, и полностью создаваемая с нуля специализированная аппаратура. IP блоки, пригодные для повторного использования, просто добавляются в проект, зачастую с некоторыми изменениями и доработками. Чаще всего доступные параметры настраиваемых RTL компонентов довольно ограничены (ширина порта, протокол ввода-вывода), и когда появляется необходимость изменения RTL блока за границы предоставленных базовых возможностей, блок должен быть верифицирован в контексте системного проектирования как новая часть логики. ESL синтез может быть использован для генерации произвольной аппаратуры, он является развитием и заменой поведенческого синтеза благодаря преодолению многих не решённых ранее проблем, неоспоримым преимуществом является использование C/C++ и их диалектов. Многие проблемы постепенно преодолеваются и становятся решаемыми благодаря инструментам, предоставляемым разными производителями.

Процессы проектирования для ASIC и FPGA очень похожи. В обоих случаях результат – RTL код, который реализует поток данных сквозь функциональные модули, где он контролируется мультиплексорами, которые контролируются конечным автоматом (автоматами).

ESL-синтез не решает автоматически все проблемы – он всего-навсего делает более продуктивным проектирование аппаратуры трансляцией высокоуровневых моделей в RTL. Как хороший инженер, создающий RTL-описания, осознаёт компромиссы между энергопотреблением, занимаемым на кристалле местом и скоростью выполнения и использует их для достижения своих целей, так и инженер, использующий ESL-синтез, должен использовать ограничения для управления генерацией RTL.

Одним из основных преимуществ ESL-синтеза является возможность раннего исследования вариантов реализаций. Варьируя ограничения при фиксированном поведенческом описании, инженер может ознакомиться с пространством решений и их чувствительностью к конкретным параметрам.

Реализация ПО предоставляет человеко-машинный интерфейс, позволяющий использовать аппаратную часть, и представляя собой самые высокоуровневые элементы проекта. Обычно, процесс разработки ПО соответствует классической водопадной модели ("waterfall" model). ПО пишется для уже существующей аппаратуры с целью удовлетворения всех требований к продукту, включая компенсацию аппаратных багов и попытки исправить возможный недостаток производительности. Альтернативный подход – использование ESL-моделей для прототипирования программных компонентов системы со спиральным процессом разработки. В ходе подобного процесса создаются серии

реализаций, каждая оценивается со связанной с ней аппаратурой в системном контексте, и затем уточняется для исследования ограничений, обнаруженных в последней реинкарнации. Подобная последовательность позволяет аппаратуре и ПО последовательно преобразовываться в решение, которое будет соответствовать требованиям к разработке.

Для оценки производительности процессоров часто используется количественные оценки скоростей выполнения инструкций (например, MIPS). Однако, семантическая плотность ("semantic density") различных инструкций, под которой понимается количество вычислительной сложности на одну инструкцию, различается от одного набора команд к другому, из-за чего подобные оценки теряют адекватность. Необходимо измерять производительность самого алгоритма для оценки его эффективности на заданном процессоре, для чего, как правило, используются стандартные тесты.

Помимо оценки объёма вычислений, также важно оценивать объём используемой памяти и её пропускную способность. Кодирование инструкций (вроде RISC или CISC) напрямую влияет на размер инструкций и программной шины, объём занимаемой скомпилированным кодом памяти, энергопотребление на выборку инструкций. Кроме того, влияние на производительность оказывают шаблоны доступа к современным SDRAM, которые оптимизированы для последовательного доступа внутри блоков. Если есть некоторая степень свободы при реализации шаблонов обращения к адресам операндов, их желательно оптимизировать для минимизации задержек.

Языки программирования и инструменты разработки, используемые при создании ПО в стиле ESL, похожи на обычные, но с некоторыми особенностями. C продолжает быть наиболее широко используемым языком, Embedded C++ видится перспективным вариантом, так как предоставляет возможности объектно-ориентированного языка без снижения производительности, характерной для полной C++ версии. Для отладки в основном используется технология ICE – внутрисхемная отладка.

Модель ESL – уже алгоритмическая программная реализация, и исходя из этого, напрашивается вопрос: «Почему бы не использовать саму модель, как реализацию?» Если предположить, что модель полностью детализирована с точки зрения абстракции, остаётся за скобками только модель параллелизма, используемая моделью, и доступные вычислительные ресурсы. Тогда как аппаратура вполне естественно вписывается в мелкогранулярную модель параллелизма, программное обеспечение – нет. ESL модель должна использовать конструкции параллельного выполнения ("parallel operation constructs"), которые хорошо

отображаются на целевой процессор, такие как векторная арифметика и VLIW инструкции. Кроме того, необходимо предусмотреть отладку модели в контексте системной реализации. В переходе от модели к реализации должны проверяться два типа ошибок. Во-первых, отказы могут быть в самой модели, и отладочное окружение должно предоставлять возможность изолировать, останавливать, пошагово отлаживать и проверять состояние самой модели внутри работающей системы. Во-вторых, отказ в окружении, вызванный моделью, должен быть отлажен в системном контексте с видимостью для структур данных и кода на системном уровне. В дополнение к требованиям, накладываемым типами отказов, программное обеспечение модель-для-реализации ("model-to-implementation") должно сохранять достаточно информации для отображения основных причин из домена реализации на исходную модель. Проектирование для верификации ("Design-For-Verification, DFV"), учитывающее простоту обнаружения, диагностики и исправления багов – должно быть одним из первоочередных явных требований при создании модели для реализации.

Использование ESL модели как платформы для разработки и как окружения для запуска ПО позволяет проектировать ПО системы одновременно с проектированием аппаратуры, благодаря созданию окружения для запуска скомпилированного машинного кода и взаимодействия с компонентами системного уровня.

Каждая модель – компромисс между точностью и скоростью выполнения. Модели, использующиеся для разработки ПО, можно классифицировать как: охвата системы ("system scope"), точности временного домена ("time domain accuracy"), платформы для выполнения приложений ("execution platform") и производительности в реальном времени ("real-time performance"). Модели охвата системы представляют для ПО компоненты системного уровня. Точность временного домена акцентируется на величине интервала между событиями. Платформа для выполнения – механизм, способный запускать ПО. Производительность реального времени – измерение способности модели выдерживать задержку между событием системного уровня и откликом от ПО.

Благодаря высокоскоростным ESL-моделям, доступна прозрачность всего взаимодействующего ПО на многопроцессорных системах на кристалле, чьи каналы связи часто закрыты для доступа извне.

Последний шаг ESL-проектирования – ***верификация реализации***. В ходе этого этапа нужно в очередной раз убедиться, что полученная система соответствует основной концепции и целям проекта. На данном этапе детализация RTL-описания и встраиваемого ПО должна быть произведена полностью, без оставшихся неясностей. Цели проекта, с которыми

производится сравнение, берутся из моделей, полученных после разделения на программные и аппаратные компоненты, остальные входы для сравнения берутся из ограничений, заданных в спецификации. Весьма вероятен вариант, когда большая часть верификации, производящейся на данном шаге, производится на субсистемном уровне из-за требований к производительности окружения, где производится выполнение, для поднятия производительности этого окружения могут быть использованы дополнительные инструменты вроде эмуляторов или прототипирующих систем. Альтернативой является исполнение со смешением абстракций ("mixed abstraction execution"), когда некоторые части системы представляются моделями, виртуальными системными прототипами, полученными после разделения, а некоторые – моделями реализации ("implementation models"). Несколько тестов должно быть запущено для всей системы на RT уровне, и эти тесты должны быть выбраны так, чтобы возможно полнее использовать доступные временные и аппаратные ресурсы.

Фундаментальными подходами для представления сохранения проектной концепции ("preservation of design intent") являются позитивная и негативная верификации. Позитивная верификация демонстрирует или доказывает, что проект удовлетворяет требованию. Негативная верификация демонстрирует или доказывает, что в реализации нет дефектов. Позитивная верификация рассматривает требования системного уровня, в то время как негативная убеждается, что низкоуровневые компоненты – аппаратные и программные – исполнены корректно. Процесс ESL проектирования управляется высокоуровневой спецификацией, которая фиксирует все требования системного уровня. Позитивная верификация используется в процессе верификации после разделения на компоненты, чтобы убедиться, что абстрактная аппаратная модель и модель ПО (RTL и встраиваемого ПО) соответствую соответствующей абстрактной модели.

На уровне реализации абстракции – за один шаг от логических вентилей для RTL и за один шаг от машинного кода для встраиваемого ПО – заметно значительное падение производительности моделирования из-за размера и точности моделей. Для моделей аппаратного обеспечения, чтобы сохранить разумную производительность, используется обычный логический симулятор для верификации блоков и подсистем. Статический формальный анализ также широко используется на уровне блоков, чтобы избежать моделирования. Более полные интеграции аппаратной логики требуют аппаратных ускорителей или эмуляторов для достижения продуктивных скоростей выполнения верификации и отладки.

Верификация «прозрачного ящика» используется в процессе верификации реализации для проверки граничных условий и

возможностей, не предусмотренных функциональной спецификацией (как правило), но добавленных в ходе реализации вследствие принятых решений. Требуется различить неописанные ранее поведения, которые идут вразрез с базовой спецификацией, от тех, которые не конфликтуют с ней. Для этого проводится позитивная верификация, заключающаяся в сравнении полученного поведения уровня реализации и поведения, заданного на уровне спецификации, а также негативная верификация, заключающаяся в сравнении реализованного поведения и поведения абстрактной модели. В обоих случаях производится сравнение поведения на самом высоком уровне абстракции двух элементов, из чего логично вытекает возможность повторного использования некоторого однажды созданного окружения.

Программное обеспечение, написанное с учётом повторного использования – Verification IP, VIP. Причём, подразумевается, что повторно используемый продукт требует только настройки и установки, после чего готов к работе, это не простое копирование старого кода в новое окружение. Существуют два вида VIP – статические и динамические, в соответствии с методами верификации. Динамические VIP – окружения, отвечающие за генерацию тестов, проверку возвращаемых значений, измерение покрытия кода. Статические VIP – обычно идут в виде библиотеки, которая фиксирует специфические требования на RT уровне. Статические VIP также могут использоваться в динамическом окружении для запуска и отладки. VIP обычно создаются для стандартных ядер или интерфейсов (x86, PCI).

Если бы не ограничения стандартных декларативных языков ("standard assertion languages", таких как PSL и SVA), взятые из них свойства ("properties") и декларации ("assertions") можно было бы использовать в верификации. Декларации – реализованные свойства, свойства пишутся в основном на естественном языке в плане верификации. Одни и те же декларации могут использоваться для статического анализа, как и для симуляции. Это удобно, так как, если статический анализ работает только на блочном уровне, те же реализованные свойства могут проверять несоответствия требования при субсистемной симуляции и симуляции всей системы в целом.

Один из способов оценки скорости верификации – покрытие (другие включают скорость обнаружения багов, скорости изменения RTL и ПО). Существуют три разновидности покрытия, использующиеся для измерения прогресса верификации: функциональное, структурное и декларационное. Функциональное покрытие производит подсчёт требований к реализации на определённом уровне точности, используя атрибуты уровня реализации, такие как регистровые поля и задержки обработки прерываний. Структурное покрытие измеряет, насколько широко было охвачено

реализованное ПО. Декларационное покрытие – измерение распространения деклараций на всём этапе реализации, а также частоты и полноты их оценки и вызова.

После верификации отдельных компонентов желательно проверить и всю систему в целом. Симуляция RTL может быть очень медленной, потому существуют некоторые альтернативы. Помимо моделей со смешанными уровнями абстракций, аппаратных ускорителей и эмуляторов, можно использовать прототипирование в FPGA.

Однако даже при самой тщательной проверке некоторые баги остаются до самой реализации, и требуется предусмотреть возможность их диагностики. Для этого, во-первых, используется стандарт JTAG, открывающий доступ ко всем, или хотя бы многим, элементам памяти. Во-вторых, логика трассировки ("trace logic") выводит данные на внешние выходы для анализа вне чипа. В-третьих, встроенные в процессоры ICE, открывающие доступ к управлению процессорными инструкциями и дающие возможности управления программами для отладчика вне исследуемого чипа. Наконец, внутренние логические анализаторы, которые опционально могут использоваться для оценки функциональности некоторых процессоров, которая выходит за рамки ICE для доступа к микроархитектурным элементам. Когда баг находится, резервные выходы, обычно расположенные в небольших количествах в разных кремниевых слоях, могут быть использованы для коррекции неправильной логики изменениями только в металлическом слое. Кроме того, риски могут уменьшаться добавлением реконфигурируемых блоков в стратегические точки аппаратуры.

Самое важное, что надо помнить про методологию ESL – то, что это рекомендованный процесс проектирования, и в каждом конкретном случае может изменяться и подгоняться под решаемую задачу. Модификации следует документировать для сохранения возможности отслеживать проектные решения. После успешного завершения, желательно, чтобы команда разработчиков сделала обзор проекта, как с технической стороны, так и со стороны процесса проектирования, с целью усвоения полученных уроков и использования накопленного опыта в будущих разработках. Кроме того, не следует забывать, что время, потраченное на ранние этапы планирования и создания спецификаций, окупается сторицей. Ошибки, сделанные в начале проектирования, часто невозможно или очень сложно исправить.

Много вопросов в методологии остаётся открытыми. При переходе на высшие уровни абстракции детали реализации скрываются, но игнорировать их при проектировании нельзя. Неясно, что именно следует поднимать на системный уровень. Переход к более мощным процессорам

и программно-центрированному (software-centric) проектированию может изменить вид процесса проектирования. Вероятно, аппаратное проектирование станет менее значимым, всё "железо" будет генерироваться из спецификаций с достаточной степенью автоматизации. Новая методология может понадобиться для надстройки над существующей ESL, для получения новых уровней параллелизма и обеспечения коммуникаций процессов, что потребует новых уровней абстракции.

1.3.1 Пример: проектирование средствами Mentor Graphics

Mentor Graphics предоставляет инструменты для ESL-ориентированного проектирования электронных систем, дающие возможность работать над проектом, начиная с создания архитектурного описания и фиксации формальных требований и вплоть до конечной реализации в виде СнК или ПЛИС.

На самом верхнем уровне предполагается исследование пространства проектных решений и выбор оптимальной архитектуры. Для этого создаётся виртуальный прототип, реализованный на уровне транзакций, который используется для примерных оценок характеристик будущей системы и применяется в качестве симулятора целевой аппаратуры для создания ПО. При этом среда разработки ПО позволяет одинаково работать как с моделями, созданными на уровне транзакций, так и с более детализированными, в том числе с эмуляторами на основе ПЛИС и с реальными, уже реализованными в ПЛИС, СнК или ИС системами.

Рис. 4. Маршрут проектирования FPGA/CPLD/ASIC/SoC [16].

На начальном этапе возможно явное задание исходной спецификации, фиксация ограничений и требований к будущему проекту при помощи системы интерактивного отслеживания требований. Она позволяет на протяжении всего процесса разработки контролировать качество создаваемого продукта, отслеживать его соответствие начальной концепции.

Следующим шагом идеализированного нисходящего проектирования, который идёт после создания архитектурного описания и определения требуемого функционала изделия, является генерация из высокоуровневых языков программирования (C, C++, SystemC), на которых создаётся высокоуровневая спецификация будущего изделия, в языки описания аппаратуры (Verilog, VHDL), для дальнейшей доработки аппаратными инженерами. Генерацию осуществляет специальный инструмент синтеза, который позволяет получать на выходе HDL-проект, представляющий собой описание на уровне регистровых передач (RTL). Дальнейшая модернизация, отладка и анализ производится уже в отдельном пакете для работы с аппаратными проектами, в дополнение к которому используется среда симуляции и верификации, а при необходимости – и аппаратная платформа-эмулятор.

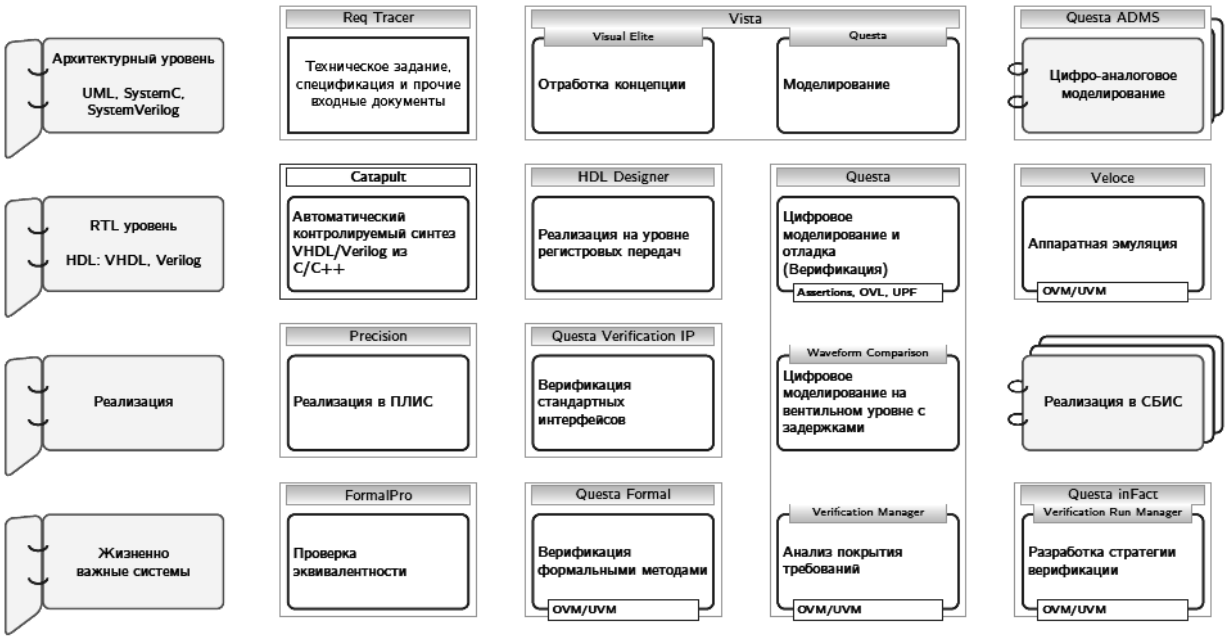


Рис. 5. Инструменты для функционального проектирования [16].

Для получения окончательной реализации применяются программы-синтезаторы, которые генерируют представления системы на вентильном уровне. Кроме того, различные вспомогательные пакеты позволяют добиться высокого качества и работоспособности продукта, что особенно важно при создании ИС и СнК. Доступны инструменты, облегчающие верификацию и анализ полученного решения.

Работа с проектом на разных этапах и на разных уровнях абстракции производится несколькими взаимодополняющими программными пакетами (Рис. 5).

Начальная проработка проекта на архитектурном уровне, ранние оценки работоспособности принимаемых решений, оптимизация и поиск компромиссов производится при помощи пакета Vista. Его средствами формируется рабочая модель аппаратной части на уровне транзакций (используется так называемая TLM 2.0 Scalable Modeling Methodology). При этом модель создаётся интуитивно понятными графическими интерфейсами, генерирующими компактный SystemC код.

Моделирование предполагает несколько разделённых уровней, позволяющих отделить коммуникации, функциональность, а также энергопотребление и временные задержки, не смешивая их друг с другом. Так, функциональный уровень совсем не учитывает временной аспект, а отражает только поведение модели с точки зрения того, «что» она делает. Задержки, ассоциируемые с конкретными реализациями микроархитектурных функций, учитываются во временном уровне, отображая примерную продолжительность вычислений. Функциональность и коммуникации отделяются друг от друга естественным образом.

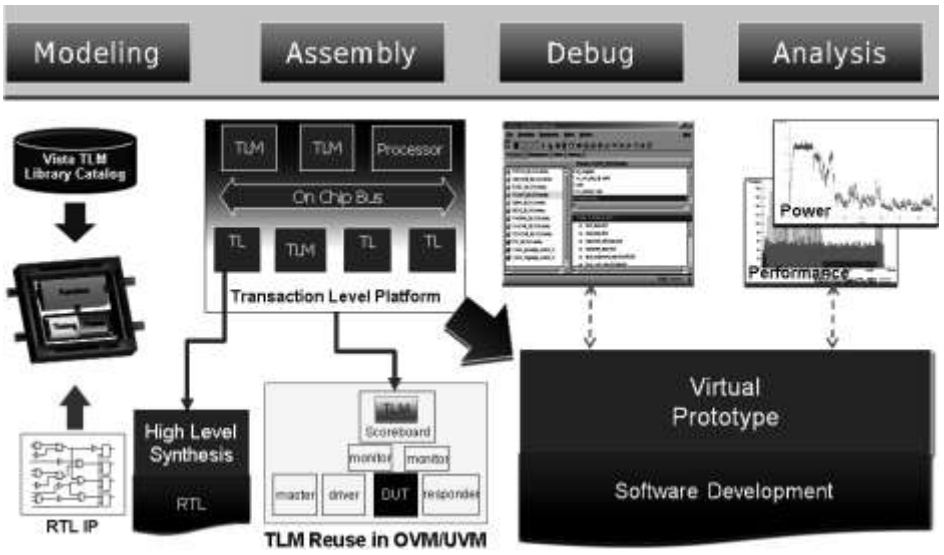


Рис. 6. Vista [17].

Всё это позволяет получать примерные данные для оценки и оптимизации таких параметров системы, как энергопотребление, производительность, место на кристалле и т. д. ещё до реализации. Полученную в итоге исследования пространства решений оптимизированную и верифицированную модель аппаратуры можно использовать впоследствии в качестве виртуального прототипа для

дальнейшей разработки программного обеспечения, а также для верификации RTL.

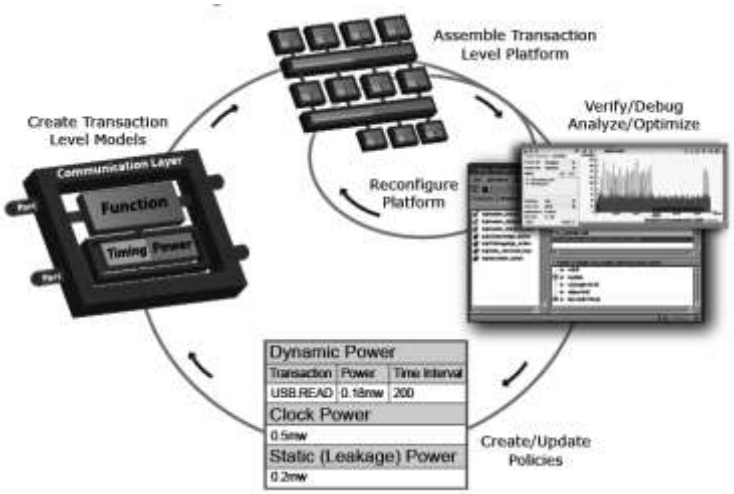


Рис. 7. Цикл разработки при помощи Vista [17].

Vista интегрируется со средой симуляции аппаратуры Questa и с окружением для разработки программного обеспечения CodeBench Virtual Edition, позволяя отлаживать и анализировать проект, как со стороны аппаратуры, так и со стороны ПО. Для моделирования могут использоваться существующие IP-блоки и модели. Поддерживается формат исполняемого файла ELF и стандартный протокол gdbstub, вследствие чего можно использовать совместимые с этим протоколом сторонние средства отладки.

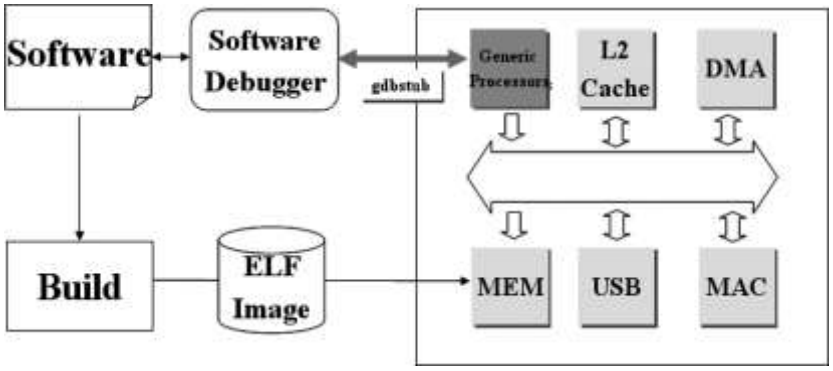


Рис. 8. Отладка при помощи gdbstub [16].

Кроме исследования пространства решений и создания виртуального прототипа, на системном уровне так же полезно бывает задать начальные требования к проекту, с целью их дальнейшего отслеживания в процессе разработки. Это позволяет быть максимально уверенным, что конечный продукт будет соответствовать изначальному техническому заданию,

помогает вовремя отслеживать проблемы, возникающие в процессе проектирования, и своевременно реагировать на них.

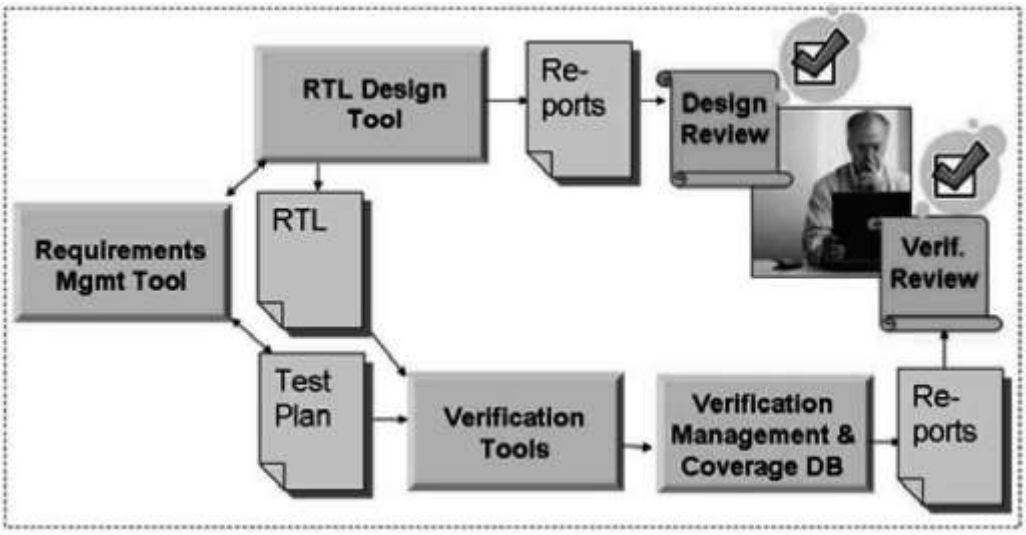


Рис. 9. Управление требованиями [54].

Для интерактивного анализа и трассировки требований используется программа ReqTracer. ReqTracer связывает между собой процесс задания начальной спецификации, реализации и верификации, структурирует все имеющиеся по проекту данные в виде требований и зависимостей между ними. ReqTracer интегрируется со всеми основными инструментами разработки от Mentor Graphics, возможен импорт исходных данных из сторонних программ.

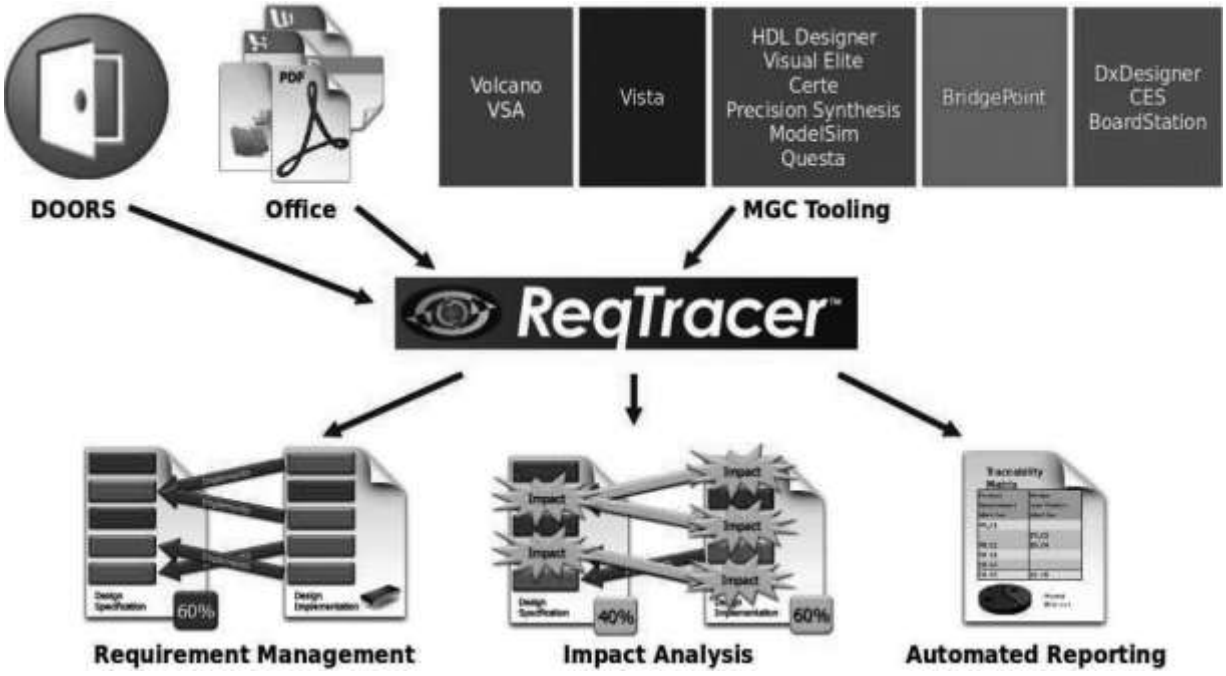


Рис. 10. Менеджер требований ReqTracer [16].

Генерацию из высокоуровневых описаний архитектуры на языках C/C++/SystemC в Verilog/VHDL осуществляет пакет Catapult, обеспечивающий настраиваемый и контролируемый синтез – получаемый RTL может оптимизироваться по энергопотреблению, производительности, объёму или удобству верификации. На вход программе подаются файлы, написанные на SystemC/C++, т.е. без добавления каких-либо дополнительных конструкций. Генерируется управляющая логика и тракт данных, возможен выбор из микроархитектурных альтернатив, возвращение на уровень наверх.

Для интеграции и верификации проектов на смешанном уровне TLM/RTL применяется входящий в состав пакета Vista инструмент Visual Elite. Используя его, системные архитекторы и инженеры могут в интуитивно понятном виде объединять SystemC и HDL, производить верификацию, сравнивая результаты, полученные в ходе симуляции, полученные моделей разной степени детализации, создавать смешанные модели для ускорения симуляции и проверки определённых областей, представляющих интерес для разработчиков.

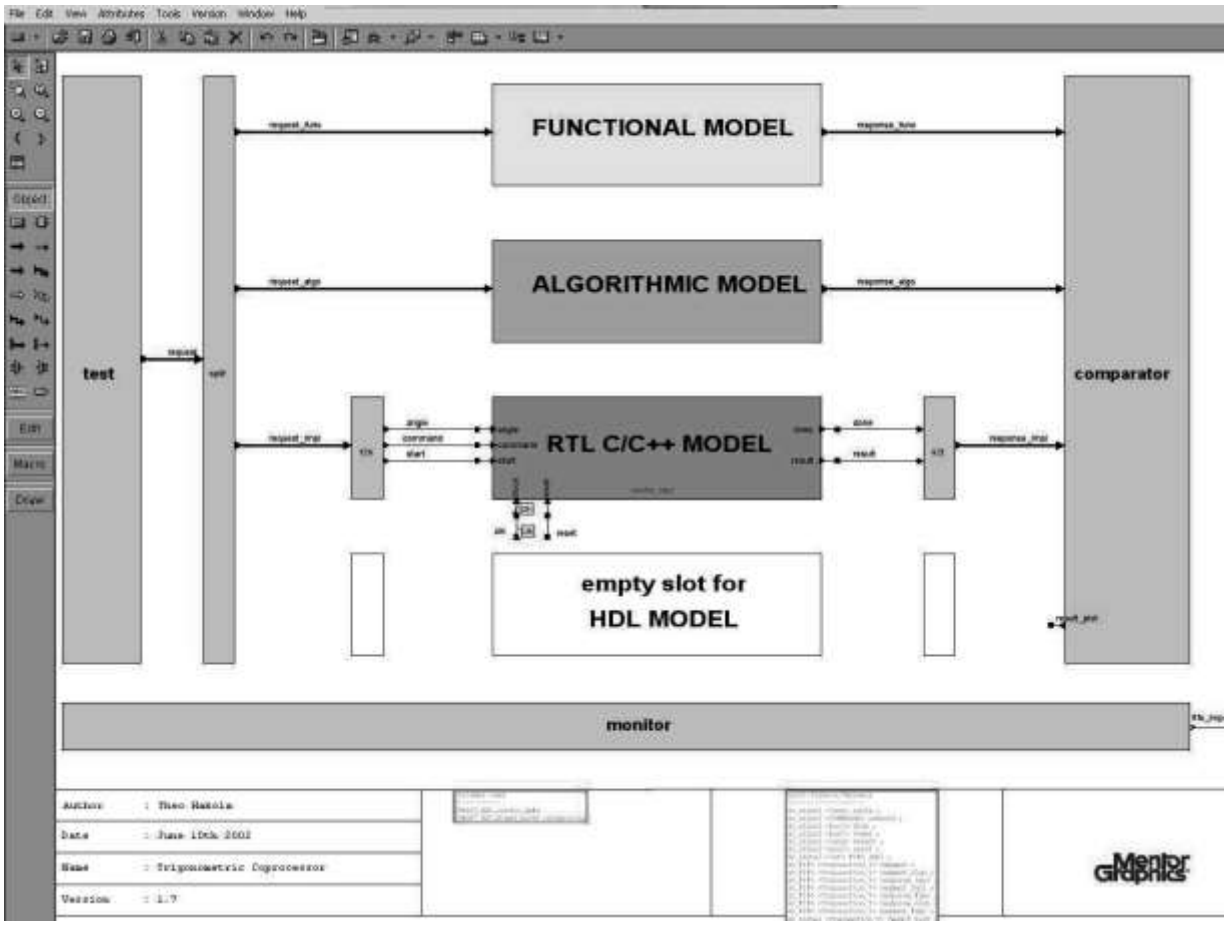


Рис. 11. Visual Elite [55].

Для ручной работы с HDL-описанием аппаратной части проекта и доведения её до работоспособного состояния применяется HDL Designer, в котором можно дорабатывать и уточнять описание аппаратуры на RTL уровне. HDL Designer поддерживает задание проектов в графическом, текстовом, табличном виде и позволяет производить запуск следующих этапов проектирования при помощи вызова таких пакетов, как Precision и Leonardo Spectrum (для синтеза более низкоуровневых описаний вплоть до уровня вентилей). Симулятор Questa используется для отладки и верификации вплоть до вентильного уровня с реальными задержками сигналов. В случае необходимости доступа к аналоговым объектам может применяться также Questa ADMS, средство для цифро-аналогового моделирования.

Повышение качества и доведение до рабочего состояния проектов, предназначенных для реализации в СнК, производится при помощи многочисленных дополнительных инструментов, позволяющих создавать физически реализуемые и удовлетворяющие заданным ограничениям варианты размещения, оптимизировать энергопотребление, производительность, занимаемый на кристалле объём, увеличивать степень параллелизма. Кроме того, при необходимости аппаратной эмуляции будущей системы на RTL-уровне может быть использован пакет Veloce.

Для оценки полученных результатов и формальной верификации используются многочисленные возможности симулятора Questa, который в том числе имеет IP-блоки для верификации стандартных интерфейсов, и программы FormalPRO.

Кроме того, в идеале параллельно с проектированием аппаратной части системы должна производиться разработка ПО, для чего используется интегрированная среда разработки (IDE) Sourcery CodeBench Virtual Edition. Sourcery CodeBench является надстройкой для широко используемой и мощной платформы Eclipse и включает в себя улучшенный GNU-компилятор (GCC) с библиотеками, позволяющий компилировать надёжный и производительный код, а также некоторые дополнительные возможности, облегчающие процесс проектирования ПО. Так, например, Sourcery CodeBench поддерживает визуальную отладку с возможностью просматривать дизассемблированный код, состояние памяти и регистров, позволяет подключаться к целевым устройствам через JTAG. Компиляция может производиться для запуска на «голом железе» или в операционной системе Linux. Встроенный Sourcery Analyzer предоставляет расширенные возможности для анализа кода, облегчая поиск и коррекцию узких мест и ограничений (например, по производительности). Имеются расширенные возможности для запуска и анализа Linux, а также часто используемой программы для работы с видео

– Gstreamer, и графических пользовательских приложений, написанных на QT.

В контексте ESL-ориентированного проектирования одной из наиболее интересных возможностей Sourcery CodeBench Virtual Edition является возможность использования виртуальных платформ, создаваемых пакетом Vista. Это позволяет отлаживать аппаратуру и ПО совместно ещё на ранних стадиях разработки, давая примерные оценки производительности и энергопотребления, и начинать разработку ПО задолго до появления работающей аппаратной части. В течение всего процесса проектирования ПО используется одна и та же среда разработки – как на начальных этапах, когда имеется лишь виртуальный прототип, так и в конце, когда уже существует реальная, работающая аппаратура.

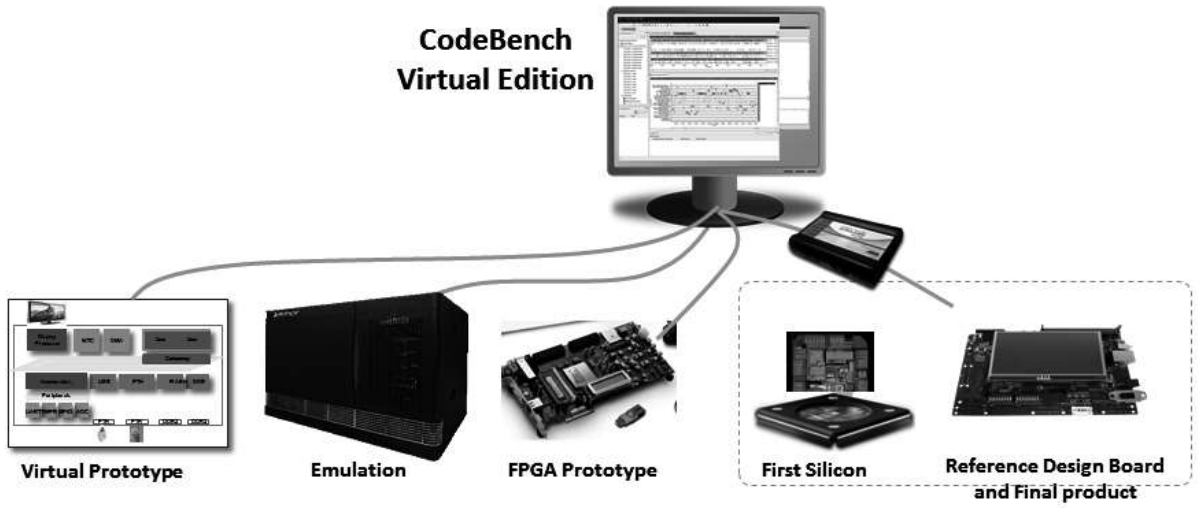


Рис. 12. Sourcery CodeBench Virtual Edition [18].

1.4 Создание технического задания на проектируемое изделие

Любой процесс проектирования начинается с составления спецификации, или технического задания, в том или ином виде. Создание спецификаций направлено, с одной стороны, на прояснение требований заказчика, а с другой – на донесение требований и ограничений ко всему изделию и к отдельным работам до непосредственных исполнителей. Недостаточная ясность в понимании поставленных задач может стать критичной – ведь даже если изделие сделано идеально, но при этом не реализует необходимой заказчику функциональности и не удовлетворяет нефункциональным требованиям, оно сделано неправильно.

Спецификация может выглядеть как задокументированное словесное описание, исполняемая модель или математическая спецификация. Спецификация определяет намеченный функционал и область действия продукта, платформу, при необходимости – архитектурные требования, а

также многие операционные и не функциональные аспекты (размер, потребляемая энергия и т.д.) [1].

Задание технических требований не должно зависеть от архитектуры реализации. Высокоуровневые технические задания должны быть достаточно неопределенными для того, чтобы не ограничиваться в выборе реализаций.

При составлении спецификаций должно учитываться, что аппаратное и программное обеспечение традиционно реализуются отдельными группами, причём, эти люди используют разные наборы методов и инструментов проектирования. Поскольку при ESL-проектировании процесс разработки начинается прежде, чем появляются четкое разделение между аппаратным и программным обеспечением, эти организационные особенности должны тоже приниматься во внимание [15].

Вообще, разработчикам ВсС всё чаще приходится иметь дело с аппаратными платформами, и довольно редко встречаются возможности для нисходящего проектирования в чистом виде. При платформно-ориентированном проектировании необходимо затратить немалые усилия на задание высокоуровневых требований для выявления стоимости реализации и воздействия нового функционала на производительность остальной части системы. Помочь в этом может жёсткое использование процесса менеджмента требований. Подобный процесс должен обеспечивать необходимую обратную связь, управление оценками изменений, согласование необходимых требований и спецификаций независимо от того, является поток проектирования нисходящим, восходящим или идёт от существующей платформы, т.е. из середины.

Гетерогенная электронная система может состоять из большого количества подсистем, создаваемых с помощью различных технологий, самые очевидные из которых – аппаратные средства, программное обеспечение и механические части. Используемые методологии проектирования и языки изменяются не только от одной подсистемы к другой, но и часто в пределах самой подсистемы

В связи с этим имеется нерешённый вопрос – достаточно ли одного формата спецификации для описания требований целой системы? Конечно, может использоваться естественный язык, способный описать виртуально все, что угодно, но возможен ли универсальный, более формальный способ, имеющий не меньшую гибкость?

Независимо от области, в которой ведётся проектирование, системные требования задаются для ожидаемого поведенческого или функционального аспектов системы, для архитектур, обеспечивающих

сборку системы из составных частей, и для нефункциональных атрибутов, которые описывают различные аспекты системы.

Практическое применение ESL в основном сконцентрировано на описании и моделировании поведенческих аспектов и технических требований систем, что естественно, поскольку поведение легко формализуется и моделируется. Моделирование поведения также позволяет решить много возникающих в процессе проектирования проблем за счёт проверки функциональности системы.

Архитектура может быть определена частично теми же методами, что используются для задания поведения. Так же, как результаты симуляции модели определяют требуемое поведение, структура модели может определять архитектуру. В этом отношении модель является просто другим способом реализации системы, а не спецификацией для нее.

Однако, легко распространить эту аналогию слишком широко. Структурирование также используется и в качестве способа разделения проблемы на относительно независимые и подъёмные для отдельных людей или команд разработчиков части, но это ни в коем случае не подразумевает, что функциональное или архитектурное деление производятся по тем же границам. Нет необходимости связывать себя какой-либо определенной функциональной моделью при принятии архитектурных решений, хотя решения, которые приводят к приемлемым функциональным моделям, могут и воздействовать на вероятную или реальную архитектуру.

Помимо поведенческих и архитектурных особенностей, другие атрибуты системы, такие как потребляемая мощность, размер, цвет и т.д., оказывают значительное влияние на коммерческий успех конечного продукта, но их труднее специфицировать в инфраструктуре ESL. Подобные аспекты обычно описываются словесно, на естественном языке. Также возможно формальное описание таких требований на декларативных языках и через формальные методы.

В идеале, было бы неплохо иметь все технические требования в форме, доступной для автоматического анализа. У исполняемых и формальных спецификаций есть три основных потенциальных применения:

- автоматизация отслеживания требований в процессе проектирования и по подсистемам;
- улучшение понимания интеграции гетерогенных систем;
- лучшая интеграция процессов реализации и верификации.

При моделировании, как правило, используется какая-либо абстракция для скрытия ненужных деталей и выделения важных аспектов.

Модели могут быть созданы, когда описываемая система или компоненты ещё не существуют физически, и подобный случай – одно из их самых важных применений. Примерами языков для создания моделей являются UML, Java, Z, или B. Теоретически, можно даже смешивать языки в одной модели. Кроме того, модели могут состоять из других моделей, каждая из которых написана на своем собственном языке.

Требования к разрабатываемому изделию, как правило, зависят от проблемной области и формируются исходя из нужд заказчиков. Например, в типичном телекоммуникационном продукте используются тысячи индивидуальных требований различной природы, задаваемых заказчиками, стандартов, технологий реализации и прочих источников. Для принятия решений при разработке продуктовой линейки товаров может быть сконструирована модель требований. Удобной практикой является её организация в древовидной форме с отношениями предок-потомок.

Процесс управления требованиями. Управление требованиями должно делать эти требования видимыми, трассируемыми и доступными в течение всего процесса проектирования. Желательно использование баз данных и формальное описание для возможности автоматизации, хотя иногда бывает достаточно ограничиваться просто фиксацией в бумажных документах в словесной форме. К тому же, некоторые требования могут описываться формально, некоторые – представляться через исполняемые модели, а некоторые – только описываться на обычном языке. Реализация системы управления требованиями зависит от размера и сложности организации.

Вводить формальный процесс управления требованиями оправдано в очень больших проектах, в небольших компаниях это может привести к катастрофическим последствиям из-за возросших временных и трудовых затрат.

Для иллюстрации процесса управления требованиями можно рассмотреть его типичную реализацию, успешно применённую в одной крупной компании. Причиной введения была возросшая сложность выпускаемых продуктов и невозможность справляться с нею дальше. Включение управления требованиями в процесс разработки всех продуктовых линеек заняло полных два года.

Сам процесс управления требованиями представлен на рис. 56. Изначально требование представляет собой некую потребность заказчика, как он себе её представляет. Оно имеет жизненный цикл и статус с

желаемыми сроками его изменения. Это требование самого высокого уровня связывается с одним или более требованиями, отражающими понимание проблемы с точки зрения подрядчика (“features”), которые, в свою очередь, привязываются к одному или нескольким требованиям самого низкого уровня(“sub-features”), каждое из которых отвечает за отдельный компонент системы.

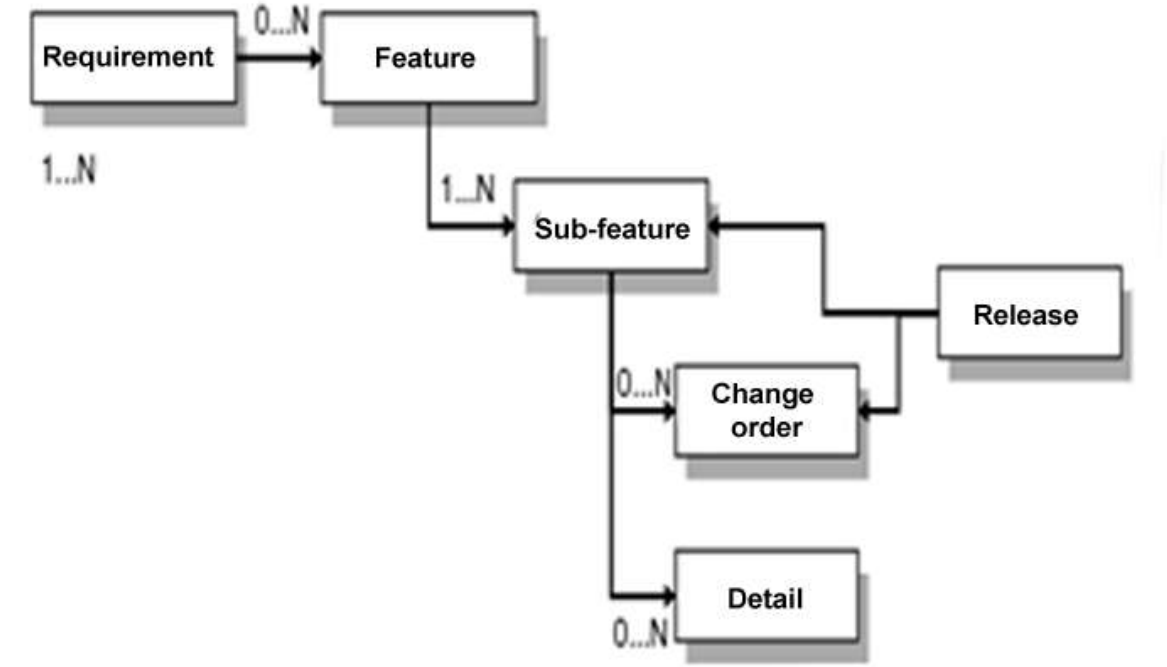


Рис. 13. Пример реализации процесса управления требованиями [15].

Помимо человека, задающего начальные требования, в процессе управления требованиями должен участвовать ответственный за требования, который отслеживает их во время всего жизненного цикла, и управляющий требованиями, администратор, который имеет полный контроль над всем деревом требований с возможностью создавать требования и связывать между собой (как правило, это один и тот же человек). Принятие требований или отказ от них производятся на собраниях, общих или отдельных групп разработчиков. У низко- и среднеуровневых требований могут быть ответственные за них лица. Эти низко- и среднеуровневые требования представляются в системе управления требованиями как словесные описания спецификаций, однако могут привязываться и к исполняемым или формальным спецификациям. В течение процесса используется инструмент, облегчающий хранение требований, управление их статусами, связывание, управление правами доступа, управление расписанием изменения статусов и создание отчётов.

Предметные области проектирования. Существует несколько областей проектирования, имеющих между собой некоторые отличия с

точки зрения ESL-проектирования, отражающиеся на создании спецификаций. В области проектирования трактов потоковой обработки данных, несмотря на то что они всегда требуют также и некоторый механизм управления, ранняя разработка и анализ алгоритмов, которые определяют преобразования данных ввода-вывода, обычно игнорируются или абстрагируются от потока управления. На ранних этапах разработки алгоритмов предназначенные для специфицирования и моделирования моменты являются статическими по природе, например, соотношение сигнал/шум в одном конкретном режиме работы алгоритма. Моделирование работы алгоритма обычно реализуется с помощью последовательностных языков программирования, например, C/C++ и MATLAB. Модели системы, которые соединяют модули алгоритма вместе, также являются последовательностными. Исходя из того, что только тракт обработки данных представляется интересным, соединения обычно моделируются как бесконечно глубокие буферы FIFO.

Тракт управления осуществляет контроль состояний системы (примерами являются реализации стеков протоколов, пользовательские интерфейсы). Тракты обработки данных и управления вместе определяют функциональность системы на высоком уровне. Поток управления может быть описан на различных уровнях абстракции (например, только с состояниями или режимами основной системы).

Другая область проектирования – стеки протоколов, представляющие собой субсистемы коммуникационных систем, которые управляют потоком данных в некотором передающем канале в соответствии с заданным протоколом. Стеки обычно разрабатываются как иерархия уровней и, как правило, имеют управляющий и пользовательский сегменты. Обычно стеки представляют собой низкоуровневое ПО, хотя нижние уровни стеков могут реализовываться аппаратно, а верхние – на процессоре персонального компьютера. С точки зрения ESL-моделирования, общими чертами стеков протоколов являются очень жёсткие требования реального времени, очень модульные архитектуры, коммуникации путём передачи сообщений, ограниченные потребности параллелизма.

Также в отдельную область выделяются ВсС. Их отличительными чертами являются встраивание в некоторый объект, требования реального времени, взаимодействие с окружающим миром не через традиционный пользовательский интерфейс, а через низкоуровневые команды и сигналы, и нефункциональные ограничения, вроде энергопотребления и размера. Для встраиваемых систем, как правило, платформа известна заранее, и проектирование с точки зрения неограниченности пространства проектирования контрпродуктивно. Кроме того, требуется очень детальное

моделирование платформы с целью проверки, удовлетворяет ли она ограничениям.

Исполняемая спецификация – поведенческое описание проектируемого компонента или объекта системы, которое отражает определенный функционал с учётом временных задержек. Основная цель исполнимой спецификации заключается в проверке того, что поведение разрабатываемой системы удовлетворяет системным требованиям после интеграции с другими компонентами системы и того, является ли реализация объекта непротиворечивой указанному поведению. Исполняемые модели являются альтернативными реализациями разрабатываемой системы или её подсистем, которые можно использовать для анализа характеристик будущего изделия. В результатах моделирования обязательно необходимо различать характеристики исполняемой модели, которые действительно отображают функциональность проекта, от артефактов реализации модели, и это является сложнейшей задачей при проектировании на системном уровне.

На раннем этапе проектирования исполняемая спецификация, как правило, является симуляцией некоторой системной функции (или функций). В процессе детализации и анализа различных архитектурных вариантов исполнения эта спецификация превращается в исполняемую архитектурную спецификацию, а затем – в исполняемую спецификацию реализации.

В качестве языков для специфицирования могут использоваться: M-Code от MATLAB, который обычно применяется на ранних фазах разработки алгоритмов ЦОС и управления; Rosetta, созданная для облегчения декларативного специфицирования; SystemC, позволяющий реализовывать параллельные процессы, события, сигналы во многом подобно языкам описания аппаратуры, но с объектно-ориентированным подходом, шаблонами и типами данных из C++; SDL (Specification and Description Language), созданный для специфицирования комплексных приложений с ограничениями по времени, интерактивных и с управляемыми событиями, пригодный, в том числе, для анализа нефункциональных и статических свойств; UML – язык графического описания для объектного моделирования, разработанный для ПО, но пригодный и для аппаратуры, и применявшийся как основной язык для моделирования аппаратных систем; XML – язык общего назначения для создания на своей основе произвольных специализированных языков, пригодный для описания разных типов информации; Bluespec (Bluespec SystemVerilog и Bluespec SystemC) – декларативный язык для создания аппаратных спецификаций, с существующим компилятором, позволяющим синтезировать SystemVerilog или SystemC код.

В дополнение к вышеперечисленным языкам, для функциональной верификации также хорошо подходят аспектно-ориентированные языки, позволяющие выделять сквозные аспекты, влияющие сразу на некоторое количество объектов при объектно-ориентированном подходе. Самые широко используемые аспектно-ориентированные языки – Aspect J (вариант для Java), AspectL (расширение Common Lisp) и много других аспектных модификаций не аспектных языков.

Таким образом, существует множество различных способов специфицирования ВС, ни один из которых не представляется идеальным одновременно для всех её составных частей. Требования для языков спецификаций многочисленны и иногда конфликтуют друг с другом, так как подобные языки должны быть и точными, и иметь возможность определять неоднозначности и управлять ими. Лучшими являются спецификации, которые не ограничивают средства реализации. Многообещающим выглядит модельно-ориентированное проектирование.

При создании спецификаций желательно придерживаться описанных ниже рекомендаций.

1) Спецификации должны быть сделаны максимально, насколько это возможно, более формальным способом для получения возможности формального анализа.
2) При невозможности использования формального анализа необходимо использовать исполняемые спецификации, при этом не забывая, что они неизбежно содержат искусственные артефакты реализации, которые должны быть учтены при анализе полученных данных.
3) Если нельзя использовать исполняемые модели, декларативные аспекты проекта должны быть зафиксированы в некотором структурированном виде, например, как при использовании системы управления требованиями. Новые языки, такие как Rosetta, могут со временем начать использоваться для формального исследования.
4) В случае невозможности перечисленных выше подходов для описания аспектов проекта и его ограничений может использоваться обычное словесное описание.
5) Спецификации должны сначала быть насколько можно более абстрактными и дорабатываться путем добавления деталей.
6) Можно объединять проектирование на основе платформ с нисходящим проектированием, если спецификация относится к существующим платформам, а нисходящее описание концентрируется на новой или производной функциональности.

7) В качестве языков для спецификаций и нотаций можно использовать UML, C/C++, SystemC, математические нотации и нотации потоков данных (такие как MATLAB, Simulink и SPW).

8) Верификация является только частью процесса валидации. Не имеет значения, насколько среда верификации является экстенсивной и высококачественной: если система плохо спроектирована и некорректна, тогда верификация не влияет на качество системы. Из этого вытекает необходимость более формальной/жесткой спецификации и формального/жесткого анализа.

1.5 Исследование пространства проектных решений

Разработка аппаратуры и программного обеспечения отдельно друг от друга может привести к реализациям системы, спроектированной с недостаточным запасом, или систем, которые не отвечают всем нефункциональным требованиям, таким как надёжность, энергопотребление и т.д. Проектные решения по назначению вычислительных ресурсов для задач могут быть излишне строгими, что приводит к слишком дорогостоящим реализациям. Бороться с этим призвано исследование пространства проектных решений [1], или же анализ перед разделением и после в ESL-маршруте проектирования [15].

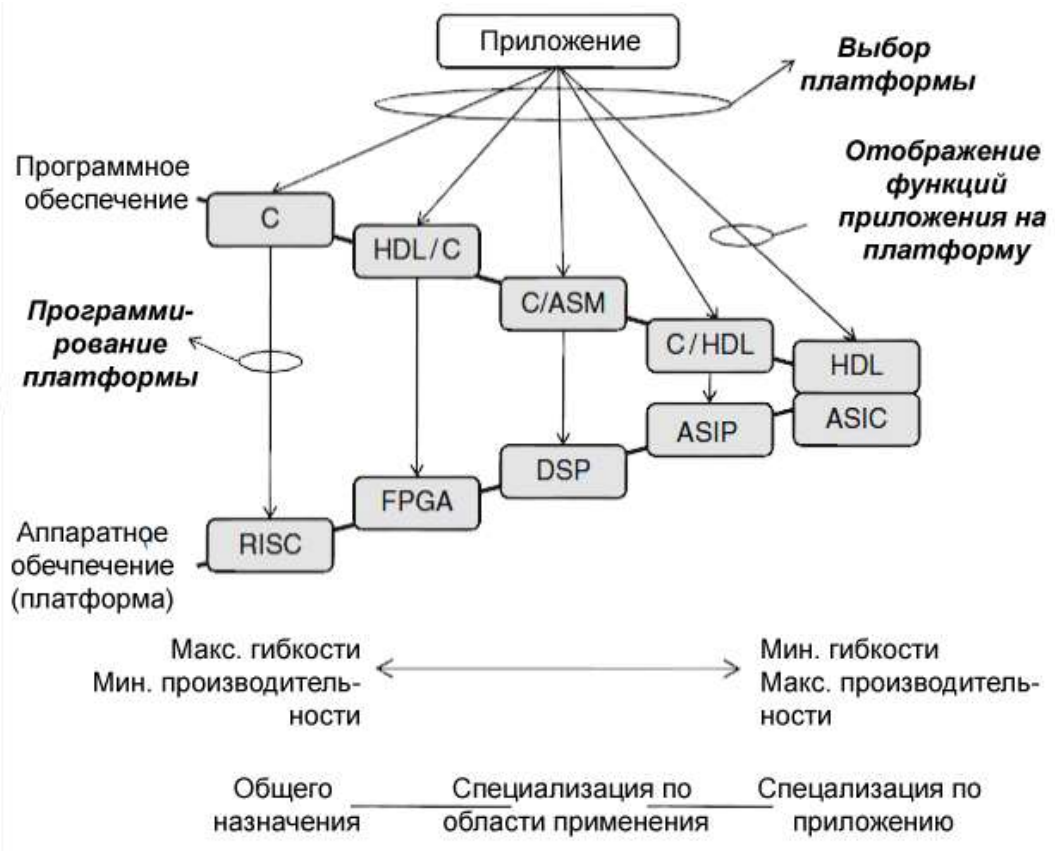


Рис. 14. Пример выбора отображений функциональности на архитектуру [2].

Исходя из того, что системный синтез включают три этапа – выделение ресурсов архитектурной модели, привязку объектов функциональной спецификации к выделенным ресурсам и планирование задач должным образом, пространство проектирования задано набором всех возможных перестановок выделений, привязок и планирования. Любую такую тройку, удовлетворяющую определенному числу дополнительных нефункциональных ограничений, таких как ограничения стоимости, производительности, электропитания, температуры и т.д., называют допустимым решением [1]. Из допустимого решения может быть легко получена соответствующая структурная реализация. Часто запланированный код для каждого ресурса может быть сгенерирован автоматически как детализация начальной спецификации.

Сейчас исследование пространства проектирования является задачей исследования допустимых реализаций эффективно и находя не только одну из них, но многие, а также оптимальные.

- *Стоимость исследования и стратегии исследования (алгоритмы).* Какие алгоритмы хороши для исследования обширного и, в общем, дискретного пространства поиска с миллионами потенциальных решений? Конечно, мог бы быть применен любой известный поисковый метод, чтобы обнаруживать допустимые реализации – рандомизированные поисковые методы, методы, полагающиеся на итеративную детализацию, такие как моделированная нормализация, или точные методы, основанные на формулировке целочисленных линейных программ (integer-linear program, ILP).
- *Многоцелевая природа и оценка нефункциональных свойств.* Классические одноцелевые поисковые методы, подобные упомянутым выше, оптимизируют единственную целевую функцию. В случае если необходимо обнаружить лучшие проектные решения для двух или больше целей, таких как стоимость и производительность, нужно обеспечить весовую функцию, чтобы объединить обе цели в одну функцию, которая соответствует принятию решений разработчиком благодаря выбору подходящего весового коэффициента. Настоящие многокритериальные методы требуются для того, чтобы выполнить действительно непредвзятое исследование пространства проектных решений и переложить решения, принимаемые обычно позже, на системного инженера, с учётом того, какие компромиссы реализации доступны.
- *Как гибко оценить качество проектирования?* Наконец, должны быть приняты во внимание не только обширное пространство поиска и время на исследования, но также и как

гибко различные настраиваемые цели могут быть определены и оценены. Средства исследования должны быть гибкими для возможности интеграции в различные инструментальные цепочки проектирования.

Важное наблюдение при использовании многокритериальных эволюционных алгоритмов для исследования пространства проектирования в ESL состоит в том, что для сильно ограниченных пространств поиска они могут не обнаружить допустимого решения вообще. Поэтому предлагаются расширения, например, чтобы добавить некоторое число нарушений ограничений в качестве дополнительной цели для минимизации.

Анализ проектного пространства предполагает детализацию модели спецификаций проектируемой системы и определение, исходя из высокоуровневых требований, ограничений проекта ещё до разделения (включающего разделение на аппаратуру и ПО, разделение на конкретные аппаратные блоки и программные функции и распределение задач между отдельными вычислительными ядрами в многопроцессорных системах). При этом необходимо помнить, что выделение в процессе проектирования ESL жесткого предварительного разделения и последующих фаз пусть и позволяет просто и понятно описать маршрут проектирования, тем не менее не всегда соответствует реальности. Во-первых, разработка продукта может быть основана на платформе, где часть проекта предварительно задана, реализована и проверена, новый проект лишь добавляет или модифицирует существующую функциональность. В этом случае являются уже фиксированными многие решения разделения, хотя для новых функций сначала может быть неясно, сколько их могло бы быть реализовано в блоках аппаратуры, а сколько в программном коде, или же способы, которым должна устанавливаться связь между блоками. Во-вторых, для любого проекта проблематично предугадать все решения на уровне реализации, включая разделение, на основе только заданных спецификаций. Вне зависимости от того, насколько можно попытаться отложить решения разделения, человеческой натуре всегда свойственно стремится разделить сложную спецификацию на агломерацию более простых функций. Несмотря на это, лучше, по крайней мере, попытаться отложить решения разделения, пока не будет сделан анализ, потому что не очевидные, но более совершенные варианты разбиения могут быть пропущены при преждевременном быстром разложении на составные части и создании разделенной спецификации.

Статический анализ. Статические методы используются при отсутствии исполняемых спецификаций и моделей, основываются на анализе спецификаций и обычно неформальны. Несмотря на предпочтительность использования динамических моделей, эти методы

также могут использоваться при проектировании. Анализ сложности системы, заимствующий принципы «анализа функциональных точек» из программной инженерии, может быть применен для оценки многих характеристик, включающих стоимость, производительность, затраты на разработку и т.д. Такой анализ может также производиться с помощью исполняемых моделей в качестве дополнения к динамической симуляции. Одной из ключевых проблем при этом является калибровка, сличение результатов анализа с фактическими результатами проектирования. При этом не столь важна абсолютная погрешность оценки, сколько то, что решения выбора оптимального соотношения, принятые на основе оценочных функций, должны быть монотонными. Если анализ вариантов A и B указывает, что A предпочтительней B, то фактические результаты проектирования системы для критериев анализа должны соответствовать этому (A будет лучше, чем B, в пределах некоторых приемлемых границ неопределенности).

Анализ функциональных точек. Хорошо зарекомендовавшим себя методом оценки трудозатрат на создание ПО является анализ функциональных точек, позволяющий оценивать затраты на разработку и сопровождение некоторого программного продукта. Кратко суть метода заключается в оценке трудоёмкости на основе функционала программного продукта, информация о котором получается из имеющихся требований, руководств, предписаний и других документов, описывающих будущую (или уже существующую) систему, так как оцениваться может разработка, доработка или готовый продукт.

В ходе анализа производится определение границ разрабатываемой системы, подсчёт функциональных точек, связанных с данными, определение сложности данных, подсчёт функциональных точек, связанных с транзакциями (процессами, осуществляющими переход системы из одного состояния в другое), определение сложности транзакций. Первоначально объём продукта оценивается в так называемых «не выровненных» функциональных точках, полученных простым суммированием полученных данных о количестве и сложности данных и транзакций. На следующем этапе определяется значение фактора выравнивания, который зависит от 14 определяющих системные характеристики продукта параметров, оцениваемых по шкале от 0 до 5.

Эти параметры:

- обмен данными – является ли продукт автономным приложением или обменивается данными по более, чем одному протоколу;

- распределённая обработка данных – обработка данных выполняется только в одном компоненте системы или во многих;
- производительность – есть ли ограничения на время отклика;
- ограничения по аппаратным ресурсам – есть ли требования к выполнению на конкретном процессоре и к невозможности разделения, или подобных ограничений нет;
- транзакционная нагрузка – определяется количество и равномерность транзакций, есть ли пики;
- интенсивность взаимодействия с пользователем – производится ли обработка в пакетном режиме или более 30% транзакций – интерактивные;
- эргономика – есть ли жёсткие требования по эффективности;
- интенсивность изменения данных пользователем – интенсивны ли изменения, есть ли жёсткие требования по восстановлению данных;
- сложность обработки – оцениваются требования безопасности, логическая и математическая сложность, многопоточность;
- повторное использование – разрабатывается ли система как многоразовый стандартный компонент или без подобных ограничений;
- удобность инсталляции – необходима ли автоматическая установка и обновление ПО;
- удобство администрирования – есть ли требования к автоматическому самовосстановлению системы;
- переносимость – есть ли требования установки на различные машины и операционные системы;
- гибкость – изменяется ли структура и модель данных пользователем в интерактивном режиме.

Финальным этапом производится расчёт количества выровненных функциональных точек, с учётом полученного коэффициента, который и является оценкой усилий разработчиков, требуемых для реализации или доработки проекта.

Варианты этого подхода к оценке трудозатрат на разработку можно было бы применить к модели "проект" – разложенному на составные части решению некоторой задачи – и иерархии модулей проекта, реализующих функции, определенные в спецификации. Но у метода имеется несколько недостатков, ограничивших его применение. Во-первых, предполагается,

что проекты достаточно похожи друг на друга, а метрики и веса, выведенные для проекта (или «разработки») А, будут применимы и к проекту В. Во-вторых, считается, что спецификация известна с самого начала и не является объектом непрерывного и радикального изменения. В-третьих, метод предполагает последовательный и скрупулёзный сбор данных и документирование в течение некоторого времени, а опыт использования в сложных проектах показывает, что организации часто не желают тратить время и усилия на последовательный многопроектный и многолетний сбор метрик. Кроме того, организациями, разрабатывающими продукты, которые изменялись значительно от проекта к проекту, было обнаружено, что методы не сходятся по прошествии длительного времени и генерируют прогнозы с чрезвычайно широкими дисперсиями.

Модель издержек разработки. Для оценки стоимости разработки ПО, наряду с методом функциональных точек, также часто применяется модель издержек разработки (COnstructive COst MOdel, COCOMO) – алгоритмическая модель оценки стоимости разработки ПО. Модель предоставляет три уровня оценки – базовый, для быстрых оценок стоимости разработки и не дающий большой точности, средний, рассчитывающий трудоёмкость разработки также исходя из множества факторов, влияющих на стоимость, и детальный, включающий в себя множество характеристик с оценкой их влияния на различные этапы процесса разработки.

За единицу размера программы берутся тысячи строк кода (KLOC – kilo lines of code). Выделяются три типа проектов – органический, представляющий собой маленькие команды с хорошим опытом работы и довольно мягкими требованиями к разработке, промежуточный, для средних по размеру команд со смешанным опытом разработки и смешанными требованиями, и встроенный, когда разработка ведётся при наличии множества жёстких ограничений.

На базовом уровне трудоёмкость, срок разработки и число разработчиков вычисляются исходя из размера программы и известных коэффициентов, соответствующих каждому типу проектов. Средний уровень учитывает также регулирующий фактор, на который влияют дополнительные характеристики, каждая из которых включает несколько дочерних:

- характеристики продукта (требуемая надёжность, размер БД, сложность продукта);
- характеристики аппаратного обеспечения (ограничения быстродействия при выполнении программы, ограничения памяти, неустойчивость окружения виртуальной машины, требуемое время восстановления);

- характеристики персонала (аналитические способности, потенциал к разработке ПО, опыт разработки, опыт использования виртуальных машин, опыт разработки на языках программирования);
- характеристики проекта (использование инструментария разработки ПО, применение методов разработки ПО, требования соблюдения графика разработки).

Каждому из факторов назначается рейтинг по шести балльной шкале (от «очень низкого» до «очень высокого», по значению или важности фактора). С учётом их произведения, а также коэффициентов, соответствующих каждому типу проектов (органический, промежуточный, встроенный), из начальных KLOC рассчитывается трудоёмкость в человеко-месяцах. Детальный уровень включает все характеристики среднего уровня и сверх этого оценку влияния данных характеристик на каждый этап процесса разработки.

Анализ спецификаций аппаратных систем или систем с преобладанием аппаратуры. Некоторые виды методов статического анализа из области программного обеспечения были также применены для анализа систем со значительными аппаратными частями или для программно-аппаратных систем.

Одно из самых интересных направлений исследования было проведено в Politecnico di Milano, Италия, и ее ассоциированном институте CEFRIEL, исследователями William Fornaeiari, Fabrio Saliee и другими. Большая часть их исследования сосредоточилась на статическом анализе проектов, полученных в форматах, таких как VHDL, Occam2, C и UML, для определения различных атрибутов, таких как потребляемая мощность, время выполнения программы, затраты на разработку, размер разработки, стоимость проектирования, площадь, емкость памяти и т.д.

Эти исследования, начавшиеся с более конкретных измерений, развились в оценки стоимости и размера разработки, явно унаследовав принципы COCOMO и анализа функциональных точек, и начали принимать во внимание аспект повторного использования. Дальнейшие исследования позволили получить значительную точность в предсказании «размера» разработки (выраженного в строках кода VHDL) для реализуемых проектов – и, таким образом, предсказать трудозатраты и размеры группы разработчиков.

Наиболее совершенной и полной формы данные исследования достигли в работе Fornaciari и коллег [19]. В ней спецификации закреплены в UML с сочетанием диаграмм класса и последовательности, используются методы анализа функциональных точек, чтобы оценить трудозатраты и время на разработку, оцениваются затраты на единицу

продукта. Все эти аналитические формулы передаются в эвристический алгоритм исследования пространства автоматизированного проектирования, который заканчивается на том, что появляется много оптимальных разделений HW/SW в зависимости от взвешивания целевой функции. Весь подход также включает и анализ повторного использования.

"Ility" анализ. Исследование "ility" (или "ilities") – исследование атрибутов, таких как надежность, пригодность для обслуживания, удобство использования, важность объекта, анализ отказа и т.д. (в английском языке эти слова оканчиваются на "-ility").

Общий алгоритм, используемый при таком статическом анализе, должен объединить прогнозы надежности для отдельных компонентов иерархически. Например, если система состоит из подсистем A и B, и если A имеет вероятность отказа в течение года в 5%, а подсистема B – 10%, то вероятность одного года успешной работы, допуская, что подсистемы независимы от видов отказа, будет произведение A и B: $0.95 \cdot 0.9 = 0.855$, что соответствует вероятности отказа в течение года 14.5% [15]. Если виды отказа подсистем коррелированы, то различные алгоритмы, основанные на исследовании вероятности надежной работы, используются для вычисления совокупной надежности системы.

Модели надежности компонента основываются на измерениях для отдельных компонентов – сложности, операционной среды (особенно температуры и вибрации), радиационной среды (что важно для космических систем) и других соответствующих параметров. Большие базы данных для стандартных компонентов были накоплены в 1970-х и 1980-х, чтобы использовать в таких предсказаниях. Для обеспечения надежности системы для ключевых или критических систем в жестких режимах работы стала использоваться избыточность разных видов. Такие области, как надежность программного обеспечения, со временем стали намного более трудными для расчёта, чем стандартная надежность аппаратного компонента, из-за плохо понятых видов и моделей отказов и огромной разницы в качестве программного обеспечения, создаваемого для различных продуктов. Однако, простые меры, такие как тысячи строк кода (KloC) могут включаться в предсказания и учитываться в общем вычислении.

Открыт вопрос относительно того, могут ли такие методы экстраполироваться из традиционных методов проектирования крупномасштабных систем в область BcC, где возможность собрать статистические данные сдерживаются конфиденциальностью и отсутствием аналитического интереса, что типично для большинства команд и компаний-разработчиков. Кроме того, вариабельность многих

встраиваемых систем означает, что даже хорошая база данных с опытом предыдущих разработок часто не очень применима к новому проекту. Отсутствие хороших моделей ошибок программного обеспечения для систем, которые стали все больше и больше основываться на программном обеспечении, исполняемом на процессорах, ограничивает полезность надежности и других "ility" прогнозирующих устройств. Наконец, поскольку архитектуры системы становятся всё более сложными, воздействие связей между модулями, которые сложно отслеживать, означает, что вероятность отказа и оценки других "ilities"-параметров в этих системах не всегда можно применять.

Анализ требований. Требования могут быть проанализированы статически для оценки «функционального веса» определенных модулей проекта, и, таким образом, усилий и времени на разработку. Они также могут быть проанализированы для оценки затрат на верификацию, необходимых для определения, удовлетворяет ли реализация спецификации всем своим требованиям. Оценки, которые могут быть получены, могут оказаться очень грубыми и с широким диапазоном погрешности, но все же могут быть использованы с точки зрения монотонности, обсужденной ранее.

Анализ при платформно-ориентированном проектировании. Подход к проектированию, основанный на заданной инструментальной платформе, часто рассматривался в противоречии с нисходящим подходом проектирования, который используется в качестве общего идеализированного маршрута ESL-проектирования. Однако, есть способы совмещать эти подходы.

Хотя платформа может быть полностью фиксирована с помощью её архитектурных моделей, моделей IP и функциональных моделей компонентов ПО и других элементов, у любого производного проекта, основанного на платформе, будет новая функциональность, которая должна быть реализована. Именно эта новая функциональность может быть специфицирована и смоделирована с помощью методов ESL-проектирования, и именно эта новая функциональность может быть проанализирована. Способ, которым моделируется функция, может зависеть от характеристик платформы, но до самого большого возможного предела команда проектирования системы должна стремиться смоделировать функцию нейтральным способом, без искажения в сторону реализации как аппаратных средств, так и программного обеспечения, и без искажения по отношению к определенным компонентам или подходам реализации.

Трудность проектирования на базе заданной платформы, которая первоначально ставит в тупик команду разработчиков, является сильным

риском, что они будут излишне склоняться к уже существующим в платформе решениям разделения. Но необходимо отрешиться от существующих в платформе решений и использовать только исходные данные технического задания, каждое новое требование и связанные спецификация и модели должны быть проанализированы, как будто проектирование началось с чистого листа. Методы как статического, так и динамического анализа должны быть выполнены для всех новых требований безотносительно к существующим моделям платформы. Введение понятия чистого листа для проекта требуется для того, чтобы повторное использование существующих частей платформы было в достаточной мере оправдано и выполнено с некоторой аналитической точностью.

Динамический анализ. В случае анализа динамического исполнения или симуляции спецификаций посредством моделей исследуются такие характеристики, как более точные оценки производительности (задержки, пропускная способность, время ожидания), факторы, которые влияют на характеристики с временными критериями, такие как причины перегрузки, арбитражные политики и приоритеты, алгоритмы планирования для исполнения моделей (которые могут использоваться после разделения, например, как политики планирования программного обеспечения). Из динамической симуляции исполняемых моделей на этапе перед разделением требуется получить важные характеристики базовой функциональности, которые впоследствии используются при разделении. Они включают: затраты вычислительных ресурсов, затраты на коммуникации, затраты ресурсов энергопотребления.

Если модель спецификации для определенной функции представлена в исполняемой форме на языке C или C++, но не удовлетворяет ограничениям какого-либо определенного комплекта инструментальных средств и ассоциированной модели вычисления, как в инструменте алгоритмического анализа, то используется выполнение на базе хоста и ассоциированный анализ, что является текущим современным подходом для динамического анализа. Инструменты для профилирования программного обеспечения могут использоваться, чтобы дать оценки затрат вычислительных ресурсов за счёт выполнения наборов тестов и сценариев тестирования. Путем взвешивания инструкций с оценками потребляемой мощности на инструкцию могут быть оценены затраты ресурсов мощности, предполагая, что известны затраты потребляемой мощности для доступа к памяти. Добавление простых функций мониторинга может быть полезным, чтобы отследить объем данных и управляющих коммуникаций, которые возникают при выполнении функциональной модели.

Конечно, важно помнить, что функциональная модель – в лучшем случае несовершенный аналог реализации. Таким образом, все оценки, выведенные или из статического, или из динамического анализа функциональной модели (или моделей), которые составляют спецификацию системы, не обязательно точны, и при принятии решений должны использоваться осторожно. Монотонность оценок может быть полезна при принятии решений, несмотря на погрешности. Кроме того, все исполняемые модели спецификации являются также частично реализациями, включающими выборы оптимальных решений реализации, позволяющих получить полную модель на известном языке, для известных инструментов разработки и известной платформы. Важно помнить, что работа с такой моделью должна производиться осторожно, должны отделяться характеристики спецификации от артефактов исполняемой реализации.

Алгоритмический анализ позволяет получать такие оценочные критерии, как ожидаемое выполнение или вычислительные затраты, объем передаваемых данных, требуемый для того, чтобы обработать кадры, пакеты или токены посредством алгоритма, частота возникновения ошибок, частота передачи ошибочных битов. Этот анализ особенно важен для алгоритмов обработки потоков данных, используемых в сигнальных приложениях и приложениях для обработки изображений. Кроме того, во время рассматриваемой фазы проектирования системы полезен процесс алгоритма преобразования данных с плавающей точкой в данные с фиксированной точкой и, если возможно, оптимизация спецификации с фиксированной точкой, что полезно делать перед финальным разделением и реализацией. Больше оптимизации возможно после разделения, когда уже лучше известны характеристики платформы для конечной реализации. Однако, начальная установка диапазона с фиксированной точкой и требований по точности на этапе перед разделением приводит к уменьшению работы после разделения.

Алгоритмический анализ включает рассмотрение моделей вычисления, требуемых алгоритмами в соответствии с проектом. Область алгоритмического анализа была одной из первых областей для коммерчески жизнеспособных инструментов ESL-анализа. Инструмент SPW (Signal Processing Workbench), первоначально поставляемый Comdisco, затем распространяемый Cadence, и впоследствии перекупленный CoWare, активно использовался ещё начиная с конца 1980-х. SPW как инструментальное средство моделирования и алгоритмического анализа поддерживает моделирование потока данных на базе блоков, и имеет библиотеки блоков обработки, взятые из многих доменов обработки сигналов и изображений и специальных моделей верификации среды. Кроме того, SPW поддерживает возможность

аппаратной реализации посредством библиотеки генераторов кода RTL – Hardware Design System (HDS). SPW также позволяет проводить анализ неразделённой части алгоритма с реализацией программного обеспечения части приложения посредством ко-симуляции с симулятором системы команд DSP, выполняющим код реализации. Этим SPW может также поддерживать подход к платформно-ориентированному проектированию, где акцент делается на новые, неразделенные алгоритмические функции.

Известны и широко используются также такие инструменты, как MATLAB от MathWorks, Simulink и родственные инструменты (Stateflow и различные библиотеки реализации). В инструменте Mirabilis используются возможности проекта Ptolemy университета Berkeley. Инструменты анализа конечных автоматов предоставляются поставщиками инструментария встроенного программного обеспечения, часто использующими UML – например, IBM Rational Rose, iLogix Rhapsody, Artisan Software Tools, Esterel Technologies, MathWorks (Stateflow), CoWare, Prosilog, CoFluent и другие. Иногда подобные пакеты интегрируются в основанную на HDL среду проектирования и верификации (Cadence, Mentor и Synopsys).

Сценарии анализа и моделирование. Набор методик анализа может зависеть от того, в которой области приложение используется. Например, в проводной и беспроводной связи используются модели окружающей среды для каналов коммуникации – "модели канала" (которые могут быть смоделированы в программном обеспечении или использованием специальных аппаратных устройств). Введение реальных и искусственных механизмов отказа (например, помехи, затухание сигнала, фазовое искажение, отражение и другие эффекты в приложениях беспроводной связи) в таких моделях окружающей среды чрезвычайно важны для того, чтобы определить устойчивость, скорость динамического отклика и адаптируемость основных базовых алгоритмов и функций, которые должны быть реализованы, и анализа новых алгоритмов в стрессовых условиях.

1.6 Разделение

Как уже было сказано выше, в основе кодизайна лежит разделение ПО и аппаратуры на системном уровне, что позволяет сполна использовать специфику этих составляющих для лучшего удовлетворения требованиям производительности, гибкости, энергопотребления и т.д. Разделение ускоряет общий процесс проектирования, благодаря вычленению отдельных работ. При введении спецификаций на межмодульные интерфейсы эти модули вообще могут разрабатываться отдельными командами разработчиков, при этом также упрощается верификация. Правда, одновременно уменьшаются возможности оптимизации из-за введения барьеров между модулями [15].

Поскольку отображение на системном уровне оперирует гетерогенными объектами, оно позволяет разделить различающиеся между собой и ортогональные аспекты [15]:

1) Вычисления и коммуникации. Вычисления обычно оптимизируются вручную, тогда как коммуникации – при помощи шаблонов.
2) Функциональность и архитектура. Эти аспекты обычно задаются и проектируются раздельно.
3) Поведенческий аспект и производительность. Производительность может представлять и нефункциональные требования.

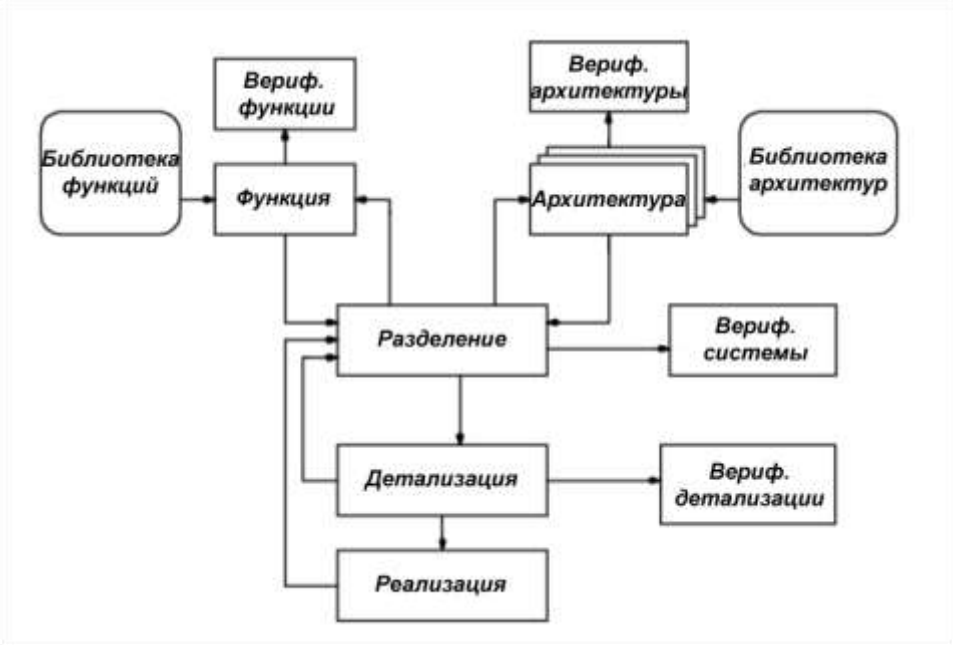


Рис. 15. Представление возможного процесса разделения [15].

Кроме того, разделение упрощает повторное использование, так как разделяет независимые аспекты, которые позволяют связать, например, заданную функциональную спецификацию с деталями низкоуровневой реализации.

В процессе разделения можно выделить несколько основных шагов. Первый, вероятно, самый сложный – выделение индивидуальных функциональных компонентов из начальной, "монолитной" спецификации. Спецификация может быть представлена в разных формах, таких как документы с перечисленными требованиями, предыдущая рабочая "legacy" реализация, или программная модель. Ключевой задачей на данном этапе является выделение адекватных по размеру и сложности фрагментов, которые могут быть поручены разным командам, работающим более-менее

независимо, и выделение параллелизма в масштабе отдельных процессов или задач. Как правило, редкая команда разработчиков может хорошо разрабатывать системы одновременно и на RTL уровне для аппаратуры, и на уровнях C/Assembler для ПО, поэтому чёткое разделение и разграничение ответственностей необходимы. И эта декомпозиция далеко не просто абстрактная операция, которая может быть применена к статической модели. Она должна учитывать [15]:

1) функциональные зависимости, так как оптимизация между модулями очень сложная задача;
2) требования производительности, так как каждый интерфейс может стать узким местом, бутылочным горлышком системы;
3) результаты анализа перед разделением, чтобы последующие шаги отображения функциональных задач на элементы архитектуры смогли обеспечить требуемые уровни производительности при удовлетворении требований к стоимости, энергопотреблению и уменьшению количества узких мест в коммуникациях.

Повторное использование в области ПО часто встречающаяся практика; в разработке аппаратуры этот подход тоже желательно применять, для ускорения процесса проектирования и уменьшения рисков.

Второй шаг при разделении – определение архитектуры системы с учётом требований, заданных в спецификации, как правило – стоимости, энергопотребления, производительности и многих других, зачастую явно неуказанных – надёжности, сопровождения, простоты использования, повторным использованием компонентов, опытом команды разработчиков и так далее. Практически всегда это не просто выбор в соответствии с ценовыми требованиями. Скорее, это комплекс из технических, рыночных и деловых решений, которые обычно диктуют успешность проекта.

Третий шаг включает в себя назначение функциональных блоков элементам архитектуры с учётом ограничений по занимаемому объёму, энергопотреблению и производительности. Ограничения объёма могут представлять собой размер памяти для хранения ПО и данных, количество CLB для FPGA, пропускную способность для коммуникаций. Этот этап, обычно ограничивающийся выбором из двух вариантов между аппаратурой и ПО, традиционно в литературе называется "разделением" ("partitioning"). Но для обеспечения пригодного для использования процесса проектирования необходимо более широко подходить к разделению, учитывая всю проблемную область.

Кроме того, данный этап может менять функциональную декомпозицию и архитектуру, если требования не будут выполняться. Для

оценки производительности и энергопотребления, как правило, используется симуляция, инструменты высокоуровневого синтеза аппаратуры и ПО, "worst-case" анализ.

Четвёртый шаг – детальное описание интерфейсов между различными частями проекта; коммуникации внутри них создаются инструментами синтеза или вручную. Выбор данных интерфейсов важен для верификации и обеспечения взаимодействия между командами разработчиков.

В принципе, каждый из описанных шагов этапа разделения может быть автоматизирован, однако, сложность значительно растёт к начальным шагам по сравнению с последующими.

Функциональная декомпозиция. Для эффективного разделения системы желательно начать с модели, максимально удалённой от деталей реализации. Предпочтительно, чтобы это была чисто функциональная модель с возможностью представления параллелизма на уровне процессов, дополненная комментариями об ограничениях. Этим требованиям удовлетворяют два подхода: описание модели на изначально параллельном функциональном языке или автоматическое извлечение параллелизма из последовательного языка.

Примерами первого подхода являются Simulink от MathWorks, LabVIEW/MatriX (National Instruments), SPW (CoWare), Lustre (Esterel). Ни один язык не может эффективно и лаконично покрывать одновременно аспекты систем с доминированием управляющей части, в которых более важны задержки и ветвления алгоритмов, и аспекты систем с доминированием данных, в которых важнее арифметические операции и пропускная способность. Основные инструменты функционального моделирования позволяют смешивать различные модели вычислений, по крайней мере, в виде поддержки параллельно работающих реализаций алгоритмов конечных автоматов и графов потоков данных. Описания потоков данных на блочном уровне лучше для представления аспектов систем с доминированием данных, а формализмы, базирующиеся на теории конечных автоматов, более подходят для аспектов с доминированием управления.

Второй подход является гораздо более сложной и многоуровневой задачей, однако при этом сулит немалые выгоды. Как правило, данные вопросы исследуются в контексте проектирования компиляторов для векторных и параллельных машин. Примеры реализаций – Pico Express, XPRES). С академической стороны основными исследователями технологий компиляторов являются T. U. Delft, University of California Irvine, University of Michigan, Georgia Tech.

Большинство техник извлекают параллелизм из анализа зависимостей, обращение к которым происходит из вложенных циклов. Это вызвано тем, что циклы обычно представляют собой ядро алгоритма, поглощающее основную массу ресурсов, а чтение и запись данных в массивы легко формализуемы и обычно представляет собой узкое место коммуникаций. Оба аспекта широко освещены в литературе, посвящённой компиляции для суперкомпьютеров и векторных машин. Однако, у реализаций подходов, основывающихся на анализе и распараллеливании циклов, слишком много ограничений.

Архитектурное описание. Архитектура обычно рассматривается как набор компонентов, которые предоставляют доступ для реализации некоторого поведения. Эти компоненты обычно соединены в виде некоторого графа или списка соединений для задания способа обмена информацией между ними. В своей самой базовой форме каждый компонент архитектурного описания может быть идентифицирован с реальным устройством, и список соединений представляет физические соединения между устройствами. Элементы в архитектуре классифицируются, основываясь на том, какой тип поведения они могут реализовывать, и обычно делятся на три основных класса, в соответствии с основными функциями: вычислительной, коммуникационной и хранения. Такое архитектурное описание наиболее близко к аппаратной реализации системы и может иногда быть рассмотрено как абстрактная схема аппаратной части проекта.

В другом, более комплексном случае архитектурное описание также включает элементы, которые не могут быть напрямую ассоциированы с физическими устройствами. Примеры таких элементов – сетевые протоколы, набор инструкций ЦП, политика доступа к разделяемым ресурсам. Также должна рассматриваться и программная архитектура, отображающая организацию задач и уровней, которые реализуют часть системного поведения, с возможным включением RTOS. Комплексные спецификации, особенно когда описание может быть перекроено для специфических случаев и повторно использовано в некоторых проектах, также называются платформами.

Отображение приложения на архитектуру или платформу требует от проектировщиков назначить каждому функциональному модулю архитектурный элемент, способный реализовать заданную функциональность. В конце процесса разделения вся системная спецификация характеризуется не только функциональным поведением, но и общими характеристиками производительности, полученными из списка аннотированных показателей, которые могут представлять собой энергопотребление, занимаемый объём, общее использование архитектурного ресурса, достигаемый уровень параллелизма,

удовлетворение каким-нибудь требованиям вроде отсутствия deadlocks и т.д.

Цель разделения – нахождение оптимальных соответствий функциональных компонентов архитектурным для обеспечения требований к проектируемому изделию. Это обычно означает нахождение лучших компромиссов между различными требованиями. Взаимоотношения между всеми требованиями обычно очень сложно смоделировать аналитически, для оценки используется динамическая симуляция.

Архитектурная модель должна задавать и проверять корректность архитектуры, а также позволять отображать на себя функциональную спецификацию и извлекать данные о производительности и взаимодействии между различными компонентами.

Характеристики, которые должны рассматриваться в архитектурном описании [15]:

1) аннотированная производительность ("annotated performance") – отображение приложения на архитектуру позволяет характеризовать систему как набор элементов с приблизительными оценками производительности;
2) исполняемая спецификация – архитектура должна иметь симулируемую спецификацию для основного механизма верификации, позволяющую предотвратить несовместимости и предоставляющую общее окружение для разных уровней процесса верификации;
3) вместимость ("capasity") – способность архитектурного элемента реализовывать разные поведения;
4) параметризация – многие компоненты могут быть сконфигурированы набором параметров, например, шириной шин адреса и данных для памяти;
5) ограничения и свойства – часто варьируемые параметры архитектур и платформ должны удовлетворять набору ограничений и свойств, обеспечивающих реализуемость системы;
6) параллелизм и арбитраж – функциональная спецификация, отображённая на архитектуру, обычно представляет собой набор параллельных процессов, поэтому должны указываться доступная параллельность в каждом компоненте и какие алгоритмы используются для обеспечения доступа к совместно используемым ресурсам;

7) иерархия – позволяет задать архитектуру и отображать её в виде нескольких иерархических уровней для упрощения проектирования и, иногда, переключения между разными уровнями абстракции.

Кроме того, важной задачей архитектурного описания является разделение между выполнением и коммуникациями. Причина этого разделения – то, что вычислительные элементы обычно компилируются или синтезируются, коммуникации же обычно описываются в виде заданных паттернов – вроде прерываний, поллинга (pooling), разделяемой памяти и т.д., что требует разных методов для планирования доступности этих ресурсов.

Разделение. Разделение параллельного приложения на архитектурные модули может быть произведено двумя путями: последовательным уточнением функциональной модели с превращением её в отображённую или заданием явного указания отображений, которое используется инструментами как для получения детализированной модели, так и для управления синтезом.

Одно из представлений разных уровней абстракции и актуальный метод реализации системы представлен на Рис. 59. Наиболее абстрактный уровень описывается функциональной несинхронизированной моделью A, также часто называемой моделью спецификации. Добавлением архитектурной информации и приблизительного функционального временного поведения получается модель B, называемая моделью компонентов сборки. Выбор арбитража шин позволяет получить некоторое представление о реальном «тайминге» коммуникаций, давая модель арбитража шин C. Из этой модели можно пойти двумя путями для достижения конечной цели – через поведенческую шинно-функциональную модель D или через потактовую модель вычислений можно получить модель RTL описания F. Данный подход можно реализовывать через SystemC.

При втором подходе конкретная реализация требует отображения функциональной модели на архитектурную, при помощи некоторой метамодели. Ключевое отличие от предыдущего подхода заключается в том, что никаких изменений функциональной модели не требуется, меняются только используемые коммуникационные примитивы и добавляются ограничения, не предусмотренные более высокими уровнями абстракции, для обеспечения возможности пользования конкретными ресурсами. Используя программную терминологию, это родственно аспектно-ориентированному программированию.

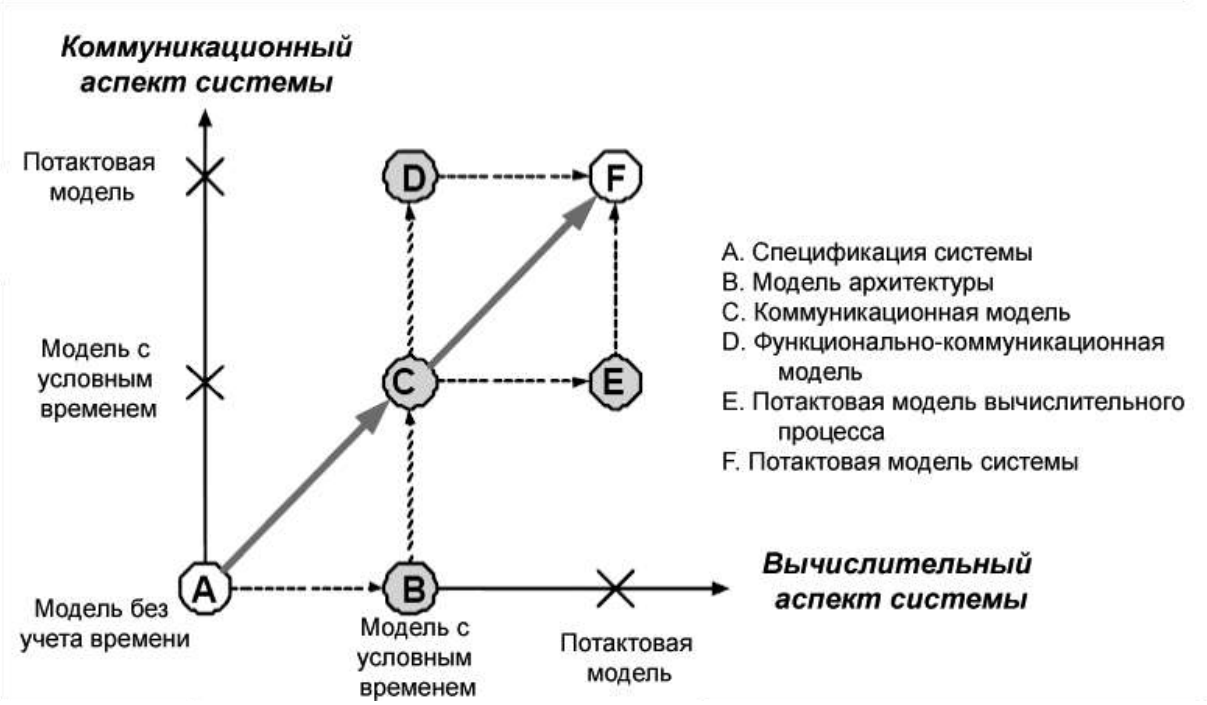


Рис. 16. Абстрактные модели [20].

За отображением детализированной системы на архитектуру компонентов может следовать детализация коммуникаций. Оптимизация должна быть проведена при помощи инструментов аппаратного и программного синтеза при генерации реализации. Так как предыдущие шаги были направлены на повышение возможности повторного использования, они могли вносить избыточность. Структура сетевых протоколов как правило модульная, что снижает эффективность. Результаты такой глобальной трансформации становятся менее пригодными для последующего повторного использования.

Для получения оптимального решения все ограничения проекта должны быть приняты во внимание при разделении. Учёт ограничений должен производиться на максимально возможно высоком уровне абстракции. Синтез системы – комплексная проблема автоматического отображения спецификации в гетерогенную аппаратно-программную архитектуру. Это включает в себя выбор оптимальной архитектуры – выделение ресурсов ("allocation"), отображение алгоритма на выбранную архитектуру в пространстве – привязка ("binding"), и во времени – пространственное планирование ("sequence planning"), исследование пространства проектных решений для удовлетворения ограничениям по производительности и стоимости. Существующие подходы к синтезу опираются на фиксированные архитектуры, концентрируются на этапе привязки, игнорируют коммуникационную проблему или требуют огромного количества времени на вычисления. Без этих ограничений настоящее исследование пространства проектных решений обычно

невозможно; кроме того, они ставят под вопрос результаты автоматического системного синтеза на практике.

Аппаратное разделение. Существование большого количества вариантов реализации аппаратуры, их растущая сложность требуют добавления аспектов сопровождаемости и расширяемости в процесс разделения, причём, даже для относительно жёсткого аппаратного сектора. Проектирование аппаратуры выглядит всё больше как конфигурирование уже существующей платформы, когда основная работа сдвигается в программную область, где более дешёвые инструменты и меньше стоимость исправления ошибок. Это также приводит к новым требованиям в плане надежности и детерминизма для процесса внедрения программного обеспечения – верификация не может больше считаться роскошью и выполняться задним числом. Формальная проверка на отсутствие ошибок в конечном продукте становится обязательным требованием.

Программное разделение. Так как сейчас только критические к задержкам и потребляющие много энергии части систем реализуются в аппаратуре, основная часть функционала остаётся на долю программного обеспечения процессоров. Ввиду того, что может требоваться выполнение сразу нескольких задач с разными требованиями к ресурсам и времени, для управления часто применяются операционные системы, с многими уровнями абстрагирования аппаратуры, как у сетевых стеков. То есть получается, что программная архитектура не менее, а в некоторых случаях и более важна, чем аппаратная и имеет существенное воздействие на производительность всей системы. В то же время хорошо продуманная программная архитектура позволяет эффективно использовать аппаратные платформы, добавлять и убирать функции, тем самым генерируя производные продукты от общей базы исходных кодов.

Разделение между несколькими процессорами позволяет использовать крупногранулярный параллелизм, однако, накладные расходы на коммуникации и синхронизацию могут превышать получаемую пользу. Для компенсации негативного воздействия необходимости обмениваться данными могут использоваться специализированные механизмы, вроде выделенных каналов между процессорами. Типичный случай – процессор общего назначение и DSP-сопроцессор.

Создание нескольких задач на одном процессоре и управление ими для встраиваемых систем может быть реализовано очень простыми техниками, вроде циклического планировщика без приоритетов, или очень сложными схемами, учитывающими ограничения реального времени. Во всех случаях задачей является максимизация использования процессора при удовлетворении всех ограничений по времени.

Иногда необходимо гарантировать, что заданная политика планирования соответствует требованиям. В таком случае требуется формальный анализ требований каждой задачи для создания метода, который позволит решить проблему. Примером таких методов является Rate Monotonic Analysis (RMA).

Коммуникации между задачами могут реализовываться разными путями, основное различие между ними – являются ли коммуникации блокирующими или неблокирующими, синхронными или асинхронными. В домене встраиваемых систем существует много схем межпроцессных коммуникаций, наиболее распространены очереди и разделяемая память. Разделяемая память потенциально более эффективна, очереди удобнее, когда требуется синхронизация. Правда, при большой задержке на доступ к памяти оба метода далеки от идеала, и в современных мультипроцессорных чипах предпочтительно наличие выделенного канала межпроцессорной связи.

Планирование программных задач во время выполнения – не лучшее решение во многих случаях: затраты на само планирование и коммуникации могут быть существенными в масштабе разрабатываемой системы. Для уменьшения этих затрат могут использоваться оптимизирующие компиляторы, которые делают основную работу статически, комбинируя набор задач в урезанный вариант, где параллельные задачи расположены в определённом порядке и на самом деле выполняются последовательно.

Для гарантированного удовлетворения временным ограничениям приближённые решения алгоритмов планирования должны знать время исполнения каждой задачи в наихудших условиях ("worst-case"). Подсчёт данных параметров вручную длителен и труден, и существуют некоторые методы и инструменты, автоматизирующие его – например, Program Analyser Generator от Absint и язык WHILE.

Для эффективного управления процессором или набором задач большинство встраиваемых задач требуют использования ОС, которые виртуализируют процессор (или набор процессоров) в набор задач, управляют доступом к разделённым аппаратным ресурсам и обеспечивают межпроцессные коммуникации, выделение памяти, защиту одних задач от сбоев в других.

Программное разделение должно учитывать наличие и характеристики операционной системы, так как многие решения могут зависеть от этого. Кроме того, может быть случай, когда отдельные свойства и функции могут быть отображены на компоненты ОС (например, драйверы).

Разделение памяти. Также влияет на характеристики системы, и потому ему должно уделяться отдельное внимание. Обычно память проектируется в виде иерархии, с виртуализацией больших, медленных и дешёвых блоков быстрыми и дорогими кэшами и сверхоперативными запоминающими устройствами. Организация памяти и способ хранения данных вместе с доступом к ним, должны проектироваться одновременно для достижения максимальной возможной производительности. На задание характеристик блоков памяти влияют также и ограничения по энергопотреблению и величине задержек.

Реконфигурирование. Использование реконфигурируемых вычислителей призвано бороться с растущей стоимостью проектирования и создания ASIC. Добавление реконфигурируемого блока к жёстко заданной аппаратуре позволяет поддерживать больше различных приложений, чем в случае традиционных случаях, когда вся аппаратная часть жёстко задана. Динамическая подстройка позволяет адаптировать аппаратуру к изменяющимся требованиям и стандартам, разделяя достоинства встраиваемого ПО, с лучшей производительностью и более низким энергопотреблением. Таким образом, заполняется ниша по производительности, стоимости и энергопотреблению между ПО и аппаратурой.

Ключевая проблема с динамически реконфигурируемой аппаратурой – сложность программирования, так как традиционные процессы разработки для аппаратуры – синтез, размещение и трассировка, для ПО – компиляция, запуск и отладка, не поддерживают её. Динамическое реконфигурирование может быть рассмотрено как гибрид между ПО, где ЦПУ "реконфигурируется" с выполнением каждой инструкции, а объём памяти большой, но пропускная способность доступа к ней ограничена и аппаратурой, где реконфигурация происходит редко (например, записью в конфигурационные регистры UART), памяти мало, но доступ к ней имеет потенциально высокую пропускную способность. Объединение процессора общего назначения с реконфигурируемым модулем требует создания модели программирования и решения проблемы пропускной способности коммуникаций.

Реализация коммуникаций. Часто коммуникации могут быть узким местом системы и практически всегда сильно влияют на её свойства. Существует два основных способа реализации интерфейсов: синтез и использование шаблонов.

При использовании шаблонов применяется набор готовых решений, которые настраиваются исходя из функциональных требований и требований производительности. Существует множество инструментов, которые рассматривают взаимодействия в системе между аппаратурой и

аппаратурой и между аппаратурой и программным обеспечением как взаимодействия между настраиваемыми модулями (драйверами, регистрами, декодерами и т.д.). Ключевой проблемой является обеспечение непротиворечивости описаний параметров интерфейсов (ширина буферных регистров, адреса каждого регистра и т.д.) в разных местах. Это подразумевает уверенность в том, что связанные интерфейсные IP-блоки и программные драйверы настраиваются, начиная с единичного представления ("single view") или с набора представлений, чья непротиворечивость может быть автоматически проверена. Например, программные драйверы должны использовать высокоуровневые представления управляющих регистров и регистров данных, чтобы при изменениях в спецификации конкретного интерфейса нужно было только перекомпилировать драйвер, и простые несоответствия могли быть сразу обнаружены.

Подходы к синтезу интерфейсов, как правило, предполагают проектирование совместимых протоколов, синтез преобразователей протоколов, а также выявление невозможности совмещения разных протоколов с существующими ограничениями. Для определения, совместимы протоколы или нет, обычно проверяется, согласуются ли между собой интерфейсы при том потоке данных, который планируется передавать. Размеры блоков, передаваемых данных, могут различаться, но средние пропускные способности приёмника и передатчика должны соответствовать друг другу. Протокол на сигнальном уровне, как правило, описывается в семантике конечных автоматов в расчёте на использование различных техник синтеза протоколов.

2 ИНСТРУМЕНТАЛЬНОЕ ОБЕСПЕЧЕНИЕ ПРОЕКТИРОВАНИЯ

2.1 Обзор рынка САПР

На начальном этапе своего развития направление кодизайна ограничивалось академическими исследованиями и не имело поддержки со стороны коммерческих САПР. Однако, сейчас ситуация меняется в лучшую сторону. Уже появляются инструментальные средства для сквозной и комплексной разработки сложных электронных систем, начиная с системного уровня, в том числе предусматривающие совместное проектирование ПО и аппаратуры. Кроме того, при наличии чёткого понимания желаемого маршрута проектирования, возможно интегрировать между собой инструменты разных производителей, покрывающие определённые задачи синтеза, анализа и т.д., даже при отсутствии их интеграции «из коробки».

Сегодняшний рынок САПР достаточно обширен и разнообразен. Наибольшее развитие получили САПР проектирования и производства интегральных микросхем класса «Система на кристалле». Стоит отметить, что САПР данного типа для системного уровня проектирования могут использоваться для проектирования аппаратуры любой вычислительной системы. Специализация системы происходит на уровне синтеза решений в конкретный базис с использованием библиотеки компонентов либо малой (транзисторов, логических вентилей, ячеек ПЛИС), либо высокой степени гранулярности (процессоров, периферийных контроллеров, вычислительных платформ и т.п.).

Также, доступны САПР для проектирования:

- печатных плат и многокристальных модулей (PCB & MCM);
- механических соединений, корпусов, составных частей бортовых систем автомобилей, оптических систем и пр. (Computer Aided Engineering, CAE);
- и многие другие.

Распределение САПР по областям показано на Рис. 17.

Главными игроками рынка САПР являются компании Cadence, Synopsys, Mentor Graphics. Они предоставляют интегрированные решения для всех этапов проектирования вычислительной техники, начиная от системного уровня и заканчивая подготовкой производства и выпуском печатных платы, аналоговой и цифровой электроники в виде интегральных микросхем и др.

Доля рынка, закрепленная за САПР различных компаний, представлена на Рис. 18.

В список других компаний входят такие компании, как Agnisys, Altium, Altera, Xilinx, National Instruments и др.

Далее подробнее рассмотрим маршруты проектирования и соответствующие САПР лидеров рынка.

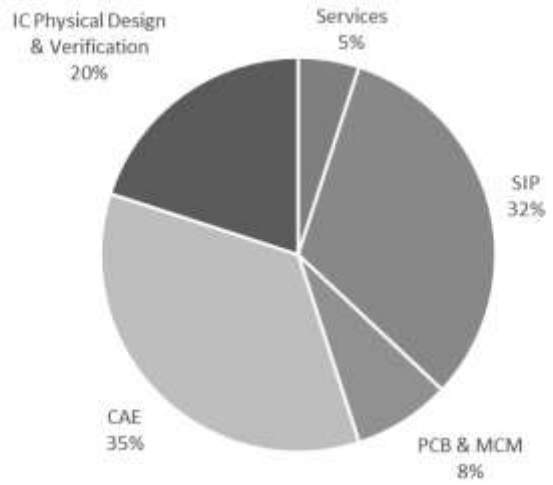


Рис. 17. Распределение САПР по областям (данные EDA Consortium, Market Statistics Service, 2015 г.).

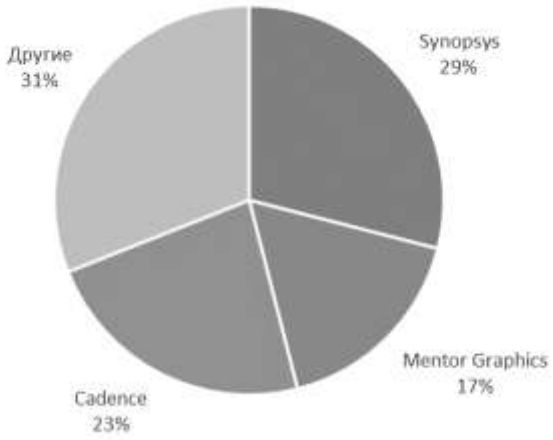


Рис. 18. Доля рынка, закрепленная за САПР различных компаний (по данным на четвертый квартал 2015 г.).

Стоит отметить, что современные вычислительные системы – это сложные технические устройства, состоящие из нескольких вычислительных модулей. Каждый модуль системы, как правило, выполняется в виде печатной платы, содержащей необходимые вычислительные ядра и периферийные устройства (устройства взаимодействия с внешним миром и другими модулями). Сами вычислительные ядра могут изготавливаться в виде интегральных

микросхем, либо систем, реализованных на микросхемах программируемой логики (ПЛИС).

В настоящее время коммерческие САПР поддерживают интегрированные маршруты проектирования, как интегральных микросхем, систем на ПЛИС, так и печатных плат, а также многокристальных модулей. При этом предоставляются средства комплексного решения задач управления проектом, разработки новых технических решений, анализа и контроля качества получаемых результатов.

В соответствии с представленными на рынке САПР можно выделить следующие стадии получения целевой вычислительной системы:

- уровень системного проектирования;
- уровень технической реализации.

Структура и взаимосвязь результатов разработки вычислительной системы представлены на Рис. 19

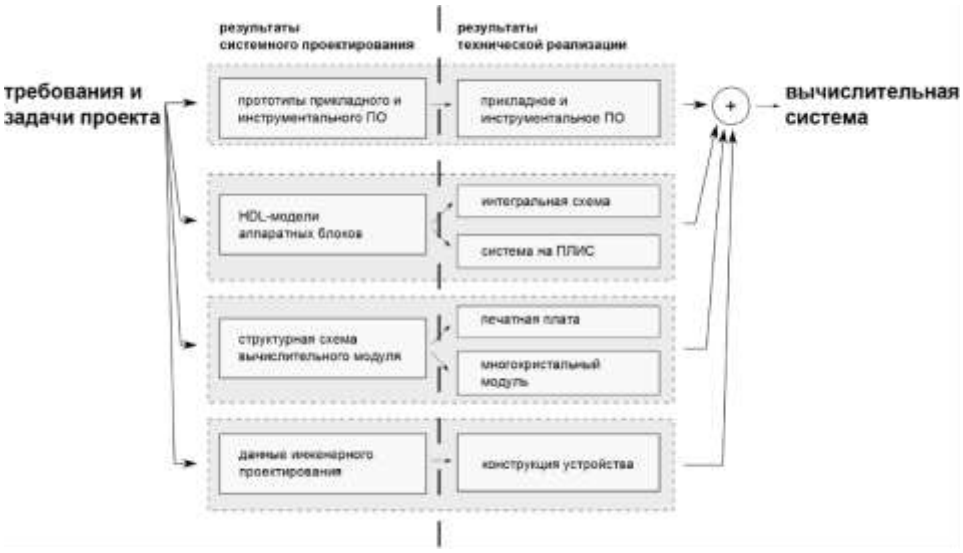


Рис. 19. Структура результатов проектирования вычислительной системы средствами САПР

На уровне системного проектирования создается проект системы, который представляется в виде набора файлов, описывающих конкретные, формализованные методы (пути) удовлетворения требований. При этом формат файлов выбирается такой, чтобы содержимое файлов возможно было однозначно интерпретировать на уровне технической реализации. В ходе системного проектирования производится разделение системы на вычислительные модули и определение требования к технологии их реализации/производства в виде печатных плат или многокристальных

модулей, производится уточнение требований к инженерному проектированию (к корпусу, жгутам, креплениям и т.п.).

Под технической реализацией понимается подготовка и непосредственно выпуск составных частей системы в виде интегральных микросхем, систем на ПЛИС, печатных плат и многокристальных модулей.

При производстве цифровых интегральных микросхем или реализации частей системы на ПЛИС результаты системного проектирования представляются в виде HDL-моделей системы и её функциональных узлов. Под HDL-моделями понимаются набор описаний поведения аппаратных блоков с точностью до такта системного сигнала синхронизации.

При производстве печатных плат и многокристальных модулей проект системы может представляться как в абстрактном виде – структурно-функциональной схемы, так и в более детальном – принципиальной схемы будущего устройства (вычислительного модуля).

Существующие технологии позволяют проводить параллельную разработку программного обеспечения (ПО) и аппаратной платформы системы. Таким образом, в идеальном случае прототипы инструментального и прикладного ПО, созданные на этапе системного проектирования, с небольшими доработками возможно использовать на выпущенном экземпляре системы, то есть её технической реализации.

Также стоит выделить процесс проектирования инженерных конструкций, механических соединений будущего устройства. Данный процесс является неотъемлемой частью комплексного системного проектирования, но в рамках данного учебного пособия он не будет затронут. Основное внимание будет уделено проектированию блоков, непосредственно участвующих в вычислительном процессе.

Рассмотрим основные технологии, используемые на каждой стадии маршрута проектирования и поддерживаемые средствами современных САПР.

2.2 САПР проектирования системного уровня

После определения требований и задач проекта начинается этап системного проектирования. Основной задачей данного этапа является создание проекта архитектуры будущей системы и анализ её системных характеристик: производительности, энергопотребления, габаритов, надежности и т.п. Под созданием проекта архитектуры понимается определение номенклатуры функциональны узлов, их назначения, алгоритмов поведения и связей с другими подсистемами.

На данном этапе применяется технология проектирования системы с использованием виртуальных компонент. Виртуальный компонент – это исполняемая модель функционального узла системы с определенной функциональностью. Поведение данного компонента детализируется в процессе проектирования. Так проектирование архитектуры начинается с использованием виртуальных компонент, в которых не учитывается параметр времени, и заканчивается созданием потактовых моделей соответствующих узлов.

Сегодня у каждого производителя есть свои средства для работы с виртуальными компонентами. Cadence предоставляет соответствующие средства в рамках Virtual System Platform, Synopsys – Virtulazer, CoMET, METeor, Mentor Graphics – Vista.

Данные средства позволяют создавать сами виртуальные компоненты, объединять их в систему, создавать тестовые окружения, проводить моделирование и измерять характеристики работы системы. Виртуальные компоненты, как правило, описываются на языке SystemC или SystemVerilog с использованием библиотеки TLM и являются транзакционными моделями соответствующих функциональных узлов системы. Примерами виртуальных компонент являются процессоры, контроллеры периферийных устройств, графические ускорители и др.

Существует три основных способа получения необходимых виртуальных компонент. Они могут быть созданы вручную «с нуля», взяты в качестве готовых блоков из библиотеки производителя и получены средствами САПР из высокоуровневого описания.

Входными данными для разработки виртуальных компонентов вручную «с нуля», так и с использованием средств САПР является набор декларативных описаний архитектуры системы и её составных частей. В качестве декларативных описаний могут выступать:

- не стандартизированное описание архитектуры, синтаксис и семантика которого определены в рамках конкретной команды разработчиков;
- описание архитектуры с использованием UML-подобных языков, например, SysML;
- описание архитектуры с использованием ADL (Architecture Description Language) и PDL (Processor Description Language) языков, например, таких как nML, LISA, ArchC и др.;
- математическое описание алгоритмов обработки данных.

Процессы создания виртуальных компонентов из ADL/PDL описания поддержаны современными средствами САПР. В настоящее время этот

подход применяется для создания специализированных процессорных ядер с программным управлением и архитектурой ISA (Instruction Set Architecture). Активные работы по созданию таких средств проводит фирма Synopsys. Она уже выпустила на рынок САПР Processor Designer, где основным языком описания архитектуры является язык LISA, а также недавно создала ещё ряд САПР для проектирования многопроцессорных систем ASIP Designer, MP Designer, в которых используется язык nML.

Получение HDL-моделей и соответствующего инструментального ПО для программирования, отладки и тестирования создаваемых блоков при таком подходе может быть выполнено также с помощью средств САПР. САПР Processor Designer, ASIP Designer, MP Designer позволяют синтезировать из ADL/PDL описания не только виртуальные компоненты, но и создавать HDL-модели на языках Verilog и VHDL, а также полный набор инструментального ПО для них: компилятор, загрузчик, отладчик, инструменты для тестирования.

Преобразование остальных типов декларативных описаний в исполняемые модели в настоящее время недостаточно формализовано и не получило должной поддержки со стороны САПР.

Для создания HDL-моделей алгоритмических блоков, например, блоков цифровой обработки сигналов возможно воспользоваться средствами высокоуровневого синтеза (HLS, High Level Synthesis). Они позволяют проводить прямой синтез описания алгоритма на языках C/C++/SystemC в HDL-модель. Данные инструменты предоставляются фирмой Cadence в рамках САПР C-to-Silicon Compiler и более нового продукта Stratus HLS. Соответствующие инструменты есть у фирмы Synopsys – Synphony C Compiler. Mentor Graphics не имела до конца 2015 года собственных инструментов HLS. В сентябре 2015 года Menor Graphics купила Calypto Design Systems с планами развивать её деятельность в рамках отдельной дочерней компании. Вместе с приобретением Calypto фирма Mentor Graphics получила контроль над разработкой САПР высокоуровневого синтеза Catapult C.

Еще одним способом автоматизации проектирования HDL-моделей является использованием средств визуального проектирования отдельных составных частей функциональных узлов, например, устройств управления, контроллеров периферийных интерфейсов. Mentor Graphics для этих целей создала отдельные инструменты: HDL Designer, HDL Author, HDL Visual Elite. В их задачи входят преобразование структурного описания системы в виде системы взаимосвязанных блоков в HDL-описания соответствующей структуры, а также синтез HDL-моделей дискретных алгоритмов управления из их описания в виде формальных моделей конечных автоматов.

Оставшаяся часть работ по созданию HDL-моделей, которая не может быть выполнена с помощью САПР, выполняется вручную. На рис. 70 показаны основные процессы получения HDL-моделей аппаратных блоков вычислительной системы. На рисунке отмечены процессы, которые выполняются разработчиками вручную (сплошная стрелка) и которые поддержаны средствами САПР (пунктирная стрелка).

Прототипы ПО разрабатываются с использованием виртуальных компонентов. Прикладное ПО запускается непосредственно на соответствующих виртуальных моделях процессоров. При разработке алгоритмов обработки данных могут использоваться средства различных математических пакетов типа Matlab, Simulink, Maple, Mathematica. Подобные инструменты есть также и у Synopsys. Они включены в САПР System Studio и имеют средства интеграции с проектами Matlab и Simulink.

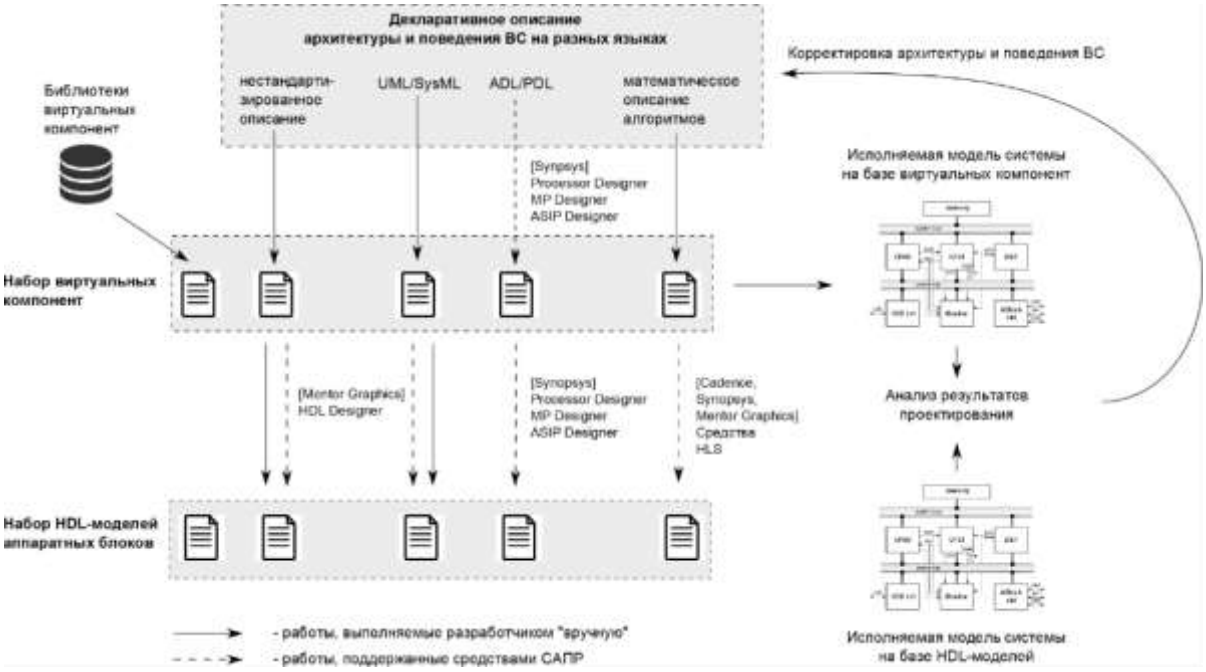


Рис. 20. Процесс получения HDL-моделей аппаратных блоков системы.

В таблице (Таблица 1) представлена сводная таблица современных технологий высокоуровневого системного проектировании и соответствующих средств САПР фирм-лидеров рынка, в которых данные технологии реализованы.

На каждой стадии проектирования важно осуществлять контроль качества получаемых результатов. Возможные виды анализа и соответствующие средства САПР представлены на Рис. 21.

Таблица 1. Технологии системного проектирования, реализованные в САПР фирм-лидеров рынка.

Фирма-производитель САПР	Cadence	Synopsys	Mentor Graphics
Средства проектирования системы на базе виртуальных компонентов	Virtual System Platform	Virtualizer, CoMET, METeor	Vista
Средства высокоуровневого синтеза	Stratus HLS	Synphony C Compiler	Catapult
Средства проектирования алгоритмов обработки данных	-	System Studio	-
Средства высокоуровневого проектирования архитектуры специализированных процессоров	-	Processor Designer, ASIP Designer, MO Designer	-
Визуальное проектирование HDL-моделей аппаратных блоков	-	-	HDL Designer, HDL Author, HDL Visual Elite

Под статическим или формальным анализом понимается анализ структурного описания проекта на предмет ошибок. В качестве примера можно привести поиск синтаксических ошибок в коде, поиск участков «мертвого кода», вычисление метрик функционального покрытия, проверку описания конечных автоматов, межмодульных интерфейсов и т.п. Под «мертвым кодом» подразумевается поиск участков кода, описывающих поведение блоков, которым не передаются сигналы управления в процессе работы устройства, либо результаты работы, которые не используются другими блоками. Также с помощью инструментов статического анализа производится предварительная оценка энергопотребления и габаритов будущей схемы, выполняется статический временной анализ и определяется верхняя граница частоты тактового сигнала для существующих доменов синхронизации, создается набор тестов и формальных правил для динамической верификации.

Для выполнения статической верификации возможно воспользоваться программным продуктом Incisive Formal Verifier фирмы Cadence. Фирма Synopsys реализует подобные средства в новом САПР Verification Compiler, где интегрирует возможности своих существующих инструментов: Certitude (проверка качества верификационного окружения), HECTOR (проверка согласованности нескольких представлений функционального блока, проверка структуры и связей моделей, написанных на языке C и HDL-моделей), VC Formal (статический анализ кода, оценка функционального покрытия), BugScope (генерация проверяемых правил утверждений – assertions – для этапа динамического анализа). Фирма Mentor Graphics реализует средства статического анализа в САПР Questa Verification Platform.

Рис. 21. Виды анализа результатов проектирования, поддерживаемые средствами САПР.

Под динамическим анализом понимается проверка требований проекта в процессе выполнения исполняемых моделей функциональных блоков в симуляторе с помощью технологий компьютерного моделирования. В процессе моделирования проверяется поведение устройства при различных вариантах входных воздействий. В существующих САПР реализуется так называемая технология гетерогенного моделирования. В одном симуляционном окружении возможно анализировать совместную работу моделей как разного уровня детализации, так и написанные на разных языках. Поддерживается одновременное моделирование виртуальных компонентов, написанных на

языках SystemC и SystemVerilog, а также более низкоуровневых HDL-моделей, написанных на языках Verilog HDL, VHDL (рис. 72).

Рис. 22. Использование средств компьютерного моделирования для визуализации поведения разрабатываемой системы.

Упомянутые средства моделирования поддерживают не только визуализацию результатов симуляции в виде временных диаграмм, но и позволяют производить анализ выполнения формальных требований-утверждений (assertion) на базе данных моделирования, а также средства визуализации/анализа проекта на транзакционном уровне, средства анализа тестового окружения, написанного с использованием библиотеки UVM. Соответствующие продукты имеют встроенную поддержку анализа формул утверждений на языках PSL (Property Specification Language), SVA (System Verilog Assertion) и язык ***e***. Средства визуализации/анализа проекта на транзакционном уровне поддерживаются в рамках библиотеки TLM для языков SystemC и SystemVerilog.

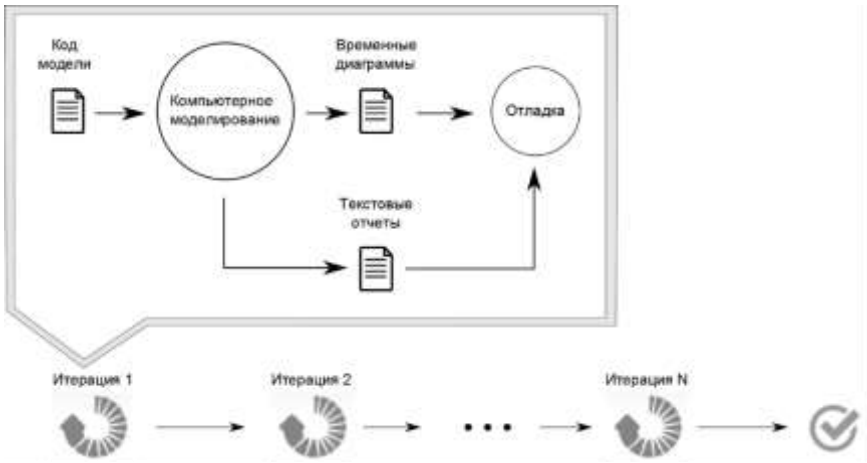


Рис. 23. Традиционный подход к поиску причин ошибок проекта.

Для задач динамического анализа Cadence разработала Incisive Enterprise Simulator. Фирма Synopsys реализует соответствующие средства

в среде моделирования VCS. Mentor Graphics для данных задач предлагает использовать QuestaSim либо свой более старый, но более стабильный и заслуживший признание множества пользователей продукт ModelSim.

Средства обработки результатов моделирования. С ростом сложности системы объем результатов моделирования также стремительно растет. В процессе моделирования в течение нескольких микросекунд образуются гигабайты данных, описывающих происходящие в системе события. Средства контроля формальных утверждений (assertion-based verification), а также вручную написанные средства контроля и анализа результатов моделирования позволяют проверить и установить причины в рамках изначально предполагаемых граничных случаях. Задача усложняется, если система на выходе формирует некорректные результаты, а существующие средства не позволяют точно локализовать и установить причины непредвиденных отказов. Один из вариантов является уточнение требований, формальных правил проверки, доработка кода тестового окружения и перезапуск процесса моделирования. Это многоитерационный подход, который может существенно увеличивать время отладки и тестирования системы (Рис. 23).

Учитывая, что на такой процесс тратится большое количество времени (Рис. 24), разработчики САПР призывают отойти от метода поиска, основанного на выдвижении предположений о причинах обнаруженных ошибок и многоитерационной их проверки в симуляторе. Продвигается тезис о том, что в таком случае разработчик работает «в темноте», отлаживая работу системы по частям, не имея полного системного представления о результатах работы всех компонентов в совокупности.

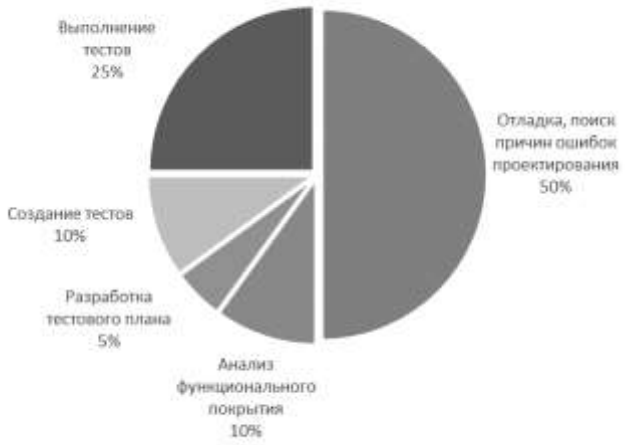


Рис. 24. Распределение усилий в процессе анализа результатов проектирования и поиска причин ошибок проектирования.

Вместо этого предлагается получать исчерпывающие данные о работе разрабатываемого устройства в процессе одного, либо нескольких последовательных циклов моделирования, а затем использовать эти данные для детального анализа и поиска причин ошибок. Результаты каждого цикла помещаются в базу данных, содержимое которой анализируется с помощью алгоритмов работы с «большими данными» (Big Data). Схема данного подхода представлена на Рис. 25.

Описанных подход значительно увеличивает пространство поиска и, по мнению специалистов, увеличивает эффективность отладки на 50%. В качестве критериев эффективности процесса отладки берут время работы ядра симулятора на рабочем компьютере или сервере и время, затраченное инженером на анализ данных моделирования, локализации ошибки и поиска её причин.

Рис. 25. Использование технологии работы с Big Data для поиска причин ошибок

В существующих САПР реализованы методы отслеживания взаимосвязей между событиями, создаваемых в тестовом окружении, процессом исполнения поведенческой модели устройства и результатами моделирования, представленными на временных диаграммах. При анализе работы отдельного функционального узла разработчик имеет возможность перемещаться между результатами моделирования различных типов и анализировать их связи.

Предоставляются средства для отслеживания причинно-следственных связей между изменениями переменных проекта. Реализованная технология получила название Root-Cause Analysis (RCA). Все факты исправления ошибок записываются в журнал, и разработчик позже может просмотреть историю изменений в проекте и оценить эффект от этих изменений, выполнив повторное моделирование, вернувшись к предыдущим правкам.

Cadence для решения данных задач разработала САПР Indago Debug Platform, фирма Synopsys – инструменты Verdi и The Siloty, Mentor Graphics реализует часть соответствующих инструментов в рамках

интегрированной платформы Questa Verification Platform, а также в Visualizer Debug Environment.

Средства ускорения симуляции и средства эмуляции у Cadence реализованы в продуктовой линейке Palladium. Cadence предоставляет потребителям сервера Palladium XP, Z1, выполненные в виде набора процессорных модулей с необходимым программным обеспечением. Cadence осуществляет продажу как самих серверов, так и услугу доступа к вычислительным мощностям своих серверов.

Стоит различить задачи ускорения симуляции и эмуляции. Под первой подразумевается интеграция с тестовым окружением проекта и выполнение отдельных его частей на более высокопроизводительной платформе, нежели рабочая станция. Под эмуляцией подразумевается моделирование проекта на более низком уровне с учетом физических особенностей работы отдельных вентилей. В последнем случае возможно проводить более специфические для аппаратной платформы виды анализа, например, анализ задержек распространения сигналов между компонентами, анализ потребляемой мощности частей схемы и др.

С помощью платформы Palladium возможно выполнять симуляцию и эмуляцию проектов, включающих до 9.2 миллиардов вентилей и 220 тыс. портов ввода вывода. Возможно одновременное использование платформы 2000 пользователями. Компиляция проекта выполняется со скоростью 140 миллионов вентилей в час. Эмуляция позволяет оценивать характеристики работы проекта в реальном времени на частоте 4 МГц.

Программное обеспечение платформы Palladium тесно интегрированы с продуктом Incisive Enterprise Simulator и позволяет работать с проектами, написанными на языках C++, SystemC, VHDL, Verilog, SystemVerilog, а также с представлением проекта на вентильном уровне (формат netlist).

Palladium возможно совместно использовать с продуктом Virtual System Platform и проводить совместное моделирование RTL-проекта в составе виртуальной платформы. Такой объединенный проект в Cadence называют гибридной платформой (Hybrid platform). Работа с такой гибридной платформой позволяет существенно ускорить процесс моделирования и эмуляции. Например, загрузку ядра Linux на серверах Palladium с использованием виртуальных компонентов возможно ускорить с 30 до 0.5 минут.

Подобные средства фирма Synopsys реализует в продукте ZeBu Server. Сервера Synopsys позволяют выполнять эмуляцию проектов до 3 миллиардов вентилей и поддерживают одновременную работу с проектом до 49 пользователей. В отличие от фирмы Cadence сервер эмуляции Synopsys построен на массиве вычислителей, реализованных на множестве

микросхем ПЛИС Xilinx Virtex-7 XC7V2000T. Такой подход помогает, по заявлениям Synopsys, снизить общее энергопотребление вычислительного сервера (при полной загрузке энергопотребление составляет не более 2.5 кВт), а также увеличить скорость моделирования. С помощью средств Synopsys можно производить эмуляцию проекта в реальном времени на частоте 30 МГц.

Mentor Graphics для решения задач эмуляции и ускорения симуляции разработала продукт Veloce Emulation Platform. В Veloce Emulation Platform используются сервера, построенные на базе массива специализированных вычислительных ядер, разработанных Mentor Graphics. Каждое ядро выполнено в виде отдельной интегральной схемы, содержащей блок программируемой логики, блоки памяти, конфигурируемую систему соединений, а также блоки отладки и тестирования. Такой подход позволяет улучшить масштабируемость системы и обеспечить тесную интеграцию платформы со средствами САПР за счет оптимизации архитектуры вычислителя под задачи эмуляции. С помощью серверов Mentor возможно производить эмуляцию проекта до 2 миллиардов вентилей. Сервера поддерживают одновременную работу с системой до 128 пользователей. При этом энергопотребление сервера значительно выше, чем у Synopsys и составляет 11 кВт при полной загрузке.

Средства прототипирования проекта на ПЛИС позволяют запускать проект в реальном времени на скоростях значительно больших, чем при эмуляции. Недостатком прототипирования является ограниченный доступ к информации о поведении внутренних блоков. Обычно в САПР реализованы соответствующие средства наблюдения за внутренними блоками и их отладки, предоставляющие разработчику интерфейс взаимодействия с проектом, схожий с анализом проекта в симуляторе.

Фирма Cadence для решения задач протитипирования разработала платформу Protium Platform. Данная платформа включает массив ПЛИС Virtex-7 XC7V2000T. В зависимости от версии предоставляются платформы, содержащие от 2 до 8 ПЛИС. С помощью Protium возможно запускать и производить отладку проектов или их частей емкостью до 100 миллионов вентилей и использовать ресурсы до 68 Мб памяти на кристалле.

Подобные средства реализованы у фирмы Synopsys в продукте HAPS Proto Compiler. Для прототипирования используется версии платформы, содержащие от 1 до 64 ПЛИС Virtex UltraScale VU440 FLGA2892. С помощью HAPS возможно выполнять прототипирование проектов емкостью до 1.6 миллиарда вентилей и использовать ресурсы до 700 Мб памяти на кристалле.

У Mentor Graphics подобных инструментов нет.

Средства верификации СнК с помощью программного обеспечения (Software-driven verification) реализованы фирмой Cadence в продукте Parspec System Verifier. Основная идея предлагаемой технологии заключается в автоматической генерации тестов на уровне программного обеспечения, которые нацелены на максимальную загрузку системы коммуникаций между вычислительными ядрами проекта. Ожидается, что такой подход позволит существенно сократить время отладки сложных проектов.

Фирма Synopsys и Mentor Graphics отдельно не выделяют данную технологию в линейках своих САПР.

САПР, реализующие различные технологии анализа результатов проектирования, представлены в таблице (Таблица 2).

Таблица 2.Технологии анализа результатов проектирования, реализованные в САПР фирм-лидеров рынка.

Фирма-производитель САПР	Cadence	Synopsys	Mentor Graphics
Средства статического анализа исполняемой модели системы	Incisive Formal Verifier	Verification Compiler	Questa Verification Platform
Средства динамического анализа исполняемой модели системы	Incisive Enterprise Simulator	VCS	Questasim
Средства ускорения симуляции и эмуляции	Palladium	ZeBu Server	Veloce Emulation Platform
Средства прототипирования проекта на ПЛИС	Protium	HAPS Proto Compiler	-
Средства обработки результатов моделирования, Root-Cause анализ	Indago Debug Platform	Verdi, Siloty	Questa Verification Platform, Visualizer Debug Environment
Средства верификации системы с помощью программного обеспечения	Parspec System Verifier	-	-

2.3 САПР этапа технической реализации

Этап технической реализации проекта интегральной микросхемы или системы на ПЛИС начинается с логического синтеза созданных HDL-моделей. В процессе логического синтеза HDL-модели преобразуются в структурные схемы соответствующих функциональных узлов, базовыми компонентами которых являются элементы технологической библиотеки.

При проектировании интегральных микросхем используются библиотеки физических ячеек, предоставляемые фабрикой. Как правило, такими элементами являются транзисторы и логические вентили (NAND, NOR), производство которых освоено на фабрике-производителе. При проектировании систем на ПЛИС в качестве библиотечных элементов используются базовые конфигурируемые блоки соответствующей микросхемы ПЛИС.

Для решения задач логического синтеза в базис элементов интегральной микросхемы Cadence предоставляет инструмент Genus, Synopsys – Design Compiler, Mentor Graphics – Oasys-RTL и Olympus-SoC.

Логический синтез в базис ПЛИС у Synopsys возможно использовать Synplify, у Mentor Graphics – Precision RTL, Leonardo Spectrum. У Cadence отдельных инструментов для решения этой задачи нет.

Стоит отметить, что задача логического синтеза решается в несколько этапов. На первых этапах производится трансляция HDL-моделей в виртуальные ячейки и производится оценочное размещение и трассировка элементов будущей схемы с учетом сформулированных технологических требований. Формируется набор вариантов топологии схемы, из них выбирается оптимальный вариант, который затем используется для создания схемы в заданном базисе. Данный многоитерационный подход позволяет улучшить результаты дельнейших этапов размещения и трассировки схемы с использованием физических ресурсов целевой платформы.

За этапом логического синтеза следует этап трассировки и размещения. При этом трассировка и размещения схемы на ПЛИС возможно выполнить только с помощью средств САПР компаний-производителей ПЛИС. Действительная структура каждой микросхемы является коммерческой тайной, которая в настоящее времени не раскрывается производителям интегрированных САПР такими, как Synopsys, Mentor Graphics и Cadence. В связи с этим выполнение дальнейших шагов будем рассматривать в контексте производства интегральных микросхем.

Для размещения и трассировки схемы возможно использовать САПР Innovus фирмы Cadence, САПР IC Compiler фирмы Synopsys и САПР Oasys-RTL, Olympus-SoC фирмы Mentor Graphics.

Результатом размещения и трассировки схемы является получение файла её топологии для производства. В прямом виде данный файл на практике нельзя использовать. Хотя большую часть требований можно удовлетворить с помощью автоматизированных средств проектирования, всегда остается часть, требующая ручного контроля специалистом-топологом. Для топологического проектирования Cadence предоставляет САПР Virtuoso, Synopsys – САПР Custom Designer, Laker, Helix, Mentor Graphics – САПР Pyxis Schematic, Implement, Layout. Помимо ручного контроля и корректировки проекта топологии схемы данные САПР могут непосредственно использоваться для проектирования файлов библиотек базовых элементов (топологий транзисторов, схем и топологий вентилей), используемых на фабрике производителе интегральных микросхем.

После получения файла топологии для производства наступает этап подготовки самого производства к выпуску партии интегральных схем. Для подготовки производства Cadence предоставляет линейку САПР Process Proximity Compensation, MaskCompose Reticle, Wafer Synthesis Suite, QuickView Signoff Data Analysis Environment, фирма Synopsys – Sentaurus, Taurus, IC WorkBench, CATS, Avalon, Yield Management, Odyssey, Yield Explorer, фирма Mentor Graphics – САПР Calibre.

Начиная с этапа логического синтеза и вплоть до получения топологии микросхемы, решить все задачи с помощью средств автоматизированного проектирования не удается с должным качеством. На некоторых этапах требуется «ручная» корректировка проекта со стороны разработчика, например, в части переопределения шаблонов синтеза, корректировки топологии и т.п. Изменения, внесенные в проект на определенной стадии, оказывают влияние не только на все последующие шаги, но и связаны с предыдущими. Встает задача отслеживания данных связей с целью возможности повторения всех этапов проектирования без участия разработчика и актуализации результатов на каждом этапе. Также решение данной задачи позволило бы ускорить процесс получения файлов топологии в случае внесения изменений в структуру первоначальных HDL-моделей. Так части системы, которых изменения не затронули, могут быть оставлены в том же виде и в топологии, а повторное размещение и трассировка проводилась бы локально в пределах модифицированных частей проекта.

Технология отслеживания взаимосвязей результатов проектирования носит название ECO (Engineering Change Order) (Рис. 26).

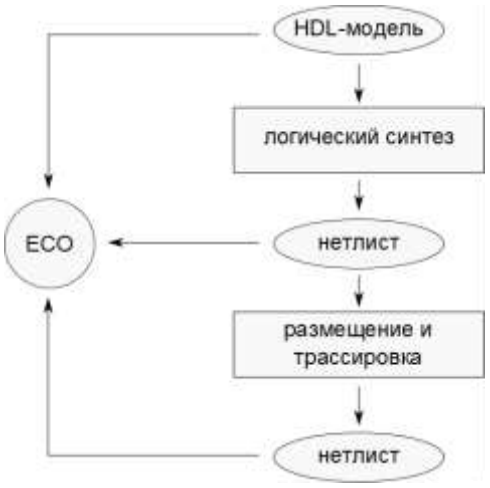


Рис. 26. Результаты проектирования, контролируемые с помощью технологии ECO.

В настоящее время технология ECO поддерживается со стороны САПР и позволяет избавиться от большого объема рутинной работы, требующей большого внимания со стороны разработчиков. Cadence с целью поддержки данной технологии разработала специальный САПР Encounter Conformal ECO Designer. Фирма Synopsys поддерживает технологию ECO в рамках САПР Design Compiler и IC Compiler, а также САПР Formality. Последний специально разработан для отслеживания взаимосвязей HDL-моделей устройства и результатов их логического синтеза. Фирма Mentor Graphics поддерживает технологию ECO в рамках САПР Olympus-SoC.

Входными данными для этапа физической/технической реализации проекта печатной платы или многокристального модуля является структурно функциональная или принципиальная схема будущего устройства. На основе этих данных проектируется топология печатной платы или многокристального модуля. Эти работы можно выполнять с помощью таких инструментов Cadence, как OrCAD, Allegro. Mentor Graphics предоставляет САПР Xpedition. Фирма Synopsys не предоставляет инструментов для решения данных задач.

Все САПР фирм Cadence, Synopsys, Mentor Graphics, ассистирующие процесс технической реализации собраны в таблице (Таблица 3).

На уровне технической реализации используются следующие средства, связанные с процедурой контроля качества и анализа результатов проектирования:

- Средства подготовки проекта для тестирования после выпуска первых экземпляров микросхем – DFT (Design for test);
- Средства проектирования тестовых векторов;
- Средства анализа принципиальных схем.

Таблица 3. САПР фирм-лидеров, ассистирующие процесс технической реализации ВС.

Фирма-производитель САПР	Cadence	Synopsys	Mentor Graphics
Средства логического синтеза в базис интегральной схемы	Genus	Design Compiler	Oasys-RTL, Olympus-SoC
Средства логического синтеза в базис ПЛИС	-	Synplify	Precision RTL, Leonardo Spectrum
Средства размещения и трассировки проекта интегральной схемы	Innovus	IC Compiler	Oasys-RTL, Olympus-SioC
Средства размещения и трассировки проекта печатной платы или многокристального модуля	OrCAD, Allegro	-	Xpedition
Средства проектирования принципиальной схемы и редактирования топологии интегральной схемы	Virtuoso	Custom Designer, Laker, Helix	Pyxis Schematic, Implement, Layout
Средства подготовки производства интегральных схем	Process Proximity Compensation, MaskCompose Reticle, Wafer Synthesis Suite, QuickView Signoff Data Analysis Environment	Sentaurus, Taurus, IC WorkBench, CATS, Avalon, Yield Management, Odyssey, Yield Explorer	Calibre
Средства отслеживания взаимосвязей результатов проектирования (технология ECO)	Encounter Conformal ECO Designer	Design Compiler, IC Compiler, Formality	Olympus-SoC

Следует заметить, что проекты для выпуска первых экземпляров микросхем отличаются от проектов для серийного производства. На первых экземплярах производится оценка качества технологического процесса фабрики для реализации конкретного проекта, то есть определяется, с каким качеством топология микросхемы может быть произведена с использованием возможностей определенной кремниевой фабрики. Для этого в проект системы встраиваются специальные блоки тестирования, посредством которых производится внутрисхемная функциональная верификация первых экземпляров схем.

Специализированные блоки тестирования, встраиваемые в микросхему, позволяют переключать входы функциональных узлов системы между целевыми каналами и каналами тестирования, а также контролировать значения выходных сигналов. Таким образом, становится возможным изолировать определенные узлы и проверить их поведение, подавая на вход заранее заготовленные наборы значений (тестовые вектора). Проект схемы со встроенными блоками тестирования называется Design For Test (DFT).

Сегодня процесс создания встроенной инфраструктуры для тестирования частично автоматизирован и для типовых функциональных блоков схемы, например, таких как арифметические блоки, конечные автоматы, используются уже проверенные методики. Компании-разработчики САПР предоставляют свои средства для автоматической вставки типовых цепей тестирования, генерации набора тестовых векторов и создания соответствующего инструментального программного обеспечения, то есть предоставляют удобные средства для разработки проекта DFT от начала до конца.

Фирма Cadence для решения перечисленных задач разработала серию продуктов: Encounter DFT Architect, Encounter True-Time ATPG, Encounter Diagnostics.

Продукты Encounter DFT Architect и Encounter Diagnostics предоставляют средства автоматизированного проектирования встроенных блоков контроля и внутрисхемной отладки: помогают встраивать в проект схемы самопроверки (схемы built-in self- test, BIST), цепи JTAG и др.

Продукты Encounter True-Time ATPG (Automatic Test Pattern Generation) и Modus Test Solution предназначены для проектирования массива тестовых воздействий для цепей BIST.

Соответствующие средства разработки DFT фирмы Synopsys представлены в продуктах DFTMAX, SpyGlass DFT ADV.

Mentor Graphics для данных целей разработала САПР Certus Silicon Debug и Tessent.

Базовой технологией анализа принципиальных схем является SPICE моделирование. Первый SPICE (Simulation Program with Integrated Circuit Emphasis) симулятор были разработан в Калифорнийском университете в Беркли в начале 1970-х гг. В дальнейшем на его основе были созданы другие программы. Необходимость моделирования вызвана тем, что создать дешевый прототип интегральной схемы практически невозможно. Для этого ее надо фактически изготовить, что требует значительных ресурсов. SPICE позволяет промоделировать работу схемы в разных условиях и выявить потенциальные проблемы до того, как она будет выпущена. SPICE содержит модели различных электронных компонентов и элементов схем – транзисторов, резисторов, конденсаторов, источников напряжения, усилителей и т.п. С помощью SPICE моделирования можно выполнить анализ схемы несколькими различными способами: построить передаточную функцию логического элемента, произвести анализ схемы по постоянному и переменному току и др.

К другим видам анализа относятся структурный анализ схемы на предмет целостности цепей питания, наличия паразитных цепей, а также проверку соответствия логической схемы её топологии (Layout Versus Schematic, LVS – анализ).

Для анализа принципиальной схемы на физическом уровне, а также результатов её размещения и трассировки, то есть топологии, фирма Cadence предлагает продукты Voltus IC Power Integrity Solution, Tempus Signoff Solution, Quantus QRC Extraction Solution.

С помощью продукта Voltus проверяется целостность цепей питания схемы, Tempus позволяет проводить анализ временных характеристик проекта, Quantus предоставляет средства автоматизированного анализа паразитных параметров топологии.

Анализ свойств и характеристик разработанных топологий физических ячеек возможно выполнять с помощью среды моделирования Virtuoso Multi-Mode Simulation (MMSIM). В данную среду интегрированы инструменты Spectre Platform, позволяющие выполнять SPICE моделирование аналоговых компонентов.

Фирма Synopsys для SPICE моделирования проекта схемы предоставляет продукт PrimeTime. Для моделирования отдельный частей схемы с более высокой точностью разработан программный продукт NanoTime. Для экстракции паразитных цепей предлагается использовать StarRC. Эти и другие инструменты для анализа проекта принципиальной схемы и её топологии Synopsys интегрировала в среду Custom Designer Environment.

Mentor Graphics для выполнения SPICE моделирования предлагает использовать Eldo. Остальные виды анализа возможно производить с помощью интегрированной среды Calibre.

Все перечисленные САПР, ассистирующие процесс анализа результатов технической реализации, представлены в таблице (Таблица 4).

Таблица 4. САПР фирм-лидеров рынка, ассистирующие процесс анализа результатов технической реализации ВС.

Фирма-производитель САПР	Cadence	Synopsys	Mentor Graphics
Средства разработки проекта DFT	Encounter DFT Architect, Encounter True-Time ATPG, Encounter Diagnostics	DFTMAX, SpyGlass DFT ADV	Certus Silicon Debug, Tessent
Средства моделирования принципиальных схем (SPICE-моделирование)	Spectre Platform	PrimeTime, NanoTime	Eldo
Статический анализ проекта топологии схемы (оценка энергопотребления, извлечение паразитных цепей и др.)	Voltus IC Power Integrity Solution, Tempus Signoff Solution, Quantus QRC Extraction Solution	StarRC, Custom Designer Environment	Calibre

2.4 Средства управления процессом проектирования.

Фирмы Cadence, Synopsys и Mentor Graphics предлагают не только средства осуществления проектных процедур, но и средства управления жизненным циклом проекта, то есть реализуют в своих САПР так называемые CALS – технологии (Continuous Acquisition and Lifecycle Support).

В данном контексте фирмы Synopsys и Cadence развивают методологию проектирования, управляемую метриками или MDV-методологию (Metric Driven Verification).

Основная концепция методологии MDV состоит в выделении определенных метрик проекта и требований к ним до начала работ, а также составления плана проверки данных требований на разных стадиях реализации. План составляется так, чтобы каждый его пункт можно было

проверить с помощью написания определенных тестов, исполняемого кода для системной модели устройства, моделей отдельных функциональных узлов и HDL-моделей конкретных аппаратных блоков.

Контроль развития проекта осуществляется по факту удовлетворения количественных показателей сформулированного списка метрик – требований проекта. Диапазон используемых техник контроля метрик не ограничен. Важно, чтобы они были изначально формализованы разработчиком. Так в рамках MDV могут быть использованы методы оценки функционального покрытия (Coverage Driven Verification, CDV), методы направленного тестирования (Direct Tests, DT), тестирование с использованием случайных воздействий (Constrained Random Verification, CRV), методы верификации на основе формул утверждений (Assertion Based Verification, ABV) и др.

Cadence реализует идеи MDV (Рис. 27) в наборе инструментов Incisive Enterprise Manager и Incisive Design Team Manager. Данный набор инструментов предоставляют среду взаимодействия разработчиков в рамках технологии клиент-сервер. На сервере сохраняется база данных метрик проекта (характеристик, которые могут быть численно измерены и проверены с помощью инструментов Cadence) и исполняемый план проекта vPlan. Разработчики работают по проекту, проводят тестирование и отладку на клиентских машинах. Тестовые окружения клиентов связаны с исполняемым планом vPlan. Таким образом, по ходу работы над проектом отдельные характеристики измеряются и отправляются на сервер. Подключаясь к серверу, каждый разработчик или руководитель проекта может просматривать ход работ в терминах удовлетворения определенных требований к различным метрикам: в процентах отображается функциональное покрытие, пропускная способность определенных блоков, поддержка определенных функций и пр.

По оценкам фирмы Cadence такой способ организации работ позволяет увеличить скорость разработки и верификации проекта до 60% по сравнению со стандартными методами без использования единого исполняемого плана. Хотя эффективность использования данных инструментов на первых проектах может быть снижена из-за необходимости преодоления определенного порога вхождения. Только с третьего проекта возможно увидеть рост продуктивности.

В настоящее время с исполняемым планом проекта интегрированы средства контроля качества системного и частично логического уровней.

Реализацию подобной технологии можно найти и в инструментах Synopsys. Технология MDV поддержана программным пакетом VCS Planning and Management Package. Фирма Synopsys также предоставляет средства упаковки и подготовку к выпуску проекта системы со всем

набором проектных файлов – данных проектирования. Данные инструменты реализованы в программном продукте GenSys.

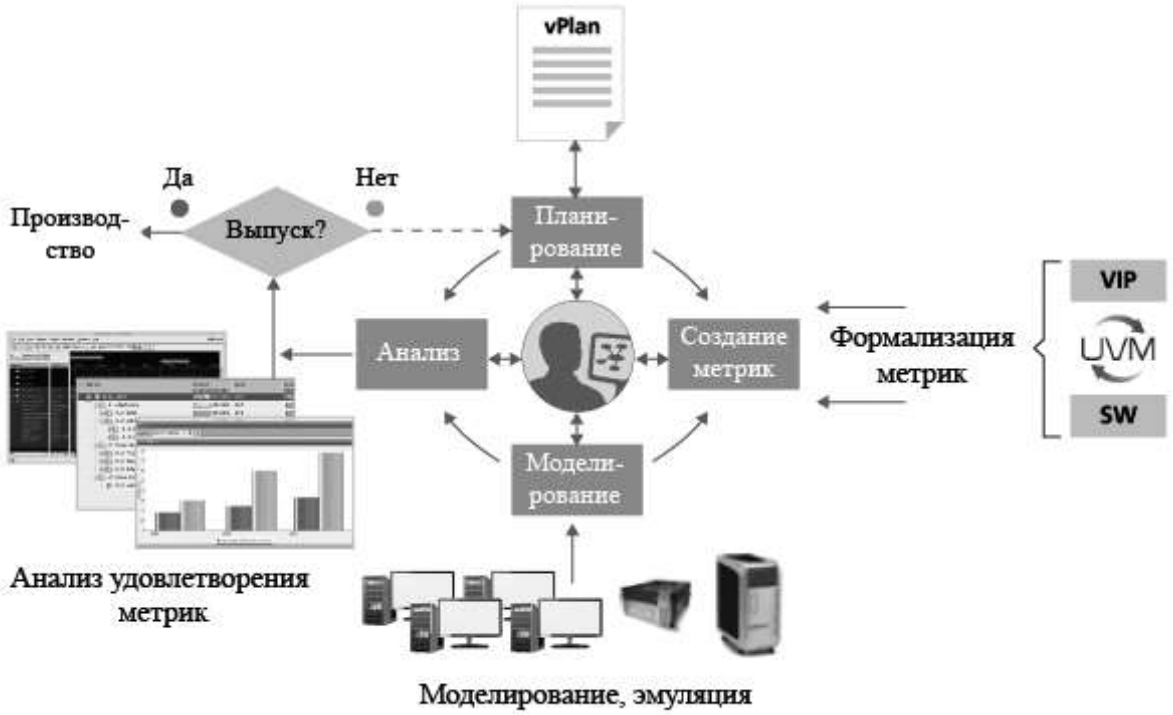


Рис. 27. Схема реализации технологии MDV в САПР фирмы Cadence.

Для интерактивного анализа и трассировки требований фирма Mentor Graphics предоставляет САПР ReqTracer. ReqTracer связывает между собой процесс задания начальной спецификации, реализации и верификации, структурирует все имеющиеся по проекту данные в виде требований и зависимостей между ними. Предусмотрена интеграция со всеми основными инструментами разработки от Mentor Graphics, поддерживается импорт исходных данных из сторонних программ.

Также Mentor Graphics предоставляет средства управления данными проекта в САПР Context SDM. Context SDM предоставляет многопользовательский доступ к данным проектирования, позволяет отслеживать результаты каждого этапа, контролировать взаимосвязи данных проектирования, полученные различными командами разработчиков.

2.5 Тенденции развития современных САПР

В настоящее время все большую формализацию получают высокоуровневые процессы проектирования. Инновации в данной сфере, как правило, начинаются с создания ряда стартапов и стремительного развития наиболее успешных из них. После того как их идеи становятся заметными не только в академических кругах, но и для индустрии,

результаты их труда покупают крупные игроки рынка и интегрируют их в свои маршруты проектирования.

Сегодня возможно проследить определенную тенденцию в развитии инструментов САПР каждой компании.

Компания Cadence в последнее время вкладывает значительные средства в развитие САПР высокоуровневого синтеза. До 2014 года Cadence развивала идеи синтеза HDL-моделей блоков из описания их функциональности на языках C/C++/SystemC в своем САПР C-to-Silicon Compiler. В феврале 2014 года Cadence приобрела компанию Forte Design Systems вместе с их наработками в области высокоуровневого синтеза. Затем Cadence сделала следующий шаг и объединила преимущества C-to-Silicon Compiler и Cynthesizer от Forte, выпустив новый продукт Stratus HLS.

Часть средств Cadence вкладывает в совершенствование технологий логического и физического синтеза. Недавно выпущены продукты Genus для логического синтеза, Innovus для физического синтеза (размещения и трассировки), характеризующиеся использованием высокопараллельных алгоритмов для выполнения соответствующих задач. Все стадии проектирования файла топологии контролируются с помощью отдельного набора инструментов, интегрированных в среду Encounter Conformal ECO Designer.

В области высокоуровневого проектирования компания Synopsys направляет свои усилия в развитие средств проектирования с использованием ADL-языков. Synopsys сначала сконцентрировала свое внимание на языке LISA и реализовала его поддержку в среде проектирования архитектуры программируемых процессоров Processor Designer. Далее Synopsys расширила возможности своих инструментов для решения задач проектирования многопроцессорных систем. В данном контексте внимание Synopsys было обращено в сторону языка nML. Поддержка данного ADL-языка была реализована в ASIP Designer и MP Designer.

Mentor Graphics специализируется на развитии технологии крупноблочного проектирования вычислительной системы на базе виртуальных компонентов, а также средств статического и динамического анализа проекта. Основные средства вкладываются в совершенствования соответствующих САПР: Vista и Questa. Mentor Graphics не упускает из внимания технологии высокоуровневого синтеза. В 2015 году она приобрела для этих целей Calypto Design Systems вместе с САПР Catapult C.

Компания Mentor Graphics также думает об удобстве и доступности своих инструментов. Для стадии проектирования HDL-моделей разработаны инструменты визуального проектирования, позволяющие снизить вероятность внесения ошибок при реализации типовых блоков. Также Mentor Graphics предоставляет возможность использовать свои инструменты академическому сообществу. Для этого, в отличии от Synopsys и Cadence, нет необходимости оформления официальных документов между ВУЗом и компанией. На открытых условиях, хотя и с некоторым набором ограничений, распространяется симулятор ModelSim, позволяющий моделировать исполняемые модели проекта как на системном уровне, используя языки SystemC и SystemVerilog, так и отлаживать потактовые модели устройств, описанные на языках VHDL или Verilog HDL.

Одновременно с развитием непосредственно средств проектирования каждая компания-разработчик САПР в последнее время уделяет все большее внимание решению задач управления процессом проектирования как таковым. Это выразилось в создании специальных средств отслеживания выполнения установленных проектных требований и развитием методологии MDV.

ЛИТЕРАТУРА

1. Teich J. Hardware/Software Codesign: The Past, the Present, and Predicting the Future // Proc. IEEE. 2012. Vol. 100, № Special Centennial Issue. P. 1411–1430.
2. Schaumont P.R. A Practical Introduction to Hardware/Software Codesign // Media. 2010. 403 p.
3. Prakash S., Parker A.C. SOS: Synthesis of application-specific heterogeneous multiprocessor systems // J. Parallel Distrib. Comput. 1992. Vol. 16, № 4. P. 338–351.
4. Gupta Giovanni De Micheli R.K. Hardware-Software Cosynthesis for Digital Systems // IEEE Des. Test Comput. 1993. Vol. 10, № 3. P. 29–41.
5. Ernst R., Henkel J., Benner T. Hardware-Software Cosynthesis for Microcontrollers // IEEE Des. Test Comput. 1993. Vol. 10, № 4. P. 64–75.
6. Liem C. et al. System-on-a-chip cosimulation and compilation // IEEE Des. Test Comput. 1997. Vol. 14, № 2. P. 16–24.
7. Zivojnovic V., Meyr H. Compiled HW/SW co-simulation // Des. Autom. Conf. Proc. 1996, 33rd. 1996. P. 690–695.
8. Ernst R. Codesign of embedded systems: Status and trends // IEEE Des. Test Comput. 1998. Vol. 15, № 2. P. 45–54.
9. Li Y.-T.S., Malik S., Wolfe A. Performance estimation of embedded software with instruction cache modeling // ACM Trans. Des. Autom. Electron. Syst. 1999. Vol. 4, № 3. P. 257–279.
10. Thiele L., Chakraborty S., Naedele M. Real-time calculus for scheduling hard real-time systems // Circuits Syst. 2000. Proceedings. ISCAS 2000 Geneva. 2000 IEEE Int. Symp. 2000. Vol. 4. P. 101–104 vol.4.
11. Henia R. et al. System level performance analysis – the SymTA/S approach // Comput. Digit. Tech. IEE Proc. 2005. Vol. 152, № 2. P. 148–166.
12. Blickle T., Teich J., Thiele L. System-level synthesis using evolutionary algorithms // Des. Autom. Embed. Syst. 1998. Vol. 40, № 1. P. 1–40.
13. Chou P.H., Ortega R.B., Borriello G. The Chinook hardware/software co-synthesis system // Syst. Synth. 1995., Proc. Eighth Int. Symp. Cannes: IEEE, 1995. P. 22–27.
14. Rompaey K. Van et al. CoWare-a design environment for heterogeneous hardware/software\nsystems // Proc. EURO-DAC ’96. Eur. Des. Autom. Conf. with EURO-VHDL '96 Exhib. 1996. P. 252–257.
15. Bailey B., Martin G., Piziali A. ESL Design and Verification // ESL Des. Verif. 2007. 113-138 p.
16. Рабоволюк А.В. Обзор решений Mentor Graphics в области проектирования СБИС. Текущее состояние и перспективы развития. Москва: МЭС-2012, 2012.
17. Vista Architect System Level Design Solution for Performance and Power. Mentor Graphics, 2009.

18. Matalon S. Vista virtual prototyping. Mentor Graphics, 2015.
19. Fornaciari W. et al. A first step towards Hw/Sw partitioning of UML specifications // Proc. -Design, Autom. Test Eur. DATE. 2003. P. 668–673.
20. Gajski D.D. System-Level Design Methodology. Irvine: Center for Embedded Computer Systems, University of California, 2003.

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра ВТ Университета ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Традиционно основной упор в подготовке специалистов на кафедре делается на фундаментальную базовую подготовку в рамках общепрофессиональных и специальных дисциплин, охватывающих наиболее важные разделы вычислительной техники.

Кафедра является одной из крупнейших в университете. Учебными курсами и научно-исследовательскими работами руководят 8 профессоров и 16 доцентов. На кафедре обучаются более 500 студентов и 30 аспирантов.

Кафедра имеет собственные компьютерные классы и специализированные исследовательские лаборатории, оснащенные современной вычислительной и оргтехникой, уникальным инструментальным и технологическим оборудованием, измерительными приборами и программным обеспечением.

В 2007-2008 гг. коллективом кафедры была успешно реализована инновационная образовательная программа СПбНИУ ИТМО по научно-образовательному направлению «Встроенные вычислительные системы».

Начиная с 2009 года кафедра вычислительной техники является активным участником реализации программы развития национального исследовательского университета НИУ ИТМО, вошла в состав крупнейшего в НИУ ИТМО научно-исследовательского центра «Интеллектуальные системы управления и обработки информации».

Начиная с 2013 года кафедра ВТ принимает активное участие в реализации мероприятий программы повышения международной конкурентоспособности Университета ИТМО «5 в 100».

Быковский Сергей Вячеславович

Горбачев Ярослав Георгиевич

Ключев Аркадий Олегович

Пенской Александр Владимирович

Платунов Алексей Евгеньевич

Сопряжённое проектирование встраиваемых систем (Hardware/Software Co-Design)

Часть 2

Учебное пособие

В авторской редакции

Редакционно-издательский отдел Университета ИТМО

Зав. РИО Н.Ф. Гусарова

Подписано к печати

Заказ №

Тираж

Отпечатано на ризографе

Редакционно-издательский отдел

Университета ИТМО

197101, Санкт-Петербург, Кронверкский пр., 49