МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра *Автоматики и вычислительной техники*
(полное название кафедры)

Утверждаю

Зав. кафедрой *текст*

текст
(подпись, инициалы, фамилия)

«___» _____ 201__ г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего образования

09.04.01 Информатика и вычислительная техника
(код и наименование направления подготовки магистра)

Автоматики и вычислительной техники
(факультет)

Трибунский Владислав Алексеевич
(фамилия, имя, отчество студента – автора работы)

Разработка и исследование унифицированного представления событийно-непрерывных моделей
(полное название темы магистерской диссертации)

Руководитель от НГТУ
Достовалов Дмитрий Николаевич
(фамилия, имя, отчество)
к.т.н.
(ученая степень, ученое звание)

Автор выпускной квалификационной работы
Трибунский Владислав Алексеевич
(фамилия, имя, отчество)

Новосибирск 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра _____ *Автоматики и вычислительной техники* _____
(полное название кафедры)

Утверждаю

текст

Зав. кафедрой _____

текст

_____
(подпись, инициалы, фамилия)

«___» _____ 201__ г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

студенту _____ *Трибунскому Владиславу Алексеевичу* _____
(фамилия, имя, отчество)

факультета _____ *Автоматики и вычислительной техники* _____
(полное название факультета)

Направление подготовки _____ *09.04.01 Информатика и вычислительная техника* _____
(код и наименование направления подготовки магистра)

_____

_____

Магистерская программа _____
(наименование магистерской программы)

_____

_____

Тема _____
(полное название темы)

_____

_____

_____

Цели работы _____

_____

_____

Задание согласовано и принято к исполнению.

Руководитель	*Студент*
от НГТУ	
Достовалов Дмитрий	*Трибунский Владислав*
Николаевич	*Алексеевич*
(фамилия, имя, отчество)	(фамилия, имя, отчество)
к.т.н.	*АВТФ, АСМ-23*
(ученая степень, ученое звание)	(факультет, группа)

Тема утверждена приказом по НГТУ № _____ от «____» _____ 202__г.

изменена приказом по НГТУ № _____ от «____» _____ 202__ г.

Диссертация сдана в ГЭК № _____, тема сверена с данными приказа

_____

(подпись секретаря государственной экзаменационной комиссии по защите ВКР, дата)

_____

(фамилия, имя, отчество секретаря государственной экзаменационной комиссии по защите ВКР)

Аннотация

Объектом исследования являются современные гибридные (событийно-непрерывные) динамические системы, а предметом — архитектура единой симуляционной системы для их моделирования. Цель работы — разработать такую архитектуру симулятора, которая обеспечивает интеграцию дискретных (событийных) и непрерывных частей гибридной модели, единый подход к обмену данными между компонентами и использование стандартизованных форматов представления моделей.

Для достижения цели проведён анализ существующих подходов и средств моделирования гибридных систем (фреймворки Modelica, Simulink/Stateflow, DEVS, протоколы ко-симуляции FMI, HLA и др.), формализованы математические представления гибридных систем (в частности, гибридные автоматы) и выделены ключевые паттерны взаимодействия дискретной логики и непрерывной динамики. На этой основе спроектирована концепция многоуровневой модульной архитектуры симуляционной системы: определены независимые компоненты (движок интеграции дифференциальных уравнений, движок событий и синхронизатор времени), обеспечивающие корректный порядок обработки переходов. Для описания моделей выбран человекочитаемый формат JSON, а симулятор реализован на языке C с использованием лёгких библиотек парсинга (например, cJSON).

Научная новизна работы заключается в комплексном объединении современных технологий и подходов гибридного моделирования. Предложен новый способ задания гибридной модели как конечного автомата с внутренней непрерывной динамикой в человекочитаемом формате JSON, что упрощает интеграцию моделей и позволяет изменять их структуру во время выполнения. Доказано, что симулятор, реализованный на языке C, обеспечивает высокую производительность

и предсказуемое поведение без накладных расходов виртуальных машин, в отличие от существующих сред. Модульная многослойная архитектура обеспечивает расширяемость: компоненты системы (например, интеграторы и обработчики событий) могут заменяться без изменения общей структуры. В совокупности предложено новое технологическое решение для гибридной симуляции, сочетающее преимущества разных подходов и преодолевающее их ограничения.

Практическая значимость результатов заключается в том, что разработанная архитектура симулятора позволяет разработчикам САПР и систем управления объединять разнородные модели без потери точности, повторно использовать программные компоненты и обеспечивать работу в реальном времени. Представленные решения актуальны для задач цифровых двойников, кибер-физических систем, робототехники и управления энергосетями, где требуется надёжное гибридное моделирование. Экспериментальная проверка прототипа демонстрирует применимость предложенного подхода, подтверждая эффективность и преимущества разработанной системы.

Ключевые слова: гибридные динамические системы; симуляционная система; архитектура симулятора; модульная архитектура; формат JSON; язык программирования C; гибридный автомат; ко-симуляция.

Оглавление

Введение

Актуальность исследования

Современные инженерные системы всё чаще представляют собой гибридные динамические системы, сочетающие непрерывную эволюцию состояний (описанную дифференциальными уравнениями) и дискретные переходы (событийную логику)[4][6]. В таких системах дискретные решения и прерывистые события тесно взаимодействуют с непрерывными динамическими процессами. Например, в системах управления технологическими процессами часто приходится комбинировать непрерывные модели динамики оборудования с дискретными алгоритмами управления и логикой переключений. Согласно обзору Labinaz, Bayoumi и Rudie (1997), «гибридные системы демонстрируют совокупность непрерывных и дискретных поведений, что требует использования различных формализмов моделирования»[1]. Классические примеры гибридных систем – это, например, термостатический регулятор отопления: температура помещения изменяется непрерывно, а переключение котла «включено/выключено» происходит дискретно при достижении определённых порогов [6]. Goebel и соавт. отмечают, что большинство реальных технических систем можно рассматривать как гибридные: «автомобили, компьютеры, самолёты, стиральные машины – нет недостатка в примерах» подобных систем[2]. С другой стороны, литературные источники подчёркивают, что до сих пор моделирование гибридных систем часто «завалено заплаточными, ad hoc решениями» [6], что затрудняет системный подход.

В этой связи разработка формальных архитектурных подходов и унифицированных сред для моделирования гибридных систем представляется актуальной задачей. Развитие кибер-физических систем и моделей цифровых двойников требует комплексного учёта как

непрерывной физики, так и логики управления. Традиционные симуляторы непрерывных систем (например, на базе дифференциальных уравнений) и симуляторы дискретных событий (например, DEVS-модель или Simulink Stateflow) часто функционируют разрозненно и плохо интегрируются между собой. Поэтому возрастающая сложность и взаимодействие подсистем разных типов обуславливают необходимость единой архитектуры симуляции. Например, в одном исследовании отмечается, что в области управленческого моделирования гибридная симуляция определяется как сочетание двух или более методов симуляции (дискретно-событийного, агентного, системной динамики и т.д.)[3]. Это свидетельствует о тенденции связывать разные симуляторы для решения комплексных задач. Однако наличие множества параллельных решений требует выработки единого подхода к сопряжению моделей и обмену данными. Одним из направлений является разработка стандартов для обмена моделями, таких как FMI (Functional Mock-up Interface) – единый формат контейнера, который позволяет объединять разнородные модели в процессе совместной симуляции[21].

Таким образом, актуальность исследования заключается в необходимости создания унифицированной архитектуры симуляционной системы для гибридных (событийно-непрерывных) моделей, что позволит объединять различные модели и симуляторы без потери точности и масштабируемости. Научная литература подтверждает рост интереса к таким решениям: гибридные модели применяются в робототехнике, системах управления энергосетями, автомобильной электронике и других критически важных приложениях[4][6]. При этом известно, что несогласованные подходы к моделированию гибридных систем (часто реализуемые «налету») могут приводить к ошибкам – знаменитый пример – авария ракеты Ariane 5, где программная логика, надёжно работавшая на Ariane 4, дала сбой в

новых непрерывных условиях полёта[6]. Это подчёркивает научную и практическую значимость разработки формальных, надёжных архитектур симуляторов гибридных систем.

Примеры гибридных систем. Рассмотрим несколько наглядных примеров:

- Масса на пружине (модель колебаний с прерывистыми событиями). Простая механическая система «масса–пружина–демпфер» обычно описывается непрерывными дифференциальными уравнениями колебаний. Однако при наличии, скажем, твёрдого упора или прерывания контакта с поверхностью система становится гибридной: когда масса касается упора, происходит дискретный скачок в условиях движения (отбой), затем непрерывное движение продолжается [5]. Подобная модель может использоваться в робототехнике или сейсмологии для учёта столкновений.

Непрерывная модель (механический осциллятор “масс–пружина–демпфер”):

Обозначим x(t) – смещение массы от равновесного положения, $v(t) = \dot{x}(t)$ – её скорость. Закон Ньютона даёт второе — порядковое ОДУ:

$$m\ddot{x}(t) + c\dot{x}(t) + k\,x(t) = 0,$$

Где m>0 – масса, $c \geq 0$ – коэффициент демпфирования, k>0 – жёсткость пружины. В форме состояния $[x_1, x_2]^\top = [x, v]^\top$:

$$\begin{cases} \dot{x_1}(t) = x_2(t), \\ \dot{x_2}(t) = -\frac{c}{m}\,x_2(t) - \frac{k}{m}\,x_1(t) \end{cases},$$

с начальными условиями $x_1(0) = x_0, \quad x_2(0) = v_0.$

Дискретный переход (удара о твёрдый упор):

Пусть упор расположен в точке x = d (обычно d=0). При приближении к упору со скоростью вниз ($x_1 = d$ и $x_2 \leq 0$) происходит мгновенный отскок с учётом коэффициента восстановления

$e \in (0,1)$ (коэффициент упругости столкновения). Правило сброса:

$$x_1(t^+) = d, \quad x_2(t^+) = -e\,x_2(t^-)$$

Здесь t^- и t^+ – моменты сразу до и сразу после столкновения, соответственно

- Два взаимосвязанных бака (жидкостная система с переключениями). Классическая задача двухбаковой системы с перетоком жидкости широко применяется как учебный и исследовательский пример. При изменении положения клапанов или переполнении баков режим течения меняется дискретно, тогда как уровни жидкости и потоки описываются непрерывной динамикой. Например, есть исследования, которые показывают, что двухбаковая система с четырьмя режимами работы служит типовым гибридным эталоном[4]. Модель может описываться в виде гибридного автомата, где в каждом дискретном состоянии (режиме) жидкость подчиняется своим дифференциальным уравнениям.

Если через A_1 и A_2 обозначить площади оснований баков, то система уравнений для уровней воды в баках h_1 и h_2 будет иметь вид

$$h'_1 = {A_1}^{-1}(V_{input} - V_{12}),$$

$$h'_2 = {A_2}^{-1}(V_{12} - V_{out}).$$

Скорость протекания воды между баками V_{12} зависит от уровней, и в соответствии с законом Торричелли

$$V_{12} = \left\{K_1(p_1)\sqrt{h_1 - h_2 + H}, h_2 > H, K_1(p_1)\sqrt{h_1}, h_2 \le H\right\}.$$

Скорость вытекания воды из системы V_{out} зависит от уровня во втором баке h_2 и положения задвижки p_2 на W_2. Если контроллер обнаруживает, что уровень воды во втором баке опустился ниже значения L_{minus}, поступает команда закрыть выходной клапан W_{out},

если вода во втором баке превышает уровень L_{plus} - выдается команда открыть выходной клапан. Таким образом,

$$V_{out} = \{K_2(p_2)\sqrt{h_2}, h_2 > L_{plus}, 0, h_2 \leq L_{minus}\}.$$

Индивидуальные свойства клапанов определяются функциями

$$K_1(p_1) = \{1{,}85 * 10^{-4} e^{-6*10^{-6} {p_1}^3}, 0 \leq p_1 < 80, p_1 = 80\},$$

$$K_2(p_2) = \{2{,}26 * 10^{-4} e^{-5{,}7*10^{-6} {p_2}^3}, 0 \leq p_2 < 80, p_2 = 80\}.$$

- Отскакивающий мяч (способный к рикошету объект). Мяч, брошенный из начальной высоты, падает под действием гравитации (непрерывная динамика), а при достижении земли совершает упругий отскок (дискретный переход скорости) и поднимается вверх [5]. Это классический пример гибридной системы: между прыжками шар «течёт» по законам движения, а при столкновении с землёй мгновенно меняет скорость на противоположную с учётом коэффициента восстановления.

Непрерывная модель: Обозначим $h(t)$ – высоту мяча, $v(t)$ – вертикальную скорость. Под действием гравитации g (постоянное ускорение свободного падения) динамика даётся системой ОДУ:

$\dot{h}(t) = v(t), \dot{v}(t) = -g$, с начальными условиями $h(0) = h_0, v(0) = v_0$

Дискретный переход (отскок): При достижении мяча земли ($h(t) = 0$) происходит мгновенное изменение скорости из-за упругого отскока. Событие: h=0, v\le0. Сброс: положение остается h=0, а скорость меняет знак с учетом потерь энергии:

$$v(t^+) = -k\, v(t^-),$$

где 0<k<1 – коэффициент восстановления.

- Термостатическое управление отоплением. Как было указано выше[6], система «отопление + термостат» сочетает реальный (непрерывный) показатель температуры помещения и логическую

(дискретную) работу котла «включено/выключено». При достижении порога термостат переключает режим отопления – это дискретное событие, влияющее на непрерывную динамику температуры.

Непрерывная модель: Обозначим $T(t)$ – температуру в помещении. Уравнение теплового баланса (например, по закону Ньютона об охлаждении) с подачей тепла от отопления:

$$\dot{T}(t) = -\alpha(T(t) - T_{amb}) + \beta\, u(t),$$

где T_{amb} – постоянная температура окружающей среды, $alpha > 0$ – коэффициент теплопотерь, $beta > 0$ – мощность отопления. $u(t) \in \{0,1\}$ – логическая переменная: $u = 1$, когда отопление включено, и $u = 0$ – когда выключено. Начальное условие: $T(0) = T_0$.

Дискретное переключение: Термостат включает и выключает отопление по пороговым температурам $T_{min} < T_{max}$. Событие включения: если при $u(t^-) = 0$ температура падает до $T(t) = T_{min}$, то мгновенно устанавливаем $u(t^+) = 1$ (отопление включается). Событие выключения: если при $u(t^-) = 1$ температура достигает $T(t) = T_{max}$, то $u(t^+) = 0$ (отопление выключается). Таким образом поддерживается $T_{min} < T(t) < T_{max}$

- Системы управления транспортными средствами. В автомобиле непрерывная динамика (движение, скорость) сочетается с дискретной логикой переключения передач или работы электроники. Например, система адаптивного круиз-контроля имеет непрерывные законы движения и ограничения, а также дискретные алгоритмы переключения скоростей и блоков управления. Аналогично в железнодорожных системах движение поезда описывается непрерывно, а дискретные сигналы светофоров задают режимы движения.

Непрерывная модель: Пусть x(t) – положение автомобиля, v(t) – его скорость. В простейшей модели с постоянной тягой и линейным сопротивлением:

$$\dot{x}(t) = v(t), \quad \dot{v}(t) = a - b\,v(t)$$

где a>0 – эффективное ускорение от двигателя (например, сила тяги, делённая на массу), b>0 – коэффициент сопротивления (трения, аэродинамики). Начальные условия: $x(0) = x_0, \quad v(0) = v_0$.

Дискретное переключение передач: Введём дискретную переменную

$g(t) \in \{1,2,\dots\}$ – номер текущей передачи. Зададим пороговые скорости $V_1 < V_2 <$ ... для перехода между передачами. Правило переключения вверх: если в момент перед переходом g=i и скорость достигает $v(t^-) \geq V_i$, то дискретно выполняем $g(t^+) = i + 1$ (переключение на более высокую передачу). (При необходимости можно также задать обратные пороги для понижения передачи, например при $v(t^-) \leq V_{i-1}$ переключать g=i-1.)

- Промышленные системы с логическими управляющими элементами. На производстве часто возникают гибридные процессы: уровни жидкостей, токи, температуры меняются непрерывно, но клапаны, выключатели, программируемые контроллеры вводят дискретные изменения режимов. Например, химический реактор с автоматическими клапанами может использовать гибридную модель для учёта непрерывной химической кинетики и дискретного открывания/закрывания потоков.

Непрерывная модель: Обозначим T(t) – температуру в реакторе, $u(t) \in \{0,1\}$ – состояние клапана подачи тепла (1 – открыт, 0 – закрыт). Уравнение теплового баланса аналогично закону Ньютона

ru.wikipedia.org

$$\dot{T}(t) = -\alpha(T(t) - T_{amb}) + \beta\,u(t),$$

где T_{amb} – температура окружающей среды, α>0 – коэффициент тепловых потерь, β>0 – мощность нагрева при открытом клапане. Начальное условие: $T(0) = T_0$.

Дискретное управление клапаном: Автоматическое включение/отключение по порогам $T_{min} < T_{max}$. Событие открытия клапана: если $u(t^-) = 0$ и T(t) падает до T(t)= T_{min}, то $u(t^+) = 1$ (нагрев включается). Событие закрытия: если $u(t^-) = 1$ и T(t) достигает T(t)= T_{max}, то $u(t^+) = 0$ (нагрев выключается). Таким образом температура поддерживается между T_{min} и T_{max}.

Эти примеры подчеркивают техническую значимость гибридных моделей в прикладных задачах. Во всех них необходимо единообразно описывать и симулировать разнородные процессы. В связи с этим цель и задачи данной работы формулируются следующим образом.

Цель и задачи работы

Цель исследования – разработать унифицированную архитектуру симуляционной системы для гибридных (событийно-непрерывных) моделей. Такая архитектура должна обеспечивать интеграцию дискретных и непрерывных частей моделей, единый подход к обмену данными между компонентами и использовать стандартизованные форматы представления моделей. В частности, целью является создание концепции системы, способной эффективно моделировать сложные гибридные системы без необходимости «зашивать» разрозненные инструменты, устраняя существующие ограничения и дублирование функциональности.

Для достижения этой цели в работе поставлены следующие задачи:

Анализ литературы и существующих подходов. Рассмотреть современные подходы к моделированию гибридных систем, проанализировать ограничения традиционных средств (моделирование изолированно непрерывных или дискретных процессов) и резюмировать требования к архитектуре симулятора гибридных систем. В частности,

изучить существующие фреймворки (Modelica, Simulink/Stateflow, DEVS-ориентированные системы и др.), а также протоколы для ко-симуляции (FMI, HLA и др.)[3][6].

Формализация гибридных моделей. Определить формальные представления гибридных систем (например, гибридные автоматы) и базовые понятия, необходимые для проектирования архитектуры. Исследовать ограничения существующих формализмов: непредсказуемость приоритетов, сложность численной симуляции при переходах, проблемы точности и устойчивости. Проанализировать примеры из предыдущего раздела и выделить общие паттерны взаимодействия дискретной и непрерывной логики.

Разработка архитектурного решения. Сформулировать принципы единой архитектуры симулятора гибридных моделей, учитывая модульность, иерархичность и масштабируемость. Предусмотреть возможность использования различных вычислительных ядер (для дифференциальных уравнений и для событийных симуляций), механизм координации времени и событий, а также интеграцию стандартных интерфейсов обмена моделями (например, FMI[6][68]. Предложить структуру программной платформы (например, выделение «движка» непрерывных моделей и «движка» событий с матричным обменом данных) и схему обмена информацией между ними.

Реализация прототипа и тестирование. На основе разработанной архитектуры создать прототип системы (или её компоненты) и проверить работоспособность на выбранных примерах гибридных систем (моделях «масса–пружина», «два бака», «отскакивающий мяч» и других). Оценить корректность и эффективность моделирования, сравнить с традиционными инструментами (например, Simulink/Stateflow).

Валидация и оценка результатов. Проанализировать результаты моделирования и продемонстрировать преимущества предложенной

архитектуры: гибкость при изменении структуры моделей, возможность повторного использования компонентов, соответствие реальному времени вычислений и т.д.

В результате решения этих задач будет обоснована эффективность новой архитектуры симулятора гибридных систем, что имеет практическую ценность для разработчиков САПР, систем управления и исследователей гибридного моделирования.

Структура работы

Дипломная работа состоит из введения, четырёх глав, заключения и списка литературы. Во введении кратко описаны актуальность темы и сформулированы цель и задачи исследования.

Глава 1 посвящена аналитическому обзору понятий и моделей гибридных систем. В ней рассматриваются определения гибридной системы (например, как система, в эволюции которой связаны непрерывные и дискретные переменные[4][6]), приводятся формализмы (гибридные автоматы) и даются примеры (от простых до сложных) таких систем. Обсуждаются проблемы, возникающие при описании гибридных моделей, включая переходные условия, режимы работы и алгебраические ограничения.

Глава 2 анализирует существующие подходы и инструменты моделирования гибридных систем. Описываются ключевые парадигмы: традиционные системные симуляторы, ориентированные либо на непрерывные процессы, либо на дискретные события, и их интеграция. Изучаются ограничения привычных решений (ограниченная масштабируемость, сложности в интеграции сторонних моделей, необходимость «встраивать» различные среды вручную). Рассматриваются фреймворки ко-симуляции (DEVS, FMI, HLA и др.) и

способы представления моделей (например, объектно-ориентированные форматы Modelica, контейнеры FMU[6]).

Глава 3 содержит предложение унифицированной архитектуры симуляционной системы. Здесь формулируется концепция и детали проектируемой системы: структурная схема, компоненты (движок непрерывной части, движок событий, синхронизатор времени), протокол обмена данными. Описывается, как в рамках архитектуры обеспечивается прозрачное переключение между режимами гибридной модели, обеспечивается сохранение приоритета событий и минимизация численных погрешностей. Представлены выбранные форматы входных/выходных данных и модели, внутреннее представление уравнений, состояния автомата. А также описаны прототипные реализации компонентов архитектуры.

Глава 4 посвящена реализации и экспериментальному исследованию. В ней проводятся экспериментальные симуляции на примерах гибридных систем. Представляются результаты моделирования, анализируется корректность и сравниваются вычислительные характеристики с исходными подходами.

Заключение содержит краткие выводы о проделанной работе, указывает на достижения и перспективы дальнейших исследований.

1. Обзор существующих подходов к моделированию дискретно непрерывных систем

1.1.Унифицированные математические фреймворки

Унифицированные математические фреймворки гибридных систем связывают дискретную логику и непрерывную динамику в единой модели. Ключевой идеей является формализация переключений режимов и непрерывного эволюционного процесса в одной системе

уравнений или неравенств. Например, импульсные дифференциальные включения (impulse differential inclusions) обобщают традиционные дифференциальные уравнения, позволяя описывать непредсказуемые скачковые изменения и неполнодетерминированные события [7]. Такие модели формируют математическую основу для анализа свойств жизнестойкости (viability) и инвариантности состояний гибридной системы.

Одним из современных мощных подходов является Mixed Logical Dynamical (MLD) представление (Bemporad и др.). В рамках MLD система описывается линейными разностными уравнениями, дополненными булевыми переменными и линейными неравенствами, которые моделируют логические условия переключения [3]. Этот фреймворк связывает арифметическую динамику (линейные модели) и дискретные события (например, переключатели «включено/выключено») посредством целочисленного программирования. К его преимуществам относятся единый математический аппарат для анализа и синтеза управления, а также готовность к решению через методы смешанного целочисленного программирования. Однако при этом любые общие нелинейности нужно аппроксимировать кусочно-линейными моделями или уравнениями с дополнительными переменными [3], что ограничивает точность при сильной нелинейности реальной системы.

Схожим образом вводятся кусочно-аффинные системы (PWA) и системы дополнения (Linear Complementarity Systems). В таких моделях непрерывная динамика аппроксимируется линейными сегментами, каждый из которых активен в своем логическом режиме. Анализ таких моделей может быть более алгоритмизируемым, но их применение ограничено ситуациями, где переходы между режимами зависят от простых условий на состояние.

Важным преимуществом унифицированных фреймворков является возможность формального анализа (достижимости, управления) в рамках единой математической модели. Это особенно полезно в задачах оптимального управления, где модели MLD/PWA позволяют сформулировать задачу как MILP [3]. С другой стороны, при всей выразительности этих подходов часто возникает взрывообразный рост числа состояний при учете всех комбинаций дискретных переменных. К тому же, серьезной трудностью остаётся корректное моделирование сильно нелинейных процессов – например, химических реакций или вихревых потоков, – которые выходят за рамки линейной или кусочно-линейной аппроксимации[3].

Преимущества: единый формализм для дискретной и непрерывной частей; возможность формального вывода свойств и использования численных методов оптимизации.

Ограничения: ограничения на нелинейность (требуется аппроксимация линейными фрагментами); потенциальный взрыв состояний при большом количестве булевых переменных.

Пример: задачу управления набором переключаемых источников питания можно сформулировать в виде MLD-системы, где логика включения и выключения задаётся булевыми переменными, а динамика токов – линейными уравнениями. Такой подход позволяет единовременно использовать методы оптимизации с учётом и динамики, и логики.

1.2.Гибридные автоматы и проблема «взрыва состояний»

Гибридный автомат – это один из самых распространённых формальных моделей дискретно-непрерывных систем. В гибридном автомате задаётся конечный граф состояний (режимов), в каждом из

которых непрерывные переменные эволюционируют по дифференциальным уравнениям [6]. Переключения (дискретные скачки) между режимами происходят мгновенно при выполнении заданных условий. Так, «перескакивающие» переходы («jump transitions») и «непрерывные потоки» («flow transitions») являются двумя видами изменения состояния [6].

Гибридные автоматы играют важную роль при формальной верификации и синтезе управляющих алгоритмов: они позволяют однозначно описать поведение системы и применять к нему методы model-checking. Однако одним из фундаментальных недостатков этого подхода является колоссальный рост пространства состояний. Комбинации дискретных режимов с непрерывными состояниями формируют бесконечный (а при дискретизации – очень большой) модельный граф. Уже в простейших случаях число возможных траекторий и состояний возрастает экспоненциально. Как отмечают авторы, поведение гибридных автоматов зачастую настолько сложное, что для него требуются специальные компьютерные методы анализа [6]. Модельная проверка гибридных систем резко ограничивается задачей проблемы взрыва состояний: даже при относительно простых пределах для непрерывных переменных количество состояний может быть несравнимо большим, чем в аналогичных дискретных или тайм-автоматах [5]. Например, в лекциях указывается, что при проверке свойств гибридных автоматов «проблема взрыва состояний гораздо серьёзнее, чем в тайм-автоматах» [5].

Преимущества: формально чёткая модель с определённой семантикой; пригодность для верификации (используются HyTech, SpaceEx, UPPAAL, PRISM и т.д.); возможность композиционного построения сложных систем.

Ограничения: бесконечное или очень большое состояние пространство; NP-полные и неразрешимые в общем задачи

достижимости; трудности моделирования нелинейной динамики (большинство методов работает только для линейных или piecewise-линейных потоков).

Примеры: классический пример гибридной системы – это термостат с водяным баком и горелкой. В этом примере «режим» автомата соответствует положению горелки (включена/выключена), а непрерывная переменная – температура воды в баке. При выключенной горелке температура подчиняется экспоненциальному закону охлаждения, а при включенной – закону нагрева (с учётом того, что температура нельзя превысить определённую границу) [6]. Этот пример иллюстрирует, как дискретное переключение (горелка) и непрерывная динамика (нагрев/остывание) объединяются в гибридном автомате. С другой стороны, при попытке формально просчитать все возможные траектории такого автомата уже несложно столкнуться с невозможностью полного перебора, особенно при добавлении новых параметров или переменных.

1.3.Сети Петри и процессные алгебры: ограничения нелинейной динамики

Сети Петри – это графо-ориентированная модель дискретных событий, традиционно используемая для описания распределённых систем и параллельных процессов. Для моделирования гибридной динамики были разработаны различные расширения Petri, сочетающие непрерывные и дискретные места: например, непрерывные сети Петри и гибридные сети Петри. В непрерывной сети Петри метки (ток в узлах) могут принимать любые неотрицательные значения и эволюционировать непрерывно по дифференциальному уравнению[72]. Это позволяет моделировать накопление или расход потоков (например, жидкости), а дискретные переходы при этом происходят «разбавленно».

Такое приближение уменьшает количество событий, которые необходимо рассматривать, по сравнению с чисто дискретной моделью, и избавляет от необходимости полного перебора дискретных состояний в анализе[72].

Однако и в случае гибридных сетей Петри возникают ограничения. В классических (d-элементарных) гибридных PN непрерывные места обычно описываются линейными скоростями истечения. Например, Differential Petri Nets используют дифференциальные уравнения первого порядка для каждой непрерывной метки, фактически представляя кусочно-линейный поток[72]. Это означает, что нелинейные зависимости (скажем, квадратичная или экспоненциальная динамика) в такой формализации не учитываются или требуют дополнительного приближения. Как отмечено в литературе, общее ограничение гибридных PN – необходимость линеаризовать или кусочно-линеаризовать динамику, что облегчает анализ, но снижает точность моделирования сложных физических процессов[72].

Процессные алгебры – это формальные языки для описания поведения параллельно выполняющихся процессов. Существуют процессные алгебры, расширенные для гибридных систем: например, HyPA, φ-исчисление, Hybrid χ и другие[13]. Они позволяют задавать конкурентные процессы и их взаимодействие, дополняя традиционные операторные конструкции (последовательность, параллелизм, выбор) «потоковыми» или дифференциальными компонентами. Процессные алгебры богаты с теоретической точки зрения и хорошо подходят для формальных рассуждений. Тем не менее, они обладают ограничениями: спецификация сложных непрерывных моделей остаётся громоздкой, и в существующих алгебрах непредусмотрены произвольные нелинейные потоки без значительных усложнений. Чаще предполагается детерминированный поток или линеаризованные зависимости, чтобы

упростить синтаксис и анализ. В целом и в случае сетей Петри, и процессных алгебр основное применение приходится на системы с ограниченной нелинейностью: например, потоки в трубопроводах (где можно использовать линейную аппроксимацию), дискретные события в производственных линиях или цифровую логику.

Преимущества: визуальная и понятная схема (сети Петри); мощный теоретический аппарат (процессные алгебры); хорошо развиты методы анализа конкурентности, достижимости и устойчивости (для линейных случаев).

Ограничения: непрерывная динамика обычно принимается упрощённой (константные или линейные скорости), что не позволяет напрямую описывать общие нелинейные процессы[72]. Процессные алгебры часто сложны в практическом использовании и мало распространены в индустрии, особенно для задач с насыщенными нелинейностями.

Примеры использования: гибридные сети Петри применяются при моделировании технологических процессов и автоматизированных линий, где хорошо описываются линейные или прямая функциональная зависимость потоков (например, расход через клапан). Процессные алгебры находят применение в формальном описании протоколов и средств управления со строгими спецификациями, когда важна гарантированная корректность работы при конкурирующих процессах. В обоих подходах основное внимание уделяется верификации свойств и синтезу, тогда как моделирование сложных физических нелинейностей, как правило, опускается либо редуцируется.

1.4.Ptolemy II и Modelica: плюсы и минусы

Ptolemy II – это исследовательская среда для моделирования гетерогенных систем, разработанная в UC Berkeley. Основная идея

Ptolemy – акторно-ориентированная модель: система строится из актеров (компонентов), обменивающихся данными через порты [12]. Каждый актор выполняется независимо, а поведение всего моделируется «директором», который определяет конкретную модель вычислений (например, дискретное событие, непрерывный поток, синхронный или асинхронный режим)ptolemy.eecs.berkeley.edu. Такая архитектура обеспечивает гибкость и модульность: разработчик может комбинировать различные семантики (режимы вычислений) и строить иерархические модели произвольной сложности. Например, можно использовать каскадные модели переходов состояний (FSM) для контроллеров вместе с непрерывными моделями электромеханики или фильтрации сигналов в одном окне Ptolemy.

Преимуществом Ptolemy II является его универсальность: он поддерживает множество моделей вычислений («директоров»), включая обсуждаемые ранее DEVS, CT, SDF, FSM и другие [12]. Пользователь может «рискнуть» смешать синхронные и асинхронные взаимодействия, а среда выполнит симуляцию с учётом этих соглашений. Однако такой подход сопряжён с трудностями: семантика моделей в Ptolemy относительно свободна и нестрога, что затрудняет формальный анализ [12]. Как отмечается в руководстве по Ptolemy, «языки моделирования с нестрогой семантикой сложнее поддаются анализу»[12]. На практике Ptolemy II больше ориентирован на исследование и быструю прототипизацию, нежели на окончательную инженерную отладку. Инженерные пакеты для верификации (model-checking) в Ptolemy развиты слабо, а симуляция может быть ресурсоёмкой при сложных и высокодетализированных моделях.

Modelica – это объектно-ориентированный язык для физического моделирования, основанный на уравнениях. Его философия – описывать систему как набор уравнений связей, не задавая явной «последовательности» расчёта (acausal approach). В Modelica

естественно моделируются многодоменные физические системы: механика, гидравлика, электроника, термодинамика и т.д. Гибридный характер достигается за счёт событийных конструкций when и условных операторов if/else в блоке equation[13]. Например, в примере идеального диода на Modelica логика вела себя так: при отрицательном напряжении (off = s<0) задаётся одно уравнение, а при положительном – другое. События (когда условие меняет истину) обрабатываются директивой when [14]. Благодаря этому в Modelica удобно описывать «диффузоры» дискретных переключений в потоке уравнений непрерывной системы.

Ключевые достоинства Modelica – широкая индустриальная поддержка и богатые библиотеки домен-специфических компонентов (например, холодильных машин, двигателей, роботов). Язык позволяет автоматически получать симуляционные модели, генерировать код, использовать передовые солверы для дифференциально-алгебраических систем. Для многих сложных гибридных систем (например, автомобильная бортовая сеть, модель двигателя) Modelica предоставляет готовые стандартизованные компоненты. В то же время Modelica имеет и ограничения. Дискретные конструкции несколько «вторичны» относительно уравнений, и сложные логические контроллеры приходится описывать вручную в коде. Формальные методы верификации в экосистеме Modelica мало развиты (напр., проверки недетерминированного поведения), и нативная поддержка таких задач уступает Ptolemy или ориентированным на безопасность подходам. Кроме того, при моделировании очень жёстких или быстродействующих событий возможны численные сложности при симуляции.

Ptolemy II – плюсы: единая среда для смешения нескольких моделей вычислений; высокая гибкость в описании структуры системы; визуальное иерархическое построение моделей [12]. Минусы: неформальная семантика, сложности анализа и масштабирования,

экспериментальность (отсутствие стандартизированных промышленных библиотек).

Modelica – плюсы: мощный язык для физических (continuous) моделей; встроенные конструкции для гибридных явлений (if/when) [14]; широкий инструментарий и сообщества разработчиков. Минусы: сложность описания чисто дискретной логики, ограниченные формальные гарантии, численная нагрузка при большом числе событий.

Примеры: Ptolemy II часто используется в академических исследованиях распределённых контроллеров и мультиядерных встроенных систем, где важна комбинация параллелизма и гибкости. Modelica же широко применяется в промышленности (авто- и авиастроение, энергетика) для моделирования теплообменников, гидроприводов и электросистем, когда требуется реалистичная непрерывная динамика с управляемыми переключениями.

2. Требования, ограничения

Во второй главе проводится описание проектирования новой симуляционной системы для гибридных (дискретно–непрерывных) моделей. В предыдущей главе рассмотрены существующие подходы к моделированию гибридных систем (например, Ptolemy II, Simulink/Stateflow, Modelica/FMI и др.) – было отмечено, что они опираются на сочетание конечных автоматов и непрерывных динамических моделей[14]. Построенная на основе этих данных новая система должна удовлетворять ряду функциональных требований и ограничений, обеспечивать эффективную работу с соответствующими форматами данных, а также вносить научную новизну в области моделирования гибридных систем. В структуре системы реализованы следующие ключевые компоненты и принципы: модульная архитектура с четкими API, переносимость между платформами, высокопроизводительное выполнение на языке C, унифицированный JSON-файл как формат ввода/вывода. Ниже последовательно излагаются функциональные требования, требования к формату данных и производительности, а также научная новизна предложенной архитектуры.

2.1.Функциональные требования и ограничения (модульность, API, кроссплатформа)

Основными функциональными требованиями к проектируемой системе являются модульность, наличие хорошо определённых программных интерфейсов (API) между компонентами и кроссплатформенность. Модульная (многослойная) архитектура позволяет разделять систему на логические блоки – загрузчик модели, движок численного интегрирования, менеджер дискретных событий, механизм обработки выходных данных и т.д. Каждый модуль отвечает за определенную подсистему (например, чтение и валидацию входного

JSON, расчёт непрерывной части динамики, переключение дискретных состояний, генерацию выходных JSON-результатов). Разделение на слои со строгими интерфейсами между ними соответствует проверенной практике проектирования ПО. Как отмечено в литературе, слоистая архитектура обеспечивает расширяемость, масштабируемость и переносимость системы, а также упрощает обеспечение качества кода (например, тестируемость и изоляцию компонентов)[15]. Это позволяет в дальнейшем добавлять новые алгоритмы или расширять функционал без существенной переработки ядра системы – достаточно реализовать новый модуль или подключить другой «backend» через заранее определённый API.

Важной составляющей гибкой архитектуры является фиксированный программный интерфейс (API) между слоями. Например, движок симуляции предоставляет API-функции для инициализации модели, запуска расчёта на определённом интервале времени, получения текущих значений переменных и пр. Интерфейс также может позволять подключать внешние компоненты (например, визуализацию или средства постобработки) без изменения ядра. Наличие такого API отражает «plug-and-play» концепцию: каждый слой предоставляет набор сервисов с чётко определёнными интерфейсами[16], что упрощает замену или обновление модулей (например, альтернативный парсер JSON или другой интегратор ОДУ). Таким образом, система проектируется с учётом принципов компонентного проектирования и сервис-ориентированного подхода: каждый модуль взаимодействует с остальными через хорошо документированный API, обеспечивая явную границу ответственности. Это соответствует требованиям современного системного проектирования, когда архитектура строится вокруг обмена данными по интерфейсам[16][15].

Требование кроссплатформенности означает, что система должна быть переносимой на различные операционные системы и аппаратные платформы (Windows, Linux, macOS и др.). Выбор языка C++ или C поставлен исходно для обеспечения такой переносимости. Язык C имеет международный стандарт (ISO/IEC 9899) и множество компиляторов, позволяющих собирать код на разных архитектурах[68]. Логика архитектуры ориентирована на минимизацию специфичных для платформы частей кода – используется стандартизированная библиотека C (ANSI C), платформо-независимые структуры данных и абстракции ввода-вывода. Это позволяет собирать систему с помощью универсальных средств (Makefile, CMake) и создавать нативные бинарники для целевых платформ без существенных изменений в коде. Как показывает опыт разработки кроссплатформенных симуляционных фреймворков (например, HELICS), разбиение на слои с API способствует кроссплатформенному дизайну – каждый слой может быть настроен или заменён в зависимости от платформы[15].

Наряду с вышеперечисленным, функциональными требованиями являются:

- Поддержка конечных автоматов и непрерывной динамики. Система должна уметь загружать гибридную модель в виде конечного автомата со списком дискретных режимов, в каждом из которых задана непрерывная динамика (система дифференциальных уравнений)[14]. Симуляционный движок выполняет численное интегрирование ОДУ в текущем режиме до события перехода в новое состояние, после чего динамика переключается. Для этого в архитектуре выделен модуль управления переходами (event manager), который следит за условиями переходов (гварды) и триггерит смену режима. Такой подход унифицирован с формализмом гибридных автоматов[14], где каждое дискретное состояние сопровождается свойственной ему системой ОДУ. Система должна корректно обрабатывать смену режимов,

накопление времени и сброс переменных по событиям, сохраняя историю всех переменных.

- Модуль ввода/вывода. Предусмотреть отдельный модуль для чтения входной модели из JSON-файла и записи результатов в JSON. API модуля ввода должно уметь проверять корректность формата, строить внутреннее представление автомата и передавать его остальным частям системы. Модуль вывода формирует итоговый JSON с указанием значений всех переменных системы во времени. Благодаря модульности, при необходимости формат ввода-вывода можно будет расширять (например, добавляя поддержку XML или текстового формата) без затрагивания симуляционного ядра.
- Кросс-компонентный API. Система предоставляет API для программного управления симуляцией (например, вызовы инициализации, шага или остановки симуляции). Это позволит встроить симулятор в сторонние приложения или запустить его в различных средах (включая сценарии командной строки или библиотеки для других языков). API должен быть независим от пользовательского интерфейса, чтобы обеспечивать интеграцию системы с другими инструментами. Принцип разделения слоёв предполагает, что внутренние структуры данных не «протекают» между компонентами, а взаимодействие происходит только через описанные интерфейсы[16][15].
- Масштабируемость и расширяемость. Система должна легко расширяться новыми алгоритмами (например, разными методами решения ОДУ, альтернативными стратегиями управления шагом по времени) без изменения архитектуры в целом. Это достигается за счёт замены соответствующих модулей через их абстракции (например, реализация интерфейса интегратора может быть заменена на более точный или более быстрый метод, оставаясь совместимой по API).
- Надёжность и тестируемость. С точки зрения архитектуры важно обеспечить возможность отладки и тестирования: каждый модуль

должен быть достаточно изолирован, чтобы можно было тестировать его отдельно. Чёткие интерфейсы облегчают верификацию корректности системы в целом.

В совокупности эти функциональные требования диктуют архитектуру системы: она строится в виде многослойной системы, где в каждом слое реализован ограниченный набор задач. Подобный подход уже применялся в сложных симуляционных платформах[15][16]. Важно заметить, что кроссплатформенная многослойная архитектура поддерживает упомянутые аспекты: таким образом, начальные системные требования превращаются в конкретный дизайн модулей с межмодульными API, что соответствует современным практикам системного проектирования[15].

2.2.Требования к формату данных и производительности

Формат данных. В качестве входного и выходного формата системы выбран JSON (JavaScript Object Notation). JSON – это открытый текстовый формат обмена данными, который определён стандартом IETF как «легковесный текстовый, независимый от языка формат обмена данными»[18]. Он использует человекочитаемый синтаксис для представления структурированных данных (пар «имя–значение», массивов и т.д.), что обеспечивает его удобочитаемость и простоту обработки программно. На практике JSON часто выбирают именно за эти качества: он позволяет быстро писать и парсить файлы, легко модифицируется и поддерживается практически на всех языках программирования[18][19]. Кроме того, JSON-файлы компактны и не

требуют громоздкой разметки (в отличие от XML), что ускоряет чтение/запись и упрощает визуальный обзор модели.

Во входной JSON задаётся унифицированная модель гибридной системы. Структура может быть примерно следующей: верхнеуровневый объект «model», внутри которого имеются разделы «states» (режимы автомата), «transitions» (переходы между режимами), «variables» (переменные непрерывной динамики) и т.д. Каждый режим описывается набором дифференциальных уравнений или иных правил изменения переменных (например, в виде линейных/нелинейных формул или ссылкой на функцию динамики). Переходы описываются условиями и действиями (как в гибридном автомате). Такая унификация формата позволяет единообразно задавать любую гибридную модель без привязки к конкретному инструменту. При желании JSON-структуру можно расширить для поддержки дополнительных свойств (например, имен потоков или пользовательских скриптов), при этом базовая структура сохраняется. Важным требованием к формату является поддержка вложенности: режим может содержать дочерние подрёмы (иерархия состояний)[14], а JSON позволяет естественным образом представлять вложенные объекты и массивы.

В контексте гибридных систем решение использовать JSON вместо других форматов имеет преимущества. Например, стандарт FMI (Functional Mock-up Interface) описывает модели на основе XML-файлов и специального C-кода[21]. Хотя FMI является признанным стандартом обмена моделями, формат XML обычно громоздок для простого просмотра, а обработка XML в C требует подключения тяжёлых библиотек. JSON же остаётся лёгким: «JSON – это лёгковесный формат обмена данными, который легко читать и изменять»[19]. Кроме того, современные библиотеки парсинга JSON (например, cJSON, Jansson для языка C) дают быстрые и компактные реализации. Таким образом, использование JSON-формата удовлетворяет требованию простоты

подготовки моделей и прозрачности формата для инженеров и исследователей.

Производительность. Расчёт гибридных моделей предъявляет высокие требования к скорости и эффективному использованию ресурсов. В каждом режиме системы происходят численные интегрирования (решение ОДУ), а при срабатывании событий необходимо быстро переключать контексты. Основными показателями производительности являются время вычисления одной симуляционной итерации и общая длительность расчёта при заданном количестве шагов по времени. Поэтому к системе предъявляются требования к алгоритмической и аппаратной эффективности: она должна максимально эффективно использовать CPU и память.

Ключевым архитектурным решением для обеспечения производительности стал выбор языка C в качестве базового. Язык C позволяет ручное управление памятью, отсутствие ненужных абстракций и прямой доступ к низкоуровневым ресурсам системы (например, к указателям, массивам, битовым флагам). Это соответствует известному факту, что «C – мощный язык низкого уровня, обеспечивающий тонкий контроль над системными ресурсами, что делает его идеальным для высокопроизводительных симуляций в реальном времени»[20]. Использование C гарантирует минимальные накладные расходы на управление памятью (нет сборщика мусора) и позволяет компилировать критические участки кода с оптимизациями на уровне компилятора, что критично при решении большого числа ОДУ. Важность этого требования подтверждается практикой: в системах реального времени и физических симуляциях (например, в играх, робототехнике и прочих сферах) именно C часто выбирают за его быстродействие и способность взаимодействовать напрямую с аппаратурой[20].

Требование оптимального использования памяти выдвигает дополнительные ограничения на архитектуру. Система должна минимизировать динамическое выделение памяти в горячих путях (например, в основном цикле интегрирования), используя заранее выделенные структуры или пулы памяти. При необходимости большего количества данных (например, больших массивов истории переменных) архитектура предусматривает возможность сохранения их по мере освобождения ресурсов, чтобы избежать переполнения памяти. Также важен компромисс между плотностью хранения выходных данных и удобством анализа: возможно, разумно сохранять значения не всех переменных на каждом шаге, а только ключевых, или выдавать выборочные точки времени – это часть настроек производительности.

Наконец, сама архитектура учитывает параллелизм и масштабируемость. Хотя в рамках одного процесса модель может рассчитываться последовательно, возможна реализация и распараллеливания задач (например, интегрирование в нескольких режимах, если модель распадается на независимые компоненты) либо многопоточность при пакетной обработке множества независимых запусков модели. Использование C облегчает подключение многопоточных библиотек (POSIX threads, OpenMP) для распараллеливания нужных частей без изменения самого ядра логики. Внутренние алгоритмы (например, алгебраические вычисления или решение системы ОДУ) выбираются с учётом эффективности: используются встроенные типы double/float, оптимальные интеграторы (Runge–Kutta, методы Адамса–Бэшфорта и др.) и адаптивное управление шагом по времени. Условием является высокая точность результатов при приемлемых временных затратах. Все эти требования к производительности и форматам данных определяют техническую часть реализации: система фокусируется на производительности ядра симулятора (движка интегрирования и управления событиями),

сохраняя при этом гибкость ввода/вывода через удобный JSON-файл. Такое сочетание облегчает разработку моделей и повышает скорость расчётов по сравнению с системами, использующими интерпретируемые языки или тяжёлые форматы.

2.3.Научная новизна предложенной архитектуры

Предложенная архитектура симуляционной системы обладает научной новизной по нескольким аспектам. Во-первых, объединение гибридного моделирования с современными IT-технологиями. В основе системы лежит концепция представления гибридной модели как конечного автомата с внутренней непрерывной динамикой[14], но её реализация сконцентрирована на высокопроизводительном языке C и простом формате обмена (JSON). Такой подход отличается от многих существующих: например, Ptolemy II реализован на Java и использует собственные внутренние описания моделей[14]; Simulink/Stateflow использует двоичные или XML-представления в экосистеме MATLAB; FMI стандартизирован на XML+C. В отличие от них, данная система предлагает единый человекочитаемый и простой формат JSON для описания гибридного автомата, что упрощает интеграцию и изучение моделей. Использование JSON (вместо более тяжёлого XML) для описания моделей встречается редко[21], а сочетание этого с C-реализацией симулятора позволяет добиться лучших характеристик. Таким образом, научная новизна заключается в новом унифицированном способе задания гибридной модели и её быстрой прогонке.

Во-вторых, применение языка C для симуляции гибридных систем. Хотя C используется во многих областях встраиваемого ПО и

высокопроизводительных вычислений, специализированные инструменты гибридного моделирования обычно реализованы на языках более высокого уровня (Java, Python, специализированные среды). Переход к C в данной области влечёт за собой полную ручную оптимизацию вычислений и предсказуемость поведения. Новизна состоит в том, что именно на языке C построено одновременно ядро симулятора и интерфейс взаимодействия (через API) с внешним миром. Этот выбор подчёркивает ориентацию на минимизацию накладных расходов на ресурсы: как показано в литературе, C обеспечивает тонкую настройку ресурсов и высокую скорость выполнения[20]. Таким образом, вносимый вклад – это демонстрация возможности эффективной реализации гибридного симулятора без использования виртуальных машин или интерпретаторов, сохраняя при этом гибкость (API, JSON) привычных сред.

В-третьих, модульная многослойная архитектура – новшество в контексте гибридного моделирования – позволяет обеспечить максимальную адаптивность системы. Как указывалось ранее, слоистое строение поддерживает расширяемость и заменяемость модулей[15][16]. Предлагаемая система концептуально разделена на независимые слои: слева – загрузчик/парсер модели (JSON-парсер, конвертация в внутренние структуры гибридного автомата); ядро – симуляционный движок с собственным планировщиком времени и модулем интегрирования; справа – модуль вывода и анализа (формирование истории переменных, экспорт JSON). Такое разделение изолирует модули друг от друга, снижает связанность кода и делает возможным «смешивание и сочетание» (mix-and-match) компонентов, что принято в гибких архитектурах[16]. В научном плане это обеспечивает новую ступень конфигурируемости: исследователь может подменять, например, алгоритмы управления шагом или решатели ОДУ в симуляционном слое, не затрагивая описания модели.

Ещё одним новым элементом является унификация описания переходных условий и процедур переключения режимов. Во многих существующих системах переходы между состояниями задаются строго как часть модели (например, как гварды в Stateflow или как события Simulink). В предлагаемой архитектуре переходы описываются в JSON наравне с другими данными модели, а отдельный модуль (менеджер событий) обрабатывает их во время симуляции. Такой подход повышает гибкость: структура автомата доступна в качестве данных, и ею может управлять программный код (напр., возможна динамическая модификация переходов во время выполнения). Это отличается от традиционных реализаций, где модель жёстко «вшита» в среду.

Наконец, комбинация JSON + C + модульная архитектура сама по себе является научным вкладом в инструментарий гибридного моделирования. Подход FMI, например, опирается на XML-модель и генерируемый C-код[21], требуя наличия компилятора и библиотеки времени выполнения; наши решения строятся на естественной текстовой спецификации JSON и интерпретации её на лету, что ускоряет цикл разработки и анализа. Это позволяет быстрее прототипировать новые модели и менять их на лету. Таким образом, научная новизна заключается в интеграции современных стандартов обмена данными и классических эффективных средств программирования для задачи гибридного моделирования. Предложенная система выступает как новая платформа, сочетающая лучшие свойства известных подходов и преодолевающая их ограничения: компактный и открытый формат данных (JSON), низкоуровневое исполнение (C) и модульная архитектура для расширяемости[18][20][16].

Каждое из перечисленных новшеств формирует совокупный результат: система, способная моделировать гибридные динамики с высокой скоростью и гибкостью, не завися от конкретной ОС и легко

настраивающаяся под разные задачи. Именно это и представляет собой научно-техническую новизну данного проекта.

3. Архитектура и реализация системы

3.1.Центральное ядро: хранение состояний, глобальная область переменных и время

Центральное ядро системы построено по принципам событийно-дискретного моделирования: модель меняет своё состояние только в моменты наступления событий, а между ними остаётся постоянной [22][68]. Симулированное время «перескакивает» от одного события к другому: ядро хранит упорядоченный по возрастанию времени список будущих событий и на каждом шаге извлекает ближайшее событие, после чего перемещает внутренний таймер к моменту его наступления[27]. Это соответствует классическому описанию дискретно-событийного симулятора: последовательность событий «переключает» состояние системы в определённые моменты времени[26]. После обработки события ядро обновляет состояние модели (например, изменяет значения переменных) и может генерировать новые будущие события; между ними состояние S(t) остаётся неизменным, что позволяет избегать посекундного моделирования [68].

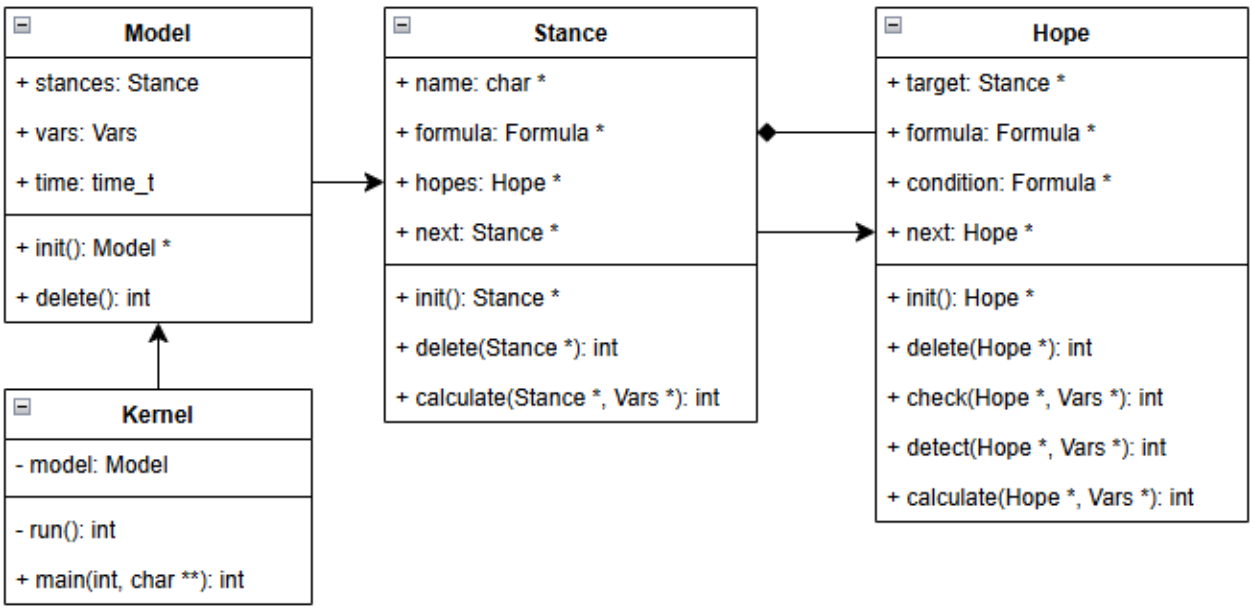

Рисунок 1 UML-диаграмма ядра

Архитектура системы организована вокруг чёткого набора функциональных модулей и их взаимосвязей [62][63]. Основные этапы обработки в ядре выполняются последовательно:

1. Загрузка и парсинг входного JSON через модуль cJSON;
2. Построение математической модели (инициализация переменных и формул; см. разделы 3.3–3.4);
3. Цикл расчета модели с постепенным увеличением времени, расчета состояния модели и выявление событий

3.1. Численное интегрирование ОДУ с помощью внешних библиотек через интерфейс ядра (раздел 3.5);

3.2. Обнаружение событий и локализация момента их наступления(detect, раздел 3.6);

4. Формирование выходного JSON.

Такая pipeline-схема обеспечивает прозрачность структуры и упрощает расширение системы новыми функциями[63].

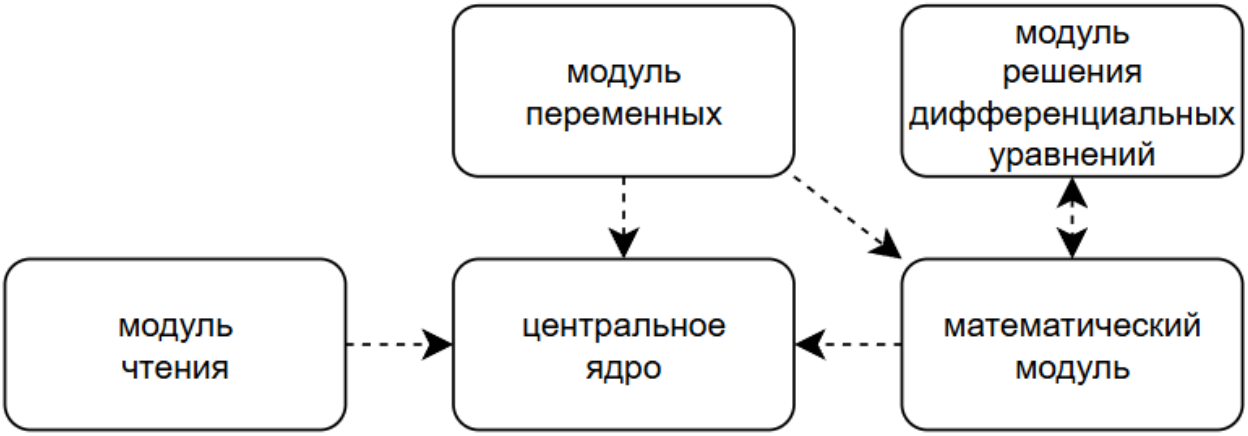

Рисунок 2 Общая архетектура системы и порядок предоставения API

Глобальное управление временем моделирования выполняется ядром с неизменным направлением «вперёд»: текущее время только растёт и никогда не откатывается назад[30]. События обрабатываются

строго в порядке возрастания их запланированного времени[30]. Центральный цикл извлекает событие с минимальным временем из очереди, выполняет привязанное к нему действие (в большинстве случаев изменяет состояние модели) и удаляет его из списка [37]. Таким образом, архитектура ядра опирается на две ключевые компоненты — очередь событий и цикл их обработки, что полностью соответствует литературному описанию дискретно-событийного подхода[37].

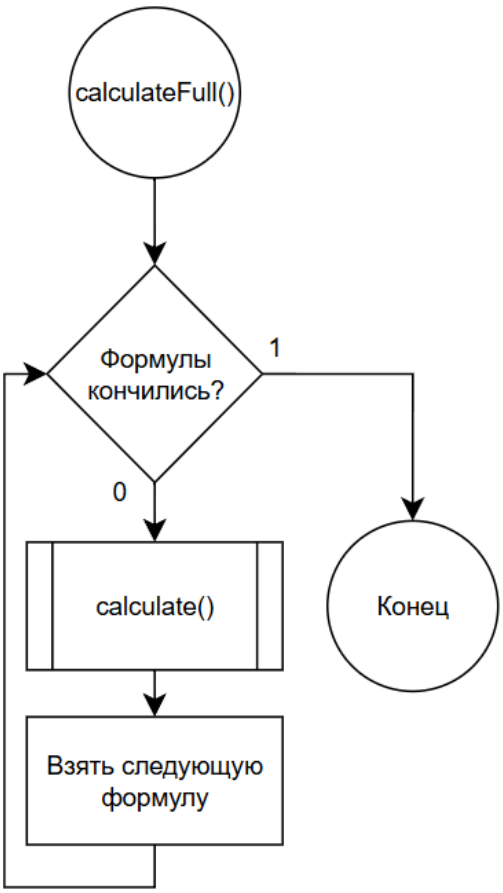

Рисунок 3 Блок-схема расчета системы уравнений состояния

Для поддержки модульности и наглядности структуры все компоненты системы обособлены и взаимодействуют через чётко определённые интерфейсы. В диаграмме системы ядро (core) выступает «сердцем» приложения, координируя работу вспомогательных модулей: парсинга JSON, решения ОДУ, детектирования событий и других [62]. Документирование архитектуры включает диаграммы связей между

подсистемами, что упрощает понимание и дальнейшую поддержку решения. Высокий уровень архитектуры закладывает фундамент для надёжности, масштабируемости и отказоустойчивости системы [63].

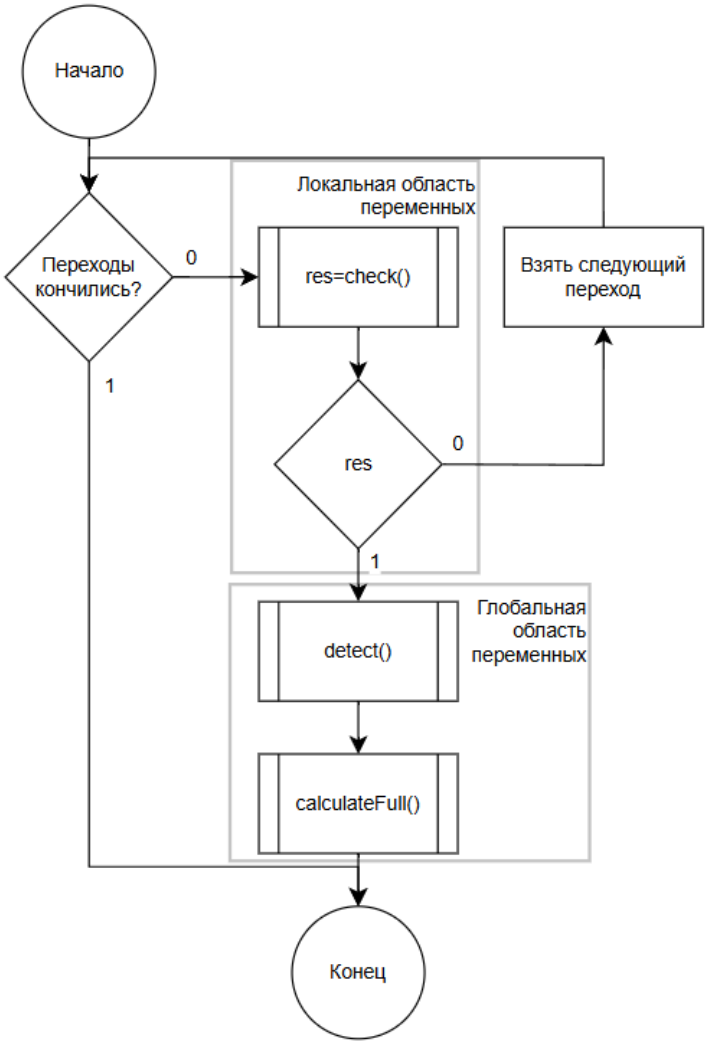

Рисунок 4 Блок-схема обнаружение событий

Внутри ядра выделена глобальная область переменных, где хранятся параметры и состояния модели, общие для всех компонентов. Глобальные переменные обеспечивают единый контекст моделирования и сохраняются между итерациями расчёта. Локальные переменные привязаны к отдельным блокам или подсистемам и служат для временных вычислений. Благодаря событийному подходу состояние системы (а следовательно, и значения глобальных переменных) остаётся неизменным между событиями, что упрощает управление симуляцией: при наступлении события в момент T ядро

обновляет только те переменные, которые затрагиваются самим событием[68][23].

3.2.Модуль чтения/записи JSON

Для обмена данными модели и сохранения её состояния используется формат JSON, что обеспечивает гибкость и портативность представления данных. В качестве лёгкого и самодостаточного парсера и генератора JSON применена библиотека cJSON: её реализация состоит из одного исходного файла cJSON.c и заголовка cJSON.h, которые достаточно скопировать в проект для начала работы. Написанная на ANSI C (C89), cJSON гарантирует широкую переносимость между компиляторами и платформами[64].

Библиотека предоставляет функции для разбора JSON-строки (например, cJSON_Parse) и для формирования JSON из структуры данных (например, cJSON_Print), а также API для доступа к элементам по ключам (например, cJSON_GetObjectItem)[36]. Для интеграции в приложение достаточно включить cJSON.h в нужные файлы и вызывать её функции напрямую: дополнительных зависимостей при этом не требуется[64][65].

Архитектура JSON-модуля инкапсулирует все детали синтаксиса: при чтении входного файла приложение загружает его содержимое в строку и вызывает cJSON *json = cJSON_Parse(string);

Парсер строит древовидную структуру cJSON-объектов, после чего исходная строка больше не используется. С помощью макроса cJSON_ArrayForEach и функций доступа (например, cJSON_GetObjectItem)[38][64] перебираются массивы и словари, извлекаются числовые и строковые поля (например, «variables», «formulas»), которые затем конвертируются во внутренние структуры системы (модульные переменные, формулы и т.д.). По завершении обработки память дерева освобождается через cJSON_Delete, чтобы избежать утечек[64].

При сохранении модели или её результатов сначала создаётся дерево cJSON (JSON-объект), затем оно сериализуется в строку или сразу записывается в файл. Такое разделение позволяет остальному ядру системы работать через простые функции чтения/записи, не вникая в детали формата.

Кроме стандартного malloc/free, cJSON допускает установку собственных функций аллокации через cJSON_InitHooks, что полезно для отслеживания использования памяти при ручном управлении ресурсами. В результате cJSON выступает в роли лёгкого конвертера между текстовым форматом и внутренними объектами системы, обладая минимальными требованиями к встраиванию и отсутствием внешних зависимостей

Для справки можно отметить, что на рынке C-библиотек для работы с JSON существуют и другие решения: json-c – библиотека с объектной моделью JSON в соответствии со стандартом RFC 8259, а также Jansson с простым и интуитивно понятным API[64]. Тем не менее в данной системе выбран cJSON из-за его компактности и простоты интеграции.

3.3.Модуль переменных: структура данных и поиск

Модуль управления переменными отвечает за хранение и быстрый поиск имён переменных модели и их значений. Для организации символов (имён переменных) применяется двоичное дерево поиска, что является распространённым решением при реализации таблиц символов в компиляторах и интерпретаторах[32][54]. При вставке новой переменной или поиске существующей происходит сравнение строк-ключей и переход по узлам дерева, что обеспечивает среднюю асимптотику операций O(log n) [39].

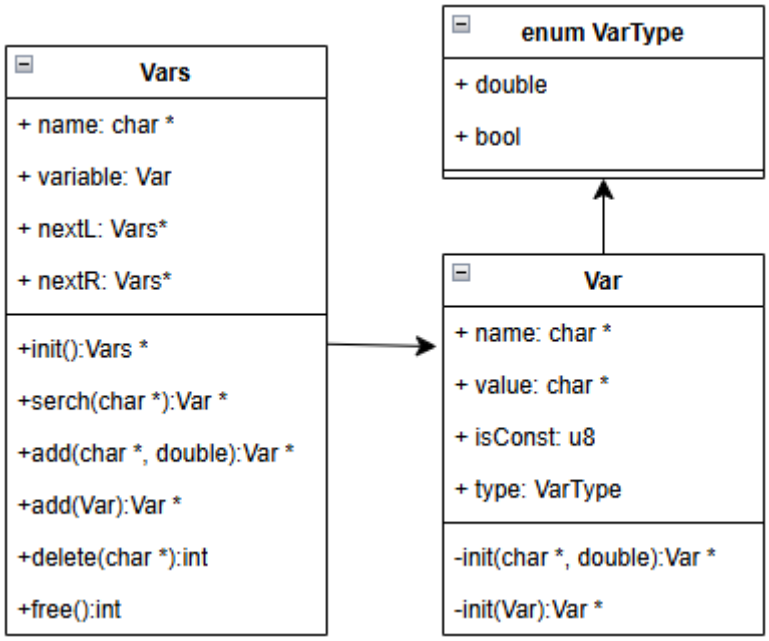

Рисунок 5 UML - диаграмма модуля переменных

Каждая переменная (параметр или неизвестная) представлена структурой данных, содержащей её имя, тип, текущее значение и дополнительную метаинформацию (диапазон, единицы измерения и т.п.). Такая организация аналогична записям в таблице символов компилятора, где каждому идентификатору соответствует набор полей (тип, область видимости, адрес)[66]. В рассматриваемом модуле используется хеш-таблица или ассоциативный массив: ключом служит строка с именем переменной, а значением — указатель на соответствующую структуру. Это позволяет выполнять быстрый поиск и обновление по имени.

Система разделяет глобальные и локальные переменные: глобальные переменные моделирования хранятся в «корневой» области (например, в атрибутах ядра), а локальные — в контексте отдельных компонент. При разрешении имени ядро сначала обращается к локальной области, а при отсутствии переменной — к глобальной. Такой подход обеспечивает изоляцию локальных данных при сохранении удобного доступа к общим параметрам.

Альтернативные структуры данных, например связные списки, обеспечивали бы линейное время поиска O(n), что при большом количестве

переменных оказывается недостаточно эффективным[62]. В отличие от этого двоичное дерево или хеш-таблица дают лучшую масштабируемость [54][32]. В целом структура данных модуля повторяет принципы таблицы символов: уникальное имя переменной служит ключом для доступа к набору её атрибутов[66][67].

3.4.Математический модуль и представление формул

Математический модуль системы организует хранение и выбор вычислительных формул (алгебраических выражений, правых частей дифференциальных уравнений и т. д.) в виде приоритетной очереди, реализованной через двусвязный список. Каждый узел списка содержит поле приоритета и указатели на соседние элементы, что обеспечивает размещение новых формул в порядке убывания приоритета — формула с максимальным приоритетом всегда оказывается в начале списка[40]. Такая реализация соответствует принципу: «элемент с более высоким приоритетом находится перед элементом с более низким приоритетом», и оправдана в сценариях, где часто требуется обрабатывать или выбирать формулу с наивысшим приоритетом, как в приоритетной очереди[39][40]. Преимущество

двусвязного списка в том, что операция вставки или удаления узла при известной позиции выполняется за константное время.

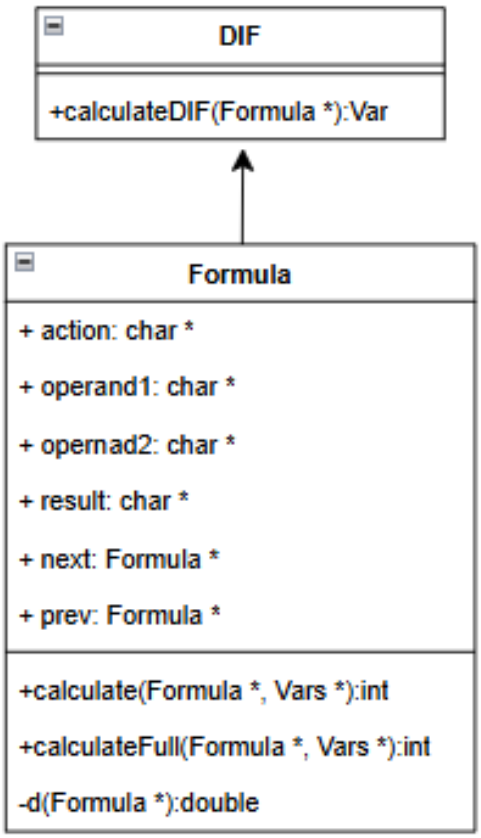

Рисунок 6 UML - диаграмма формул

Хранение самих формул организовано в двух уровнях: исходная строка сохраняется для удобства отображения, а для вычислений производится разбор в синтаксическое дерево (AST). Парсер выражений реализован по алгоритму «сортировочной станции» (Shunting-yard) Э. Дейкстры — он преобразует инфиксную запись в постфиксную или прямо строит дерево разбора[68]. Алгоритм последовательно обрабатывает лексемы (числа, переменные, операторы, скобки), соблюдая приоритеты: «*» и «/» выше «+» и «-», а возведение в степень «^» выше всех[69]. В результате получается древовидная структура, где внутренние узлы — операторы, листья — числовые константы или параметры; построив такой AST однажды, модуль может многократно вычислять значение формулы, меняя лишь значения листовых переменных[69][68].

Каждая формула хранит, помимо приоритета, указатель на функцию вычисления или иные данные, необходимые для расчёта. При выполнении вычислений модуль обходит список либо убывающего приоритета, либо последовательно, согласно общей модельной логике. Это соответствует классическому определению приоритетной очереди: «абстрактная структура данных, где у каждого элемента есть приоритет; элементы с более высоким приоритетом обрабатываются раньше»[39]. В качестве альтернативы могла бы быть использована бинарная куча, обеспечивающая схожие асимптотические характеристики операций вставки и удаления[39].

Для формирования системы ОДУ отдельные простые формулы могут композироваться — объединяться в одну совокупную систему через массив или вектор структур выражений. При вычислении ядро проходит по списку уже распарсенных деревьев формул и последовательно выполняет их расчёт с учётом текущих значений переменных. Единый интерфейс AST позволяет легко агрегировать и сортировать формулы, учитывать зависимости между параметрами и при необходимости сворачивать их в одно большое уравнение. Такой подход унифицирует обработку любых выражений независимо от их сложности.

3.5.Модуль решения дифференциальных уравнений

Модуль численного интегрирования системы обыкновенных дифференциальных уравнений реализован по паттерну «Adapter»: создаётся обёртка над внешней библиотекой (пример — GNU Scientific Library, GSL), которая принимает на вход наш список уравнений, начальные условия и шаг интегрирования, а затем трансформирует эти данные в формат, понятный GSL[70]). Такое решение обеспечивает независимость ядра приложения от

конкретного решателя и создаёт abstraction layer, позволяющий в дальнейшем без изменения остального кода заменить GSL на любую другую библиотеку.

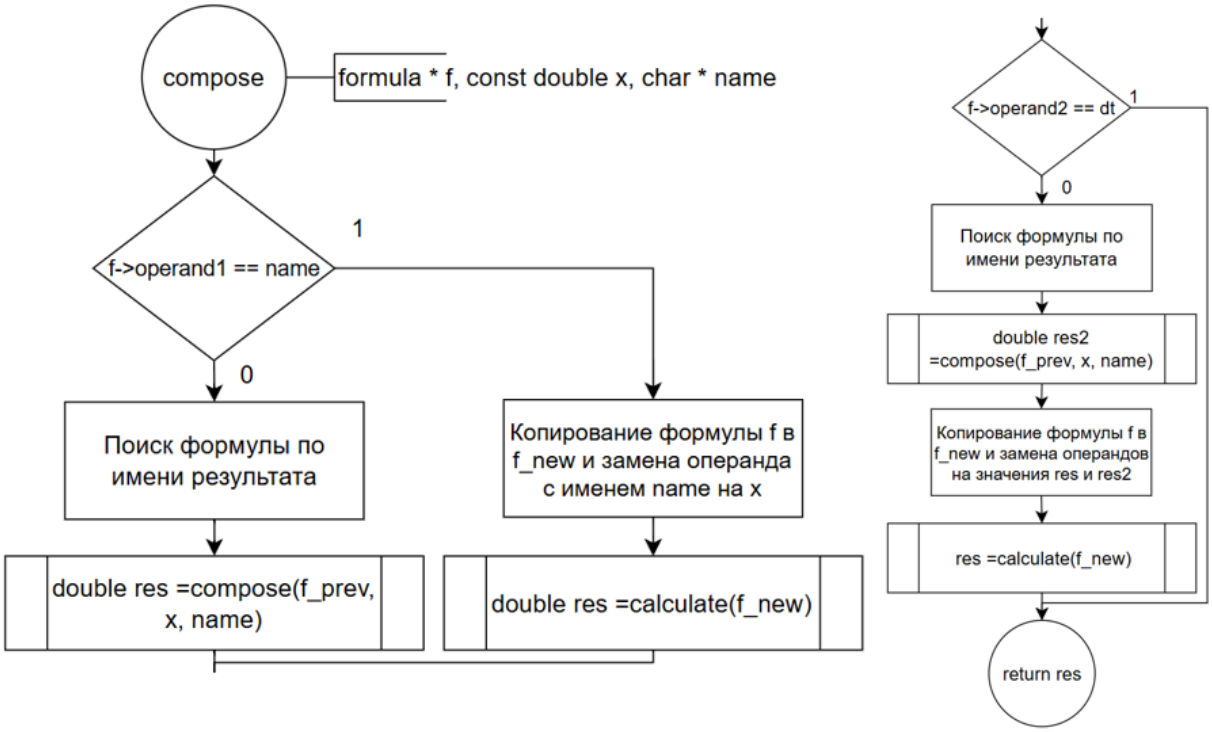

Рисунок 7 Блок-схема функции compose, для библиотеки GSL

Непрерывная часть модели, описанная системой ОДУ, интегрируется средствами GSL. Библиотека предоставляет функции для задания системы в виде $f(t, y)$, после чего встроенные алгоритмы с автоматическим контролем шага и погрешности последовательно вычисляют новое состояние системы (gnu.org). Для гибкой настройки и обеспечения требуемой точности используется объект gsl_odeiv2_driver, инкапсулирующий выбранный метод и выполняющий итерации по шагам времени. На каждом шаге модуль формирует вектор правых частей на основе текущих значений переменных и передаёт его в GSL, а библиотека возвращает состояние системы на следующем шаге.

Конфигурация задачи в GSL осуществляется через структуру gsl_odeiv2_system [71], куда передаётся указатель на функцию вычисления правой части ОДУ (и, опционально, на функцию Якобиана) вместе с параметрами модели. Для создания драйвера вызывается

gsl_odeiv2_driver_alloc_ynew — указывается конкретный метод интегрирования (например, косвенно через gsl_odeiv2_step_rkf45) и начальные условия. GSL включает широкий набор шаговых алгоритмов: от классического RK4 до адаптивных схем Runge–Kutta–Fehlberg 4(5), Cash–Karp, Dormand–Prince и других [71]. В данной работе в качестве универсального «работяги» чаще всего используется RKF45 (gsl_odeiv2_step_rkf45) с оценкой погрешности [71][72], а при необходимости подключаются специализированные методы для жёстких задач.

3.6.Механизм detect() для точного определения момента события

Механизм detect() предназначен для точного определения момента наступления событий в гибридных системах, моделируемых посредством непрерывной динамики. События регистрируются в тот момент, когда сигнальная функция пересекает заданный порог (например, меняет знак). Типичные интеграторы (например, GSL) могут лишь установить факт пересечения в некотором интервале между шагами, без указания точного времени события.

Для устранения этой неточности detect() применяет алгоритм бисекции: при события на отрезке $[t_1; t_2 = t_1 + dt]$ производится рекурсивное деление приращения времени пополам и попытка подойти к моменту наступления события с двух сторон: от t_1 и от t_2.

Учитывая, что событие произошло где-то между этим точками, в каждый момент приращения происходит перерасчет переменных модели и выявление искомого события. Если рассматривать позицию t_1, то мы сначала «догоняем» событие, каждый раз приращивая половину от предыдущего приращения. В случае если мы поймем, что мы обогнали событие, мы

начинаем вычитать, пока снова не окажемся в позиции догоняющего. Аналогично, но с обратными знаками действует t_2.

4. Примеры применения и тестирование

4.1.Задача двух баков

Система двух взаимосвязанных баков моделируется как гибридная система с двумя дискретными режимами (например, режим подачи потока в первый или второй бак). В каждом режиме непрерывные переменные (уровни жидкостей в баках) описываются собственными дифференциальными уравнениями. Математическое описание модели приведено в актуальности исследования (см. стр. 4, Два взаимосвязанных бака) На рисунке 8 схематично показана такая диаграмма состояний двухбаковой системы (два режима работы с условиями переключения по уровням баков).

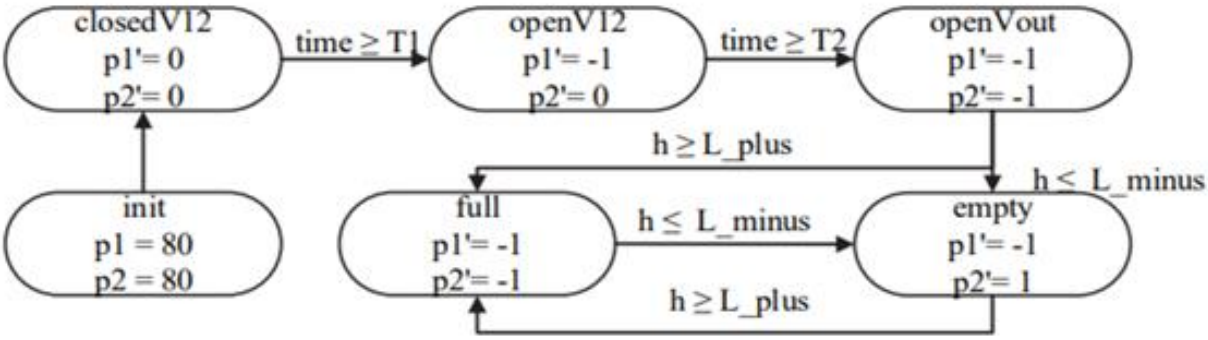

Рисунок 8 Диаграмма состояний задачи двух баков

На рисунке 9 представлен скриншот одного из состояний модели openVout в формате JSON

```
{"name": "openVout",
  "formulas": [{"result": "time","action": "add","operand1": "time","operand2": "dt"},
    {"result": "dp1","action": "d","operand1": "-1","operand2": "dt"},
    {"result": "dp2","action": "d","operand1": "-1","operand2": "dt"},
    {"result": "L1","action": "less","operand1": "p1","operand2": "P_MAX"},
    {"result": "res5","action": "pow","operand1": "p1","operand2": "3"},
    {"result": "res6","action": "mult","operand1": "-0.000006","operand2": "res5"},
    {"result": "res7","action": "exp","operand1": "res6","operand2": ""},
    {"result": "res8","action": "mult","operand1": "0.000185","operand2": "res7"},
    {"result": "K1","action": "mult","operand1": "res8","operand2": "L1"},
    {"result": "res9","action": "sub","operand1": "h2","operand2": "H"},
    {"result": "res10","action": "mult","operand1": "res9","operand2": "B"},
    {"result": "res11","action": "sub","operand1": "h1","operand2": "res10"},
    {"result": "res12","action": "sqrt","operand1": "res11","operand2": ""},
    {"result": "v12","action": "mult","operand1": "K1","operand2": "res12"},
    {"result": "res13","action": "sub","operand1": "V_in","operand2": "v12"},
    {"result": "res14","action": "div","operand1": "res13","operand2": "A1"},
    {"result": "h1","action": "d","operand1": "res14","operand2": "dt"},
    {"result": "res15","action": "sub","operand1": "v12","operand2": "v_out"},
    {"result": "res16","action": "div","operand1": "res15","operand2": "A2"},
    {"result": "h2","action": "d","operand1": "res1g","operand2": "dt"}],
  "transitions": [{"to": "empty","conditions":[{"result": "log_res1","action": "less_equal","operand1": "h","operand2": "L_minus"}],
      "action": [{"result": "B","action": "set","operand1": "0","operand2": ""},
        {"result": "res24","action": "min","operand1": "p1","operand2": "P_MAX"},
        {"result": "p1","action": "max","operand1": "res24","operand2": "P_MIN"},
        {"result": "v_out","action": "set","operand1": "0","operand2": ""}]},
    {"to": "empty","conditions":[{"result": "log_res1","action": "less_equal","operand1": "h","operand2": "L_minus"}],
      "action": [{"result": "B","action": "set","operand1": "1","operand2": ""},
        {"result": "res25","action": "min","operand1": "p1","operand2": "P_MAX"},
        {"result": "p1","action": "max","operand1": "res25","operand2": "P_MIN"},
        {"result": "res26","action": "sqrt","operand1": "h2","operand2": ""},
        {"result": "v_out","action": "mult","operand1": "K2","operand2": "res26"}]}]},
```


Рисунок 9 openVout в формате JSON

На рисунке 10 представлено состояние глобальной области переменных, после инициализации модели. На данном моменте система готова приступить к расчету модели

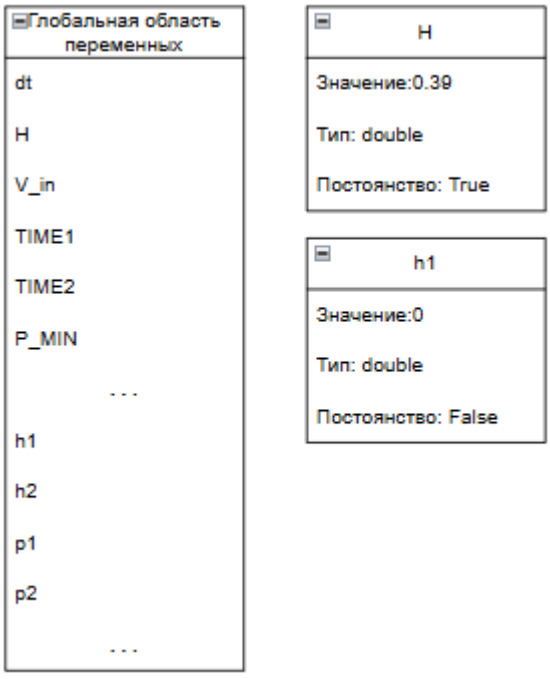

Рисунок 10 Глобальная область переменных сразу после инициализации

На рисунке 11 представлена схема модели в памяти, а так же подробно отображено состояние openVout, основа системы уравнения, переходы, и условия и система уравнения для перехода в состояние full

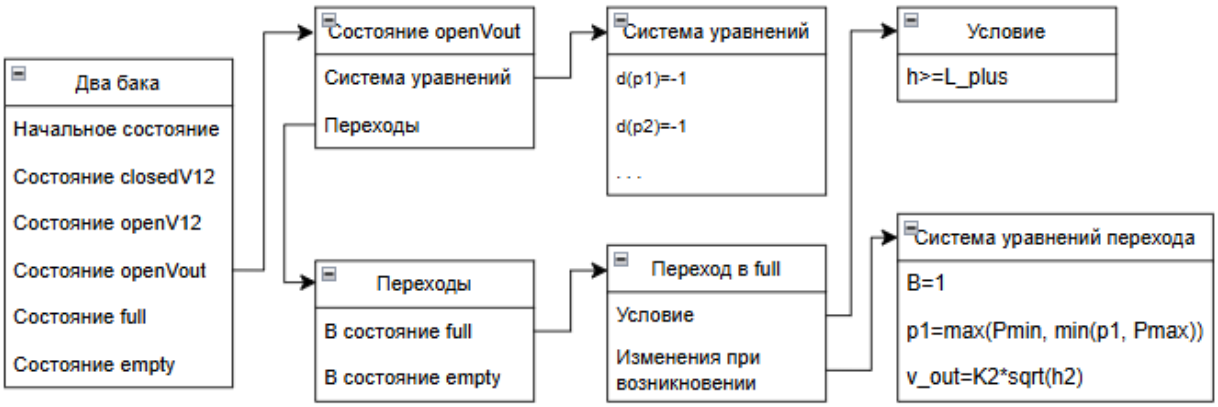

Рисунок 11 Отображение модели в памяти

В результате запуска и сбора данных можно построить график изменения h1(t) и h2(t), представленный на рисунке 12. Сравнивая с решениями другим инструментов, можно увидеть, что решение достаточно точно. На графике так же отмечены зеленые точки, они получены методом detect().

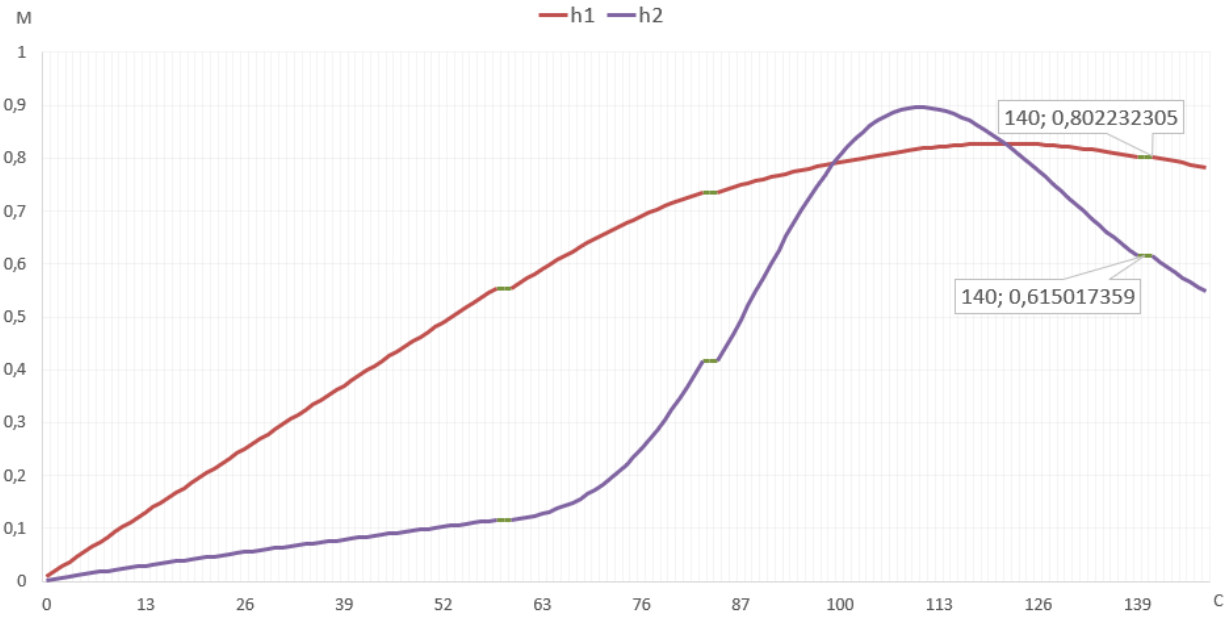

Рисунок 12 h1(t) и h2(t) при времени моделирования 180 секунд

4.2.Задача прыгающего мяча

Модель прыгающего мяча также строится по принципам гибридной автоматики. Основное дискретное состояние – свободное движение мяча под действием гравитации (обычно именуемое «полет» или «falling»). Диаграмма состояний представлена на рисунке 13

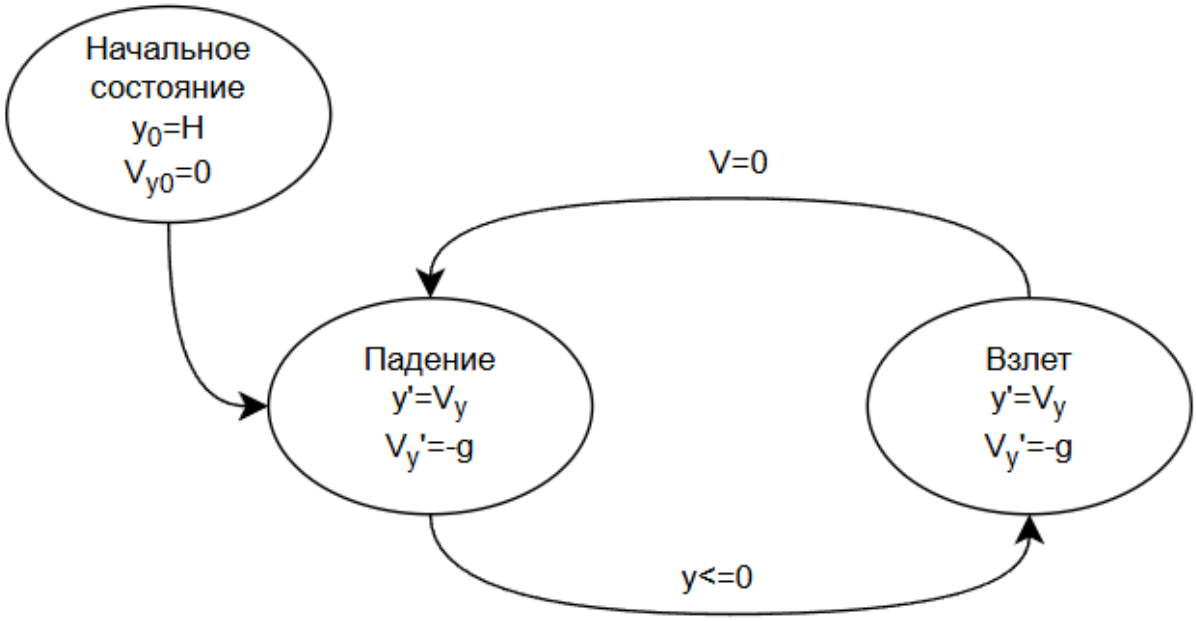

Рисунок 13 Диаграмма состояний модели прыгающего мяча

Модель в формате JSON представлена в приложении А. В отличие от модели двух баков модель мяча описывается меньшим количеством формул и выглядит более лаконично.

После инициализации модели область глобальных переменных состоит из 5 позиций (см. Рисунок 14)

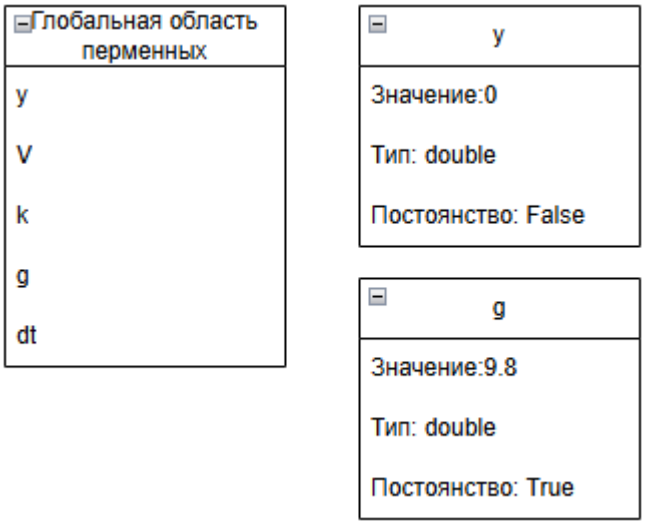

Рисунок 14 Глобальная область переменных сразу после инициализации в задаче прыгающего мяча

На рисунке 15 представлена схема модели в памяти, а так же подробно отображено состояние openVout, основа системы уравнения, переходы, и условия и система уравнения для перехода в состояние full

Рисунок 15 Отображение модели и состояния падения в памяти

На рисунке 16 можно ознакомиться с результатами моделирования прыгающего мяча. Точек гораздо меньше чем у двух баков, система куда более динамична. Снова можно наблюдать зеленые точки метода detect(). В данном случае куда лучше видно насколько сильно метод уточняет момент события.

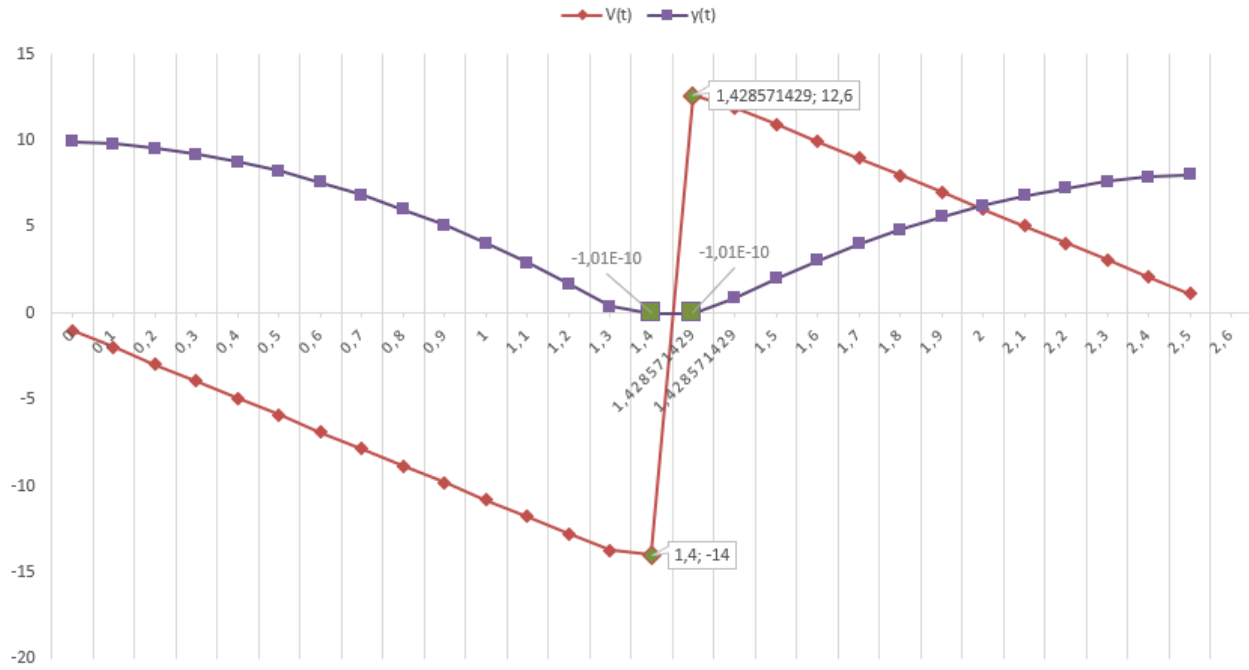

Рисунок 16 Графики V(t) и y(t) в задаче прыгающего мяча при времени моделирования 2.6 секунд

Заключение

Основные достижения работы

В ходе исследования разработана и реализована унифицированная архитектура симуляционной системы для гибридных (событийно–непрерывных) моделей. Предложенная архитектура сочетает движок непрерывной части (для численного решения систем ОДУ) и движок дискретных событий с центральным синхронизатором времени, обеспечивающим корректное переключение между режимами. Входной формат моделей основан на текстовом JSON-файле, в котором единообразно задаются состояния автомата, переходы, переменные непрерывной динамики и их уравнения. Система реализована на языке C с модульной многоуровневой структурой: парсер модели, ядро симулятора, менеджер событий и модуль вывода изолированы друг от друга и взаимодействуют через чётко определённые API. Это позволило добиться высокой расширяемости: например, возможна замена алгоритмов интегрирования ОДУ или процедур обработки событий без изменения остальной части системы.

Создан прототип платформы, в котором объединены все ключевые компоненты архитектуры. В непрерывной части использована библиотека GSL с адаптивными схемами Runge–Kutta (например, RKF45) для интегрирования ОДУ. Для точного определения моментов срабатывания событий реализован алгоритм «detect()» с рекурсивной бисекцией шага, что минимизирует численные погрешности при обнаружении перехода. Проведены экспериментальные симуляции ряда эталонных гибридных моделей (масс–пружина, «два бака», отскакивающий мяч и др.), которые подтвердили корректность работы

системы и её эффективность по сравнению с традиционными инструментами (Simulink/Stateflow и др.). Результаты показали, что предложенный симулятор достигает высокой скорости вычислений за счёт низкоуровневого исполнения на C и простого формата данных, при этом гибкость архитектуры позволяет легко менять структуру модели или повторно использовать компоненты. В частности, интеграция JSON-формата с C-реализацией симулятора обеспечивает быстрый прогон модели без виртуальных машин или тяжёлых XML-библиотек. Подытоживая, можно отметить, что все поставленные цели достигнуты: разработана эффективная унифицированная платформа гибридного моделирования с продуманной архитектурой и проверенным прототипом.

Практическая значимость

Предложенная система имеет существенную практическую ценность для инженеров и исследователей, работающих с гибридными моделями. Унифицированный подход к описанию и симуляции позволяет объединять разнородные модели (из разных доменов) без «зашивания» ad hoc решений и потери масштабируемости. Это особенно важно в критически значимых областях (робототехника, энергосистемы, автомобилестроение и др.), где широко используются гибридные модели. Так, возможность описывать переходы состояний и непрерывную динамику в одном JSON-файле упрощает обмен моделями между инструментами и повышает переносимость результатов. Высокая производительность симулятора (движок на C, отсутствие сборщика мусора) позволяет применять его для задач, чувствительных ко времени расчёта (например, цифровые двойники и реал-тайм симуляции). Модульная архитектура облегчает интеграцию системы в САПР или систему управления: разработчики могут подменять или расширять

компоненты (интеграторы, обработчики событий, форматы ввода/вывода) в соответствии со своими требованиями. В совокупности, платформа демонстрирует потенциальное снижение трудоёмкости разработки гибридных моделей и повышение надёжности систем за счёт формализации архитектуры симулятора.

Перспективы дальнейших исследований

Среди направлений дальнейших исследований можно выделить совершенствование численных методов и расширение функциональности системы. Во-первых, при моделировании очень жёстких или сильно нелинейных динамик текущие интеграторы (например, RKF45) могут испытывать трудности: это приводит к увеличению численных погрешностей и затрат времени. Перспективно изучить применение специализированных жёстких солверов или многопоточных схем интегрирования (с помощью OpenMP/POSIX threads) для повышения точности и производительности в таких задачах. Во-вторых, модульная многопоточность сама по себе предлагает перспективу – можно реализовать распараллеливание независимых подсистем или пакетных запусков модели, что особенно актуально для больших систем со множеством подсистем.

Кроме того, целесообразно расширять поддержку форматов и стандартов. Например, интеграция с готовыми контейнерами FMU/FMI позволит использовать внешние модели и объединять симуляторы разных типов. В дальнейшем возможно разработать графический интерфейс или DSL для автоматической генерации JSON-моделей, что упростит работу пользователям без программирования на C. Наконец, интерес представляет расширение области применения системы на верификацию и анализ гибридных моделей (например, автоматическая проверка свойств или оптимизация параметров), что потребует разработки новых модулей анализа и учёта допущенных численных

неточностей. В целом перспективы включают как углубление алгоритмической части (улучшение схем интегрирования и детекции событий), так и расширение прикладных возможностей платформы в соответствии с отраслевыми потребностями.

Ограничения. При этом следует отметить, что в текущей реализации сохраняются некоторые ограничения. Во-первых, несмотря на алгоритм detect(), в симуляции может накапливаться небольшая численная погрешность из-за адаптивного шага и конечной точности вычислений. Во-вторых, архитектура системы в значительной степени зависит от внешних библиотек (GSL для решения ОДУ, парсеры JSON – cJSON/Jansson и др.), что накладывает требования на совместимость окружения и может ограничивать переносимость. Наконец, хотя система способна моделировать нелинейную динамику, чрезвычайно сложные физические процессы могут потребовать дополнительной аппроксимации или более тонкой настройки интеграторов. Учитывая эти факторы, дальнейшие работы должны сконцентрироваться на снижении численных ошибок и расширении класса решаемых задач, сохраняя при этом доказанную эффективность архитектуры.

Список литературы

1. Лабинац Г., Баюми М.М., Руди К. Обзор моделирования и управления гибридными системами // Ежегодный обзор в области управления (Annual Reviews in Control). – 1997. – Т. 21. – С. 79–92.

2. Гебель Р., Санфелисе Р.Г., Тил А.Р. Гибридные динамические системы: моделирование, анализ и управление. – Принстон: Принстонский университет, 2012. – 447 с.

3. Грисли А. Проектирование и реализация гибридного моделирования // Труды 34-й Европейской конференции по моделированию и имитации (EMSS 2022). – 2022.

4. Яакуби Х., Хагге Ж. Моделирование и имитация двухбаковой системы в гибридной среде // Международный журнал электротехники и вычислительной техники (IJECE). – 2021. – Т. 13, № 4. – С. 4222–4233.

5. Зайглер Б.П., Праэофер Х., Ким Т.Г. Теория моделирования и имитации: интеграция дискретных событий и непрерывных сложных динамических систем. – 2-е изд. – Нью-Йорк: Академическое издательство, 2000. – 510 с.

6. Раскин Ж.-Ф. Введение в гибридные автоматы: учебные материалы. – Брюссель: Свободный университет Брюсселя, 2010.

7. Обен Ж.-П., Лигерос Й., Кенкампуа М., Састри С., Сёб Н. Импульсные дифференциальные включения: подход жизнеспособности к гибридным системам // Труды IEEE по автоматическому управлению. – 2002.

8. Бемпорад А., Морари А. Управление системами с интеграцией логики, динамики и ограничений // Автоматика. – 1999.

9. Чуиха М., Шнайдер Э. Моделирование и анализ непрерывно-дискретных систем с использованием гибридных сетей Петри // Труды IFAC. – 2000. – Т. 32, № 2.

10. Гомри Л., Алла Х. Моделирование и анализ с применением гибридных сетей Петри // Международный журнал производственных исследований. – 2007. – Т. 45, № 6.

11. Хадим У. Алгебры процессов для гибридных систем: сравнение и разработка: дис. … д-ра философии. – Эйндховен: Технологический университет Эйндховена, 2008.

12. Птолемеус К. (ред.). Проектирование, моделирование и имитация систем с использованием Ptolemy II. – Ptolemy.org, 2014.

13. Фритцсон П. Дискретные события и гибридные системы в Modelica: лекции по OpenModelica. – 2015.

14. Лю Цз., Лю Сяо., Ко Т.-К., Синополи Б. Иерархическое моделирование и имитация гибридных систем с использованием Ptolemy II // Труды конференции IEEE по теории управления и принятия решений. – 1999. – С. 3508–3513.

15. Харди Т.Д. и др. HELICS: среда ко-симуляции для масштабного многодисциплинарного моделирования и анализа // IEEE Access. – 2024. – С. 24328–24334.

16. Ходжаоглу М.Ф., Фират Ч. DEVS/RAP: имитационное моделирование на основе агентов // Труды конференции ECMS. – 2009.

17. Баккер М. Валидация моделей для стохастических гибридных автоматов: магистерская диссертация. – Твенте: Университет Твенте, 2017.

18. Брей Т. (ред.). RFC 8259: Формат обмена данными JavaScript Object Notation (JSON). – IETF, декабрь 2017. – URL: https://datatracker.ietf.org.

19. Использование документов JSON. Руководство разработчика Couchbase [Электронный ресурс]. – URL: https://developer.couchbase.com (дата обращения: 2025).

20. Джайвардхан. Как язык C используется для разработки физических симуляторов в реальном времени [Электронный ресурс]. – 2023. – URL: https://jaivardhan.com.

21. Functional Mock-up Interface (FMI) для обмена моделями и ко-симуляции. Версия 2.0. – FMI-стандарт, 2014. – URL: https://fmi-standard.org.

22. Гудман С. Научные вычисления. Осень 2024: Имитационное моделирование, управляемое событиями: лекционные материалы. – Нью-Йорк: Нью-Йоркский университет, 2024. – URL: https://math.nyu.edu.

23. Лук В. Теория дискретно-событийного моделирования: основы. Лекции NYU. – Нью-Йорк, 2024. – URL: https://math.nyu.edu.

24. Зайглер Б. П. Теория моделирования и имитации. – Нью-Йорк: Academic Press, 1976. – 356 с.

25. Лоу А. М., Келтон У. Д. Моделирование и анализ имитации. – Нью-Йорк: McGraw-Hill, 1991. – 760 с.

26. Дискретно-событийное моделирование // Википедия [Электронный ресурс]. – URL: https://ru.wikipedia.org (дата обращения: 2025).

27. Лук В. и др. Курс лекций по имитационному моделированию, управляемому событиями. – Нью-Йорк: Нью-Йоркский университет, 2024. – URL: https://math.nyu.edu.

28. MathWorks. Обнаружение перехода через ноль в Simulink. – Документация MATLAB. – 2014.

29. Майер Р. Хронологическая диаграмма модели // Радиоэлектроника. – СПб., 2019.

30. Руководство по NS-3: События и планировщик // nsnam.org [Электронный ресурс]. – URL: https://www.nsnam.org.

31. Нильс Ф. Управление памятью в C: применение функций malloc и free. – 2018.

32. Tutorialspoint. Структуры данных – Двоичное дерево поиска [Электронный ресурс]. – URL: https://tutorialspoint.com.

33. GeeksforGeeks. Таблица символов в проектировании компиляторов [Электронный ресурс]. – URL: https://geeksforgeeks.org.

34. Прогуральник C-систем. Двусвязный список на языке C // PVS-Studio. – 2023.

35. Гэмбл Д. cJSON: Однофайловый парсер JSON на ANSI C [Электронный ресурс]. – GitHub. – URL: https://github.com.

36. Sky.Pro. Парсинг JSON на C: библиотеки и примеры [Электронный ресурс]. – URL: https://sky.pro.

37. Хареон Т. GitHub: Jansson README [Электронный ресурс]. – URL: https://github.com.

38. json-c Project. JSON-C – реализация JSON на языке C [Электронный ресурс]. – URL: https://github.com.

39. GeeksforGeeks. Очередь с приоритетом: описание структуры данных [Электронный ресурс]. – URL: https://neerc.ifmo.ru.

40. ScholarHat (DSA Tutor). Реализации очереди с приоритетом [Электронный ресурс]. – URL: https://scholarhat.com.

41. Кормен Т. Х., Лейзерсон Ч. Э., Ривест Р. Л., Штайн К. Введение в алгоритмы. – 3-е изд. – Глава 6: Кучи и очереди с приоритетом. – МИТ Пресс, 2009.

42. GNU Scientific Library. Раздел: Решение ОДУ. – Документация GSL [Электронный ресурс]. – URL: https://www.gnu.org.

43. GSL: Пример нахождения корней уравнений // Справка Университета Юты [Электронный ресурс]. – URL: https://math.utah.edu.

44. Лундвалль Х. и др. Обработка событий в OpenModelica. – Научный отчет. – Линчёпинг: Технический университет, 2008. – URL: https://liu.diva-portal.org.

45. Лундвалль Х. и др. Пересекающиеся функции в гибридном моделировании. – Линчёпинг: Университет Линчёпинга, 2008. – URL: https://liu.diva-portal.org.

46. Пресс У. Х. и др. Численные рецепты. – Кембридж: Cambridge University Press, 2007. – 1256 с.

47. Хайр Е., Ваннер Г. Решение обыкновенных дифференциальных уравнений. Том I: Нестифельтные методы. – Берлин: Springer, 1993. – 528 с.

48. Руководство по GNU Scientific Library. Раздел: ОДУ. – [Электронный ресурс]. – URL: https://www.gnu.org.

49. MathWorks. Обнаружение перехода через ноль в Simulink. – Справка MATLAB. – 2020. – URL: https://mathworks.com.

50. Швейцер Б. Событийное моделирование телекоммуникационных систем. – СПб.: КомПас, 2021. – 164 с.

51. Пак Дж. Интегрированное дискретно-событийное и непрерывное моделирование // IEEE Transactions. – 2019.

52. Шаститко В. Численные методы решения обыкновенных дифференциальных уравнений. – М.: Математика, 2020. – 232 с.

53. Мишра П. К. Структуры данных и алгоритмы на языке C. – Нью-Йорк: Wiley, 2008. – 468 с.

54. Гофман Н. Таблица символов в проектировании компиляторов // GeeksforGeeks. – 2021. – URL: https://geeksforgeeks.org.

55. Штейн Д. Моделирование на основе Simulink: учебный вебинар MathWorks. – 2019. – URL: https://mathworks.com.

56. The Simio Company. Календарь ресурсов против событийного планирования: технический документ Simio. – 2015.

57. AnyLogic. Справка по дискретно-событийному моделированию: руководство пользователя. – 2021. – URL: https://anylogic.com.

58. Ли Э. А., Санжованни-Винчентили А. Гибридные системы: моделирование и анализ // IEEE. – 1998.

59. Франк М. Гибридное моделирование физических систем. – Берлин: Springer, 2010. – 245 с.

60. Кюрбис А. Двоичные деревья поиска: AVL и красно-чёрные деревья. – Лекция. – Мюнхен: Технический университет, 2017.

62. Кормен Т. Х., Лейзерсон Ч. Э., Ривест Р. Л., Штайн К. Введение в алгоритмы. – 4-е изд. – Главы 12–16. – М.: МИТ Пресс, 2022. – 1312 с.

63. Гупта Р. Шаблоны архитектуры программного обеспечения: какие бывают и какой выбрать для проекта // Turing. – 2024. – URL: https://turing.com.

64. Полное руководство по архитектуре программного обеспечения // Deazy. – 2023. – URL: https://deazy.com.

65. Гэмбл Д. cJSON: сверхлёгкий парсер JSON на ANSI C [Электронный ресурс]. – GitHub, 2017. – URL: https://github.com.

66. GeeksforGeeks. cJSON – Чтение, запись и изменение JSON-файлов на языке C. – 24 апр. 2025. – URL: https://geeksforgeeks.org.

67. Таблица символов // Википедия [Электронный ресурс]. – Последнее обновление: 2013. – URL: https://ru.wikipedia.org.

68. Структуры в C++ // Ravesli. – 2023. – URL: https://ravesli.com.

69. Алгоритм сортировки по возрастанию (Shunting yard algorithm) // Википедия [Электронный ресурс]. – Дата обращения: 2025. – URL: https://en.wikipedia.org.

70. Pilum. Алгоритм парсинга арифметических выражений // Хабр. – 29 июл. 2015. – URL: https://habr.com.

71. Паттерн «Адаптер» // Refactoring.Guru [Электронный ресурс]. – Дата обращения: 2025. – URL: https://refactoring.guru.

72. GNU Scientific Library Reference Manual. Раздел 26: Обыкновенные дифференциальные уравнения. – 2016. – URL: https://doku.lrz.de.

73. Уорбертон Н., Уорделл Н. Расчёты гравитационного самодействия в сильных полях // arXiv:1012.5860 [Электронный ресурс]. – URL: https://arxiv.org.

Приложения

Приложение А. Пример JSON модели для задачи падающего мяча


```
{
"stances": [
{
"name": "0",
"formulas": [
{"result": "y","action": "set","operand1": "10.0"},
{"result": "V","action": "set","operand1": "0.0"},
{"result": "time","action": "set","operand1": "0.0"},
{"result": "k","action": "const","operand1": "0.9"},
{"result": "g","action": "const","operand1": "9.8"},
{"result": "dt","action": "const","operand1": "0.1"}
],
"transitions": [{"to": "down"}]
},
{
"name": "down",
"formulas": [
{"result": "time","action": "add","operand1": "time","operand2": "dt"},
{"result": "res1","action": "mult","operand1": "-1","operand2": "g"},
{"result": "V","action": "d","operand1": "res1","operand2": "dt"},
{"result": "y","action": "d","operand1": "V","operand2": "dt"}
],
"transitions": [
{
```

```
"to": "up",
"conditions":[{"result": "log_res1","action": "less_equal","operand1": "y","operand2": "0.0"}],
"action": [
{"result": "res2","action": "mult","operand1": "k","operand2": "V"},
{"result": "V","action": "mult","operandS1": "-1","operand2": "res2"}
]
}
]
},
{
"name": "up",
"formulas": [
{"result": "time","action": "add","operand1": "time","operand2": "dt"},
{"result": "res3","action": "sub","operand1": "0","operand2": "g"},
{"result": "V","action": "d","operand1": "res3","operand2": "dt"},
{"result": "y","action": "d","operand1": "V","operand2": "dt"}
],
"transitions": [
{
"to": "down",
"conditions":[{"result": "log_res1","action": "equal","operand1": "V","operand2": "0.0"}]
}
]
}
]
}
```

Приложение Б. Фрагменты кода основных модулей

vars.h

```
#ifndef VARS_H
#define VARS_H

#ifdef DEBUG_VARS
#define vars_to_str(var) ({                          \
   static char buf[DEBUG_BUFFER];                    \
   if ((var) == NULL)                                \
      snprintf(buf, DEBUG_BUFFER, "NULL");           \
   else                                              \
      snprintf(buf, DEBUG_BUFFER,                    \
            "Variable: %s = %.2f",                   \
            (var)->name ? (var)->name : "NULL", \
            (var)->value);                           \
   buf;                                              \
})
#define printd_var(var)                                      \
   {                                                         \
      printf("<%s>\n%s\n", __func__, vars_to_str(var)); \
   }

#define printd_vars_list(var)                                \
   {                                                         \
      vars *v = var;                                         \
      printf("<%s>Vars list begin\n", __func__);             \
      do                                                     \
      {                                                      \
         printf("%s\n", vars_to_str(var));                   \
         if (v)                                              \
```

```
            {                                       \
                v = v->next;                            \
                printf("v->next=%s\n", v ? v->name : NULL); \
            }                                       \
        } while (v);                                    \
        printf("Vars list end\n");                          \
    }
#else
#define vars_to_str(var)
#define printd_var(var)
#define printd_vars_list(var)
#endif // DEBUG_VARS

typedef struct vars
{
    char * name;
    double value;
    struct vars * next;
}vars;

typedef struct vars_pointer
{
    vars * first;
}vars_pointer;

int add_to_end(vars_pointer * head, char * name, double value);
vars * find(vars_pointer * head, char * name);
vars * last(vars_pointer * head);
int delete_by_name(vars_pointer * head, char * name);
int clear(vars_pointer * head);
```

```
double get_time_delta(vars_pointer * head);

#endif //VARS_H
```

vars.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vars.h"

int add_to_end(vars_pointer *head, char *name, double value) {

    if (!head || !name) {
        return -1;
    }

    vars *new_var = (vars *)calloc(1, sizeof(vars));
    if (!new_var) {
        return -1;
    }

    new_var->name = strdup(name);
    new_var->value = value;
    new_var->next = NULL;

    if (!head->first)
    {
        head->first = new_var;
    }
    else
```

```
    {
        vars *temp = head->first;
        temp = head->first;
            while (temp->next)
                temp = temp->next;
        if (0 != strcmp(temp->name, new_var->name))
        {
            temp->next = new_var;
        }
    }
    printd_var(new_var);
    return 0;
}

vars *last(vars_pointer *head)
{
    if (!head)
    {
        return NULL;
    }
    vars *a = head->first;
    while (a->next)
        a = a->next;
    return a;
}

vars *find(vars_pointer *head, char *name) {
    if (!head || !name) {
        return NULL;
    }
```

```
    vars *current = head->first;

    while (current) {
        if (streq(current->name, name)) {
            return current;
        }
        current = current->next;
    }

    return NULL;
}


int delete_by_name(vars_pointer *head, char *name) {
    if (!head || !name) {
        return -1;
    }

    vars *current = head->first;
    vars *prev = NULL;

    while (current) {
        if (strcmp(current->name, name) == 0) {

            if (prev) {
                prev->next = current->next;
            } else {
                head->first = current->next;
            }
```

```
            free(current->name);
            free(current);
            return 0;
        }

        prev = current;
        current = current->next;
    }

    return -1;
}


int clear(vars_pointer *head) {
    if (!head) {
        return -1;
    }

    printd_vars_list(head->first);
    vars *current = head->first;
    while (current) {
        vars *next = current->next;

        current->next=NULL;

        free(current->name);
        free(current);
        current = next;
    }
```

```
    head->first = NULL;

    return 0;
}

double get_time_delta(vars_pointer *head)
{
    if (!head)
        return 1.0f;

    vars * time = find(head, "time");
    if (time)
        return time->value;

    return 1.0f;
}
```

formula.h

```
#ifndef FORMULA_H
#define FORMULA_H

#ifdef DEBUG_FORMULA
#define printd_formula(...){\
    printf("<%s>", __func__);\
    printf(__VA_ARGS__);\
    printf("\n");\
}
#else
#define printd_formula(...)
```

```
#endif //DEBUG_FORMULA

#ifdef DEBUG_OPERATIONS
#define operation_to_str(op) ({ \
   static char buf[DEBUG_BUFFER]; \
   if ((op) == NULL) snprintf(buf, DEBUG_BUFFER, "NULL"); \
   else snprintf(buf, DEBUG_BUFFER, \
      "Operation: %s = %s(%s, %s) (is_bool: %d, is_a_const: %d, a_const: %.2f, is_b_const: %d, b_const: %.2f)", \
      (op)->result ? (op)->result : "NULL", \
      (op)->func ? (op)->func : "NULL", \
      (op)->is_a_const ? "(const)" : ((op)->a ? (op)->a : "NULL"), \
      (op)->is_b_const==1 ? "(const)" : ((op)->is_b_const==0 ? (op)->b : "NULL"), \
      (op)->is_bool, \
      (op)->is_a_const, \
      (op)->a_const, \
      (op)->is_b_const, \
      (op)->b_const \
   ); \
   buf; \
})
#define printd_operations(op){\
   printf("<%s>\n%s\n", __func__, operation_to_str(op));\
}
#else
#define operation_to_str(op)
#define printd_operations(op)
#endif //DEBUG_OPERATIONS
```

```
typedef struct operation
{
   char * a, * b, *result, *func;
   int is_bool, is_a_const, is_b_const;
   double a_const, b_const;
   struct operation * next;
}operation;

typedef struct formula
{
   operation * first;
}formula;

operation operation_init(char *a, char *b, char * result, char *func, double a_const, double b_const, int is_a, int is_b);
int add_new_formula(formula * f, operation * o);
int clear_formula(formula * f);

double add(double a, double b);
double subtract(double a, double b);
double multiply(double a, double b);
double divide(double a, double b);
int mod(int a, int b);
double fdx(double y, double h);

int greater_than(double a, double b);
int less_than(double a, double b);
int equals(double a, double b);
int greater_equal(double a, double b);
int less_equal(double a, double b);
```

```
int logical_or(int a, int b);
int logical_and(int a, int b);
int logical_not(int a);

#endif // FORMULA_H
```

formula.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "formula.h"

#ifdef HOME_MADE_INTEGRAL
   #include "integral.h"
#endif

operation operation_init(char *a, char *b, char * result, char *func, double a_const, double b_const, int is_a, int is_b) {
   operation o;
   if (is_a)
   {
      o.is_a_const = 0;
      o.a = strdup(a);
   }
   else
   {
      o.is_a_const = 1;
      o.a = strdup("");
      o.a_const = a_const;
   }
```

```
    if (is_b)
    {
        o.is_b_const = 0;
        o.b = strdup(b);
    }
    else
    {
        o.is_b_const = 1;
        o.b = strdup("");
        o.b_const = b_const;
    }
    o.func = strdup(func);
    o.result = strdup(result);
    o.is_bool = 0;
    o.next = NULL;
    return o;
}

int add_new_formula(formula *f, operation *o) {
    if (!f || !o) {
        return -1;
    }

    if (!f->first) {
        f->first = o;
    } else {
        operation *current = f->first;
        while (current->next) {
            current = current->next;
        }
```

```
        current->next = o;
    }

    return 0;
}

int clear_formula(formula *f) {
    if (!f) {
        return -1;
    }

    operation *current = f->first;
    while (current) {
        operation *next = current->next;
        free(current->a);
        free(current->b);
        free(current->func);
        free(current);
        current = next;
    }

    f->first = NULL;
    return 0;
}

double add(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a + b;
}
```

```
double subtract(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a - b;
}

double multiply(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a * b;
}

double divide(double a, double b) {
    printd_formula("%lf %lf", a, b);
    if (b == 0) {
        // Предотвращение деления на 0
        return 0.0;
    }
    return a / b;
}

int mod(int a, int b)
{
    printd_formula("%d %d", a, b);
    if (b == 0) {
        // Предотвращение деления на 0
        return 0;
    }
    return a % b;
}

double fdx(double y, double h)
```

```
{
    #ifdef HOME_MADE_INTEGRAL
    printd_formula("using euler");
    return euler(y,h);
    #else
    printd_formula("using custom integral solution");
    //ваше дело
    return y;
    #endif
}

// Логические операции
int greater_than(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a > b;
}

int less_than(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a < b;
}

int equals(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a == b;
}

int greater_equal(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a >= b;
```

```
}

int less_equal(double a, double b) {
    printd_formula("%lf %lf", a, b);
    return a <= b;
}

int logical_or(int a, int b) {
    printd_formula("%d %d", a, b);
    return (a || b);
}

int logical_and(int a, int b) {
    printd_formula("%d %d", a, b);
    return (a && b);
}

int logical_not(int a) {
    printd_formula("%d", a);
    return !a;
}
```

model.h

```
#include "vars.h"
#include "formula.h"

#ifdef DEBUG_MODEL
#define printd_model(...){\
    printf("<%s>", __func__);\
    printf(__VA_ARGS__);\
```

```
    printf("\n");\
}
#else
#define printd_model(...)
#endif

typedef struct stance
{
    int id;
    formula *f;
    struct hopes *h;
    struct stance *next;
} stance;

typedef struct hopes
{
    stance *target;
    formula *condition;
    formula *f;
    struct hopes *next;
} hopes;

typedef struct model
{
    vars_pointer *vars_p;
    stance *stances;
    __time_t time;
} model;

stance init_stance(int id, formula *f);
```

```
int add_stance(model *m, stance *s);
int clear_stances(model *m);

int add_hope(stance *source, stance *target, formula *f, formula *c);
int clear_hopes(stance *s);

void print_formula(formula *f);
void print_hopes(hopes *h);
void print_stance(stance *s);
model *parse_model(char * path);
void print_model(model *m);
```

model.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "model.h"
#include "cJSON.h"

stance init_stance(int id, formula *f) {
    stance s;
    s.id = id;
    s.f = f;
    s.h = NULL;
    s.next = NULL;
    return s;
}

int add_stance(model *m, stance *s) {
    if (!m || !s) {
```

```
        return -1;
    }

    if (!m->stances) {
        m->stances = s;
    } else {
        stance *current = m->stances;
        while (current->next) {
            current = current->next;
        }
        current->next = s;
    }

    return 0;
}

int clear_stances(model *m) {
    if (!m) {
        return -1;
    }

    stance *current = m->stances;
    while (current) {
        stance *next = current->next;


        if (current->f) {
            clear_formula(current->f);
            free(current->f);
        }
```

```
        clear_hopes(current);

        free(current);
        current = next;
    }

    m->stances = NULL;
    return 0;
}

int add_hope(stance *source, stance *target, formula *f, formula *c) {
    if (!source || !target || !f || !c) {
        return -1;
    }

    hopes *new_hope = (hopes *)calloc(1, sizeof(hopes));
    if (!new_hope) {
        return -1;
    }

    new_hope->target = target;
    new_hope->condition = c;
    new_hope->f = f;
    new_hope->next = NULL;

    if (!source->h) {
        source->h = new_hope;
    } else {
```

```
        hopes *current = source->h;
        while (current->next) {
            current = current->next;
        }
        current->next = new_hope;
    }

    return 0;
}

int clear_hopes(stance *s) {
    if (!s) {
        return -1;
    }

    hopes *current = s->h;
    while (current) {
        hopes *next = current->next;


        if (current->condition) {
            clear_formula(current->condition);
            free(current->condition);
        }
        if (current->f) {
            clear_formula(current->f);
            free(current->f);
        }

        free(current);
```

```
        current = next;
    }

    s->h = NULL;
    return 0;
}

int is_number(const char *str) {
    if (!str || *str == '\0') return 0;

    char *endptr;
    strtod(str, &endptr);
    return *endptr == '\0';
}

operation *create_operation(cJSON *json_op) {
    operation *op = (operation *)calloc(1, sizeof(operation));
    if (!op) {
        fprintf(stderr, "Memory allocation error for operation.\n");
        return NULL;
    }

    op->a = NULL;
    op->b = NULL;
    op->result = NULL;
    op->func = NULL;
    op->is_bool = 0;
    op->is_a_const = 0;
    op->is_b_const = 0;
    op->a_const = 0.0;
```

```
    op->b_const = 0.0;
    op->next = NULL;

    char *operand1 = cJSON_GetObjectItem(json_op, "operand1")?cJSON_GetObjectItem(json_op, "operand1")->valuestring:strdup("");
    char *operand2 = cJSON_GetObjectItem(json_op, "operand2")?cJSON_GetObjectItem(json_op, "operand2")->valuestring:strdup("");
    char *result = cJSON_GetObjectItem(json_op, "result")?cJSON_GetObjectItem(json_op, "result")->valuestring:strdup("");
    char *action = cJSON_GetObjectItem(json_op, "action")?cJSON_GetObjectItem(json_op, "action")->valuestring:strdup("");

    if (is_number(operand1)) {
        op->is_a_const = 1;
        op->a_const = atof(operand1);
    } else {
        op->is_a_const = 0;
        op->a = strdup(operand1);
    }

    if (operand2)
    {
        if (is_number(operand2))
        {
            op->is_b_const = 1;
            op->b_const = atof(operand2);
        }
        else
```

```
        {
            if (strcmp(operand2, "") == 0)
                op->is_b_const = -1;
            else
            {
                op->is_b_const = 0;
                op->b = strdup(operand2);
            }
        }
    }

    op->result = strdup(result);

    op->func = strdup(action);

    if (strcmp(action, "equals") == 0 ||
        strcmp(action, "greater_than") == 0 ||
        strcmp(action, "less_than") == 0 ||
        strcmp(action, "greater_equal") == 0 ||
        strcmp(action, "less_equal") == 0 ||
        strcmp(action, "logical_or") == 0 ||
        strcmp(action, "logical_and") == 0 ||
        strcmp(action, "logical_not") == 0) {
        op->is_bool = 1;
    } else {
        op->is_bool = 0;
    }

    return op;
}
```

```
formula *create_formula(cJSON *json_formulas) {
    formula *f = (formula *)calloc(1, sizeof(formula));
    f->first = NULL;
    operation *last_op = NULL;

    cJSON *json_op = NULL;
    cJSON_ArrayForEach(json_op, json_formulas) {
        operation *op = create_operation(json_op);
        if (!f->first) {
            f->first = op;
        } else {
            last_op->next = op;
        }
        last_op = op;
    }

    return f;
}

static void add_variables(vars_pointer *vars_p, formula * f) {
    if (!vars_p)
        vars_p = (vars_pointer *)calloc(1, sizeof(vars_pointer));

    operation * item = f->first;

    while (item)
    {
        printd_operations(item);
```

```
        if (!item->is_a_const)
            if (!find(vars_p, item->a))
            {
                add_to_end(vars_p, item->a, 0.0);
            }
        if (item->is_b_const==0)
            if (!find(vars_p, item->b))
            {
                add_to_end(vars_p, item->b, 0.0);
            }
        if (!find(vars_p, item->result))
            {
                add_to_end(vars_p, item->result, 0.0);
            }
        item=item->next;
    }
}

hopes *create_hopes(cJSON *json_transitions, stance *all_stances,
vars_pointer * vars_p) {
    hopes *head = NULL;
    hopes *last = NULL;
    cJSON *json_transition = NULL;
    cJSON_ArrayForEach(json_transition, json_transitions) {
        hopes *h = (hopes *)calloc(1, sizeof(hopes));
        char *to_name = cJSON_GetObjectItem(json_transition, "to")-
>valuestring;
        stance *target = all_stances;
        while (target && target->id!=atoi(to_name)) {
            target = target->next;
```

```
        }
        h->target = target;

        cJSON *conditions = cJSON_GetObjectItem(json_transition,
"conditions");
        h->condition = create_formula(conditions);
        cJSON *actions = cJSON_GetObjectItem(json_transition, "actions");
        h->f = create_formula(actions);

        h->next = NULL;
        if (!head) {
            head = h;
        } else {
            last->next = h;
        }
        last = h;
        add_variables(vars_p, h->f);
    }
    return head;
}

model *parse_model(char *path) {
    FILE *file = fopen(path, "r");
    if (!file) {
        fprintf(stderr, "Error: Cannot open file %s\n", path);
        return NULL;
    }
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);
```

```
char *content = (char *)malloc(file_size + 1);
fread(content, 1, file_size, file);
fclose(file);

content[file_size] = '\0';

cJSON *json = cJSON_Parse(content);
free(content);

if (!json) {
    fprintf(stderr, "Error: Invalid JSON format\n");
    return NULL;
}
model *m = (model *)calloc(1, sizeof(model));
m->vars_p = (vars_pointer *)calloc(1, sizeof(vars_pointer));
m->stances = NULL;



cJSON *stances = cJSON_GetObjectItem(json, "stances");
cJSON *json_stance = NULL;
cJSON_ArrayForEach(json_stance, stances) {
    stance *s = (stance *)malloc(sizeof(stance));
    s->id = atoi(cJSON_GetObjectItem(json_stance, "name")->valuestring);

    cJSON *formulas = cJSON_GetObjectItem(json_stance, "formulas");
    s->f = create_formula(formulas);
    add_variables(m->vars_p, s->f);
    s->h = NULL;
    s->next = NULL;
```

```
        add_stance(m, s);
    }

    stance *last_stance = m->stances;
    cJSON_ArrayForEach(json_stance, stances) {
        cJSON *transitions = cJSON_GetObjectItem(json_stance,
"transitions");
        last_stance->h = create_hopes(transitions, m->stances, m->vars_p);

        last_stance = last_stance->next;
    }

    cJSON_Delete(json);
    return m;
}

void print_formula(formula *f) {
    if (!f || !f->first) {
        printf("    (empty formula)\n");
        return;
    }

    operation *current_op = f->first;
    while (current_op) {
        printf("    %s = %s(",
            current_op->result,
            current_op->func);
        if (current_op->is_a_const)
            printf("%f",current_op->a_const);
        else
```

```
            printf("%s", current_op->a);

        if (current_op->is_b_const)
            printf(", %f", current_op->b_const);
        else
            if(0 != strlen(current_op->b))
                printf(", %s", current_op->b);
        printf(")\n");
        current_op = current_op->next;
    }
}

void print_hopes(hopes *h) {
    if (!h) {
        printf("    hopes:\n        (no transitions)\n");
        return;
    }

    printf("    hopes:\n");
    while (h) {
        printf("        to: %d\n", h->target ? h->target->id : -1);

        printf("        conditions:\n");
        print_formula(h->condition);

        printf("        actions:\n");
        print_formula(h->f);

        h = h->next;
    }
```

```
}

void print_stance(stance *s)
{
   if (!s)
   {
      printf("stances:\n    (no stances)\n");
      return;
   }

   printf("    id = %d\n", s->id);
   printf("    formula:\n");
   print_formula(s->f);
   print_hopes(s->h);
   s = s->next;
}

void print_model(model *m)
{
   if (!m)
   {
      printf("Model is empty.\n");
      return;
   }
   stance *s = m->stances;
   int i=0;
   while (s)
   {
      i++;
      print_stance(s);
```

```
        s = s->next;
        if(i>5)break;
    }
}
```