Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА/GRADUATION THESIS

Верификация системы автоматического проектирования специализированных процессоров

Автор/ Author
Костючик Артем Геннадьевич
Направленность (профиль) образовательной программы/Major
Компьютерные системы и технологии 2019
Квалификация/ Degree level
Магистр
Руководитель ВКР/ Thesis supervisor
Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")
Группа/Group
P42191
Факультет/институт/кластер/ Faculty/Institute/Cluster
факультет программной инженерии и компьютерной техники
Направление подготовки/ Subject area
09.04.01 Информатика и вычислительная техника

Обучающийся/Student

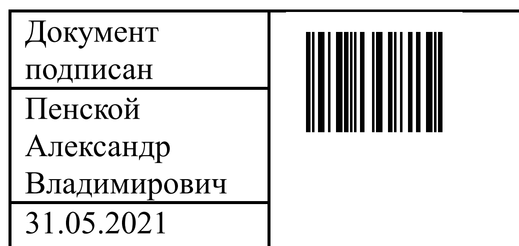
Документ подписан
Костючик Артем Геннадьевич
31.05.2021

Костючик Артем Геннадьевич

(эл. подпись/ signature)

(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor


Документ подписан
Пенской Александр Владимирович
31.05.2021

Пенской Александр Владимирович

(эл. подпись/ signature)

(Фамилия И.О./ name and surname)

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ / OBJECTIVES FOR A GRADUATION THESIS

Обучающийся / Student Костючик Артем Геннадьевич
Группа/Group P42191
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники
Квалификация/ Degree level Магистр
Направление подготовки/ Subject area 09.04.01 Информатика и вычислительная техника
Направленность (профиль) образовательной программы/Major Компьютерные системы и технологии 2019
Специализация/ Specialization Встраиваемые и киберфизические системы
Тема ВКР/ Thesis topic Верификация системы автоматического проектирования специализированных процессоров
Руководитель ВКР/ Thesis supervisor Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Срок сдачи студентом законченной работы до / Deadline for submission of complete thesis 31.05.2021

Техническое задание и исходные данные к работе/ Requirements and premise for the thesis
1) Необходимо проанализировать существующие методы и средства для верификации сложного программного обеспечения. Особый акцент необходимо сделать на методах верификации САПР. 2) На основе анализа выбрать готовые инструменты для верификации САПР специализированных вычислителей или разработать технические требования и реализовать собственные специализированные средства верификации. 3) Применить выбранные или разработанные средства верификации для верификации САПР специализированных вычислителей. 4) Для оценки результатов тестирования и сравнения с используемыми на данный момент методами необходимо определить критерии качества тестирования. 5) Оценить эффект от применения методов и средств верификации к САПР специализированных вычислителей с использованием ранее определенных критериев.

Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)/ Content of the thesis (list of key issues)
1) Анализ методов и средств верификации САПР.

2) Разработка методов и подходов в верификации САПР.

3) Реализация и применение методов и подходов в верификации САПР.

4) Анализ полученных результатов.

Перечень графического материала (с указанием обязательного материала) / List of graphic materials (with a list of required material)
Презентация по проделанной работе (в формате Microsoft PowerPoint)

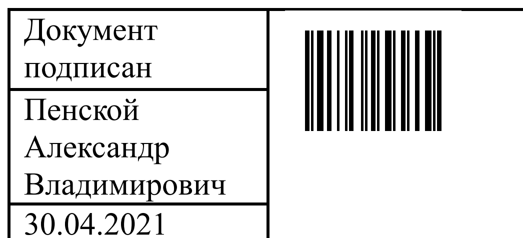
Исходные материалы и пособия / Source materials and publications

1) Prohorov D., Penskoi A. Verification of the CAD System for an Application-Specific Processor by Property-Based Testing // 2020 9th Mediterranean Conference on Embedded Computing, MECO 2020. Institute of Electrical and Electronics Engineers Inc., 2020.
2) Lee, E. A., & Seshia S.A. Introduction to Embedded Systems. A Cyber-Physical Systems Approach. Second Edition // Studies in Systems, Decision and Control. 2017. Vol. 195.
3) Penskoi A. et al. Hybrid NISC/TTA high-level synthesis tool // International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM. 2018. Vol. 18, № 2.1.
4) Piccolboni L., Di Guglielmo G., Carloni L.P. KAIROS: Incremental verification in high-level synthesis through latency-insensitive design // Proceedings of the 19th Conference on Formal Methods in Computer-Aided Design, FMCAD 2019. 2019.
5) Dossis M. High-level Synthesis Integrated Verification // Eng. Technol. Appl. Sci. Res. 2015. Vol. 5, № 5.
6) Mami S., Lahbib Y., Mami A. A New HLS Allocation Algorithm for Efficient DSP Utilization in FPGAs // J. Signal Process. Syst. 2020. Vol. 92, № 2.
7) Penskoi A.V. et al. High-level synthesis system based on hybrid reconfigurable microarchitecture // Sci. Tech. J. Inf. Technol. Mech. Opt. 2019. Vol. 19, № 2.
8) Pelcat M. et al. Design productivity of a high level synthesis compiler versus HDL // Proceedings - 2016 16th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016. 2017.
9) Harris C.B., Harris I.G. Generating formal hardware verification properties from Natural Language documentation // Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing, IEEE ICSC 2015. 2015.
10) Jozwiak L. et al. ASAM: Automatic architecture synthesis and application mapping // Microprocess. Microsyst. 2013. Vol. 37, № 8 PARTC.

Дата выдачи задания/ Objectives issued on 30.04.2021

СОГЛАСОВАНО / AGREED:

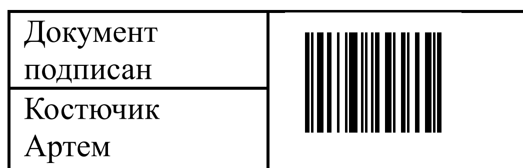
Руководитель ВКР/
Thesis supervisor



(эл. подпись)

Пенской
Александр
Владимирович

Задание принял к исполнению/ Objectives assumed by

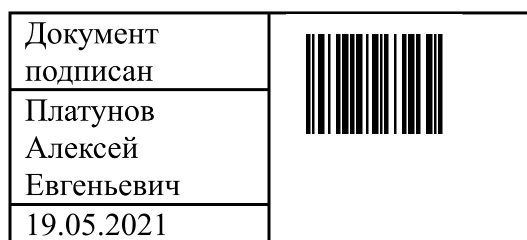


Костючик

	Геннадьевич		Артем Геннадьевич
	30.04.2021		

(эл. подпись)

Руководитель ОП/ Head of educational program

Документ подписан	
Платунов Алексей Евгеньевич	
19.05.2021	

(эл. подпись)

Платунов Алексей Евгеньевич

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

АННОТАЦИЯ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ / SUMMARY OF A GRADUATION THESIS

Обучающийся/ Student
Костючик Артем Геннадьевич
Наименование темы ВКР / Title of the thesis
Верификация системы автоматического проектирования специализированных процессоров
Наименование организации, где выполнена ВКР/ Name of organization
Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ/ DESCRIPTION OF THE GRADUATION THESIS

1. Цель исследования / Research objective
Сокращение трудозатрат на верификацию САПР и повышение качества тестирования за счет применения встраиваемых специализированных языков и проверки целостности результатов процесса синтеза.
2. Задачи, решаемые в ВКР / Research tasks
1) Необходимо проанализировать существующие методы и средства для верификации сложного программного обеспечения. Особый акцент необходимо сделать на методах верификации САПР. 2) На основе анализа выбрать готовые инструменты для верификации САПР специализированных вычислителей или разработать технические требования и реализовать собственные специализированные средства верификации. 3) Применить выбранные или разработанные средства верификации для верификации САПР специализированных вычислителей. 4) Для оценки результатов тестирования и сравнения с используемыми на данный момент методами необходимо определить критерии качества тестирования. 5) Оценить эффект от применения методов и средств верификации к САПР специализированных вычислителей с использованием ранее определенных критериев.
3. Краткая характеристика полученных результатов / Short summary of results/conclusions
Осуществлен обзор методов и средств верификации САПР, проанализированны готовые инструменты для верификации САПР, разработаны технические требования на разработку специализированного языка, выделены критерии для оценки результатов тестирования, реализован встроенный специализированный язык программирования для верификации моделей целевой вычислительной системы, реализован метод проверки целостности процесса синтеза, выявлены и исправлены ошибки в нескольких вычислительных блоках, увеличено тестовое покрытие, расширено количество тестов и тестируемых модулей, а также дана оценка эффективности полученного решения.
4. Наличие публикаций по теме выпускной работы/ Have you produced any

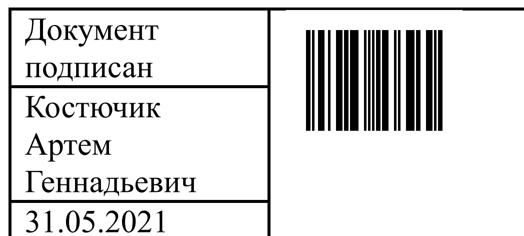
publications on the topic of the thesis

5. Наличие выступлений на конференциях по теме выпускной работы/ Have you produced any conference reports on the topic of the thesis

6. Полученные гранты, при выполнении работы/ Grants received while working on the thesis

7. Дополнительные сведения/ Additional information

Обучающийся/Student


Документ подписан
Костючик Артем Геннадьевич
31.05.2021

(эл. подпись/ signature)

Костючик Артем Геннадьевич

(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor

Документ подписан
Пенской Александр Владимирович
31.05.2021

(эл. подпись/ signature)

Пенской Александр Владимирович

(Фамилия И.О./ name and surname)

Содержание

Определения, обозначения и сокращения ... 8
Введение ... 10
1 Анализ методов и средств верификации САПР ... 13
1.1 Методы верификации сложных систем ... 13
1.2 Вычислительная платформа NITTA ... 24
1.3 Верификация в проекте NITTA ... 28
1.4 Постановка задачи ... 33
2 Разработка методов и подходов в верификации САПР ... 35
2.1 Выбор инструментов для верификации САПР ... 35
2.2 Разработка предметно-ориентированного языка ... 39
2.3 Разработка модуля проверки целостности процесса синтеза ... 43
2.4 Определение критериев качества тестирования ... 45
2.5 Вывод ... 45
3 Применение методов и подходов в верификации САПР ... 46
3.1 Реализация предметно-ориентированного языка ... 46
3.2 Реализация модуля проверки целостности процесса синтеза ... 53
3.3 Анализ полученных результатов ... 57
3.4 Вывод ... 60
Заключение ... 61
Список использованных источников ... 62
Приложение А. Исходный код модуля DSL (обязательное) ... 65
Приложение Б. Исходный код модуля проверки целостности (обязательное) ... 71
Приложение В. Исходный код реализованных тестов (обязательное) ... 74
Приложение Г. Исходный код примера doctest (необязательное) ... 77

Определения, обозначения и сокращения

В настоящей пояснительной записке применяются следующие термины:

ASIC (Application-Specific Integrated Circuit) – интегральная схема, специализированная для решения конкретной задачи.

ASIP (Application-Specific Instruction Set Processor) – проблемно-ориентированный процессор.

BDD (Behavior Driven Development) – методология разработки на основе поведения системы.

CAD (Computer Aided Design) – САПР (Система автоматизированного проектирования).

CPS (Cyber-Physical System) – киберфизическая система.

DSL (Domain Specific Language) – предметно-ориентированный язык.

NITTA (Not Instruction Triggered Transport Architecture) – специализированный реконфигурируемый вычислитель для которого ведется разработка средств верификации.

NISC (No Instruction Set Computing) – тип процессорной архитектуры с вычислениями без набора команд.

TTA (Transport Triggered Architecture) – тип процессорной архитектуры где программа напрямую управляет внутренней шиной данных процессора.

CISC (Complex Instruction Set Computing) – тип процессорной архитектуры с полным набором команд.

RISC (Reduced Instruction Set Computing) – тип процессорной архитектуры с сокращенным набором команд.

FPGA (Field-Programmable Gate Array) – программируемая логическая интегральная схема (ПЛИС).

GHC (Glasgow Haskell Compiler) – компилятор языка Haskell.

Lua – высокоуровневый язык программирования, использующийся в качестве входного языка в системе NITTA.

HLS (High Level Synthesis) – высокоуровневый синтез.

HDL (Hardware Description Language) – язык описания аппаратуры. Применяется для описания структуры и поведения электронных схем.

RTL (Register Transfer Level) – уровень регистровых передач.

PBT (Property Based Testing) – свойство-ориентированное тестирование.

Coq – интерактивное программное средство доказательства теорем, также применяется для формальной верификации программных комплексов.

CI (Continuous Integration) – практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта.

ПО – Программное обеспечение.

Введение

Постоянно растущий разрыв между программной и аппаратной областью вычислительной техники, а также высокая сложность разработки на языках регистровых передач привели к созданию систем высокоуровневого синтеза (HLS) [1]. Они позволяют преобразовать код с языка высокого уровня в схему уровня регистровых передач (RTL) на языке описания аппаратуры (HDL). Процесс синтеза представляет собой многоступенчатую последовательность, которая состоит из нескольких взаимосвязанных подзадач, таких как: компиляция, размещение, планирование, привязка и генерация управляющих сигналов [2]. Инструменты высокоуровневого синтеза обладают высокой сложностью, и чаще всего состоят из десятка тысяч строк кода, соответственно программа такого объёма с высокой долей вероятности содержит ошибки, которые могут встретиться как в самой системе, так и в синтезируемой схеме. Если неполадка выявляется на поздних этапах, таких как изготовление или крупномасштабное производство, то это приводит к многомилионным потерям.

Таким образом, верификация САПР является трудной задачей из-за сложности алгоритмов, высокой вариативности входных данных и большого количества параметров тестирования. Важность данной задачи обусловлена высокой стоимостью исправления ошибок на более поздних этапах проектирования.

NITTA – это САПР, предназначенный для синтеза реконфигурируемых специализированных процессоров реального времени [3]. По большинству критериев, его можно отнести к средству высокоуровневого синтеза третьего поколения [4], одной из особенностей которого является модельно-ориентированная инженерия, что позволяет проектировщикам сосредоточиться на решаемой задаче, а не на средствах её реализации.

Ранее приведенная проблема верификации решается в NITTA [5] с помощью различных методов, одним из которых является – тестирование на основе проверки свойств (property based testing, PBT) [3].

Свойство-ориентированное тестирование – в отличии от всем привычного модульного тестирования проверяет свойства тестируемого объекта, а не соответствие входных данных выходным, например, таким свойством может являться – коммутативность. Для проверки каждый раз генерируется набор случайных данных по заданным проверяемым свойствам, благодаря чему обеспечивается высокий уровень тестового покрытия.

Применение свойство-ориентированных тестов не снижает сложности, связанной с поддержкой старых и написанием новых тестов. В методологии разработки на основе поведения (Behavior driven development, BDD) использование встроенного специализированного языка позволяет писать более простые и понятные тесты [6]. По этой причине было решено проанализировать BDD библиотеки с вариантом написания собственного встроенного предметно-ориентированного языка. Критериями выбора являются: снижение сложности и повышение удобочитаемости тестов.

Процесс синтеза в NITTA является критической частью САПР. Ошибки в нем могут привести к скрытым неполадкам синтезируемого устройства. Пользователь не будет удовлетворен такой системой, что негативно повлияет на качество САПР. Кроме того, если неполадка в синтезируемом устройстве будет обнаружена поздно, то это приведет к высокой стоимости её исправления. Необходимо выявлять подобные проблемы путем проверки межуровневых связей в модели вычислительного блока.

Таким образом, скомбинировав вышеприведенные подходы можно значительно упростить разработку и поддержку тестов, а также повысить их качество, где под качеством имеется в виду их ценность для заинтересованных сторон, в данном случае это разработчики.

Целью настоящей работы является сокращение трудозатрат на верификацию САПР и повышение качества тестирования за счет применения встраиваемых специализированных языков и проверки целостности результатов процесса синтеза.

Для достижения поставленной цели были выделены следующие задачи:

1. Необходимо проанализировать существующие методы и средства для верификации сложного программного обеспечения. Особый акцент необходимо сделать на методах верификации САПР.
2. На основе анализа выбрать готовые инструменты для верификации САПР специализированных вычислителей или разработать технические требования и реализовать собственные специализированные средства верификации.
3. Применить выбранные или разработанные средства верификации для верификации САПР специализированных вычислителей.
4. Для оценки результатов тестирования и сравнения с используемыми на данный момент методами необходимо определить критерии качества тестирования.
5. Оценить эффект от применения методов и средств верификации к САПР специализированных вычислителей с использованием ранее определенных критериев.

1 Анализ методов и средств верификации САПР

Для понимания и выделения предметной области данной работы необходимо привести определения базовых понятий, таких как жизненный цикл, верификация, валидация. Кроме этого, будет приведена классификация существующих методов верификации САПР для обоснования выбора подходов к тестированию NITTA.

1.1 Методы верификации сложных систем

1.1.1 Жизненный цикл встраиваемых систем

Жизненный цикл – развитие системы, продукта, услуги, проекта или других изготовленных человеком объектов, начиная со стадии разработки концепции и заканчивая прекращением применения [7].

В связи с уникальностью процесса разработки любого программного продукта невозможно описать универсальный жизненный цикл, подходящий для любой существующей системы, по этой причине реализации жизненного цикла выражены в его моделях.

Модель жизненного цикла – структурная основа процессов и действий, относящихся к жизненному циклу которая также служит в качестве общего эталона для установления связей и понимания [7].

Существует множество моделей жизненного цикла, каждая из которых раскрывает свой потенциал в определенной области применения. Прежде, чем привести определения верификации и валидации, необходимо рассмотреть какое место они занимают в модели жизненного цикла, для этого рассмотрим V-model [8], приведенную на рисунке (Рисунок 1):

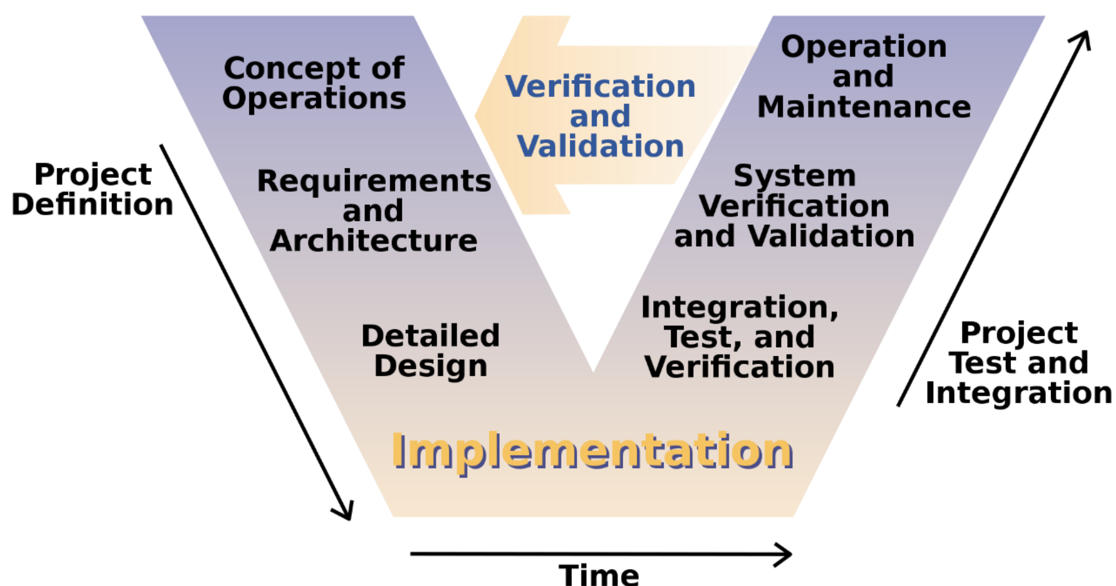


Рисунок 1 – V-model жизненного цикла [9]

В области встраиваемых систем этап валидации и верификации занимает значительную часть времени разработки и играет важную роль. Это обусловлено резко возрастающей стоимостью исправления ошибки при переходе на следующий этап разработки, например, при обнаружении ошибок после массового выпуска разрабатываемой продукции может приводить к отзыву всей партии, в свою очередь, это ведет к огромным убыткам. Как и верификация, так и валидация направлены на осуществление общей цели – контроль качества путем выявления дефектов[1] в программном и аппаратном обеспечении разрабатываемого продукта. Однако, для осуществления этой цели они используют разные методы.

---

[1] Дефект – изъян в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию, например неверный оператор или определение данных. Дефект, обнаруженный во время выполнения, может привести к отказам компонента или системы [10].

1.1.2 Верификация и валидация

Верификация – подтверждение экспертизой и предоставление объективных доказательств того, что указанные требования были выполнены [7]. Данный процесс носит предиктивный характер: если разработанный продукт будет произведен, то есть перейдет на следующий этап жизненного цикла, то он будет соответствовать проектным требованиям и спецификации.

Валидация – подтверждение путем исследования и предоставления объективных свидетельств того, что требования для конкретного предполагаемого использования или применения были выполнены [7]. Если верификация направлена на проверку соответствия техническим требованиям и спецификации, то валидация необходима для выявления соответствия удовлетворения требованиям для конкретного использования.

Для верификации и валидации существует множество техник, которые можно разделить на две большие группы, называемые статические и динамические. Существуют разные классификаций методов верификации, в данной работе в основном будут приведены методы, используемые в NITTA.

1.1.3 Классификация методов верификации

Рассмотрим статические и динамические методы верификации и приведем краткое описание каждого из методов, изображенных на рисунке (Рисунок 2).

Статические методы подразумевают тестирование полученных артефактов разработки программного обеспечения, например требований, дизайна или кода, без их выполнения [10].

Для динамической верификации необходимо непосредственно запускать тестируемую программу [10]. Отсюда следует необходимость наличия исполняемого артефакта разработки. Для рассматриваемого в работе САПР под артефактом подразумевается не только разрабатываемое программное обеспечение,

но также модели для симуляции и физические прототипы, генерируемые программным обеспечением.

Рисунок 2 – Классификация методов верификации

Из рисунка (Рисунок 2) следует, что статические и динамические методы включают несколько подгрупп, рассмотрим их по отдельности.

1.1.3.1 Статические методы верификации

Формальная верификация – процесс, при котором проверяется, удовлетворяет ли дизайн некоторым свойствам или спецификации [10]. Их делят на несколько групп:

– Методы, основанные на доказательстве теорем (дедуктивная верификация). Верификация путем доказательства теорем в подходящей логической системе. В данном направлении разработка введется с того момента, как Роберт Флойд и Энтони Хоар впервые формально постановили задачу верификации программ [11]. На сегодняшний день популярностью пользуются интерактивные программные средства доказательства теорем, например, к ним относят такой инструмент, как Coq. Проблема таких программ заключается в высокой сложности для программиста, не имеющего опыта в математике, в частности в области доказательств теорем. Кроме того, данная процедура не поддается полной автоматизации.

– Проверка на модели (Model checking) – техника автоматизированной проверки формальных свойств, выраженных с помощью темпоральной логики, на модели конечного автомата рассматриваемой системы [12]. Каждое свойство проверяется на валидность, таким образом при успешной проверке возвращается значение – истина. В случае обнаружение ошибки возвращается контрпример, то есть та последовательность действий в системе, которая приводит к невыполнению рассматриваемого свойства. Несмотря на полную автоматизацию процесса, выполнение верификации подразумевает наличие специалиста способного построить формальную модель системы.

– Статический анализ кода – многие языки программирования имеют строгую типизацию, что позволяет выявлять ошибки несоответствия типов на ранних этапах до компиляции. Кроме этого, существуют ана-

лизаторы статического кода, позволяющие проанализировать код на наличие распространенных уязвимостей и ошибок.

Неформальная верификация – процесс, использующий инструменты и методы, основанные на субъективных оценках разработчиков без строгого математического формализма. Неформальность не предписывает отсутствие структуры и методики проведения этого процесса.

Стандарт IEEE 1028–2008 регламентирует активности для рецензирования и аудита программного обеспечения. Следует отметить, что в первоисточнике используется слово «формальный» при описании одноименных методов и означает оно наличие строгой методики применения и структуры. Стоит отметить, что у многих методов расплывчатая граница между верификацией и валидацией, однако в рамках данного анализа это не играет роли. Рассмотрим неформальные методики согласно стандарту IEEE 1028–2008 подробнее:

- Рецензия менеджмента – оценка программного продукта или процесса его проектирования, которая выполняется руководством и необходима для отслеживания прогресса и определения: статуса плана и расписания, оценки эффективности управленческих подходов и подтверждения выполнения технических требований.
- Аудит – независимая экспертиза программного обеспечения или процесса его проектирования, осуществляемая независимыми экспертами, на соответствие спецификации, стандартам, контрактным договорённостям или другим критериям.
- Инспекция – визуальный осмотр артефактов разработки программного продукта для выявления аномалий программного обеспечения, включая ошибки и отклонения от стандартов спецификации. Проверка должна осуществляться группой лиц, обладающих достаточным опытом и экспертизой для выявления аномалий.

– Сквозная проверка – техника, согласно которой разработчик ведет членов команды и других заинтересованных лиц по программному обеспечению, при этом участники должны задавать вопросы, делать замечания: о возможных аномалиях, нарушениях стандартов и других проблемах.
– Техническое рецензирование – оценка программного продукта командой разработчиков, которая проверяет пригодность программного продукта для предполагаемого использования и выявляет ошибки, а также несоответствия со спецификациями и стандартами.

Рецензирование кода давно зарекомендовало себя, как полезный инструмент для выявления ошибок и поэтому используется в большинстве крупных компаний [13]. По этой причине на сегодняшний день большинство сервисов для хостинга IT-проектов (например, Github) имеют встроенные инструменты для его проведения. Таким образом, эти сервисы предписывают использование рецензирования при слиянии в основную ветку проекта.

1.1.3.2 Динамические методы верификации

Модульное тестирование – уровень тестирования, ориентированный на проверку отдельных компонентов, как аппаратных, так и программных [10]. Является наименьшей единицей тестирования, благодаря чему при возникновении ошибки довольно просто локализовать причину. По этой же причине тесты относительно просты в написании и поддержке.

Интеграционное тестирование – уровень тестирования, ориентированный на проверку взаимодействия между отдельными компонентами. Реализация и поддержка таких тестов сложнее модульных, как и процесс поиска причины возникновения ошибки.

Системное тестирование – уровень тестирования, ориентированный на проверку соблюдения системой, как единого целого, заданных требований. Являются самыми трудоемкими единицами тестирования для реализации. Иногда

данный уровень называется – тестирование пользовательского интерфейса, что обусловлено высокой популярностью веб-приложений.

Три вышеприведенных метода тестирования образуют, так называемую, пирамиду тестирования, приведенную на рисунке (Рисунок 3):

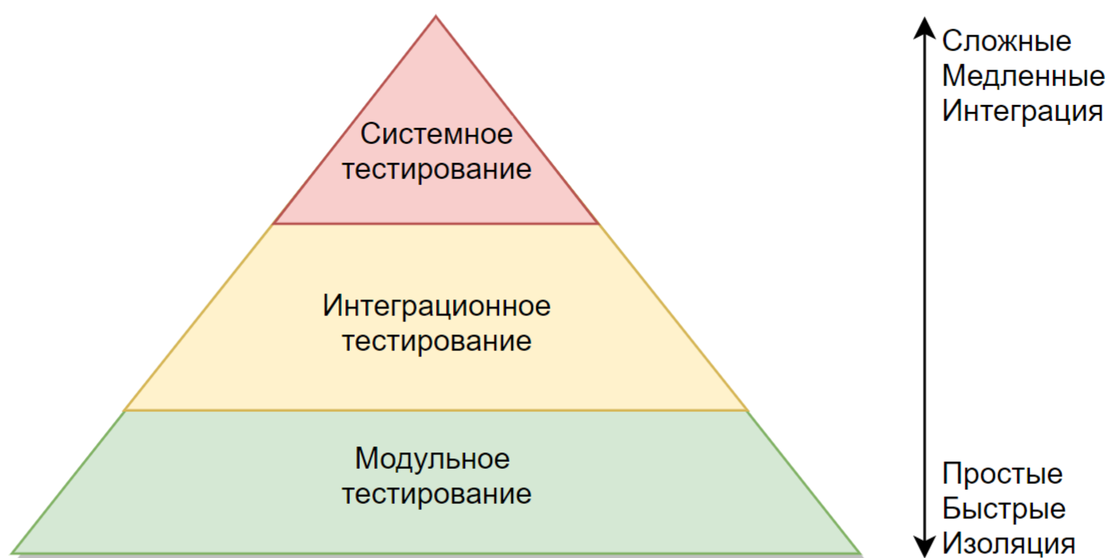


Рисунок 3 – Пирамида тестирования

Суть данной концепции заключается в классификации используемых тестов по группам с различной гранулярностью, что следует из их вышеприведенных названий и определений. Данное разбиение позволяет разделить тесты по сложности, скорости и степени охвата компонентов системы.

Свойство-ориентированное тестирование – метод тестирования, направленный на проверку выполнения ранее заданных свойств. Данный метод по смыслу схож с формальными методами, в частности с методом проверки на модели, с той разницей, что отсутствует необходимость в составлении отдельной формальной модели системы. Рассмотрим три концепции, использующихся в свойство-ориентированном тестировании:

– Свойство (property) – это инвариант тестируемой функции или подсистемы. Например, таким свойством может являться – неизменяемость результата арифметической операции при перестановке слагаемых или

коммутативность. Постоянность свойства позволяет генерировать входные данные. Для проверки свойства каждый раз генерируется набор случайных данных по заданным в генераторе правилам.

– Генератор (generator) – это механизм для генерации случайных значений. От реализации генератора во многом зависит успех в нахождении ошибки.
– Свертывание (shrinking) – если в тесте происходит ошибка и тестовые данные, созданного генератором, очень большие, то программисту будет сложно понять причину ошибки. Для решения этой проблемы тестовые данные «сужаются» перед выводом ошибки на экран, то есть начинается подбор значений меньшего размера, при которых ошибка воспроизводится.

Свойство-ориентированное тестирование позволяет обеспечить высокий уровень тестового покрытия благодаря генерации входных данных, стоит понимать, что покрытие зависит от используемого генератора и при его неудачной реализации не будет преимуществ перед другими методами тестирования.

Кроме самих методик тестирования существуют методологии для организации этого процесса, рассмотрим некоторые из них.

Непрерывная интеграция (CI) – техника разработки программного обеспечения, согласно которой команда разработчиков интегрирует свою работу на регулярной основе [14]. При осуществлении данного процесса каждый раз производится автоматическая верификация полученного проекта. Данная техника позволяет сократить количество ошибок и уменьшить время на интеграцию работы разных членов команды. По этой причине она имеет широкое распространение в индустрии, что привело к появлению множества инструментов для осуществления процесса автоматической сборки.

Разработка на основе тестов (TDD) – суть методологии заключается в написании теста до реализации задуманной функциональности [15]. Данная ме-

тодология имеет следующую последовательность: изначально пишется тест для планируемой функциональности, для этого необходимо знать входные и ожидаемые выходные данные. В первый раз полученный тест закончится неудачей, так как планируемая функциональность еще не реализована в системе. После написания кода тест начинает успешно выполняться, однако на этом процесс не останавливается и далее следует этап модернизации полученного решения. Схематично данный алгоритм представлен на рисунке (Рисунок 4):

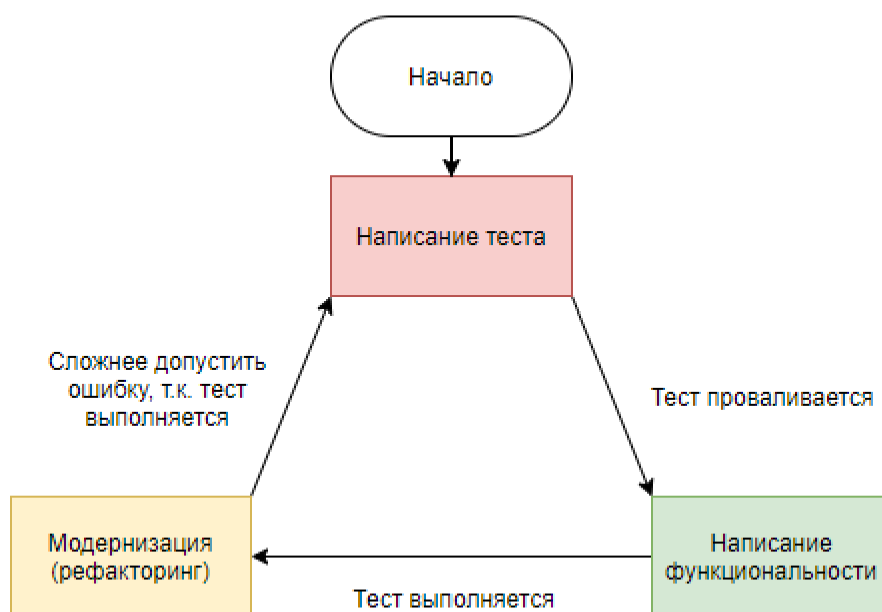


Рисунок 4 – Алгоритм разработки на основе тестов

Данный процесс позволяет:

– Проектировать функции и модули таким образом, чтобы их было легко протестировать.
– Обеспечить гранулярность системы, что достигается путем разграничения функциональности отдельных компонентов с помощью тестов.
– Упростить модернизацию системы благодаря наличию рабочих версий кода и тестов.

– Облегчить поддержку кода с помощью отдельного этапа модернизации, целью которого является повышение читаемости кода для упрощения его понимания.
– Расширить документацию. Сами тесты позволяют понять функциональность определенного компонента, таким образом, он выступает, как часть спецификации.

Таким образом, исходя из совокупности вышеприведенных особенностей методологии, можно заключить, что её использование приводит к повышению качества кода и дизайна системы.

Разработка на основе поведения – является расширением разработки на основе тестов с той разницей, что для описания тестовых сценариев используется специализированный язык [16]. Чаще всего синтаксис такого языка похож на естественные языки, благодаря чему достигается близость к спецификации и бизнес-процессам. Рассмотрим пример нотации языка из фреймворка Cucumber, приведенной на рисунке (Рисунок 5):

```
# Comment
@tag
Feature: Eating too many cucumbers may not be good for you

  Eating too much of anything may not be good for you.

  Scenario: Eating a few is no problem
    Given Alice is hungry
    When she eats 3 cucumbers
    Then she will be full
```

Рисунок 5 – Пример синтаксиса предметно-ориентированного языка из фреймворка Cucumber [17]

Из рисунка (Рисунок 5) видно, что в первую очередь дается описание тестируемого компонента, внутри которого описываются сценарии, представляющие собой набор действий для взаимодействия с ним. Каждый сценарий поделен на три шага:

– Блок «Given» содержит входные данные для сценария.

– Блок «When» запускает тестируемую функцию или компонент.

– Блок «Then» осуществляет проверку. Чаще всего полученные в результате работы теста выходные данные сверяются с ожидаемыми.

Использование особого синтаксиса для тестов позволяет обеспечить:

– Взаимодействие с заказчиком – тесты становятся понятны обычным пользователям, таким образом, уменьшается барьер между заказчиком и разработчиком, в свою очередь, это приводит к повышению качества конечного продукта.

– Вовлеченность пользователей – сами пользователи системы могут описывать их ожидания от разрабатываемой функциональности. Кроме того, из-за тесного сотрудничества, разработчики получают отзывы и пожелания пользователей на ранних этапах разработки программного обеспечения.

– Формализация тестов – благодаря использованию встраиваемого специализированного языка проще придерживаться единого стиля тестов.

Вышеприведенные особенности также порождают недостатки у данной методологии. Для взаимодействия с заказчиком потребуется больше времени, чем при использовании других методологий. К тому же продуктивность этих взаимодействий субъективна и в некоторых проектах может не привести к положительному результату. Таким образом, при выборе разработке на основе поведения необходимо учесть специфику проекта перед её применением.

1.2 Вычислительная платформа NITTA

1.2.1 Обзор и область применения

Проект NITTA основан на идее объединения концепций вычислений без набора команд (Non instruction set computing, NISC) и транспортно-

ориентированной архитектуры (Transport triggered architecture, TTA). Для понимания работы проекта в целом следует рассмотреть эти концепции отдельно.

NISC предполагает статически запланированный набор управляющих сигналов. Статически запланированный означает, что вся последовательность сигналов становится известной на этапе компиляции проекта. Эти сигналы являются простыми запросами к управляемым элементам без необходимости интерпретации команд как в архитектурах вычислитель с полным набором команд (Complex instruction set compute, CISC) и вычислитель с сокращенным набором команд (Reduced instruction set computer, RISC).

TTA предполагает наличие общей шины для передачи данных между присоединенными к ней вычислительными блоками (processor units, PU), что позволяет использовать единую структуру для всех операций, например, чтения и записи. Областями применения рассматриваемой системы являются:

– Разработка проблемно-ориентированных процессоров (Application-Specific Instruction-set Processor, ASIP) [18] .
– Разработка аппаратных программируемых ускорителей и сопроцессоров.
– Разработка проблемно-ориентированных программируемых интегральных схем специального назначения (Application-specific integrated circuit) [19].
– Разработка динамически реконфигурируемого софт-ядра для программируемых логических интегральных схем (Field-Programmable Gate Array, FPGA) [5].
– Расчет моделей системной динамики [20].
– Разработка киберфизических систем [21], основанных на алгоритмах адаптивного управления и искусственного интеллекта с высокими требованиями к задержкам и вычислительному объему, мощности и потребляемой площади.

1.2.2 Структура проекта

Одной из важнейших функций разрабатываемого САПР заключается в синтезе конечного устройства из высокоуровневого кода. Далее приведены основные компоненты, обеспечивающие функционирование системы:

– Алгоритм – представляет собой код, написанный на языке высокого уровня Lua или XMILE [18].
– Библиотека вычислительных блоков – содержит
– Микроархитектура конечного процессора – набор вычислительных блоков, а также их соединений с помощью шин.
– Пользовательский интерфейс – позволяет проследить за процессом синтеза, а также управлять им путем принятия решения.
– Модель конечной системы – в отличии от микроархитектуры представляет собой набор физических блоков, на которых будет работать синтезированная система.
– Метод синтеза – описание процесса синтеза, рассмотрим решаемые им задачи далее.
– Генератор конечной системы – предназначен для генерации низкоуровневой схемы из модели конечной системы.
– Подсистема верификации – генерирует тестовое окружение (testbench) для проверки конечной системы путем логической симуляции.

Рассмотрим подробнее какие задачи решает метод синтеза [22]:

– оптимизация прикладного алгоритма с учетом конфигурации процессора и уровня загрузки отдельных ресурсов;
– формирование конфигурации процессора (коммуникационная и управляющая инфраструктура, состав и схема подключения вычислительных блоков);
– планирование потока управления и потока данных;

– планирование вычислительного процесса с точки зрения передачи данных;

– управление внутренними ресурсами вычислительных блоков.

Вычислительный блок – это имитационная модель конкретного физического устройства, позволяющая объяснить синтезатору какие операции она может выполнять.

1.2.3 Используемые инструменты

Проект NITTA охватывает программную и аппаратные части, а цикл разработки следует от высокоуровневого кода до конечного устройства, поэтому в проекте используется широкое разнообразие инструментов и технологий. На рисунке (Рисунок 6) представлены основные компоненты NITTA [23]:

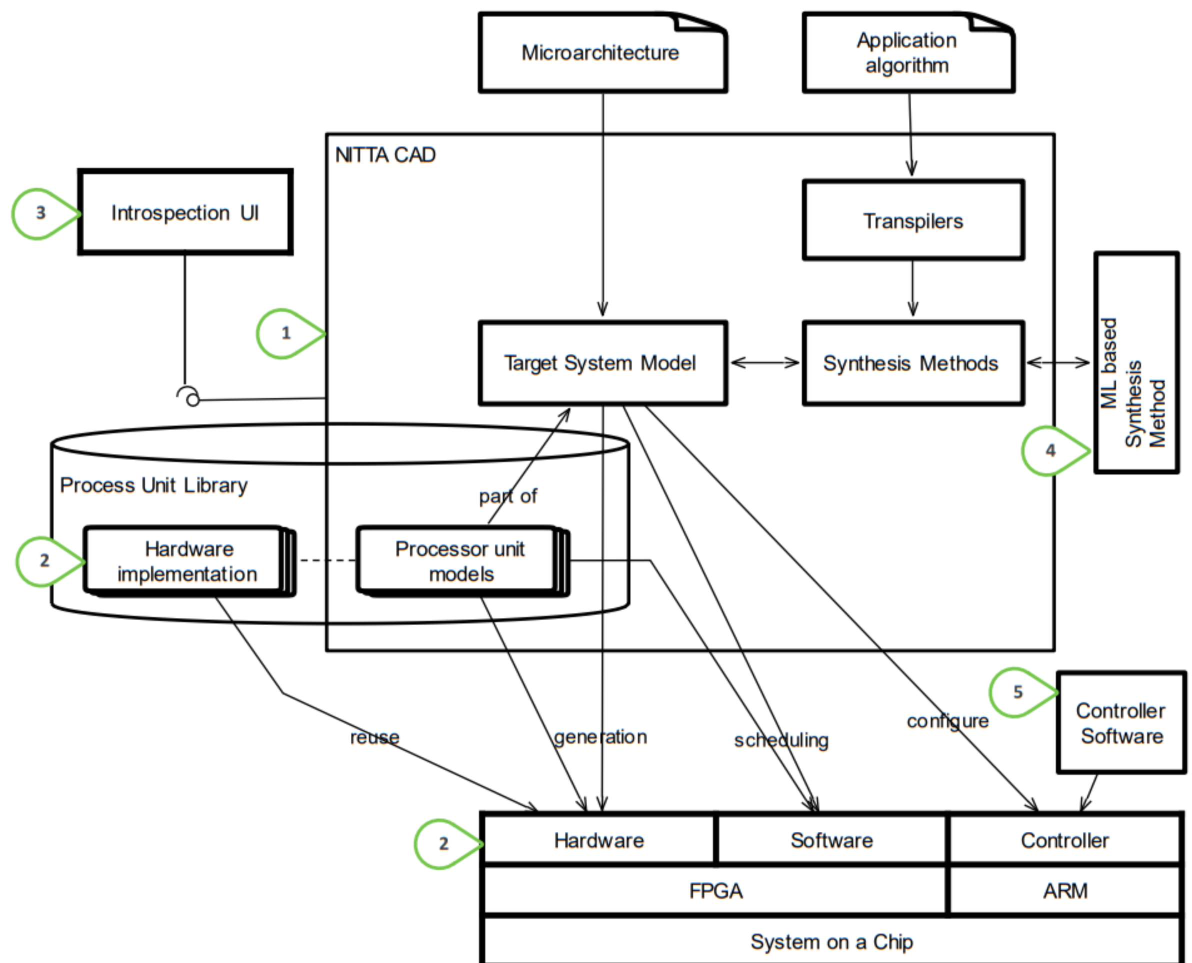


Рисунок 6 – Структура проекта NITTA [23]

Нумерация на рисунке (Рисунок 6) соответствует расшифровке используемых инструментов, представленной далее:

1. САПР реализован на языке Haskell.
2. В результате синтеза генерируется описание аппаратуры на языках FPGA, Verilog.
3. Интерфейс пользователя написан на Typescript и React.
4. Решения алгоритма синтеза принимаются с помощью методов машинного обучения, реализованных с помощью соответствующих библиотек на языке Python.
5. Управляющее программное обеспечение на языке Rust (в будущем).

Вышеприведенные методы также представлены на рисунке (Рисунок 6) с соответствующими номерами, кроме них еще стоит отметить, что схема конечного устройства синтезируется программой Quartus, а сборка проекта осуществляется с помощью сборщика Stack и компилятора для Haskell – GHC.

Рассмотрим какие методы верификации, которые используются в проекте NITTA.

1.3 Верификация в проекте NITTA

1.3.1 Статические методы

Проект NITTA использует язык Haskell, который является строго типизированным и имеет систему вывода типов Хиндли – Милнера [24]. Данная особенность позволяет автоматически генерировать тип данных выражений, что приводит к сокращению объёма кода. Кроме этого, вывод типов можно считать средством статической верификации, позволяющее выявлять ошибки на этапе компиляции кода.

Других формальные методы не используются из-за различных факторов, например, для дедуктивного анализа необходим разработчик с глубокими познаниями в области доказательств теорем и соответствующих инструментов.

Метод проверки на модели по принципу довольно похож на свойство-ориентированные тесты – в них обоих генерируются большие объемы тестовых данных для проверки ранее определенных свойств. При этом трудозатраты у первого метода выше, так как необходимо реализовывать модель на специальном языке с помощью соответствующего инструментария, кроме того, для поддержания и интерпретации тестов необходимо обучать разработчиков.

Использование формальных методов потенциально может принести огромную пользу при поиске дефектов в проекте, но на данный момент эти методы не реализуется по вышеприведенным причинам. Схожим по принципу с методом проверки на модели являются свойство-ориентированные тесты, которые активно используются в проекте.

1.3.2 Свойство-ориентированное тестирование

Одной из самых популярных и заслуженных библиотек из этой области является – Quickcheck. Именно с помощью этой свойство-ориентированной библиотеки написано большинство тестов в NITTA на данных момент.

В Quickcheck [25] для реализации идей свойство-ориентированного тестирования используется класс типов (в парадигме объектно-ориентированного программирования ближе подходит понятие – интерфейса) «Arbitrary», который содержит функцию «арбитраж», отвечающую за генерацию тестовых данных, и функцию «свертывание», позволяющую сократить тестовый вывод в случае ошибки.

Рассматриваемая библиотека обладает следующими достоинствами:

– Документация, уроки и множество примеров использования в различных проектах.

– Многие библиотеки имеют обратную совместимость с Quickcheck.

При этом имеются следующие недостатки:

– Отсутствует стандартная имплементация «Arbitrary» для новых типов данных – это связано с тем, что сами разработчики не знают, как сде-

лать её качественной. Таким образом, необходимо самому реализовывать генератор и функцию для свертывания его значений или можно воспользоваться сторонней библиотекой для создания шаблонных генераторов.

– Отсутствуют правила или семантика для класса типов «Arbitrary», что вредит при написании негативных тестов, ведь также необходимо проверять, что свойство не выполняется при определенных тестовых наборах. Проблема заключается в том, что генератор не знает о возможных негативных значениях и формирует любые возможные входные данные. Существует решение данной проблемы, однако оно реализовано вне рамок библиотеки.

Из-за этих недостатков возникают сложности при написании нового и поддержании старого кода. По этой причине было запланировано перейти к использованию альтернативного решения, что на данный момент постепенно осуществляется. Претендентом стала относительно новая библиотека свойство-ориентированного тестирования, призванная для решения вышеприведенных проблем, рассмотрим её подробней.

В Validity [26] используются два разных генератора для валидных и не валидных данных, что позволяет без проблем писать негативные тесты. Кроме того, библиотека обладает следующими достоинствами:

– Обратная совместимость с Quickcheck.
– Наличие стандартных генераторов и свойств для тестирования большого количества классов типов.
– Возможность относительно легкого переопределения существующих функций, например генератора.

Стоит отметить, что большинство стандартных генераторов и свойств не применимы в NITTA из-за сложности входных данных. Однако, их по-прежнему можно генерировать с помощью Quickcheck.

1.3.3 Динамические методы

В проекте самым распространённым видом тестирования является модульный. Во многих из них тестируются вычислительные блоки, к которым относятся блок умножителя, делителя, сумматора и так далее. Однако, из-за принципиальной сложности разрабатываемой системы новичку довольно сложно понять устройство и принципы функционирования применяемых модулей, несмотря на их интуитивные наименования.

Для решения этой проблемы используется разновидность модульного теста инструмент для тестирования интерактивных примеров (doctest), имеющий широкое распространение в языке Python. Принцип его работы заключается в поиске фрагментов кода с командами языка, после запуска которого происходит сравнение с ожидаемыми выводом. Сами тесты представляют собой набор команд, которые применяются при ручном выполнении процесса синтеза, тем самым, они имитируют ручной запуск этих команд. Таким образом, рассматриваемые тесты решают проблему сложности обучения новых разработчиков, стоит отметить, что он также позволяет поддерживать документацию в актуальном состоянии. В связи с простотой используемых в тестах команд возникают ограничения на данный метод в виде возможность реализовывать только несложные сценарии, кроме того, их тестовое покрытие небольшое относительно свойство-ориентированных и модульных тестов, из-за неудобства объявления входных данных. Фрагмент примера с использованием данного вида тестирования продемонстрирован на рисунке (Рисунок 7).

Проект обладает высокой сложностью тестовой инфраструктуры, что приводит к необходимости проверять используемые инструменты. Такие проверки помогают избегать ситуаций, когда успешно пройденные тесты на самом деле ничего не проверяют. К таким последствиям также постепенно приводит постоянное обновление кода системы, однако это решается поддержкой актуального состояния тестового инструментария в соответствии с системой.

```
0. >>> let f = F.multiply "a" "b" ["c", "d"] :: F String Int
1. >>> f
2. a * b = c = d
3. >>> let st0 = multiplier True :: Multiplier String Int Int
4. >>> st0
5. Multiplier {remain = [], targets = [], sources = [], currentWork
6.  Nothing, process_ = Process
7.     steps =
8. <BLANKLINE>
9.     relations =
10.     <BLANKLINE>
11.         nextTick = 0
12.         nextUid = 0
13.     , isMocked = True}
14.     >>> endpointOptions st0
15.     []
16.     ...
17.     >>> let Right st1 = tryBind f st0
18.     >>> st1
19.     Multiplier {remain = [a * b = c = d], targets = targets = [],
20.       sources = [], currentWork = Nothing, process_ = Process
21.     <BLANKLINE>
22.         relations =
23.     <BLANKLINE>
24.         nextTick = 0
25.         nextUid = 0
26.     , isMocked = True}
27.     >>> endpointOptions st1
28.     [?Target "a"@(0..∞ /P 1..∞),?Target "b"@(0..∞ /P 1..∞)]
29.     >>> let st2 = endpointDecision st1 $ \
30.     >>>                        EndpointSt (Target "a") (0...2)
31.     >>> st2
32.     Multiplier {remain = [], targets = ["b"], sources = ["c","d"],
   currentWork = Just a * b = c = d, process_ = Process
33.         steps =
34.             0) Step {pID = 0, pInterval = 0 ... 2, pDesc = End-
   point: Target a}
35.             1) Step {pID = 1, pInterval = 0 ... 2, pDesc = In-
   struction: Load A}
36.         relations =
37.             0) Vertical 0 1
38.         nextTick = 3
39.         nextUid = 2
40.     , isMocked = True}
41.     ...
```

Рисунок 7 – Фрагмент примера doctest для модуля умножителя

Проверка существующей тестовой инфраструктуры в NITTA осуществляется с использованием негативных тестов. САПР оперирует вычислительными блоками, поэтому для проверок такого рода был введен новый блок, способный

имитировать поломки. Внутри теста объявляется такой блок и указывается тип его поломки, затем в результате работы возникают предвиденная ошибка, которая успешно обрабатывается.

Одновременное тестирование нескольких вычислительных блоков осуществляется интеграционными тестами, в которых используются блоки, соединенные общей шиной. При возникновении ошибки сложно локализовать причину, в отличии от модульных тестов. Одним из инструментов поиска подобных неполадок выступает добавление отладочного вывода, использование которого обычно отнимает много времени.

1.4 Постановка задачи

Тестовая инфраструктура проекта содержит ряд проблем, затронутых ранее:

1. Высокая сложность поддержки тестов.
2. Реализация тестов требует написания большого количество шаблонного кода.
3. Сложности с локализацией и отладкой ошибок тестов.
4. Процесс синтеза представляет собой черный ящик.

Решения первых трех проблем может являться использование методологии разработки, основанной на поведении (Behavior driven development, BDD), благодаря наличию предметно-ориентированного языка. Поэтому необходимо составить критерии, которые позволят проанализировать и подобрать библиотеку. В случае отсутствия удовлетворительного варианта следует разработать требования и на их основе собственный предметно-ориентированный язык. В свою очередь, непрозрачность внутреннего устройства процесса синтеза для разработчика приводит к трудностям при выявления возникающих там ошибок. В данной ситуации стоит сосредоточится не на улучшения отладки возникающих проблем, а на методах их предупреждении. Проверка целостности процесса синтеза, как раз и является таким методом.

Из рассмотренных проблем следует цель данной работы: сокращение трудозатрат на верификацию САПР и повышение качества тестирования за счет применения встраиваемых специализированных языков и проверки целостности результатов процесса синтеза.

На основе предложенных методов, а также решений к ним, были сформированы следующие задачи:

1. Анализ методов и средств верификации САПР.
2. Определение критериев качества тестирования.
3. Разработка семантики и реализация встроенного специализированного языка программирования для верификации моделей целевой вычислительной системы.
4. Разработка метода проверки целостности модели функционального блока.
5. Оценка эффективности полученных решений с использованием ранее определенных критериев.

2 Разработка методов и подходов в верификации САПР

2.1 Выбор инструментов для верификации САПР

В предыдущей главе была поднята проблема сложности написания и поддержки тестов. Для её решения было решено подобрать библиотеку разработки на основе поведения, в частности особый интерес представляет идущий в комплекте предметно-ориентированный язык. Перед анализом необходимо сформировать требования, затем отталкиваясь от них сравнить все найденные решения.

2.1.1 Формирование критериев для анализа библиотек

Основным критерием выбора является наличие встроенного предметно-ориентированного языка, который позволит:

– Наглядно описать тесты, состоящие из нескольких действий или цепочки вычислений.
– Предоставит средства для хранения и передачи тестовых данных между шагами теста, что позволило бы удобно тестировать процесс синтеза.

При этом желательно, чтобы сама библиотека:

– Активно поддерживалась разработчиком и сообществом, что позволит избежать грубых ошибок на стороне инструментария.
– Содержала понятную документацию, включающую описание функций и примеры использования.

Даже в случае отсутствия удовлетворительной библиотеки результаты будут полезны при разработке собственного инструмента.

2.1.2 Анализ и сравнение существующих решений

В результате поиска было найдено три, на первый взгляд, подходящие библиотеки.

2.1.2.1 Библиотека Haskell-bdd

Данная библиотека [27] предоставляет только удобочитаемый синтаксис для записи тестов, что не решает проблему передачи данных между шагами теста. На рисунке (Рисунок 8) приведен пример теста:

```
0. testThat
1.    `given_` now "2014-07-15 12:00"
2.    `given_` noInvoices
3.    `when_` invoiceIsIssued
4.    `then_` invoiceDate ^?= "2014-07-15"
5.    `then_` invoiceNo ^?= "1/2014"
```

Рисунок 8 – Пример теста для библиотеки Haskell-bdd

Стоит отметить, что репозиторий проекта последний раз обновлялся шесть лет назад, то есть он не поддерживается разработчиком. Также в проекте отсутствует документация, единственным источником информации по использованию библиотеки являются примеры тестов. По вышеприведенным причинам данную библиотеку нецелесообразно использовать.

2.1.2.2 Библиотека Tasty-bdd

Рассматриваемая библиотека является частью заслуженного фреймворка – Tasty, который регулярно используется в проекте. Пример тестового сценария приведен на рисунке (Рисунок 9):

```
0. tests = testBdd "Test example"
1.    $ Given (print "Some effect")
2.    $ Given (print "Another effect")
3.    $ GivenAndAfter (print "Get resource" >> return "Resource 1")
4.                    (print . ("Release "++))
5.    $ GivenAndAfter (print "Get resource" >> return "Resource 2")
6.                    (print . ("Release "++))
7.    $ When (print "Action returning" >> return ([1..10]++[100..106])
8.    $ Then (@?= ([1..10]++[700..706]))
9.    $ End
```

Рисунок 9 – Пример теста для библиотеки Tasty-bdd

В результате анализа данной библиотеки [28] были выделены следующие достоинства:

– Сценарии составляется из операторов: «дано», «когда», «тогда», что придает тестам выразительность.

– В NITTA тестовые сценарии описываются с помощью библиотеки Tasty, таким образом, не возникнут проблемы с совместимостью.

– Библиотека поддерживается автором.

К недостаткам стоит отнести:

– Отсутствие механизма для хранения состояний.

– Мало примеров и поверхностная документация.

Данная библиотека могла использоваться в паре с собственным предметно-ориентированным языком, однако из-за недостаточного количества примеров возникают сложности, даже с запуском прототипа.

2.1.2.3 Библиотека HSpec

Библиотека использует блоки describe для соединения групп тестов по смыслу и оператор it, в котором дается название теста, описывается его поведение и задается ожидаемый результат. Пример теста представлен на рисунке (Рисунок 10):

```
0. tests_hspec = describe "Tests example" $ do
1.    context "when the numbers are small" $
2.    it "Should match the our expected value" $
3.    simpleMathFunction 3 4 5 `shouldBe` 7
4.    context "when the numbers are big" $
5.    it "Should match the our expected value" $
6.    simpleMathFunction 22 12 64 `shouldBe` 200
```

Рисунок 10 – Пример теста для библиотеки HSpec

Данный синтаксис, в отличии от предшественников, имеет встроенную валидацию структуры тестов, то есть при перестановке блоков местами возникнет ошибка, явно сообщающая об этом. Эта библиотека не подходит для использования по ряду причин:

– Отсутствует механизм для хранения состояний.

– Оператор it отвечает за проверку одной функции, а не цепочки вычислений, таким образом нет возможности реализовать последовательное принятие решений в процессе синтеза.

– Использование блоков describe и оператора it не добавляет тестам выразительности, относительно уже используемого решения – Tasty.

Таким образом, HSpec не целесообразно использовать, так как она не удовлетворяет ранее приведенным критериям.

2.1.2.4 Итоговое сравнение

В результате анализа BDD библиотек стало очевидным, что они по большей части предоставляют только новый синтаксис для описание тестовых сценариев и не позволяют решить проблему передачи состояния между различными функциями внутри теста, итог представлен в таблице (Таблица 1):

Таблица 1 – Сравнение библиотек

	Библиотека Haskell-bdd	Библиотека Tasty-bdd	Библиотека HSpec
Наглядность	+	+	+
Сохранение	-	-	-
Актуальность	-	+	+
Документация	+	-	+

Пункты таблицы соответствуют ранее сформированным критериям для выбора библиотеки. Таким образом, было принято решение написания собственного встроенного предметно-ориентированного языка для реализации тестов.

2.2 Разработка предметно-ориентированного языка

2.2.1 Формирование требований

Основные и необходимые требования были сформированы исходя из приведенных в прошлом разделе критериев поиска библиотеки, а именно:

– Язык должен предоставлять названия функций, соответствующие предметной области.

– Язык должен иметь возможность хранить состояние модели вычислительного блока на протяжении всего тестового сценария. При этом нужно освободить пользователя от необходимости передавать состояние в функцию и возвращать его из нее.

– Язык должен предусматривать возможность вывода отладочной информации о состоянии вычислительного блока по желанию пользователя.

Необходимо также ограничить используемую платформу для разрабатываемого языка:

– При реализации необходимо использовать язык программирования Haskell.

– Полученное решение должно быть совместимо с существующей тестовой инфраструктурой.

Таким образом, можно приступать к следующему шагу – разработке синтаксиса и семантики языка.

2.2.2 Синтаксис и семантика языка

2.2.2.1 Разработка структуры тестов

Все рассмотренные библиотеки для разработки на основе тестов в своей основе используют нотацию «Given», «When», «Then» для записи тестовых сценариев. По этой причине изначальные реализации базировались на ней, так как это позволило бы использовать общепринятую структуру построения те-

стовых сценариев. Однако было решено не использовать данную идею, из-за усложнения читаемости кода.

В рассмотренных примерах кода на рисунках видно (Рисунок 8, Рисунок 9 и Рисунок 10), что структура тестов организуется с помощью явного разделения каждой функции путем добавления оператора в начало строки. Такой подход целесообразен в рамках открытой библиотеки и связан с необходимостью предписывать пользователю правильный порядок тестового сценария. Однако для разрабатываемого специализированного языка выгодней будет отказаться от явного указания тестового этапа, что приведет к сокращению тестов и повышению читаемости кода.

В процессе синтеза для описания вычислительного блока используется имитационная модель, которая по запросу сообщает методу синтеза, какие операции может выполнить блок, таким образом, принятие решений можно сравнить с диалоговой формой. Повторение такой структуру тестовых сценариев позволит методу синтеза шаг за шагом принимать решения синтеза, что повысит выразительность языка.

2.2.2.2 Разработка используемых функций

Анализ входных данных и процесса синтеза позволит выделить необходимые функции и придумать к ним емкие названии. Рассмотрим подробнее все известные входные данные для тестов вычислительного блока:

– Название тестового сценария – уникальное наименование, разумнее задавать один раз в начале теста.
– Модель вычислительного блока. В рамках модульного теста проверяется отдельный вычислительный блок, поэтому не имеет смысла вводить отдельную функцию для его установки, а следует это делать один раз в начале теста.

– Функции, привязываемые к вычислительному блоку. Каждый вычислительный блок может выполнять множество функций, поэтому должна быть возможность их задавать.

Таким образом, в язык целесообразно добавить функцию, которая будет привязывать переменные к вычислительному блоку – «bindFunc».

Процесс синтеза представляет собой пошаговое принятие решений, где решение содержит: тип переменной – входной или выходной, наименование переменной, начальный и конечный такт времени, в течение которых решение должны быть принято процессом синтеза. Соответственно, как минимум необходима функция, принимающая все приведенные входные данные. Полезными будут функции, позволяющие автоматически задать ближайший такт времени, а для упрощения написания тестов будут использоваться функция, принимающая первое доступное решение.

В тесте необходимо проверять выходные данные и влияния принятых решений на вычислительный блок. Для этого были добавлены функции, которые начинаются с приставки «assert».

Используемая в библиотеках нотация не подразумевает вывод отладочной информации, к тому же, эта операция не поддается классификации по блокам теста. Для удовлетворения требования к предметно-ориентированному языку придется проигнорировать нотацию и добавить публичные функции для отладки, каждая из которых будет печатать отладочной вывод определенной части текущего состояния. Описание всех функций языка приведены в таблице (Таблица 2):

Таблица 2 – Описание функций языка

Название функции	Описание функции
bindFunc	Привязывает функцию к вычислительному блоку.
doDecision	Принимает решение синтеза.
beTargetAt	Формирует решение синтеза входной или выходной переменной с возможностью задать такт времени.
beSourceAt	
doDecisionFst	Принимает первое доступное решение синтеза.
doDecisionWithSource	Принимает решение синтеза с выходной переменной в первый доступный такт времени.
doDecisionWithTarget	Принимает решение синтеза с входной переменной в первый доступный такт времени.
traceFunctions	Выводит отладочную информацию о привязанных функциях.
tracePU	Выводит отладочную информацию о состоянии вычислительного блока.
assertBindFullness	Проверяет привязаны ли функции.
assertSynthesisDone	Проверяет завершенность процесса синтеза.
assertCoSimulation	Запускает симуляцию блока на тестовом окружении.

Для удовлетворения оставшихся требований необходимо определиться с некоторыми техническими особенностями реализации языка.

2.2.2.3 Выбор технических особенностей реализации

Требования к разрабатываемому языку предписывают использование языка Haskell. Он является функциональным языком и в нем отсутствует возможность изменять состояние переменных, поэтому для реализации встроенного предметно-ориентированного языка необходимо каким-нибудь образом сохранять состояние. Библиотек для хранения состояния или облегчающие решение этой проблемы не было найдено, оставшимся вариантом является стандартный механизм языка – монада State [29], которая может позволить хранить модель вычислительного блока и функции, привязанные к нему. Было решено использовать именно этот механизм так, как он является встроенным.

2.3 Разработка модуля проверки целостности процесса синтеза

С программной точки зрения в условном синтезированном процессе представлены следующие уровни:

- Intermediate – уровень функции, привязываемой к вычислительному блоку.
- Endpoint – уровень переменных, которые извлекаются из функции с одноименного уровня.
- Instruction – уровень инструкции, которые формируются относительно переменных, например, загрузить некоторое значение в вычислительный блок, при этом он сам не знает ни о инструкциях, ни о других вычислительных блоках, а работает с управляющими сигналами.
- CAD – уровень САПР, который является необязательным и возникает в случае осуществления оптимизаций встроенного компилятора, в частности – раскрутки циклов.

При этом направление соседние уровни межуровневые связи подчиняются некоторым правилам, проверку которых будет осуществлять разрабатываемый модуль. Они имеют следующий вид:

- Каждая инструкция, с одноименного уровня, имеет связь с переменной, представленной уровнем выше.
- Каждая переменная связана с функцией.
- Если во время синтеза была осуществлена оптимизация встроенного компилятора, то необходимо проверить связи с уровнем САПР.

Для решения этой задачи следует организовать функции для проверки следующих межуровневых отношений, представленных в таблице (Таблица 3):

Таблица 3 – Описание межуровневых проверок

Нижний уровень	Верхний уровень	Описание проверки	Назначение проверки
Переменные	Функции	Проверяет все ли переменные из функции представлены в модели.	Позволяет избежать отсутствие переменной в вычислительном блоке.
Инструкции	Переменные	Проверяет все ли инструкции привязаны к переменным.	Позволяет избежать отсутствие инструкции в вычислительном блоке.
CAD	Функции	Проверяет все ли функции имеют соответствующие шаги CAD.	Необходима для выявления ошибок в оптимизациях САПР.

Наиболее очевидным вариантом реализации является перебор связей в вычислительном блоке и их проверка, поэтому прототип будет сделан таким образом.

2.4 Определение критериев качества тестирования

Филипп Кросби один из пионеров дисциплины управление качеством считал, что качество — это соответствие требованиям. Однако в области компьютерных наук определение отличается. Качество – это степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон (имеется в виду stakeholders), которая позволяет, таким образом, оценить достоинства [30]. При этом ценность для заинтересованного лица может формироваться из требований и пожеланий к системе.

В тестировании проекта NITTA, в первую очередь, заинтересованными лицами выступают сами разработчики, соответственно критерии качества тестирования будут исходить из их требований. Были выделены следующие пожелания пользователей к разрабатываемому предметно-ориентированному языку: удобство использования, простота написания и поддержки тестов при высокой «глубине» тестирования. То есть при удовлетворении этих нефункциональных требований можно будет сделать заключение об удовлетворительном качестве разработанного языка.

Кроме критериев качества необходимо проанализировать тестовые метрики, под ними имеется в виду количественные показатели, используемые для оценки различных характеристик рассматриваемой системы, например тестовое покрытие.

2.5 Вывод

В данной главе были выбраны методы и инструменты для верификации САПР. Рассмотренные библиотеки разработки на основе тестов не удовлетворили критериям выбора, в результате было принято решение разрабатывать собственный встраиваемый предметно-ориентированный язык для написания тестовых сценариев. Проверка целостности процесса синтеза будет осуществляться путем сравнения межуровневых связей вычислительного блока. Для оценки качества тестирования были сформированы критерии.

3 Применение методов и подходов в верификации САПР

3.1 Реализация предметно-ориентированного языка

3.1.1 Прототип языка

В полученном предметно-ориентированном языке входными данными для тестов являются: модель вычислительного блока и функции, которые к нему следует привязать. Привязка (bind) – одна из операций процесса синтеза, позволяющая указать вычислительному блоку какую функцию следует выполнить. После привязки необходимо каждой используемой переменной из функции назначить синхронизирующий такт, то есть момент времени, в который блок обработает данную переменную. Затем после планирования тактов всех переменных полученный блок можно использовать для симуляции. Данная последовательность шагов выражается функциями языка, рассмотрим их подробнее, а затем монаду State на примере теста, приведенного на рисунке (Рисунок 11):

```
0. puUnitTestCase "multiplier smoke test" u $ do
1.    bindFunc fDef
2.    traceFunctions
3.    assertBindFullness
4.    doDecision $ beTargetAt 1 2 "a"
5.    doDecisionWithTarget "b"
6.    doDecision $ beSourceAt 5 5 ["c"]
7.    tracePU
8.    doDecisionFst
9.    assertSynthesisDone
10.   assertCoSimulation [("a", 2), ("b", 7)]
```

Рисунок 11 – Пример синтаксиса прототипа разрабатываемого языка

Каждый тестовый сценарий начинается с объявление теста на первой строке, включающее в себя:

– Название теста.

– Вычислительный блок.

– Тело теста.

На второй строке осуществляется привязка функции, производится это в теле теста, а не на входе, как в случае вычислительного блока, что позволяет привязать несколько функций и осуществить проверки между этими действиями. В качестве входа выступает функция умножение переменных «a» на «b» и присвоение результата переменным «c» и «d». Непосредственно проверка привязки функции к вычислительному блоку осуществляется на 4 строке. На строках 3 и 8 расположены функции отладочного вывода для привязанных функции и вычислительного блока соответственно. С 4 по 9 строку происходят принятия решений, для этого имеется 5 операторов, какой использовать в конкретном случае зависит от требований по времени. Например, на 5 строке переменная «a» будет занесена в блок умножения между 1 и 2 синхронизирующим тактом, когда как на 6 строке такт не указан, следовательно, будет подобран ближайший доступный такт. В случае обозначения такта двумя одинаковыми числами, как на 7 строке, операция осуществится строго на 5 такте. Проверка корректности процесса синтеза осуществляется на 10 строке, в нем проверяется, что все доступные решения синтеза исчерпаны. На 11 строке генерируется и запускается симуляция в тестовом окружении (testbench), который произведет операцию вычислительного блока и проверит результат. В случае умножения были заданы два входных параметра «a» и «b», которым присвоили числа 2 и 7 соответственно, следовательно тест проверит, что результат умножения равен 14.

Монада[2] State [31] – описывает функции, одна из которых хранит в некотором контейнере заданное состояние, а другая позволяет получить пару, состоящую из результата вычисления и обновленного состояния [32]. В разрабатываемом языке с помощью монады State сохраняется вычислительный блок в

---

[2] Монада в области программирования – это это абстракция линейной цепочки связанных вычислений. Её основное назначение — инкапсуляция функций с побочным эффектом от чистых функций, а точнее их выполнений от вычислений [34]

начале теста на первой строке. Функции языка устроены таким образом, что они ожидают модель вычислительного блока в качестве состояния, в свою очередь, это позволяет передавать его неявно по всем тестовым функциям, начиная со второй строки. Содержимое состояния представлено в таблице (Таблица 4):

Таблица 4 – Содержимое состояния

Название	Содержимое	Предназначение
unit	Вычислительный блок	Хранение состояния, как предписывают требования.
functions	Список функций	Хранение привязываемых функций для проверки привязки.

При этом пользователь не имеет прямого доступа к состоянию и ему не нужно заботиться о передаче вычислительного блока между функциями теста, однако, он может проверить текущее значение полей с помощью отладочного вывода. Использование монады State предоставило возможность избежать написание дополнительного кода для передачи и возврата модели вычислительного блока в тестовых функциях, что привело к сокращению кода тестов.

3.1.2 Работа над ошибками прототипа

После написания прототипа встраиваемого специализированного языка он был применен для тестирования модуля делителя, в результате которого были выделены следующие недостатки:

– громоздкие названия функций;
– неудобность и контринтуитивность процесса принятия решения в языке;
– непривычное и неэффективное расположение для установки входных значений процесса симуляции и другие проблемы;
– наличие лишних функций в языке.

Было решено переработать синтаксис языка, чтобы решить вышеприведенные проблемы. На основе изменений в языке была составлена таблица (Таблица 5), где слева название функции, а справа соответствующее изменение в ней:

Таблица 5 – Изменение в функциях

Старое название функции	Изменение в функции
bindFunc	Новое название – assign.
doDecision	Новое название – decideAt. Переделан механизм обработки входных данных.
doDecisionFst	Удалена из-за возникающей непрозрачности, которая приводит к понижению интуитивности тестовых сценариев.
doDecisionWithSource	Объединены в одну функцию – decide.
doDecisionWithTarget	Объединены в одну функцию – decide.
beSourceAt	Новое название – provide. Возможность задавать такт времени была перенесена.
beTargetAt	Новое название – consume. Возможность задавать такт времени была перенесена.
assertBindFullness	Новое название – isFullyBinded.
assertCoSimulation	Установка входных данных была вынесена в новую функцию – setValue.

В результате работы над ошибками были удалены лишние функции, сокращены названия, что положительно сказалось на удобочитаемости тестов и

скорости их написания, одним из наиболее важных изменений оказалось новая механика работы принятия решений. Во-первых, была удалена функция для принятия первого доступного решения в связи с тем, что она усложняет дальнейшую поддержку тестов и снижает понимание того, какое именно действие осуществляет данный оператор. Во-вторых, из функций для формирования переменных была перенесена возможность задавать синхронизирующий такт времени в функцию принятия решений. В совокупности эти изменения упростили применение функции принятия решений.

Также изменениям подверглась структура данных состояния, в нее добавились новые поля, представленные в таблице (Таблица 6):

Таблица 6 – Обновленное содержимое состояния

Название	Содержимое	Предназначение
testName	Строка с названием теста	Вывод названия теста при выводе ошибок.
unit	Вычислительный блок	Хранение состояния вычислительного блока для принятия решений.
functions	Список функций	Хранение привязываемых функций для дальнейшей проверки успешности их привязки.
cntxCycle	Список переменных	Выступают входными данными при симуляции.

Для наглядности преимуществ внесенных изменений следует сравнить новый синтаксис старой и новой версии языка на основе ранее рассмотренного примера.

3.1.3 Результаты модернизации

Старые тесты были переписаны в соответствии с изменениями встроенного предметно-ориентированного языка. Разберем возможности обновленной версии языка на основе сравнения с предыдущей. Примеры теста приведены на рисунке (Рисунок 12) где слева прежняя, а справа новая версия:

```
0. puUnitTestCase "multiplier smoke test" u $ do
1.    bindFunc fDef                             | assign $ multiply "a" "b"
      ["c", "d"]
2.    traceFunctions                            | setValue "a" 2
3.    assertBindFullness                        | setValue "b" 7
4.    doDecision $ beTargetAt 1 2 "a"           | decideAt 1 2 $ consume
      "a"
5.    doDecisionWithTarget "b"                  | decide $ consume "b"
6.    doDecision $ beSourceAt 5 5 ["c"]         | decideAt 5 5 $ provide
      ["c"]
7.    tracePU                                   | tracePU
8.    doDecisionFst                             | decide $ provide ["d"]
9.    assertSynthesisDone                       | assertSynthesisDone
10.   assertCoSimulation [("a", 2), ("b", 7)]   | assertCoSimulation
```

Рисунок 12 – Сравнение синтаксиса старой и новой версий разрабатываемого языка на примере теста блока умножителя

Объявление функции не претерпело изменений поэтому начнем сравнение с второй строки, на которой видно, что наименование функции привязки сократилось, кроме этого, для упрощения понимания тестовых сценариев было решено использовать полное название функций внутри тела теста. На 3 и 4 строках расположены операторы для присвоения начального значения входным переменным процесса симуляции, аналог в прежнем варианте находится на одиннадцатой строке. Данное расположение обусловлено близостью к месту объявления этих переменных, ведь во многих языках программирования имя переменным задается рядом с местом её объявления. Стоит отметить, что новые функции имеют дополнительную проверку, не позволяющую задавать значение для несуществующей переменной. Рассмотрим обновленные функции для принятия решений процесса синтеза, они расположены с 5 по 9 строку. Во-первых,

удалось привести к единому стилю использование функции для выбора переменных, удалив из них установку интервала синхронизирующего такта, во-вторых, была удалена функция выбора первого доступного решения, несмотря на её простоту, она вносила долю неопределенности относительно принятого решения, что в конечном счете лишь усложняло отладку и понимание теста. Функция на десятой строке для проверки завершенности синтеза не претерпела изменений. Запуск и проверка симуляции на одиннадцатой строке стала проще, потеряв входные параметры, о чем было сказано ранее. К уже имеющимся функциям трассировки отладочного вывода добавились пару дополнительных: для отображения текущего состояния процесса, а также для получения списка доступных на данный момент решений синтеза.

Таким образом, проблемы существующего решения были решены следующим образом:

– Громоздкие функции заменены на короткие, а мало используемые удалены из языка.
– Новый синтаксис и семантика принятия решения исправили проблему неудобства использования.
– Новая функция setValue решает проблему с ранее неэффективным расположением установки начальных значений симуляции. Формально местоположение функции в сценарии не регулируется, следовательно её можно разместить в любом месте, однако эта проблема решается с помощью приведения кода к общему стилю. Выявляются такие проблему с помощью рецензирования кода.

После модификации языка были реализованы тесты для других вычислительных модулей, а также негативные тесты с имитацией неисправного модуля. Дальнейшие изменения осуществлялись в рамках других задач по мере необходимости добавления новых функций язык:

– Функция «наивной» симуляция, которая берет вычислительный блок, принимает в нем все доступные решения и запускает симуляцию.
– Небезопасное принятие решений. Производит принятие решений, игнорируя проверки со стороны языка.
– Размотка цикла. Запускает одноименную оптимизацию САПР.
– Проверка решения на доступность.
– Проверка блокировки решения для выбора.

В дальнейшем по мере покрытия новых модулей тестами функции языка будут пополняться.

3.2 Реализация модуля проверки целостности процесса синтеза

3.2.1 Прототип модуля проверки целостности

Разработанный прототип проверял целостность между тремя уровнями: функций, переменных и инструкций. Соответственно для этого было необходимо проверить, что вычислительный блок подчиняется следующим правилам:
– Каждая инструкция, с одноименного уровня, имеет связь с переменной, представленной уровнем выше.
– Каждая переменная связана с функцией.

Прототип алгоритма представлял собой следующую последовательность шагов. В первую очередь, происходило формирование хранилища типа ключ-значение, где ключ — это идентификатор шага вычислительного процесса, а значение — это описание этого шага. В свою очередь, межуровневые связи представляют собой пару идентификаторов шагов, то есть с помощью полученного хранилища появилась возможность проверять уровни каждого из шагов. По завершению формирования хранилища осуществлялся циклический обход всех имеющихся связей в вычислительном блоке и проводилось сравнение на то, что каждая связь ссылается на верные вычислительные блоки. При возникновении несоответствия проверка прекращалась и выводилась ошибка.

3.2.2 Работа над ошибками прототипа

Из-за специфики языка Haskell полученное решение осуществляло для каждой связи полный обход списка шагов, теоретически сложность такого алгоритма равна произведению n на m, где n количество идентификаторов в списке межуровневых связей, а m количество шагов вычислительного блока. Таким образом, было решено изменить подход и начать формировать связи на основе имеющихся данных.

Первоначальный шаг старого алгоритма претерпел изменения, содержимое каждого хранилища типа ключ-значение изменилось, оно представлено в таблице (Таблица 7):

Таблица 7 – Содержимое хранилищ для проверки целостности

Уровень хранилища	Ключ	Значение
Функции	Идентификатор шага	Сама функция
Переменные	Переменная	Идентификатор шага
Инструкции	Идентификатор шага	Описание шага

Новая версия алгоритма заключается в генерация новых связей на основе их соответствия. Сначала происходит итерация по всем используемым функциям, каждая из которых разбивается на переменные, результат помещается в соответствующее хранилище. Для каждой переменной функции происходит поиск пары на основе их равнозначности друг другу, например, если в функции переменная «a», при этом хранилище переменных содержит ту же самую переменную, то на основе их идентификаторов формируется связь. В результате получается список связей по структуре аналогичный тому, который содержится внутри вычислительного блока. На основе сравнения формируется конечное сообщение, при обнаружении несоответствия – с ошибкой. Такой подход к алгоритму позволил:

– Получить выигрыш по скорости, так как исчезла необходимость делать лишние обходы списков.

– Сократить размер кода в модуле.

Полученное решение не осуществляло всех ранее задуманных проверок целостности, к тому же оно потребовало других доработок.

3.2.3 Модернизация проверки целостности

3.2.3.1 Обнаруженные недочеты

Следующая итерация разработки модуля проверки целостности заключается в добавлении дополнительной проверки – уровня САПР. Однако, были найдены несколько недочетов.

Первая проблема возникает при генерации списка межуровневых связей для вычислительных блоков, содержащих несколько функции, где выходные переменные одной являются входными для другой. В таких ситуациях формируется ошибочные связи. Для решения проблемы было решено исправить содержимое хранилища ключ-значения и исходный алгоритм.

Вторая проблема проявляется в вычислительных блоках с оптимизацией компилятора, соответственно в них имеются шаги уровня САПР, что в результате видоизменяет связь функций с переменными. Между ними добавлялся промежуточный транспортный уровень. Он представляет собой инструкцию, внутри которой содержится переменная и название инструкции – транспорт. Решить проблему поможет добавление в модуль дополнительной межуровневой проверки. Стоит отметить, хоть уровень транспорта и является инструкцией, однако для удобства они будут разделяться по разным уровням.

Перед модификацией работоспособного решения необходимо добавить негативные тесты для выявления регрессионных ошибок при модификации существующего кода. Написание этих тестов потребует использовать вычислительный блок с возможностью симулировать неполадки, кроме того, понадо-

биться внести новые виды ошибок в этот блок, чтобы была возможность имитировать ошибки в межуровневых связях.

Исправление вышеупомянутых проблем, а также добавления новой проверки уровня САПР, повлияло на содержимое хранилищ, соответствующие изменения представлены в следующей таблице (Таблица 8):

Таблица 8 – Обновленные хранилища

Уровень хранилища	Ключ	Значение
Функции	Идентификатор шага	Сама функция.
Переменные	Переменная	Список, содержащий пары, где первый элемент идентификатор шага.
Инструкции	Идентификатор шага	Описание шага.
САПР	Идентификатор шага	CAD функция.
Транспорт	Переменная	Пара из идентификатора и инструкции.

Проверка целостности уровня САПР схожа с ранее реализованными функциями целостности, соответственно необходимо ввести отдельное хранилище. Идея алгоритма остается прежней, на основе имеющихся шагов формируются новые связи, которые сравниваются с существующими в вычислительном блоке.

3.2.3.2 Результаты модернизации

Исправлены ошибки с повторяющимися переменными. Изменения в первую очередь затронули хранилище переменных, теперь оно содержит список идентификаторов, ссылающихся на переменную. Кроме того, если при

формировании связей встречается несколько переменных, то формируются списки для каждой из них, например, если переменная «b» является входной и выходной, то окончательно будет сформировано два списка, в одном переменная будет входной, а в другом выходной. При проверке необходимо, чтобы хотя бы один список межуровневых связей совпал со списком из вычислительного блока.

Решение проблемы транспортного уровня заключалось в запуске дополнительной проверки в случае возникновения несоответствия при проверке связи переменных и функций. По мере исправления данной неполадки возникла новая. После добавления проверки транспортного уровня модуль перестал компилироваться из-за возникающей ошибки. Её причина связана с разными представлениями шагов уровня транспорта у вычислительного блока при его тестировании отдельно и в составе микроархитектуры. Решение данной проблемы заключалось в использовании класса типов. В результате в зависимости от типа подаваемого блока используется тот или иной экземпляр класса типов.

Алгоритм проверки целостности САПР формирует два хранилища ключ-значение для CAD функций и для соответствующих шагов. Затем происходит поиск функций и совпадающие с ней CAD шаги, при нахождении формируется межуровневая связь, которая помещается в список. В результате полученный список сравнивается с межуровневыми связями вычислительного блока.

3.3 Анализ полученных результатов

3.3.1 Тестовое покрытие

Процент тестового покрытия изменился незначительно, несмотря на добавление тестов в нескольких модулях, таких как: умножитель, делитель, аккумулятор, память (fram), а также самого DSL. Данный результат не является негативным, так как основной целью предметно-ориентированного языка было сокращение трудозатрат на верификацию.

Тестовое покрытие было получено с помощью опции сборщика проекта Stack [33] – coverage. Результаты представлены в таблице (Таблица 9):

Таблица 9 – Сравнение тестового покрытия полученного решения

Модуль	Top Level Definitions				Alternatives				Expressions			
	%		П / В		%		П / В		%		П / В	
	Д	П	Д	П	Д	П	Д	П	Д	П	Д	П
Intermediate.DataFlow	69	69	9/13	9/13	71	71	5/7	5/7	83	83	61/73	61/73
Intermediate.Functions	76	76	92/121	92/121	93	93	15/16	15/16	84	84	554/653	554/653
Intermediate.Functions.Accum	86	86	20/23	20/23	90	90	28/31	28/31	96	96	285/295	285/295
Intermediate.Simulation	100	100	4/4	4/4	75	75	12/16	12/16	79	79	183/229	183/229
Intermediate.Types	57	57	48/83	50/87	84	78	16/19	18/23	85	69	326/383	332/476
Intermediate.Value	53	53	88/164	87/164	75	70	15/20	14/20	67	66	545/812	536/812
LuaFrontend	56	56	34/60	34/60	60	60	88/146	88/146	76	76	863/1135	863/1135
Model.Problems.Endpoint	66	86	14/21	19/22	100	100	12/12	15/15	79	100	85/107	114/114
Model.ProcessorUnits.Accum	52	54	28/53	29/53	71	71	33/46	33/46	85	86	720/839	713/829
Model.ProcessorUnits.Broken	41	39	20/48	20/51	76	86	26/34	38/44	87	89	476/541	512/573
Model.ProcessorUnits.Divider	50	55	34/67	32/58	63	69	46/72	37/53	87	92	646/738	647/703
Model.ProcessorUnits.Fram	49	50	32/65	32/64	93	93	60/64	60/64	90	90	1072/1183	1043/1154
Model.ProcessorUnits.Multiplier	50	50	22/44	22/44	79	79	19/24	19/24	92	93	509/551	508/545
Model.ProcessorUnits.Shift	37	37	19/51	19/51	66	66	24/36	24/36	87	87	504/573	495/563
Model.ProcessorUnits.Types	48	47	36/75	41/86	72	56	21/29	23/41	79	76	234/295	237/311
Model.Types	9	14	2/21	3/21	0	28	0/7	2/7	1	23	2/120	28/120
Project	77	77	7/9	7/9	66	68	10/15	13/19	59	63	230/389	259/408
Project.Template	76	76	13/17	13/17	58	58	7/12	7/12	69	69	209/300	209/300
Project.TestBench	30	35	6/20	7/20	88	88	8/9	8/9	79	96	402/506	497/515
Synthesis	28	28	4/14	4/14	66	83	4/6	5/6	77	77	120/154	136/176
Synthesis.Explore	81	83	9/11	10/12	50	43	8/16	7/16	77	77	244/316	251/324
Synthesis.Method	91	91	11/12	11/12	70	70	12/17	12/17	81	81	169/208	174/213
Synthesis.OptimizeAccum	100	100	3/3	3/3	-	-	0/0	0/0	87	87	7/8	7/8
Synthesis.Refactor	100	100	1/1	1/1	100	100	5/5	5/5	76	76	13/17	13/17
Utils	100	100	20/20	23/23	92	88	23/25	24/27	92	92	235/255	242/263
Utils.ProcessDescription	94	95	18/19	22/23	-	100	0/0	2/2	90	95	148/164	200/209
Model.IntegrityCheck	-	92	-	12/13	-	62	-	17/27	-	80	-	346/431
Program Coverage Total	55	56	778/ 1395	810/ 1425	68	69	624/ 908	660/ 955	83	84	14163/ 16865	14800/ 17485

Из полученной таблицы (Таблица 9) были удалены некоторые записи для модулей, в которых тестовое покрытие не изменилось. В первом столбце приведены названия модулей, на первой строке идут названия метрик, на второй и третьей строках используются следующие символы и сокращения:

– Знак процента – процент покрытия тестового решения.

– П/В – покрыто и всего используемых единиц, единица измерения которой зависит от используемой метрики.

– Д – количество до разработки решения.

– П – количество после разработки решения.

В результате общее тестовое покрытие, приведенное на последней строке, выросло на 1 процент или на 637 строк кода, но при этом удалось найти много проблем, связанных с процессом синтеза. Оба метода верификации были проверены негативными тестами, использующие вычислительный блок с имитацией поломок, что в дальнейшем упростит их модернизацию, а также позволит предотвратить некоторые ошибки.

3.3.2 Выявленные ошибки

С помощью модуля предметно-ориентированного языка были обнаружены следующие ошибки:

– В модуле делителя.

– В синхронизирующих тактах. Нельзя было принять решение в нулевой такт.

– В блокировке переменных модуля умножителя.

Кроме того, была выявлена возможность задавать решения синтеза в недоступные синхронизирующие такты. В результате вышеприведенные ошибки были исправлены.

Проверка целостности выявила неполадки в вычислительном блоке памяти (fram), которые были успешно исправлены. Остальные найденные ошибки возникали, в течении разработки, внутри самого модуля.

3.3.3 Количество строк кода

Наглядным примером различия в количестве строк является тест вычислительного блока умножителя. До разработки предметно-ориентированного языка уже существовал doctest. Сравним его с DSL тестом на рисунке (Рисунок 12) по количеству строк. В doctest сто двадцать четыре строки, исходный код которых приведен в приложении Г, а в DSL десять строк. Благодаря разнице в двенадцать раз можно заключить, что DSL тест проще читать, писать и поддерживать. Данные требования выступали основными у пользователей языка, поэтому можно сделать вывод об удовлетворительном качестве разработанного решения.

3.4 Вывод

В данной главе были реализованы методы, разработанные в предыдущих разделах данной работы. В частности, был получен DSL для тестирования вычислительных блоков и модуль проверки целостности результатов процесса синтеза. С их помощью были протестированы различные модули САПР, таким образом, был выявлен ряд ошибок, которые удалось исправить. Кроме того, выполнение ранее поставленной цели и задач подтверждается результатами анализа полученных решений на основе ранее сформированных критериев.

Заключение

В рамках данной работы были рассмотрены методы верификации сложного программного обеспечения, предметная область, вычислительная платформа NITTA и способы верификации в ней. Исходя из выделенных недостатков тестовой инфраструктуры были поставлены задачи по сокращению трудозатрат на верификацию и повышение качества тестирования с помощью применения встраиваемых специализированных языков и проверки целостности результатов процесса синтеза. Для решения задач были проанализированы существующие BDD библиотеки, ни одна из них не удовлетворила критериям.

Результатом работы являются предметно-ориентированный язык и модуль проверки целостности процесса синтеза, приведенные в приложении А и приложении Б соответственно. Приложение В содержит позитивные и негативные тесты полученных решений. В результате были найдены ошибки в нескольких вычислительных блоках, увеличилось тестовое покрытие, расширилось количество тестов и тестируемых модулей, а разработанный язык упростил написание и поддержку тестов, следовательно, поставленная цель была успешно выполнена.

Развитием данной работы может служить: модернизация проверки целостности путем расширения проверяемых элементов, внедрение в проект альтернативных свойство-ориентированных библиотек, или их комбинирование с предметно-ориентированным языком.

Список использованных источников

1. Coussy P. et al. An introduction to high-level synthesis // IEEE Des. Test Comput. IEEE, 2009. Vol. 26, № 4. P. 8–17.
2. Mami S., Lahbib Y., Mami A. A New HLS Allocation Algorithm for Efficient DSP Utilization in FPGAs // J. Signal Process. Syst. 2020. Vol. 92, № 2.
3. Prohorov D., Penskoi A. Verification of the CAD System for an Application-Specific Processor by Property-Based Testing // 2020 9th Mediterranean Conference on Embedded Computing, MECO 2020. Institute of Electrical and Electronics Engineers Inc., 2020.
4. Martin G., Smith G. High-level synthesis: Past, present, and future // IEEE Des. Test Comput. 2009. Vol. 26, № 4.
5. Penskoi A. et al. Hybrid NISC/TTA high-level synthesis tool // International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM. 2018. Vol. 18, № 2.1.
6. Holmes R. Writing BDD Test Scenarios [Electronic resource]. URL: https://www.departmentofproduct.com/blog/writing-bdd-test-scenarios/.
7. ISO/IEC/IEEE© Std. ISO/IEC/IEEE International Standard - Systems and software engineering -- System life cycle processes // ISO/IEC/IEEE 15288 First edition 2015-05-15. 2015. P. 1–118.
8. Osborne L. et al. Clarus: Concept of Operations. 2005. 123 p.
9. Osborne L. et al. Clarus Concept of Operations. 107 Carpenter Dr. Suite 230 Sterling VA 20164 1Meridian Environmental Technology Inc., Grand Forks, ND, 2005.
10. ISTQB, Veenendaal E. van. Standard Glossary of Terms used in Software Testing. ISTQB, 2018. P. 67.
11. Глебович К.Ю. MODEL CHECKING. Верификация параллельных и распределенных программных систем. БХВ-Петербург, 2010.
12. O’Regan G. Model Checking BT - Mathematics in Computing: An Accessible Guide to Historical, Foundational and Application Contexts / ed. O’Regan G. Cham: Springer International Publishing, 2020. P. 383–392.
13. McIntosh S. et al. An empirical study of the impact of modern code review practices on software quality // Empir. Softw. Eng. 2016. Vol. 21, № 5.
14. Fowler M., Foemmel M. Continuous integration // Thought-Works) http//www. thoughtworks. com/Continuous Integr. pdf. 2006. Vol. 122, № 14. P. 1–7.

15. Dalton J. Test-Driven Development // Great Big Agile. 2019.
16. Diepenbeck M. Behavior driven development for tests and verification // Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015. 2015.
17. Cucumber web page [Electronic resource].
18. Penskoi A.V. et al. High-level synthesis system based on hybrid reconfigurable microarchitecture // Sci. Tech. J. Inf. Technol. Mech. Opt. 2019. Vol. 19, № 2.
19. Taraate V. ASIC Design and Synthesis // ASIC Design and Synthesis. 2021.
20. V E.D. Designing SPI based protocol for the real-time computational platform // Proc. Sci. Pract. Conf. young Sci. “Computing Syst. networks (Mayorov’s readings).” 2018. P. 74–77.
21. Lee, E. A., & Seshia S.A. Introduction to Embedded Systems. A Cyber-Physical Systems Approach. Second Edition // Studies in Systems, Decision and Control. 2017. Vol. 195.
22. Пенской А.В. Интерфейс управления процессом высокоуровневого синтеза специализированного вычислителя. Сборник тезисов докладов конгресса молодых ученых., 2018. P. 2.
23. Penskoi A. Software Verification on the ASIP CAD Example or How to Trust Your Team and Yourself // Open Dais. 2021. P. 29.
24. Hindley R. et al. Hindley–Milner type system // WIKI. 2018.
25. Claessen K., Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs // Acm sigplan Not. ACM New York, NY, USA, 2011. Vol. 46, № 4. P. 53–64.
26. Kerckhove T.S. QuickCheck, Hedgehog, Validity [Electronic resource]. 2019. URL: https://www.fpcomplete.com/blog/quickcheck-hedgehog-validity/.
27. Haskell-bdd [Electronic resource]. URL: https://github.com/humane-software/haskell-bdd.
28. Tasty-bdd [Electronic resource]. URL: https://hackage.haskell.org/package/tasty-bdd.
29. Jones M.P. Functional programming with overloading and higher-order polymorphism // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 1995. Vol. 925.
30. Standardization I.O. for. Systems and Software Engineering: Systems and

Software Quality Requirements and Evaluation (SQuaRE): Measurement of System and Software Product Quality. ISO, 2016.

31. Monad State. Hackage web page [Electronic resource]. URL: https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html.
32. Haskell wikibook [Electronic resource]. 2018.
33. Haskell Stack [Electronic resource]. URL: https://docs.haskellstack.org/en/stable/README/.
34. O'Sullivan B. Real world Haskell. 2009.

Приложение А. Исходный код модуля DSL (обязательное)

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# OPTIONS_GHC -Wno-redundant-constraints #-}

{- |
Module      : NITTA.Model.ProcessorUnits.Tests.DSL
Description : Provides functions to test PU, by making syntesis decisions
Copyright   : (c) Artyom Kostyuchik, 2021
License     : BSD3
Maintainer  : aleksandr.penskoi@gmail.com
Stability   : experimental

= Module description

DSL (domain-specific language) is a module for testing Processor Units (PU).

= Algorithm

1. Choose PU and provide it into unitTestCase.
2. Assign function to this PU.
3. Schedule computational process for every variable in function.
4. Assert (check) the resulting PU.

= Example

Test case (numbers to the right correspond to the algorithm steps):

@
unitTestCase "multiplier smoke test" pu $ do     -- 1. Created test case for provided PU

         assign $ multiply "a" "b" ["c", "d"]      -- 2. Bind function 'a * b = c = d' to PU
         setValue "a" 2                            --    Set initial input values
         setValue "b" 7                            --    for further CoSimulation

         decideAt 1 2 $ consume "a"                -- 3. Bind input variable "a" from 1 to 2 tick
         decide $ consume "b"                      --    Bind input variable "b" at nearest tick
         decideAt 5 5 $ provide ["c"]              --    Bind output variable "c" at 5 tick
         decide $ provide ["d"]                    --    Bind output variable "d" at nearest tick

         traceProcess                              --    Print current process state to console

         assertSynthesisDone                       -- 4. Check that all decisions are made
         assertCoSimulation                        --    Run CoSimulation for current PU

...

pu = multiplier True :: Multiplier String Int Int  -- 1. Chose PU: Multiplier
@

= Algorithm steps description

* You can use any PU which instantiated with 'NITTA.Model.ProcessorUnits.Types.ProcessorUnit' and
'NITTA.Model.Problems.Endpoint.EndpointProblem' type class

* There are 4 functions for assign:

    * assign - binds function to PU right at the moment.

    * assigns - binds like 'assign', but uses a list of functions as an input.

    * assignNaive - store function in Test State and binds it only at naive synthesis.
                    Don't forget to call 'decideNaive' function.
```

```
    * assignsNaive - works like 'assignNaive', but uses a list of functions as an input.

* You can bind variables from the function to PU:

    * for first you need to wrap variables:

        * consume - for input variable.

        * provide - for output variables.

    * For second, you can pass wrapped variables to 'decide' function and
      schedule (make synthesis decisions) them. There 3 types of 'decide':

        * decide               - bind variable at the next tick of PU (nearest).

        * decideAt             - bind variable at provided moment.

        * decideNaiveSynthesis - runs naive synthesis (makes all available decisions).
                                 Requires using 'assignNaive' function.

* Assert function could be at any place in the test case.
   For a positive test case it usually at the end.

= CoSimulation:

To run simulation use 'assertCoSimulation' function.
Don't forget to set initial input values with 'setValue' function.

= Debug:

For debugging use functions starting with trace*, e.g. 'tracePU'.
-}
module NITTA.Model.ProcessorUnits.Tests.DSL (
    unitTestCase,
    assign,
    assigns,
    assignNaive,
    assignsNaive,
    setValue,
    setValues,

    -- *Process Unit Control
    decide,
    decideAt,
    decideAtUnsafe,
    consume,
    provide,
    breakLoop,
    decideNaiveSynthesis,

    -- *Asserts
    assertBindFullness,
    assertCoSimulation,
    assertSynthesisDone,
    assertEndpoint,
    assertAllEndpointRoles,
    assertLocks,

    -- *Trace
    tracePU,
    traceFunctions,
    traceEndpoints,
    traceProcess,
) where

import Control.Monad.Identity
import Control.Monad.State.Lazy
import Data.CallStack
import Data.List (find)
import qualified Data.Set as S
import NITTA.Intermediate.Types
import NITTA.Model.Networks.Types (PUClasses)
import NITTA.Model.Problems
import NITTA.Model.ProcessorUnits
import NITTA.Model.ProcessorUnits.Tests.Utils
import NITTA.Model.Types
```

```
import NITTA.Project
import NITTA.Utils
import Numeric.Interval.NonEmpty hiding (elem)
import Test.Tasty (TestTree)
import Test.Tasty.HUnit (assertBool, assertFailure, testCase)

unitTestCase ::
    (HasCallStack, ProcessorUnit pu v x t, EndpointProblem pu v t) =>
    String ->
    pu ->
    DSLStatement pu v x t () ->
    TestTree
unitTestCase name pu alg = testCase name $ do
    void $ evalUnitTestState name pu alg

-- | State of the processor unit used in test
data UnitTestState pu v x = UnitTestState
    { testName :: String
    , -- | Processor unit model.
      unit :: pu
    , -- | Contains functions assigned to PU.
      -- There two types of assign function:
      -- 1. assign - binds to PU.
      -- 2. assignNaive - will be binded during naive synthesis.
      functs :: [F v x]
    , -- | Initial values for coSimulation
      cntxCycle :: [(String, x)]
    }
    deriving (Show)

type DSLStatement pu v x t r = (HasCallStack, ProcessorUnit pu v x t, EndpointProblem pu v t) =>
StateT (UnitTestState pu v x) IO r

evalUnitTestState name st alg = evalStateT alg (UnitTestState name st [] [])

-- | Binds several provided functions to PU
assigns alg = mapM_ assign alg

-- | Binds provided function to PU
assign :: F v x -> DSLStatement pu v x t ()
assign f = do
    st@UnitTestState{unit, functs} <- get
    case tryBind f unit of
        Right unit_ -> put st{unit = unit_, functs = f : functs}
        Left err -> lift $ assertFailure $ "assign: " <> err

{- | Store several provided functions and its initial values
for naive coSimulation
-}
assignsNaive alg cntxs = mapM_ (`assignNaive` cntxs) alg

{- | Store provided function and its initial values
for naive coSimulation
-}
assignNaive f cntxs = do
    st@UnitTestState{functs, cntxCycle} <- get
    put st{functs = f : functs, cntxCycle = cntxs <> cntxCycle}

-- | set initital values for coSimulation input variables
setValues :: (Function f String, WithFunctions pu f) => [(String, x)] -> DSLStatement pu String x t
()
setValues = mapM_ (uncurry setValue)

-- | set initital value for coSimulation input variables
setValue :: (Function f String, WithFunctions pu f) => String -> x -> DSLStatement pu String x t ()
setValue var val = do
    pu@UnitTestState{cntxCycle, unit} <- get
    when (var `elem` map fst cntxCycle) $
        lift $ assertFailure $ "The variable '" <> show var <> "' is already set!"
    unless (isVarAvailable var unit) $
        lift $ assertFailure $ "It's not possible to set the variable '" <> show var <> "'! It's not
present in process"
    put pu{cntxCycle = (var, val) : cntxCycle}
    where
        isVarAvailable v pu = S.isSubsetOf (S.fromList [v]) $ inpVars $ functions pu
```

```
-- | Make synthesis decision with provided Endpoint Role and automatically assigned time
decide :: EndpointRole v -> DSLStatement pu v x t ()
decide role = do
    des <- epAt <$> getDecisionSpecific role
    doDecision False $ EndpointSt role des

-- | Make synthesis decision with provided Endpoint Role and manually selected interval
decideAt :: t -> t -> EndpointRole v -> DSLStatement pu v x t ()
decideAt from to role = doDecision False $ EndpointSt role (from ... to)

decideAtUnsafe :: t -> t -> EndpointRole v -> DSLStatement pu v x t ()
decideAtUnsafe from to role = doDecision True $ EndpointSt role (from ... to)

doDecision :: Bool -> EndpointSt v (Interval t) -> DSLStatement pu v x t ()
doDecision unsafe endpSt = do
    st@UnitTestState{unit} <- get
    let isAvailable = isEpOptionAvailable endpSt unit
    if unsafe || isAvailable
        then put st{unit = endpointDecision unit endpSt}
        else lift $ assertFailure $ "doDecision: such option isn't available: " <> show endpSt <> "
from " <> show (endpointOptions unit)

isEpOptionAvailable EndpointSt{epRole = role, epAt = atA} pu =
    case find (isSubroleOf role . epRole) $ endpointOptions pu of
        Nothing -> False
        Just EndpointSt{epAt = atB} ->
            atA `isSubsetOf` tcAvailable atB
                && member (width atA + 1) (tcDuration atB)

-- |Bind all functions to processor unit and decide till decisions left.
decideNaiveSynthesis :: DSLStatement pu v x t ()
decideNaiveSynthesis = do
    st@UnitTestState{unit, functs} <- get
    when (null functs) $
        lift $ assertFailure "You should assign function to do naive synthesis!"
    put st{unit = naiveSynthesis functs unit}

-- | Transforms provided variable to Target
consume = Target

-- | Transforms provided variables to Source
provide = Source . S.fromList

getDecisionSpecific :: EndpointRole v -> DSLStatement pu v x t (EndpointSt v (Interval t))
getDecisionSpecific role = do
    let s = variables role
    des <- getDecisionsFromEp
    case find (\case EndpointSt{epRole} | S.isSubsetOf s $ variables epRole -> True; _ -> False) des
of
        Just v -> return $ endpointOptionToDecision v
        Nothing -> lift $ assertFailure $ "Can't provide decision with variable: " <> show s

getDecisionsFromEp :: DSLStatement pu v x t [EndpointSt v (TimeConstraint t)]
getDecisionsFromEp = do
    UnitTestState{unit} <- get
    case endpointOptions unit of
        [] -> lift $ assertFailure "Failed during decision making: there is no decisions left!"
        opts -> return opts

-- | Breaks loop on PU by using breakLoopDecision function
breakLoop :: BreakLoopProblem pu v x => x -> v -> [v] -> DSLStatement pu v x t ()
breakLoop x i o = do
    st@UnitTestState{unit} <- get
    case breakLoopOptions unit of
        [] -> lift $ assertFailure "Break loop function is not supported for such type of PU"
        _ -> put st{unit = breakLoopDecision unit BreakLoop{loopX = x, loopO = S.fromList o, loopI =
i}}

assertBindFullness :: (Function f v, WithFunctions pu f, Show f) => DSLStatement pu v x t ()
assertBindFullness = do
    UnitTestState{unit, functs} <- get
    isOk <- lift $ isFullyBinded unit functs
    unless isOk $
        lift $ assertFailure $ "Function is not binded to process! expected: " ++ concatMap show
functs ++ "; actual: " ++ concatMap show (functions unit)
```

```
assertAllEndpointRoles :: (Show (EndpointRole v)) => [EndpointRole v] -> DSLStatement pu v x t ()
assertAllEndpointRoles roles = do
    UnitTestState{unit} <- get
    let opts = S.fromList $ map epRole $ endpointOptions unit
    lift $ assertBool ("Actual endpoint roles: " <> show opts) $ opts == S.fromList roles

assertEndpoint :: t -> t -> EndpointRole v -> DSLStatement pu v x t ()
assertEndpoint a b role = do
    UnitTestState{unit} <- get
    let opts = endpointOptions unit
        ep = EndpointSt role (a ... b)
    case find (\EndpointSt{epAt, epRole} -> tcAvailable epAt == (a ... b) && epRole == role) opts of
        Nothing -> lift $ assertFailure $ "assertEndpoint: " <> show ep <> " not defined in: " <>
show opts
        Just _ -> return ()

isFullyBinded pu fs = do
    assertBool ("Outputs not equal, expected: " <> show fOuts <> "; actual: " <> show outs) $ outs
== fOuts
    assertBool ("Inputs not equal, expected: " <> show fInps <> "; actual: " <> show inps) $ inps ==
fInps
    return $ not $ null fu
    where
        fu = functions pu
        outs = S.fromList $ map outputs fu
        inps = S.fromList $ map inputs fu
        fOuts = S.fromList $ map outputs fs
        fInps = S.fromList $ map inputs fs

assertSynthesisDone :: DSLStatement pu v x t ()
assertSynthesisDone = do
    UnitTestState{unit, functs, testName} <- get
    unless (isProcessComplete unit functs && null (endpointOptions unit)) $
        lift $ assertFailure $ testName <> " Process is not done: " <> incompleteProcessMsg unit
functs

assertLocks :: (Locks pu v) => [Lock v] -> DSLStatement pu v x t ()
assertLocks expectLocks = do
    UnitTestState{unit} <- get
    let actualLocks0 = locks unit
        actualLocks = S.fromList actualLocks0
    lift $ assertBool ("assertLocks: locks contain duplicates: " <> show actualLocks0) $ length ac-
tualLocks0 == S.size actualLocks
    lift $ assertBool ("assertLocks: expected locks: " <> show expectLocks <> " actual: " <> show
actualLocks0) $ actualLocks == S.fromList expectLocks

assertCoSimulation ::
    ( PUClasses pu String x Int
    , WithFunctions pu (F String x)
    , Testable pu String x
    , DefaultX pu x
    ) =>
    DSLStatement pu String x Int ()
assertCoSimulation =
    let checkInputVars pu fs cntx = S.union (inpVars $ functions pu) (inpVars fs) == S.fromList (map
fst cntx)
     in do
            UnitTestState{unit, functs, testName, cntxCycle} <- get
            unless (checkInputVars unit functs cntxCycle) $
                lift $ assertFailure "you forgot to set initial values before coSimulation."

            report@TestbenchReport{tbStatus} <-
                lift $ puCoSim testName unit cntxCycle functs False
            unless tbStatus $
                lift $ assertFailure $ "coSimulation failed: \n" <> show report

inpVars fs = unionsMap inputs fs
tracePU = do
    UnitTestState{unit} <- get
    lift $ putStrLn $ "PU: " <> show unit
    return ()

traceFunctions = do
    UnitTestState{functs} <- get
    lift $ putStrLn $ "Functions: " <> show functs
    return ()
```

```
traceEndpoints = do
    UnitTestState{unit} <- get
    lift $ do
        putStrLn "Endpoints:"
        mapM_ (\ep -> putStrLn $ "- " <> show ep) $ endpointOptions unit
    return ()

traceProcess = do
    UnitTestState{unit} <- get
    lift $ putStrLn $ "Process: " <> show (process unit)
    return ()
```

Приложение Б. Исходный код модуля проверки целостности (обязательное)

```
tests =
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}

{- |
Module      : NITTA.Model.IntegrityCheck
Description : Module for checking PU model description consistency
Copyright   : (c) Artyom Kostyuchik, 2021
License     : BSD3
Maintainer  : aleksandr.penskoi@gmail.com
Stability   : experimental
-}
module NITTA.Model.IntegrityCheck (
    ProcessConsistent (..),
) where

import Control.Monad
import Data.Data
import Data.Either
import qualified Data.Map.Strict as M
import Data.Maybe
import qualified Data.Set as S
import NITTA.Intermediate.Functions
import NITTA.Intermediate.Types
import NITTA.Model.Networks.Bus (BusNetwork, Instruction (Transport))
import NITTA.Model.ProcessorUnits
import NITTA.Utils
import NITTA.Utils.ProcessDescription

class ProcessConsistent u where
    checkProcessConsistent :: u -> Either String ()

instance {-# OVERLAPS #-} (ProcessorUnit (pu v x t) v x2 t2) => ProcessConsistent (pu v x t) where
    checkProcessConsistent pu =
        let isConsistent =
                [ checkEndpointToIntermidiateRelation (getEpMap pu) (getInterMap pu) M.empty pu
                , checkInstructionToEndpointRelation (getInstrMap pu) (getEpMap pu) $ process pu
                , checkCadToFunctionRelation (getCadFunctionsMap pu) (getCadStepsMap pu) pu
                ]
         in checkResult isConsistent

instance  {-# INCOHERENT #-} (ProcessorUnit (pu v x t) v x2 t2, UnitTag (pu v x t)) => ProcessCon-
sistent (BusNetwork (pu v x t) v x2 t2) where
    checkProcessConsistent pu =
        let isConsistent =
                [ checkEndpointToIntermidiateRelation (getEpMap pu) (getInterMap pu) (get-
TransportMap pu) pu
                , checkInstructionToEndpointRelation (getInstrMap pu) (getEpMap pu) $ process pu
                , checkCadToFunctionRelation (getCadFunctionsMap pu) (getCadStepsMap pu) pu
                ]
         in checkResult isConsistent
    checkProcessConsistent pu =
        let isConsistent = [Left "Trying to run BusNetwork"]
         in checkResult isConsistent

checkResult res =
    if any isLeft res
        then Left $ concat $ lefts res
        else Right ()

checkEndpointToIntermidiateRelation eps ifs trans pu =
    let checkIfsEmpty = M.size eps > 0 && M.size ifs == 0
```

```haskell
        checkEpsEmpty = M.size ifs > 0 && M.size eps == 0
        rels = S.fromList $ filter isVertical $ relations $ process pu
        lookup' v = fromMaybe (showError "Endpoint to Intermidiate" "enpoint" v pu) $ eps M.!? v
        makeRelationList =
            map S.fromList $
                concatMap
                    ( \(h, f) ->
                        sequence $
                            concatMap
                                ( \v -> [[Vertical h $ fst p | p <- lookup' v]]
                                )
                                $ variables f
                    )
                    $ M.toList ifs
     in do
            when checkEpsEmpty $
                Left "endpoints are empty"
            when checkIfsEmpty $
                Left "functions are empty"
            if any (`S.isSubsetOf` rels) makeRelationList
                then Right True
                else checkTransportToIntermidiateRelation ifs rels trans pu

checkTransportToIntermidiateRelation ifs rels transMap pu =
    let lookup' v = fromMaybe (showError "Transport to Intermidiate" "transport" v pu) $ transMap
M.!? v
        makeRelationList =
            map S.fromList $
                concatMap
                    ( \(h, f) ->
                        concatMap
                            ( \v -> [[Vertical h $ fst $ lookup' v]]
                            )
                            $ variables f
                    )
                    $ M.toList ifs
     in if any (`S.isSubsetOf` rels) makeRelationList
            then Right True
            else Left "Endpoint and Transport to Intermideate (function) not consistent"

checkInstructionToEndpointRelation ins eps pr =
    let checkInsEmpty = M.size eps > 0 && M.size ins == 0
        checkEpsEmpty = M.size ins > 0 && M.size eps == 0
        eps' = M.fromList $ concat $ M.elems eps
        rels = S.fromList $ map (\(Vertical r1 r2) -> (r1, r2)) $ filter isVertical $ relations pr
        consistent =
            and $
                concatMap
                    ( \(r1, r2) -> case eps' M.!? r1 of
                        Just _ | Just (InstructionStep _) <- ins M.!? r2 -> [True]
                        _ -> []
                    )
                    rels
     in do
            when checkInsEmpty $ Left "instructions are empty"
            when checkEpsEmpty $ Left "enpoints are empty"
            if consistent
                then Right True
                else Left "Instruction to Endpoint not consistent"

-- now it checks LoopBegin/End
checkCadToFunctionRelation cadFs cadSteps pu =
    let consistent = S.isSubsetOf makeCadVertical rels
        rels = S.fromList $ filter isVertical $ relations $ process pu
        showLoop f = "bind " <> show f
        lookup' v = fromMaybe (showError "CAD" "steps" v pu) $ cadSteps M.!? v
        makeCadVertical =
            S.fromList $
                concatMap
                    ( \(h, f) ->
                        concatMap
                            ( \v -> [uncurry Vertical (lookup' v, h)]
                            )
                            [showLoop f]
                    )
                    $ M.toList cadFs
```

```
     in if consistent
            then Right True
            else Left $ "CAD functions not consistent. Excess:" <> show (S.difference makeCadVerti-
cal rels)

getInterMap pu =
    M.fromList
        [ (pID, f)
        | step@Step{pID} <- steps $ process pu
        , isFB step
        , f <- case getFunction step of
            Just f -> [f]
            _ -> []
        ]

getEpMap pu =
    M.fromListWith (++) $
        concat
            [ concatMap (\v -> [(v, [(pID, ep)])]) $ variables ep
            | step@Step{pID} <- steps $ process pu
            , isEndpoint step
            , ep <- case getEndpoint step of
                Just e -> [e]
                _ -> []
            ]

getInstrMap pu =
    M.fromList
        [ (pID, instr)
        | step@Step{pID} <- steps $ process pu
        , isInstruction step
        , instr <- case getInstruction step of
            Just i -> [i]
            _ -> []
        ]

getTransportMap pu =
    let getTransport :: (Typeable a, Typeable v, Typeable x, Typeable t) => pu v x t -> a -> Maybe
(Instruction (BusNetwork String v x t))
        getTransport _ = cast
        filterTransport pu' pid (InstructionStep ins)
            | Just instr@(Transport v _ _) <- getTransport pu' ins = Just (v, (pid, instr))
            | otherwise = Nothing
        filterTransport _ _ _ = Nothing
     in M.fromList $ mapMaybe (uncurry $ filterTransport pu) $ M.toList $ getInstrMap pu

getCadFunctionsMap pu =
    let filterCad (_, f)
            | Just Loop{} <- castF f = True
            | Just (LoopBegin Loop{} _) <- castF f = True
            | Just (LoopEnd Loop{} _) <- castF f = True
            | otherwise = False
     in M.fromList $ filter filterCad $ M.toList $ getInterMap pu

getCadStepsMap pu =
    M.fromList
        [ (pDesc', pID)
        | step@Step{pID} <- steps $ process pu
        , pDesc' <- case getCAD step of
            Just msg -> [msg]
            _ -> []
        ]

showError name mapName v pu =
    error $
        name
            <> " relations contain error: "
            <> show v
            <> " is not present in "
            <> mapName
            <> " map."
            <> "proc: "
            <> show (process pu)
```

Приложение В. Исходный код реализованных тестов (обязательное)

```
tests =
    testGroup [
      ...
      puUnitTestCase "multiplier smoke test" u $ do
            assign $ multiply "a" "b" ["c", "d"]
            assertBindFullness
            decideAt 1 2 $ consume "a"
            decide $ consume "b"
            decideAt 5 5 $ provide ["c"]
            decide $ provide ["d"]
            assertSynthesisDone
        , puUnitTestCase "multiplier coSim smoke test" u $ do
            assign $ multiply "a" "b" ["c", "d"]
            setValue "a" 2
            setValue "b" 7
            decide $ consume "a"
            decide $ consume "b"
            decide $ provide ["c", "d"]
            assertCoSimulation
        , expectFail $
            puUnitTestCase "coSim test should fail because synthesis not complete" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                setValues [("a", 2), ("b", 7)]
                decide $ consume "b"
                assertCoSimulation
        , expectFail $
            puUnitTestCase "should error, when proccess is not done" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                decideAt 1 2 $ consume "a"
                assertSynthesisDone
        , expectFail $
            puUnitTestCase "should not bind, when PU incompatible with F" u $ do
                assign $ sub "a" "b" ["c"]
        , expectFail $
            puUnitTestCase "decide should error, when Target in Decision is not present" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                decideAt 1 1 $ consume "aa"
        , expectFail $
            puUnitTestCase "Multiplier should error, when Source in Decision is Targets" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                decideAt 1 1 $ provide ["a"]
        , expectFail $
            puUnitTestCase "decide should error, when Target in Decision is Source" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                decide $ consume "a"
                decide $ consume "b"
                decideAt 4 4 $ consume "c"
        , expectFail $
            puUnitTestCase "decide should error, when Interval is not correct" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                decideAt 2 2 $ consume "a"
                decideAt 1 1 $ consume "b"
        , expectFail $
            puUnitTestCase "should error: breakLoop is not supportd" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                breakLoop 10 "a" ["c"]
        , expectFail $
            puUnitTestCase "should error: setValue variable is unavailable" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                setValue "e" 10
        , expectFail $
            puUnitTestCase "should error: setValue variable is unavailable" u $ do
                assign $ multiply "a" "b" ["c", "d"]
                setValue "a" 10
                setValue "b" 11
                setValue "a" 15
                assertSynthesisDone -- to force evaluation
```

```
...
, puUnitTestCase "accum smoke test" accumDef $ do
            assign $ sub "a" "b" ["c"]
            assertBindFullness
            decide $ consume "a"
            decide $ consume "b"
            decide $ provide ["c"]
            assertSynthesisDone
...
, puUnitTestCase "test BreakLoop" u2 $ do
            assign $ loop 10 "b" ["a"]
            setValue "b" 64
            breakLoop 10 "b" ["a"]
            decideAt 1 1 $ provide ["a"]
            decideAt 2 2 $ consume "b"
            traceProcess
            assertCoSimulation
...
, puUnitTestCase "division simple" u2 $ do
            assign $ division "a" "b" ["c"] ["d"]
            setValue "a" 64
            setValue "b" 12
            decideAt 1 1 $ consume "a" -- but I would like to write: decide $ consume "a"
            decideAt 2 2 $ consume "b"
            decideAt 10 10 $ provide ["c"]
            decideAt 11 11 $ provide ["d"]
            assertCoSimulation
, puUnitTestCase "division only mod" u2 $ do
            assign $ division "a" "b" ["c"] []
            setValue "a" 64
            setValue "b" 12
            decideAt 1 1 $ consume "a"
            decideAt 2 2 $ consume "b"
            decideAt 10 10 $ provide ["c"]
            assertCoSimulation
, puUnitTestCase "division only rem" u2 $ do
            assign $ division "a" "b" [] ["d"]
            setValue "a" 64
            setValue "b" 12
            decideAt 1 1 $ consume "a"
            decideAt 2 2 $ consume "b"
            decideAt 11 11 $ provide ["d"]
            assertCoSimulation
, puUnitTestCase "division success pipeline" u2 $ do
            assign $ division "a" "b" ["c"] []
            assign $ division "e" "f" ["g"] []
            setValues [("a", 64), ("b", 12), ("e", 64), ("f", 12)]
            decideAt 1 1 $ consume "a"
            decideAt 2 2 $ consume "b"
            decideAt 3 3 $ consume "e"
            decideAt 4 4 $ consume "f"
            decideAt 7 7 $ provide ["c"]
            decideAt 9 9 $ provide ["g"]
            assertCoSimulation
]
...
, testGroup
            "broken relations integrity check negative base"
            [ expectFail $ finitePUSynthesisProp "finitePUSynthesisProp lost instr and ep"
u{lostInstructionRelation = True, lostEndpointRelation = True} fsGen
            , expectFail $ finitePUSynthesisProp "finitePUSynthesisProp lost Endpoints"
u{lostEndpointRelation = True} fsGen
            , expectFail $ finitePUSynthesisProp "finitePUSynthesisProp lost Instruction"
u{lostInstructionRelation = True} fsGen
            , expectFail $ finitePUSynthesisProp "finitePUSynthesisProp lost Function"
u{lostFunctionRelation = True} fsGen
            , expectFail $ puCoSimProp "puCoSimProp lost Endpoints" u{lostEndpointRelation = True}
fsGen
            , expectFail $ puCoSimProp "puCoSimProp lost Instruction" u{lostInstructionRelation =
True} fsGen
            , expectFail $ puCoSimProp "puCoSimProp lost Function" u{lostFunctionRelation = True}
fsGen
            , expectFail $ puCoSimTestCase "puCoSimTestCase lost Endpoints" u{lostEndpointRelation =
True} [("a", 42)] [brokenBuffer "a" ["b"]]
            , expectFail $ puCoSimTestCase "puCoSimTestCase lost Instructions"
u{lostInstructionRelation = True} [("a", 42)] [brokenBuffer "a" ["b"]]
```

```
            , expectFail $ puCoSimTestCase "puCoSimTestCase lost Function" u{lostFunctionRelation =
True} [("a", 42)] [brokenBuffer "a" ["b"]]
            ]
        , testGroup
            "broken relations integrity check negative coSim"
            [ expectFail $ nittaCoSimTestCase "nittaCoSimTestCase lost Endpoints" (maBroken
u{lostEndpointRelation = True}) alg
            , expectFail $ nittaCoSimTestCase "nittaCoSimTestCase lost Instruction" (maBroken
u{lostInstructionRelation = True}) alg
          , expectFail $ nittaCoSimTestCase "nittaCoSimTestCase lost Function" (maBroken
u{lostFunctionRelation = True}) alg
            , expectFail $ typedLuaTestCase (maBroken def{lostEndpointRelation = True}) pInt
"typedLuaTestCase lost Endpoints" lua
            , expectFail $ typedLuaTestCase (maBroken def{lostInstructionRelation = True}) pInt
"typedLuaTestCase lost Instruction" lua
        , expectFail $ typedLuaTestCase (maBroken def{lostFunctionRelation = True}) pInt
"typedLuaTestCase lost Function" lua
            ]
```

Приложение Г. Исходный код примера doctest (необязательное)

```
> :module +NITTA.Model.Types NITTA.Intermediate.Functions Numeric.Interval.NonEmpty Data.Set
> :set prompt "ESC[34mλ> ESC[m"
@
Now create the function and multiplier model initial state. Unfortunately, it
is not enough information for GHC deduction of its type, so let's define its
implicitly.
>>> let f = F.multiply "a" "b" ["c", "d"] :: F String Int
>>> f
a * b = c = d
>>> let st0 = multiplier True :: Multiplier String Int Int
>>> st0
Multiplier {remain = [], targets = [], sources = [], currentWork = Nothing, process_ = Process
    steps =
<BLANKLINE>
    relations =
<BLANKLINE>
    nextTick = 0
    nextUid = 0
, isMocked = True}
>>> endpointOptions st0
[]
Bind a function to the multiplier unit. This operation could be executed at
any time of working with a model, including when a computation process is
fully scheduled (new work can be added). The main rules are: 1) if work is
fully scheduled, then it is necessary to perform it, and any part of it
cannot be "lost" inside the model; 2) if a unit has its internal resources,
there should be enough to finish schedule, even it is inefficient.
>>> let Right st1 = tryBind f st0
>>> st1
Multiplier {remain = [a * b = c = d], targets = [], sources = [], currentWork = Nothing, process_ =
Process
    steps =
<BLANKLINE>
    relations =
<BLANKLINE>
    nextTick = 0
    nextUid = 0
, isMocked = True}
>>> endpointOptions st1
[?Target "a"@(0..∞ /P 1..∞),?Target "b"@(0..∞ /P 1..∞)]
As we can see, after binding, we have two different options of computational
process scheduling that match different argument loading sequences: @a@ or
@b@. We can see that they are similar from an execution sequence point of
view: loading can be started from 0 tick or after an arbitrary delay; for
loading of one argument needed only one tick, but it can continue for an
arbitrary time. Choose the variant.
>>> let st2 = endpointDecision st1 $ EndpointSt (Target "a") (0...2)
>>> st2
Multiplier {remain = [], targets = ["b"], sources = ["c","d"], currentWork = Just a * b = c = d,
process_ = Process
    steps =
        0) Step {pID = 0, pInterval = 0 ... 2, pDesc = Endpoint: Target a}
        1) Step {pID = 1, pInterval = 0 ... 2, pDesc = Instruction: Load A}
    relations =
        0) Vertical 0 1
    nextTick = 3
    nextUid = 2
, isMocked = True}
>>> mapM_ print $ endpointOptions st2
?Target "b"@(3..∞ /P 1..∞)
>>> let st3 = endpointDecision st2 $ EndpointSt (Target "b") (3...3)
>>> st3
Multiplier {remain = [], targets = [], sources = ["c","d"], currentWork = Just a * b = c = d, pro-
cess_ = Process
    steps =
        0) Step {pID = 0, pInterval = 0 ... 2, pDesc = Endpoint: Target a}
        1) Step {pID = 1, pInterval = 0 ... 2, pDesc = Instruction: Load A}
```

```
        2) Step {pID = 2, pInterval = 3 ... 3, pDesc = Endpoint: Target b}
        3) Step {pID = 3, pInterval = 3 ... 3, pDesc = Instruction: Load B}
    relations =
        0) Vertical 2 3
        1) Vertical 0 1
    nextTick = 4
    nextUid = 4
, isMocked = True}
>>> mapM_ print $ endpointOptions st3
?Source "c","d"@(6..∞ /P 1..∞)
After loading both arguments, we can see that the next option is unloading
@c@ and @d@ variables. Note, these variables can be unloaded either
concurrently or sequentially (for details, see how the multiplier works
inside). Consider the second option:
>>> let st4 = endpointDecision st3 $ EndpointSt (Source $ S.fromList ["c"]) (6...6)
>>> st4
Multiplier {remain = [], targets = [], sources = ["d"], currentWork = Just a * b = c = d, process_ =
Process
    steps =
        0) Step {pID = 0, pInterval = 0 ... 2, pDesc = Endpoint: Target a}
        1) Step {pID = 1, pInterval = 0 ... 2, pDesc = Instruction: Load A}
        2) Step {pID = 2, pInterval = 3 ... 3, pDesc = Endpoint: Target b}
        3) Step {pID = 3, pInterval = 3 ... 3, pDesc = Instruction: Load B}
        4) Step {pID = 4, pInterval = 6 ... 6, pDesc = Endpoint: Source c}
        5) Step {pID = 5, pInterval = 6 ... 6, pDesc = Instruction: Out}
    relations =
        0) Vertical 4 5
        1) Vertical 2 3
        2) Vertical 0 1
    nextTick = 7
    nextUid = 6
, isMocked = True}
>>> mapM_ print $ endpointOptions st4
?Source "d"@(7..∞ /P 1..∞)
>>> let st5 = endpointDecision st4 $ EndpointSt (Source $ S.fromList ["d"]) (7...7)
>>> st5
Multiplier {remain = [], targets = [], sources = [], currentWork = Nothing, process_ = Process
    steps =
        0) Step {pID = 0, pInterval = 0 ... 2, pDesc = Endpoint: Target a}
        1) Step {pID = 1, pInterval = 0 ... 2, pDesc = Instruction: Load A}
        2) Step {pID = 2, pInterval = 3 ... 3, pDesc = Endpoint: Target b}
        3) Step {pID = 3, pInterval = 3 ... 3, pDesc = Instruction: Load B}
        4) Step {pID = 4, pInterval = 6 ... 6, pDesc = Endpoint: Source c}
        5) Step {pID = 5, pInterval = 6 ... 6, pDesc = Instruction: Out}
        6) Step {pID = 6, pInterval = 7 ... 7, pDesc = Endpoint: Source d}
        7) Step {pID = 7, pInterval = 7 ... 7, pDesc = Instruction: Out}
        8) Step {pID = 8, pInterval = 0 ... 7, pDesc = Intermediate: a * b = c = d}
    relations =
        0) Vertical 8 6
        1) Vertical 8 4
        2) Vertical 8 2
        3) Vertical 8 0
        4) Vertical 6 7
        5) Vertical 4 5
        6) Vertical 2 3
        7) Vertical 0 1
    nextTick = 8
    nextUid = 9
, isMocked = True}
>>> endpointOptions st5
[]
```