ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ НАУКИ
«ИНСТИТУТ АВТОМАТИКИ И ЭЛЕКТРОМЕТРИИ»
СИБИРСКОГО ОТДЕЛЕНИЯ РОССИЙСКОЙ АКАДЕМИИ НАУК

На правах рукописи

УДК 004.05

Лях Татьяна Викторовна

ДИНАМИЧЕСКАЯ ВЕРИФИКАЦИЯ ПРОЦЕСС-ОРИЕНТИРОВАННЫХ ПРОГРАММ УПРАВЛЕНИЯ КИБЕРФИЗИЧЕСКИМИ СИСТЕМАМИ

Специальность: 05.13.18 – «Математическое моделирование, численные методы и комплексы программ»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:

д-р техн. наук., доцент
Зюбин Владимир Евгеньевич

Новосибирск – 2020

Оглавление

Введение

Актуальность работы и степень ее разработанности

На сегодняшний день киберфизические системы (КФС) с повышенными требованиями к надежности получили широкое распространение как в области промышленности, так и в области пользовательских устройств (ПЛК, промышленный интернет вещей, встраиваемые системы, системы управления и т.д.). Алгоритмы управления (АУ) КФС характеризуются неопределенной продолжительностью работы, взаимодействием с окружающей средой, зависимостью реакций от событий окружающей среды, необходимостью согласовывать реакцию алгоритма с динамическими характеристиками внешней среды и логическим параллелизмом (наличие независимых и слабо связанных потоков управления в алгоритме). Эта специфика приводит к появлению специализированных языковых средств и технологий программирования (языки МЭК 61131-3, MATHLAB (Simulink), G (NI LabVIEW), язык Дракон, модельно-ориентированное проектирование), в частности, развиваемых в Институте автоматики и электрометрии (ИАиЭ СО РАН) в виде процесс-ориентированного программирования (языки Reflex и IndustrialC). Процесс-ориентированное программирование базируется на концепции гиперпроцесса – множества взаимодействующих процессов с выполнимыми состояниями.

ПО современных КФС достигло такой сложности, что его верификация является отдельной областью исследования. Для верификации ПО развиваются методы динамической и так называемой формальной верификации. Применение формальных методов к верификации ПО КФС затруднено языковой гетерогенностью, большим объемом ручных операций, а также возрастанием сложности создаваемых моделей.

Наиболее распространенный подход – динамическая верификация ПО КФС на реальном объекте. Чаще всего специалист проводит ручную проверку на этапе приемо-сдаточных испытаний. Специалист задает значения на входах

управляющей программы, выступая как имитатор окружающей среды КФС, и визуально контролирует значения выходных сигналов. Этот трудоемкий подход, и он позволяет провести верификацию только простейших свойств управляющих программ.

Свойство открытости означает, что верификация свойств ПО КФС требует наличия объекта управления. С другой стороны, исследование ПО на реальном объекте управления может привести к критической ситуации (поломке оборудования, аварии и т. п.).

Современная тенденция для контроля качества управляющих программ предлагает интеграцию методов динамической верификации в итерационный процесс разработки ПО КФС: разработка, управляемая моделями (model-driven development, MDD), модельно-ориентированное проектирование (model-based design, MBD), разработка через тестирование (test-driven development, TDD). Активно развиваются подходы к верификации ПО КФС, в которых различные типы тестирования: тестирование на основе модели (model-based testing, MBT), системное, модульное, регрессионное, интеграционное, пассивное, параллельное (Back-to-Back) – сочетаются с использованием физических или программных моделей объекта управления (Е. Брингман, Р. Изерман, В. В. Кулямин). Другим перспективным подходом к динамической верификации ПО КФС является метод мониторинга поведения системы в процессе ее штатной работы (runtime verification) (Э. Барточчи). Однако он не позволяет исследовать поведение ПО при различных условиях.

На сегодняшний день слабо изучены модели и методы динамической верификации процесс-ориентированных программ управления КФС. Отсутствие контроля качества создаваемого процесс-ориентированного ПО серьезно увеличивает риски проекта при разработке киберфизических систем. Таким образом, исследование и разработка моделей и методов динамической верификации процесс-ориентированного ПО КФС, объединяющих методы тестирования, мониторинга и моделирования, является актуальной проблемой.

Объект исследования – программное обеспечение КФС.

Предмет исследования – методы динамической верификации процесс-ориентированных программ управления КФС.

Цель работы – разработка методов и моделей динамической верификации процесс-ориентированных программ управления КФС.

В соответствии с поставленной целью, в работе решаются **следующие основные задачи**:

1. Исследовать существующие подходы к динамической верификации программ управления КФС, сформулировать требования к разрабатываемым моделям, вычислительным методам и комплексам программ динамической верификации процесс-ориентированных программ управления КФС.

2. Предложить формальную модель динамической верификации процесс-ориентированных программ управления КФС.

3. Определить численный метод количественной оценки меры соответствия программ управления КФС требованиям.

4. Разработать программный комплекс для динамической верификации процесс-ориентированных программ управления КФС на основании предложенных методов и моделей.

5. Практически апробировать разработанный подход динамической верификации программ управления КФС в области промышленной автоматизации на реальных технологических объектах.

Основная гипотеза

Программы управления КФС, специфицируемые процесс-ориентированными средствами, образуют отдельный класс программ, динамическая верификация которых на программном имитаторе окружающей среды позволит повысить качество разрабатываемых систем.

Научные результаты, выносимые на защиту:

1. Формальная модель динамической верификации процесс-ориентированных программ управления КФС, которая включает четыре взаимодействующих расширенных гиперпроцесса, специфицирующих верифицируемый алгоритм

управления, виртуальный объект управления, диспетчер управления тестовыми сценариями и блок верификации (П. 1 паспорта специальности).

2. Численный метод определения вектора результатов выполнения тестовых сценариев, в котором блок «диспетчер» управляет порядком прохождения тестов, а «верификатор» вычисляет значение вектора результатов (П. 4 паспорта специальности).

3. Архитектура программного комплекса динамической верификации на базе среды LabVIEW, в которой бесшовно интегрируются модули на языке Reflex (П. 5 паспорта специальности).

4. Результаты апробации разработанных методов динамической верификации на задаче управления вакуумной подсистемой Большого солнечного телескопа (БСВТ) (П. 5 паспорта специальности).

Научная новизна. В диссертационной работе предложены, разработаны и исследованы модели и методы динамической верификации процесс-ориентированных программ управления КФС, которые использует средства процесс-ориентированного программирования и концепцию виртуальных объектов.

Принципиальный вклад в развитие технологии процесс-ориентированного программирования вносят следующие научные результаты, полученные автором:

1. Предложена формальная модель системы динамической верификации процесс-ориентированных программ управления КФС. Отличительная особенность – наличие верификатора, который анализирует поведение алгоритма управления (АУ) по срезу данных между АУ и виртуальным объектом управления (ВОУ). Модель состоит из четырех взаимодействующих расширенных гиперпроцессов, специфицирующих верифицируемый алгоритм управления, виртуальный объект управления, диспетчер управления тестовыми сценариями и блок верификации.

2. Разработан численный метод определения вектора результатов выполнения тестовых сценариев, специфицированный для процесс-ориентированных ПО. Отличительная особенность метода заключается в последовательной активации

блоков “объект управления”, “алгоритм управления” и “верификатор”. Блок “диспетчер” управляет порядком прохождения тестов через механизм обмена сообщениями, при этом блоки “алгоритм управления” и “объект управления” взаимодействуют через выделенную область данных, а блок “верификатор” на основании анализа данных вычисляет значение вектора результатов.

3. Разработана архитектура программного комплекса динамической верификации на базе среды LabVIEW, особенность которой в бесшовной интеграции модулей “объект управления”, “алгоритм управления”, “верификатор” и “диспетчер”, специфицированных на языке Reflex.

Теоретическая и практическая значимость результатов исследования

Разработанные метод и модель динамической верификации процесс-ориентированных программ управления КФС упрощают процесс разработки, верификации и сопровождения управляющего ПО на протяжении жизненного цикла системы управления КФС. Применительно к области промышленной автоматизации, получаемые преимущества дают возможность снизить сроки разработки, сократить сроки пуско-наладочных работ и повысить качество создаваемого программного обеспечения. Это имеет особое значение при автоматизации технологических процессов и научных исследований, а также при отработке и внедрении новых наукоемких технологий.

Результаты диссертации применялись в проекте по автоматизации Большого солнечного вакуумного телескопа (БСВТ, Институт солнечно-земной физики СО РАН, пос. Листвянка, Иркутская обл.). Был верифицирован алгоритм управления подсистемой вакуумирования телескопа. Также решение было апробировано на практике при разработке виртуальных лабораторных стендов, используемых для обучения студентов технических специальностей (ФИТ НГУ) разработке программ управления КФС. Разработанные механизмы бесшовной интеграции алгоритмических блоков, описанных на языке Reflex, в среду LabVIEW, были апробированы на задаче автоматизации углоизмерительной машины НОНИУС.

Документальные подтверждения эффективности полученных результатов при практическом использовании приведены в Приложении Б. к диссертации.

Методология и методы исследования

Задачи, поставленные в работе, решались с использованием процесс-ориентированного подхода к разработке программ управления КФС. Подходы к верификации программ управления КФС были выявлены на основе анализа доступных научных публикаций русских и зарубежных авторов. Эмпирическим путем была доказана эффективность разработанного подхода в задачах автоматизации технологических процессов и физико-технических исследований.

Личный вклад автора

Выносимые на защиту результаты получены при непосредственном участии соискателя. В опубликованных работах участие автора заключалось в непосредственном участии в разработке концептуального подхода к верификации программ управления КФС, самостоятельном проведении исследовательских работ и написании публикаций. Автор самостоятельно разработала архитектуру комплексов автоматизированной и автоматической верификации, реализовала ПО комплексов верификации и виртуальных лабораторных стендов и подобрала тестовые задачи для комплекса виртуальных лабораторных стендов.

Внедрение полученных результатов

Полученные результаты были использованы в работах по созданию автоматизированных цифровых комплексов:

1. Разработка и тестирование алгоритма управления вакуумной подсистемой Большого солнечного вакуумного телескопа (БСВТ).
2. Разработка и тестирование алгоритма управления углоизмерительной машины НОНИУС.

Апробация работы

Результаты, представленные в диссертации, докладывались на международных и всероссийских конференциях, в том числе:

- Индустриальные информационные системы – 2013 (г. Новосибирск, 24-28 сентября 2013 г.);

- Всероссийская научно-техническая конференция «Современные проблемы радиоэлектроники» (г. Красноярск, Россия, 6-8 мая 2014 г.);
- Девятая международная Ершовская конференция PSI-2014 (г. Санкт-Петербург, Россия, 24-27 июня 2014 г.);
- Индустриальные информационные системы – 2015 (г. Новосибирск, 20-24 сентября 2015 г.);
- Seventh Workshop Program Semantics, Specification and Verification: Theory and Applications (г. Санкт-Петербург, PSSV 2016, June 14-15, 2016 г.);
- 17th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (Эрлагол, Алтай, June 30 - July 4, 2016 г.);
- V международная конференция «Математическое и компьютерное моделирование» (г. Омск, Россия, 1 декабря 2017);
- International Conference on Electrical, Electronics, Computers, Communication, Mechanical and Computing (EECCMC) (Madras, India, 28-29 January 2018);
- International Siberian Conference on Control and Communications (SIBCON-2019) (Tomsk, Russia, 18-20 April 2019);
- International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON 2019) (г. Новосибирск, Россия, 21-22 октября 2019 г.).

Публикации

По теме работы опубликовано 29 работ, из которых 10 публикаций в трудах и материалах международных конференций и 6 статей в рецензируемых журналах из Перечня ВАК. Кроме этого, 4 работы опубликованы в рецензируемых научных журналах, индексируемых в международной базе Scopus и 2 работы опубликованы в Web of Science.

Структура работы

Диссертация состоит из введения, четырех глав, заключения и приложения. Объем работы – 133 страницы основного текста, содержит 17 рисунков и 3 таблицы. Список литературы включает в себя 106 наименований.

Глава 1. Исследование подходов к верификации программ управления киберфизическими системами

В главе обсуждается специфика алгоритмов управления КФС. Приводится сравнение программных языковых средств, использующихся при описании управляющих программ КФС. Проводится критический анализ существующих подходов к верификации программного обеспечения КФС. Приводится критический анализ математических моделей, используемых для описания КФС. Сформулированы требования к разрабатываемым моделям, вычислительным методам и комплексам программ динамической верификации процесс-ориентированных программ управления КФС.

1.1. Киберфизические системы. Специфика алгоритмов управления киберфизическими системами.

Киберфизические системы (КФС) – это системы, в которых цифровой вычислительный алгоритм активно взаимодействует с внешней средой [1]. – Цифровой компонент собирает и анализирует данные, а также выполняет управляющую функцию. Физический компонент содержит различные физические процессы. Объединение этих компонентов позволяет с помощью КФС решать те задачи, которые ни одна из ее компонент не способна разрешить поодиночке. КФС состоит из физических объектов и встроенной компьютерной системы, которая собирает и обрабатывает цифровые данные с помощью датчиков и взаимодействует с физическими процессами через управляющие элементы. Системы управления беспилотными аппаратами и автоматизированные промышленные системы управления являются классическими представителями

КФС, поскольку их взаимодействие с окружающей средой задается цифровым алгоритмом управления. Разработка программ управления КФС усложняется тем, что поведение цифровой части тесно связано с поведением физической части.

Также в КФС, помимо взаимодействия с физическими объектами, может быть включено взаимодействие с пользователем (оператором системы). В таких системах (human-in-the-loop cyber-physical systems (HiTLCPSs)) [2] оператор является важным компонентом системы. На сегодняшний день операторы КФС или: (1) осуществляют прямое управление системой, (2) являются частью физической составляющей для КФС, т.е. система отслеживает состояние оператора и принимает решения на основе полученных данных, а также (3) совмещают две предыдущие роли.

Поскольку КФС содержат как цифровую, так и физическую составляющую, для исследования поведения таких систем недостаточно исследовать каждую ее часть по отдельности. Требуется исследовать их совместную работу и взаимодействие, а также учитывать роль оператора КФС.

В настоящее КФС получают все более широкое распространение в авиастроении, различных видах промышленности, задачах обработки больших данных, робототехнике и т. д. Однако рост уровня компьютеризации, комплексная автоматизация множества областей науки и промышленности, разработка новых сложных объектов управления приводят к увеличению сложности алгоритмов управления КФС. Также возрастает стоимость ошибок в таких алгоритмах: некорректные реакции алгоритма на события на объекте может привести к нештатным ситуациям, простою оборудования и даже авариям.

Алгоритмы управления (АУ) КФС обладают рядом свойств, отличающих их от алгоритмов для программного обеспечения общего назначения [3, 4, 5, 6]:

1. **Открытость**. Алгоритм управления взаимодействует с «окружающей средой», физическим компонентом КФС. Это приводит к тому, что алгоритмы управления КФС существенно отличаются от алгоритмов управления ПО общего назначения. Самый распространенный случай окружающей среды для алгоритма управления открытой системой – это технологический объект.

2. **Событийность**. Алгоритм управления получает информацию от внешней среды (объекта управления) и формирует управляющие воздействия согласно полученным данным. В случае технологических объектов это могут быть сигналы от датчиков обратной связи и управляющие команды от оператора.

3. **Неопределенная продолжительность работы.** Заранее определить продолжительность работы невозможно: управляющее ПО функционирует до поступления специальной команды извне о выключении системы или в случае возникновения специфической ситуации на объекте управления, например, в случае физического разрушения объекта управления.

4. **Синхронизм**. Внешняя среда для алгоритма управления КФС является реальным физическим объектом. Это означает, что события на объекте не происходят мгновенно, а имеют некоторую протяжённость во времени. Объект обладает инерцией, и потому не может мгновенно реагировать на управляющие сигналы алгоритма управления. При реализации программ управления КФС необходима синхронизация реакций с событиями окружающей среды. Также важно учитывать временные задержки.

5. **Логический параллелизм.** В окружающей среде, с которой взаимодействует КФС, одновременно протекает множество процессов. Эти процессы возникают в произвольные моменты времени и не зависят друг от друга. Алгоритм управления структурно отражает параллелизм физических процессов окружающей среды. В случае несоблюдения этой структуры – например, описания реакций алгоритма управления на события в окружающей среде в виде единого блока с перебором всех возможных случаев – это приведет к низкоуровневому программированию и потере контроля над кодом.

1.2. Языковые средства разработки программ управления киберфизическими системами

На сегодняшний день для описания программ управления КФС используются различные языковые средства:

1. Языки общего назначения

Императивные языки, такие как язык Си и его диалекты, Паскаль и т. д. широко используются для разработки программ управления киберфизическими системами [7, 8, 9]. Язык Си является самым распространенным языком программирования для описания управляющего ПО киберфизических систем [10, 11]. Несмотря на отсутствие в этих языках языковых средств, учитывающих специфику КФС, эти языки программирования часто используются при разработке алгоритмов управления КФС. Это вызвано преемственностью кода: многие используемые библиотеки переиспользуются из проекта в проект, поскольку в них уже исправлены наиболее частые ошибки, и их надежность доказана многолетним процессом отладки и корректным функционированием заимствованного кода во множестве проектов.

Также для разработки программ управления КФС используются **объектно-ориентированные языки.** В эту группу входят широко распространённые объектно-ориентированные языки (Си++ [12, 13, 14], Java [15], Python [16] и т.д.). На сегодняшний день объектно-ориентированный подход преобладает в разработке ПО общего назначения, а также широко используется для разработки специализированных программных систем (в том числе и программ управления КФС) [10]. Однако если при разработке ПО общего назначения объектно-ориентированные языки привлекают своей гибкостью и широким спектром технологий, при использовании их для задач управления КФС, разработчики сталкиваются с рядом трудностей. Эти трудности вызваны с упомянутой ранее спецификой алгоритмов управления КФС. Для сложных технологических систем

возрастает трудоемкость использования объектно-ориентированных языковых средств при разработке программ управления КФС. Управляющие программы КФС используют операции с временными интервалами, а поступающие внешние события приводят к изменению поведения таких программ. Поступающие внешние сигналы от датчиков обрабатываются параллельно. Все эти особенности приводят к тому, что при попытке описать алгоритм в объектно-ориентированном стиле, происходит экспоненциальный рост сложности логических связей в коде. Уменьшается надежность результирующего кода, такой код становится невозможно модифицировать и сопровождать. Эти проблемы не устраняются и при использовании ООП-языков, специализированных для разработки КФС, таких, как язык Eiffel [17]. Язык Eiffel со встроенными средствами контрактного программирования используется в разработках систем управления КФС, однако не является широко распространённым.

Функциональные языки программирования (Лисп [18], Эрланг[19], Хаскель [20] и т. д.) также используются для программирования киберфизических систем [21], поскольку предоставляют широкие возможности для описания параллелизма происходящих процессов. Однако к недостаткам функциональных языков программирования относится необходимость постоянного автоматического выделения и освобождения памяти (наличия сборщика мусора). Также непредсказуемый порядок вызова функций может привести к проблемам при вводе-выводе данных на периферийные устройства.

2. Предметно-ориентированные языки программирования.

Предметно-ориентированные языки программирования (domain specific language – DSL), в противовес языкам общего назначения, разрабатываются для использования в конкретной области применения [22]. Синтаксис и семантика таких языков учитывает особенности области, в которой они применяются.

Широко распространены графические языки программирования (G LabVIEW [23], язык Дракон [24], MATLAB (Simulink) [25]). Графические языки обличаются обилием выразительных средств и возможность подбора «графических метафор». Однако по сравнению с тестовыми языками программирования,

графические языки справляются хуже при увеличении архитектуры программ. Также возникает проблема поиска подходящих «метафор». Наглядность разрабатываемого кода ухудшается с увеличением сложности программы.

В отдельную группу стоит выделить предметно-ориентированные языки программирования стандарта МЭК-61131-3 [26]. Стандарт содержит пять языков программирования (два текстовых и три графических). Он применяется в разработке разработки автоматизированных систем управления при программировании логических контроллеров. Однако языки МЭК 61131-3 обладают рядом недостатков. Низкая выразительность этих языков приводит к росту трудоемкости программирования при усложнении программ [27, 28, 29]. Также из-за низкой выразительности разработчики зачастую вынуждены или реализовывать часть управляющей программы на языках общего назначения (например, на Си), или же пользоваться расширениями этих языков. Расширения языков МЭК зачастую предлагают компании, производящие ПЛК [30, 31, 32, 33, 34]. Это приводит сложностям интеграции кода, реализованного на языках МЭК, в сторонние управляющие системы. Зачастую разработчик управляющей программы, начав разработку на оборудовании конкретного производителя, впоследствии не может уже от него отказаться.

В Институте автоматики и электрометрии СО РАН была создана концепция процесс-ориентированных языков программирования для разработки управляющих программ КФС. Процесс-ориентированные языки основываются на модели конечного автомата. Их эффективность была доказана практически рядом проектов. Язык Reflex является одним из активно развивающихся на данный момент процесс-ориентированных языков [35, 36, 37, 38].

В основе процесс-ориентированных языков (в том числе и языка Reflex) лежит математическая модель гиперпроцесса - множества взаимодействующих процессов с выполнимыми состояниями. Гиперпроцесс – это расширение модели конечного автомата. Использование модели гиперпроцесса позволяет отразить перечисленные ранее особенности алгоритмов управления КФС: открытость, синхронизм, цикличность, событийность и логический параллелизм [39]. На языке

Reflex разрабатываются управляющие программы для робототехники и промышленной автоматизации. Преимущества языка Reflex:

– Си-подобный синтаксис. Это позволит облегчить его изучение тем разработчикам, которые уже имели опыт с Си-подобными языками.

– Русскоязычный и англоязычный синтаксис.

– Программа состоит из набора равнозначных процессов.

– Процессы в языке Reflex исполняются параллельно.

– В целях инкапсуляции, внешние данные о состоянии процесса, которые могут запросить другие процессы, ограничены. Процесс может узнать о другом процессе, является ли он активен сейчас, пассивен или же завершил свою работу с ошибкой.

– В языке Reflex присутствуют служба времени и операции с временными интервалами.

– В Reflex включены языковые средства, которые позволяют отображать показания датчиков и значения управляющих сигналов на внутренние переменные программы.

Язык Reflex используется в проектах по разработке управляющих программ промышленными КФС. Было доказано удобство его использования, адекватность языка специфике задач разработки управляющих программ КФС и простота сопровождения уже созданного ПО.

1.3. Проверка корректности программ управления КФС

При использовании процесс-ориентированных языков в при разработке управляющих программ КФС [40, 41] отдельную задачу представляет проверка качества разрабатываемого ПО.

Программное обеспечение КФС должно удовлетворять требованиям контроля качества [42]:

• Надежность (Reliability). Надежность программного обеспечения – это предсказуемая и последовательная работа системы, выполнение требуемых функций в штатных условиях в течение неопределенного периода времени. Это требование важно для безопасности функционирования системы, поскольку оно уменьшает вероятность возникновения ошибок, которые могут привести к сбоям на объекте управления;

• Устойчивость (Robustness). Устойчивость системы – это способность системы работать при ненормальных или критических ситуациях. Это свойство также отвечает за безопасность системы, поскольку управляющей программе приходится сталкиваться с исключительными ситуациями на объекте управления, справляться с ними, восстанавливаться после внутренних сбоев и предотвращать распространение ошибок, возникающих в нештатных ситуациях;

• Верифицируемость (Traceability). – Возможность доказать, что результирующая программа удовлетворяет требованиям, заявленным в спецификации;

• Поддерживаемость (Maintainability) – возможность безопасно вносить изменения в уже запущенную систему – при этом внесенные изменения не должны приводить к возникновению новых ошибок. Также является важным аспектом безопасности.

Для контроля качества ПО используются технологии верификации. Верификация – это процесс, позволяющий объективно доказать, что разрабатываемое ПО соответствует требованиям, накладываемым на него на текущем этапе разработки [43].

1.4. Технологии верификации управляющих программ КФС

Исследуем применимость методов, использующиеся для верификации программного обеспечения КФС, при разработке процесс-ориентированного ПО.

При верификации ПО необходимо учитывать специфику верифицируемых программ. В случае с КФС необходимо учитывать открытость, событийность, параллелизм, синхронизм и неопределенное время функционирования. Свойство открытости означает, что только простейшие свойства управляющего ПО можно протестировать автономно. Поэтому программы управления КФС чаще всего тестируются вручную, непосредственно на объекте управления, во время пусконаладочных испытаний [10]. Специалист имитирует объект управления, задавая показания датчиков, и проверяет управляющие сигналы программы.

Однако запуск неотлаженной программы управления КФС на реальной системе грозит поломкой оборудования или даже серьезной аварией. Такой подход увеличивает трудоемкость и время приемо-сдаточных испытаний. Он позволяет провести контроль трассировки проводов, однако при верификации таким способом проверяются только простейшие требования, накладываемые на управляющую программу. Это сильно увеличивает риски при разработке управляющих программ КФС. Поэтому на сегодняшний день активно разрабатываются методики автоматической верификации программ управления КФС [44, 45, 46, 47, 48].

Для исследования свойств управляющего ПО КФС используются методы динамической верификации и методы формальной верификации [49, 50]. При динамической верификации для анализа поведения исследуемого ПО используются результаты реальной работы исследуемой системы. Формальные методы предполагают анализ исходного кода и документации (требования к ПО, спецификации интерфейсов и модели), используют формальные модели исследуемых программ и требований, но без исполнения программы.

Для верификации программ управления КФС используются методы:

1. *Статический анализ кода.*

2. *Методы формальной верификации:*
   2.1. Проверка на моделях.
   2.2. Дедуктивная верификация.
3. *Методы динамической верификации:*
   3.1. Тестирование: модульное тестирование (unit testing), тестирование на основе модели (model-based testing), различные виды тестирования, основанного на симуляции (X-in-the-loop), системное, интеграционное и др.
   3.2. Мониторинг поведения верифицируемой системы в процессе исполнения (Runtime Verification).

1.4.1. Статический анализ кода

При статическом анализе в исходном коде ПО ищутся участки, способные привести к ошибкам [51]. Методы статического анализа могут варьироваться от самых обыденных (например, подсчет статистики по плотности комментариев) до более сложных, основанных на анализе семантики. Как правило, код исследуется на наличие часто встречающихся ошибок по заданным шаблонам – к примеру, некорректная работа с ресурсами (память, потоки), некорректные арифметические операции, недосягаемый код, незавершенный код (использование неинициализированных переменных и функций). Разработчики методов статического анализа стремятся к надежности и полноте оценки качества кода. Ключевой задачей является оценка «степени опасности» исследуемого участка кода. Некорректное задание «критериев опасности» может привести возникновению множества ложных сигналов об ошибках в коде – или же, напротив, допустить пропуск реальных ошибок.

При статическом тестировании производится логический анализ программного кода. Для статического анализа существует множество программных средств, как широко распространённых – компилятор – так и узко специализированных – к примеру, PSV-Studio [52], Clang-Tidy [53] и Coverity [54]. Статический анализ может выполняться людьми. Также имеется возможность полностью автоматизировать статический анализ [55].

Однако статический анализ не позволяет анализировать логику работы проверяемой программы. Таким образом, при разработке алгоритмов управления КФС метолами статического анализа невозможно проверить корректность функционирования верифицируемой программы.

1.4.2. Методы формальной верификации

Методы формальной верификации программ, в отличие от большинства динамических подходов к верификации, призваны доказывать не наличие, а отсутствие ошибок в программе [56, 57]. При этом тестируется не конкретная часть программы, а гарантируется, что никакие события в системе, внешние или внутренние, приведут к некорректному функционированию. Наиболее распространёнными подходами в формальной верификации являются: проверка моделей и дедуктивная верификация.

1. Проверка моделей – Model checking [58, 59]. Этот метод применяется для автоматической верификации систем с конечным числом состояний. Основная идея подхода проверки моделей состоит в том, чтобы определить, достигается ли корректность системы путем исчерпывающего изучения достижимых состояний системы. Пространство состояний ПО, как правило, слишком велико, чтобы его можно было анализировать напрямую – поэтому используются механизмы абстракции, чтобы снизить количество состояний. Система представляется в виде

абстрактной модели (чаще всего используется модель Крипке [60]). Требования к системе также описываются в формальном виде (как правило, применяется темпоральная логика [61, 62]). Если проверяемое требование корректности не удовлетворяется, алгоритм проверки на моделях создает контрпример – ситуацию, при которой проверяемое свойство не выполняется. Метод Model Checking может быть полностью автоматизирован.

Основной недостаток такого подхода к верификации КФС связан с возможным комбинаторным взрывом в пространстве состояний системы. Если верифицируемая система состоит из большого количества компонентов, взаимодействующих друг с другом, количество возможных состояний системы лавинообразно\экспоненциально увеличивается. Для уменьшения пространства состояний используется подход ограниченной проверки моделей (Bounded Model Checking [63]). При таком подходе не анализируется все пространство состояний: поведение программы исследуется только до определенной глубины. Однако такой подход приводит к тому, что часть ошибок остаются не выявленными.

Наиболее известные верификаторы методом Model Checking (Spin [64], UPAAL [65] и т.д. [66]) в большинстве своем требуют для верификации модели проверяемой системы, описанной на специализированном для конкретного верификатора языке. Это приводит к увеличению трудоемкости процесса разработки и накладным расходам. Кроме того, общепризнанной проблемой верификаторов методом Model Checking является ограниченность сложности верифицируемых ими систем [67]. Для верификации свойств управляющего ПО КФС подход model checking зачастую используется совместно с динамическими подходами к верификации [10].

2. Дедуктивная верификация программ [68]. При таком подходе система описывается в виде набора аксиом. Доказательство выполнения требований проводится дедуктивно на основе этих аксиом и правил вывода. Процедура не может быть полностью автоматизирована и требует постоянного вмешательства человека.

Вычисления, используемые при использовании подхода автоматического доказательства теорем, базируются на подходе, разработанном Э. Хоаром. Подход Хоара к доказательству корректности программ включает концепцию «троек Хоара» [69], которые состоят из предусловия, пост-условия и функции (программы) перехода между этими двумя состояниями. Таким подходом можно верифицировать или части программы, или программу целиком.

Различные верификаторы, использующие методы дедуктивной верификации, как правило, анализируют не исполняемый код программы, а описание ее на специализированном DSL (обычно это диалекты языка LISP). Ключевое отличие между методами дедуктивной верификации и методом проверки моделей: при дедуктивной верификации нет необходимости исчерпывающе исследовать все состояния системы, чтобы проверить ее свойства. Следовательно, подход дедуктивной верификации может исследовать системы с бесконечным числом состояний, включающие в себя сложные типы данных и рекурсию. Это достигается с помощью того, что при дедуктивной верификации выводы делаются на основании ограничений состояний, а не самих экземпляров состояний. Верификаторы этого типа ищут доказательства методами синтаксического анализа.

Дедуктивная верификация не может быть полностью автоматизирована. Набор аксиом и правил вывода при верификации реальных программных систем может оказаться чрезвычайно велик. Кроме того, полученные доказательства могут быть объемны и трудны для понимания. Также, для использования этого подхода, требуется серьезные усилия разработчика с большим опытом работы. Эти недостатки препятствуют широкому внедрению подхода в процесс разработки реальных программных систем [70, 71]. Для дедуктивной верификации программ управления КФС используются различные автоматические верификаторы (Z3 [68, 72], KeY [73], KeYmaera [74]), однако эти подходы на данный момент не получили широкого распространения.

Также на сегодняшний день активно развивается область автоматического синтеза программ [75]. Этот метод предлагает автоматическую генерацию кода системы из спецификации, описанной на формальном языке: по формуле

темпоральной логики, набору входных и выходных сигналов автоматически генерируется дискретно-событийная программная система, которая удовлетворяет описанной формуле темпоральной логики. Один из недостатков подхода автоматического синтеза систем – на практике не уделяется никакого внимания качеству сгенерированного кода. Действительно, то, что синтезированная система верна и удовлетворяет заданному условию, не говорит о ее качестве [76].

На данный момент этот подход активно развивается и поддерживается многими исследователями, предлагаются различные варианты оценки качества сгенерированного кода. Однако полная автоматизация этого метода невозможна.

Таким образом, распространенные подходы к формальной верификации при применении их к задачам разработки алгоритмов управления КФС обладают рядом недостатков:

1. Проверяется только алгоритм управления, но не проверяется его совместное взаимодействие с реальным объектом управления. Введение объекта управления в схему верификации приводит к экспоненциальному увеличению количества состояний системы и, следовательно, усложнению процесса верификации.

2. В отличии от динамических подходов к верификации, где анализируется непосредственно результирующий код, при верификации проверяется не результирующий код, а некая формальная модель алгоритма, которая может не учитывать особенности реального кода.

3. В случае разработки алгоритмов управления КФС верификацию невозможно сделать полностью автоматической: это приводит к чрезмерному росту проверяемых состояний системы и чрезвычайно увеличивает трудоёмкость описания правил вывода.

1.4.3. Методы динамической верификации

При динамической верификации выводы о корректности поведения верифицируемой программы делаются по результатам реального запуска программы. В отличие от обычных программных систем (для пользовательских ПК, веб-приложений или мобильных приложений), у программ управления КФС большинство интерфейсов является интерфейсами взаимодействия с аппаратной частью КФС. У таких систем может полностью отсутствовать графический пользовательский интерфейс (graphical user interface – GUI) оператора. Отдельной трудностью при проверке программ управления КФС является локализация ошибок. Как следствие, динамическая верификация управляющих программ является нетривиальной задачей: прямое применение уже существующих подходов динамической верификации затруднено и требует специальной адаптации.

1. Тестирование

Тестирование является наиболее распространённым способом проверки корректности программ управления КФС. Например, в промышленной автоматизации тестирование используют ведущие игроки в области SCADA-систем [77]. Методы формальной верификации, такие как: статический анализ кода, проверка моделей, дедуктивная верификация – имеют большой потенциал, однако ни одна из этих технологий не проработана достаточно, чтобы занять доминирующую позицию и заменить тестирование. Тестирование ПО является эмпирическим исследованием. При этом чаще всего проводится системное тестирование на реальном объекте управления, когда проверяется функционирование всей системы в целом [10]. Также проводится интеграционное тестирование, модульное тестирование (unit-testing), регрессионное [78], пассивное [79] и параллельное тестирование (Back-to-Back) [80]. При этом основные усилия исследователей новых подходов к динамической верификации ПО КФС прилагаются к обеспечению выполнения тестов и автоматизации тестового процесса. Тестирование обычно начинается с разработки тестового случая (формулируются последовательность определенных действий над системой и ожидаемый результат или на основании критериев тестирования, или на основании знаний специалиста). Затем запускается тест и исследуемая система. На

заключительной фазе оцениваются результаты тестирования и делается вывод о качестве исследуемой системы. Автоматизация тестирования является одним из самых популярных подходов для снижения затрат при разработке программ управления КФС.

Тестирование не доказывает полное отсутствие ошибок в программе, поскольку проверяет лишь конкретные случаи функционирования системы. В случае программ управления КФС применением методов тестирования к программам управления КФС обладает следующими недостатками:

1. Как правило, тестирование выполняется на поздних этапах разработки, когда исполняемый код уже реализован. В случае алгоритмов управления КФС свойство открытости этих систем, требующее учитывать наличие внешней среды, приводит к водопадной модели разработки.

2. При тестировании не учитываются особенности алгоритмов управления КФС, не учитывается неопределенность времени функционирования системы и наличие окружающей среды, что приводит к потенциальной бесконечности множества проверяемых ситуаций. Тестированием можно проверить лишь строго заданный набор сценариев работы системы.

3. Методы, разработанные для тестирования программного обеспечения в области ПО общего назначения предполагают, что код тестируется автономно – однако в случае программ управления КФС существует еще динамически изменяющаяся внешняя среда, от которой проверяемая программа получает значения сигналов датчиков.

Виды тестирования, применяемые для проверки программ управления КФС:

А) Модульное тестирование (Unit-testing)

Идея модульного тестирования в том, что для наиболее важных программных компонентов создается набор тестов [81]. Задаются входные и выходные данные, запускается тестируемый код с заданными входными параметрами – и проверяется, что выходные данные совпадают с ожидаемыми (заданными в тесте).

Б) Интеграционное тестирование [82] – тестирование совместного функционирования нескольких компонент системы.

В) Системное тестирование [10, 83]. Этот тип тестирования выполняется на полноценно запущенной системе. Чаще всего выполняется на реальном объекте управления.

Г) Регрессионное тестирование [78] – повторная проверка тестовых ситуаций после внесения исправлений в код.

Д) Пассивное тестирование [79] – во время работы системы пассивно формируется кэш сигналов проверяемой системы, а затем по прошествии времени, отведенного на тестирования, эти данные анализируются.

Е) Параллельное (сравнительное) тестирование (Back-to-Back) [80] – создаваемая система тестируется параллельно с системой другой версии, уже отлаженной и доказавшей свою корректность.

Ж) Тестирование на основе модели (Model-based testing)

Тестирование на основе модели [84] – это метод тестирования ПО, чаще всего использующийся при модельно-ориентированном проектировании (model-based design – MBD) [85]. Этот подход на данный момент активно развивается [10]. Модели могут использоваться как для представления желаемого поведения тестируемой системы (ТС), так и для представления стратегий тестирования и среды тестирования. Для описания моделей часто используются модели на основе конечных автоматов или их расширений.

Модель, описывающая ТС, обычно является частичной абстракцией желаемого поведения ТС. Из этой модели извлекается описание тестовых случаев, в виде функциональных тестов на том же уровне абстракции, что и модель. Этот набор тестовых случает называется набором абстрактных тестов. Набор абстрактных тестов не является исполняемым на исследуемой системе, поскольку тесты эти описывают другой уровень абстракции. Исполняемые тесты генерируются из абстрактных тестов. Исполняемые тесты взаимодействуют непосредственно с исследуемой системой.

Автоматической генерация тестов позволяет достигнуть такой же степени покрытия кода, что и созданные вручную наборы тестовых ситуаций. Однако было обнаружено, что использование автоматической генерации тестов не всегда

приводит к лучшему обнаружению ошибок управляющей программы по сравнению с ручным тестированием – например, эмпирическое исследование показало, что наборы созданных вручную тестов более эффективно обнаруживают логические ошибки.

К недостаткам подхода MBT относят «проблему отображения» абстрактных тестовых ситуаций в исполняемые тесты [86, 87]. Поскольку тестирование – это эксперимент, который проводится над исследуемой системой, не существует единого наилучшего подхода к генерации исполняемых тестов из абстрактных тестов. Обычно для генерации исполняемых текстов вводится дополнительный набор параметров, которые называются «требованиями к тестам». В этом пакете содержатся настроечные параметры и критерии успешности и неуспешности результатов тестирования.

Кроме того, разработчики порой сталкиваются с ограничениями моделирования (используемый инструмент не масштабируется до сложных моделей). Осложняется это тем, что внедрение и использование MBT в разработке само по себе является нетривиальной задачей. Оно требует дополнительного обучения для разработчиков и тестировщиков, а также вносит языковую гетерогенность в процесс разработки – для описания моделей необходимо использовать специализированные языки (используются языки, основанные на модели конечного автомата, средства MATLAB Simulink или диаграммы последовательностей UML). Не всегда можно найти подходящую абстракцию, чтобы исследовать специфические свойства тестируемой системы.

Кроме того, несмотря на существенное облегчение возможностей для генерации проверяемых тестовых ситуаций, MBT при разработке алгоритмов управления КФС не позволяет избежать недостатков модульного тестирования, связанных с наличием окружающей среды и неограниченного времени функционирования.

Наибольшую популярность для разработки программ управления КФС MBT получило в сочетании с подходами, сочетающими в себе тестирование и симуляцию (X-in-the-loop) [10, 88, 89, 90]:

А) Model-in-the-loop (MIL) – разрабатываемая система описывается в виде модели, задавая логику функционирования разрабатываемой программы. Также в виде модели описывается и объект управления.

Б) Software-in-the-loop (SIL) – этот шаг следует после MIL. Из модели управляющей программы генерируется исполняемый код. Затем полученная программа запускается на модели объекта управления и исследуется ее поведение.

С) Processor-in-the-loop (PIL) – полученный в предыдущем шаге код загружается на целевой процессор и также исследуется на модели объекта управления.

Д) Hardware-in-the-loop (HIL) – модель объекта управления загружается на тестовый стенд и контроллер подключается к этому имитатору.

Тестирование на основе моделей на данный момент развивается как научными исследователями [91], так продвигаются активно разработчиками крупных программных систем, используемыми для разработки управляющих программ КФС (MATLAB Simulink, Simintech [92], NI VeriStand [93] и т. д.). Однако ситуация на рынке такова, что использование предлагаемых программных решений вынуждает разработчиков пользоваться продуктами только одного производителя, что уменьшает гибкость системы и затрудняет интеграцию результирующего продукта в сторонние системы.

2. Мониторинг поведения системы во время ее работы (Runtime Verification)

Runtime Verification (RV) – это метод динамической проверки качества ПО, основанный на проверке запущенной системы на соответствие накладываемым на нее формальным требованиям [94]. На вход RV подаются: (1) система, которую требуется проверить (2) набор свойств, которые требуется проверить во время запуска системы. Свойства могут быть описаны как на языках формальных спецификаций, или в виде программ на языках общего назначения. Обычно процесс RV состоит из трех шагов:

1) Из проверяемого свойства генерируется т.н. «монитор». Монитором называется сущность, позволяющая отслеживать то, что запущенная система удовлетворяет накладываемому на нее требованию. Этот шаг часто называют

синтезом монитора. Монитор обрабатывает события, происходящие в системе, и формирует на основе этих данных решение о том, выполняется ли проверяемое им свойство;

2) «Оборудование» системы (system instrumentation). К системе разрабатывают интерфейсы, которые предоставляют системе возможность генерировать передачу монитору данных для анализа состояния;

3) Анализ времени исполнения (execution analysis) – монитор анализирует поведение системы во время ее исполнения.

Недостатком метода верификации путем мониторинга формальных свойств проверки являются накладные расходы, которые возникают при дополнительных проверках во время работы системы. Также дополнительные накладные расходы возникают на этапе «оборудования» системы, когда к самой верифицируемой системе приходится добавлять интерфейсы, необходимые для взаимодействия монитора с системой. Чем больше усложняются проверяемые требования, тем больше накладных расходов возникает при мониторинге. При использовании методов RV требуется баланс между выразительностью требований и эффективностью верификации.

1.4.3.1. Моделирование окружающей среды при динамической верификации программ управления КФС

Поскольку динамическая верификация свойств управляющих программ КФС сопряжена с рядом трудностей, активно развиваются подходы, в которых вместо объекта управления используется его модель [77]:

1) Физическое моделирование. Для проверки алгоритма управления КФС строится физический имитатор - реальный объект со свойствами, схожий со свойствами реального объекта управления. При таком подходе алгоритм

взаимодействует не с реальной окружающей средой, а с ее физическим имитатором. К недостаткам такого подхода причисляют: высокую стоимость физических имитаторов, сложность подбора и конструирования имитатора, проблемы безопасности при исследовании критических режимов и трудоемкость модификации имитатора.

2) Программные имитаторы — это отдельный программно-аппаратный комплекс, который содержит программный модуль, имитирующий поведение окружающей среды. формирует аналоговые и цифровые входные сигналы алгоритма управления и считывает значения управляющих аналоговых и цифровых сигналов, поступающих от отлаживаемого алгоритма управления. На основании полученных команд, программный имитатор изменяет свое состояние и генерирует выходные сигналы. Использование программные имитаторов позволяет устранить основные недостатки физических имитаторов, обеспечивая безопасность тестирования в граничных условиях и позволяет модифицировать поведение моделируемого объекта.

На сегодняшний день ситуация такова, что в большинстве случаев программные имитаторы создаются средствами языков общего назначения. При этом использование программных имитаторов позволяет реализовывать их код на том же языке программирования, что и тестируемый алгоритм. Это связано с гибкостью и распространенностью этих языков. Однако это серьезно увеличивает стоимость разработки: усилия, затрачиваемые на разработку поведения алгоритма управления, сравнимы с усилиями, затрачиваемыми на разработку самого тестируемого алгоритма.

Существуют специализированные программные пакеты, призванные облегчить процесс создания программных имитаторов. Например, в области промешенной автоматизации, ведущие игроки на рынке SCADA-систем предлагают дополнительные пакеты для моделирования.

Нетривиальной задачей остается выбор формальной модели для программной имитации окружающей среды КФС (объекта управления) при динамической верификации.

Для верификации алгоритмов управления КФС исследователь задает верифицируемые требования. Необходимость использования имитатора в процессе верификации алгоритмов управления КФС обусловливается свойством открытости таких алгоритмов - алгоритмы управления КФС невозможно верифицировать автономно. Таким образом, при динамической верификации алгоритмов управления КФС необходимо описывать как имитатор объекта управления, так и проверяемые требования. Кроме того, одним из наиболее часто упоминаемых недостатков у большинства подходов к верификации – отсутствие унификации при описании моделей системы для верификации, требований и самого кода. Унифицированное описание требований, имитатора и верифицируемой системы позволила бы упростить процесс разработки, не усложняя его введением новых языков описанием математических моделей и формализмов.

Поведение объекта управления КФС характеризуется теми же свойствами, что и алгоритм, управляющий этим объектом (открытость, событийность, параллелизм, синхронизм и зависимость от времени функционирования). Таким образом от модели, использующейся для описания имитаторов физической части КФС, требуется возможность описывать системы с теми же свойствами, что и алгоритмы управления.

Для динамической же верификации применимы только те модели, которые возможно запустить на исполнение – это модели, основанные на моделих конечного автомата [95].

Эти модели являются расширением и обобщением концепции конечного автомата: конечный автомат, расширенные конечные автоматы, взаимодействующие конечные автоматы и т.д.). Памятуя об особенностях алгоритмов управления КФС, рассмотрим те модели, которые позволяют моделировать событийность и параллелизм.

1. Иерархические автоматы [96] позволяют организовать вложенность состояний конечного автомата. В иерархическом автомате определяется переходы из групп состояний. Такой подход уменьшает количество эквивалентных переходов между множествами состояниями. Также в иерархических автоматах

присутствует возможность задавать параллельность исполнения нескольких групп состояний, т.е. после перехода в высокоуровневое состояние, содержащее в себе параллельные группы состояний, автомат оказывается сразу в нескольких состояниях. Использование иерархических автоматов предполагает наличие внутренней иерархии моделируемых процессов, т.е. объединение всех моделируемых процессов в подмножества, среди которых возможны переходы только внутри друг друга или в другую группу состояний. В случае отсутствия четко выраженной иерархии между процессами возможности параллельного функционирования множества процессов иерархические автоматы теряют свои преимущества и вырождается в множество параллельно запущенных конечных автоматов.

2. Временные конечные автоматы (ВКА) [97]. Временной автомат содержит дополнительный набор таймеров, значения которых используются для задания условий перехода. Однако в модели ВКА отсутствует параллелизм.

3. Сети Петри [98]. В сетях Петри существует множество состояний и множество переходов, и каждый переход может связывать два произвольных множества состояний. Однако моделирование логического параллелизма с помощью сетей Петри чревато ошибками [99].

4. Гиперпроцессы [6]. Гиперпроцесс состоит из набора упорядоченных процессов. Есть выделенный начальный процесс. Все процессы активируются с заданным периодом. В формальном виде, процесс – это конечный автомат, расширенный таймером, отмечающим продолжительность нахождения процесса в текущем состоянии. Модель гиперпроцесса разрабатывалась для того, чтобы формально описывать программы управления КФС. Она отражает специфику алгоритмов управления КФС (открытость, событийность, цикличность, синхронизм и логический параллелизм). Таким образом, использование модели гиперпроцесса для имитации поведения физической составляющей КФС позволит:

1) унифицировать разработку алгоритма управления и имитатора объекта управления;

2) отразить свойства событийности, открытости, синхронизма, параллелизм аи неопределенного времени функционирования объекта управления КФС.

1.5. Современные тенденции при разработке программ управления КФС

Несмотря на то, что ручная верификация управляющих программ на объект управления по-прежнему занимает лидирующие позиции, современные подходы к разработке программ управления КФС предполагают интеграцию методов верификации в итерационный процесс разработки управляющего ПО. Исследователи предлагают вести разработку программ управления КФС через разработку через тестирование – test-driven development (TDD) [100], а также сочетать разработку, управляемую моделями – model-driven development (MDD) [90, 101] с модельно-ориентированным проектированием – model-based design (MBD) [10].

КФС включает себя две составляющие - цифровую и физическую, и при верификации алгоритмов управления КФС при верификации требуется учитывать взаимодействие алгоритма и объекта управления. Также важным элементом, влияющим на поведение КФС, является оператор системы. В Институте автоматики и электрометрии была разработана общая схема верификации управляющих программ КФС [77] (рис. 1), учитывающая эти особенности. Схема состоит из следующих шагов:

1) реализация управляющей программы КФС (или ее части), отражающей алгоритм управления (АУ);

2) программная реализация имитатора физической внешней среды КФС (ее части) – виртуального объекта управления (ВОУ);

3) верификация: проводятся тестовые сценарии и оцениваются реакции верифицируемой программы управления;

4) по результатам верификации проводится коррекция верифицируемой программы или кода ВОУ.

Рис. 1. Общая схема динамической верификации процесс-ориентированных программ управления КФС:

АУ – алгоритм управления КФС, ВОУ – виртуальный объект управления.

Эта схема используется для итерационной разработки программ управления КФС.

Выводы главы

Алгоритмы управления КФС обладают рядом специфических свойств:

1) открытость – алгоритм управления взаимодействует с окружающей средой;

2) событийность – алгоритм реагирует на события окружающей среды и воздействует на происходящие в окружении физические процессы;

3) продолжительность функционирования управляющего алгоритма не определена;

4) синхронизм – реакции управляющего алгоритма должны синхронизоваться с событиями окружающей среды;

5) логический параллелизм – структура управляющего алгоритма отражает параллелизм процессов, происходящих в окружающей среде.

Использование языков общего назначения при разработке программ управления КФС приводит чрезмерной запутанности программной архитектуры. Поэтому для разработки управляющих программ КФС используются предметно-ориентированные языки (МЭК 61131-3, MATLAB (Simulink), G (NI LabVIEW), Reflex и др.). В Институте автоматики и электрометрии была разработана концепция процесс-ориентированных языков программирования, основанных на математической модели гиперпроцесса (Reflex, Industrial C). Процесс-ориентированный подход был использован в ряде проектов по автоматизации и доказал свою практическую эффективность.

При использовании процесс-ориентированных языков при разработке управляющих программ КФС отдельную задачу представляет доказательство корректности разрабатываемого ПО. Верификация является одним из основных методов обеспечения корректности ПО. При разработке процесс-ориентированного ПО, методы, использующиеся для верификации программного обеспечения общего назначения, слабо применимы. Поэтому отладка программ управления КФС водится к тому, что оператор тестирует и отлаживает программу во время пусконаладочных испытаний. Этот подход усложняет разработку управляющих программ КФС и не гарантирует качество проведенной верификации, поскольку требует большое количество ручных операций.

Поэтому разработка моделей и методов верификации процесс-ориентированных программ управления КФС является важной теоритической и

практической задачей. Формальные методы верификации в данный момент слабо применимы для проверки программ управления КФС, поскольку накладывают серьёзные ограничения на сложность проверяемых программ.

Современная тенденция заключается в интеграции методов динамической верификации управляющих программ КФС в итерационный процесс разработки ПО с использованием программных имитаторов окружающей среды. Одним из ключевых преимуществ при использовании программных имитаторов является возможность регулировать физические параметры окружающей среды, а также имитировать ситуации, которые было бы невозможно или опасно проверять на реальном объекте (к примеру, аварии). Синхронизация программ управления КФС с физическими событиями окружающей среды может привести к увеличению сроков разработки при верификации на реальном объекте. При динамической верификации использование программного имитатора позволяет минимизировать эти задержки.

Программы управления КФС реагируют как на события физической составляющей КФС, так и на воздействия от оператора. Поэтому при динамической верификации свойств ПО КФС требуется моделировать:

1) поведение физической составляющей КФС;
2) поведение алгоритма управления КФС;
3) поведение оператора КФС;
4) планирование сценариев работы на объекте управления;
5) мониторинг реакций алгоритма управления в контексте текущего сценария.

Исследования показывают, что автоматизация динамической верификации управляющего ПО КФС приводит к снижению затрат на проверку корректности ПО. Исследователи также отмечают, что языковая гетерогенность увеличивает нагрузку на разработчиков и затрудняет внедрение новых методов верификации.

На основании проведенного исследования были сформулированы требования к разрабатываемому подходу верификации управляющих программ КФС:

1. Верификация программ управления КФС должна происходить динамически на программном имитаторе объекта управления.

2. Процесс верификации должен быть интегрирован в итерационную разработку ПО КФС.

3. Проверка корректности разрабатываемый программы управления КФС должна происходить автоматически.

4. Модель должна включать проверку корректности реакций алгоритма управления (АУ), имитацию поведения объекта управления и управление прохождением тестовых сценариев.

5. Метод должен обеспечивать возможность изменений масштаба времени;

6. При моделировании тестовых ситуаций должна предоставляться возможность изменения параметров внешней среды КФС и взаимодействия АУ с оператором;

7. Описание поведения АУ, имитатора объекта управления, а также мониторинга поведения АУ и управления тестовыми сценариями, должно быть унифицированным.

Глава 2. Четырёхкомпонентная формальная модель динамической верификации процесс-ориентированных программ. Численный метод определения вектора результатов исполнения тестовых сценариев.

Во второй главе описывается четырёхкомпонентная формальная модель динамической верификации процесс-ориентированных программ управления КФС с использованием программной модели объекта управления (ВОУ). Также описывается численный метод определения вектора результатов исполнения тестовых сценариев.

1.1. Общая схема верификации

При динамической верификации исследуются реакции управляющей программы КФС на различные события, возникающие в физической составляющей КФС, а также на воздействие от оператора системы. Алгоритм управления реализуется в виде программы в цифровой составляющей КФС. Программа управления КФС получает данные о физической компоненте, получая считанные значения от датчиков, а также управляющие команды от оператора КФС.

Отсюда, события окружающей среды делятся на два типа: управляющие команды от оператора и события, порождаемые объектом управления (изменение значений его датчиков). События могут происходить в различной последовательности и с различными временными задержками – при этом некоторые события могут быть противоречащими друг другу (то есть не могут встречаться в одной цепочке событий). Для динамической верификации алгоритмов управления КФС предлагается исследовать поведение алгоритма относительно спецификации при различных последовательностях событий.

Таким образом, при динамической верификации свойств программ управления КФС возникают следующие роли:

1) модель физической составляющей КФС;
2) модель верифицируемой программы управления КФС;
3) модель, отвечающая за воздействия оператора КФС на управляющую мрограмму;
4) планирование сценариев работы на объекте управления;
5) проверка реакций алгоритма управления в контексте текущего сценария.

В первой граве были сформулированы требования к матмоделями и методам динамической верификации процесс-ориентированных программ управления КФС, одно из которых говорит о том, описание поведения АУ, имитатора объекта управления, а также мониторинга поведения АУ и управления тестовыми сценариями, должно быть унифицированным. Поскольку верифицируемая программа, описанная в процесс-ориентированном стиле, описывается в виде гиперпроцесса, данной работе было предложено для моделирования перечисленных задач предлагается также воспользоваться математической моделью гиперпроцеса. Чтобы моделировать события окружающей среды, которые прямо не влияют на сигналы датчиков (например, управляющие команды оператора) было решено расширить модель гиперпроцесса.

Это позволит избежать языковой гетерогенности при разработке программ управления КФС, которая будет требовать дополнительных усилий у разработчиков.

.

1.2. Расширенная математическая модель гиперпроцесса

Гиперпроцесс - это расширение классической модели конечного автомата. Гиперпроцесс представляет собой кортеж:

$$H = (T_h, P, p_1) \qquad (1)$$

где P – множество процессов $\{p_1, \ldots p_n\}$. Процесс определяется как множество функций-состояний $F = \{f_0, ..., f_g\}$. В конкретный момент времени активна только одна функция-состояние. Множество функций-состояний делится на два непересекающихся подмножества: активные функции F_i^a и пассивные функции F_i^p. В случае, если автомат находится в пассивном состоянии, процесс не производит никаких реакций на внешние события.

Пассивные функций-состояния:

– f^{NS} - штатное завершение процесса;

– f^{ES} - ошибка (нештатное завершение).

T_h – период активизации гиперпроцесса (такт).

p_1 – начальный процесс из множества P. При запуске гиперпроцесса является единственным активным процессом.

Процессы способны запускать и останавливать друг друга, реагировать на внешние события и изменять свои функции-состояния (изменять текущую активную функцию).

Для динамической верификации процесс-ориентированных программ требуется предоставлять возможность изменения параметров модели окружающей среды, т.е. физических характеристики объекта управления. Также в зависимости от текущего состояния объекта управления может изменяться и сама логика его работы – например, в случае аварий. Таким образом, требуется предоставить возможность изменения и самих режимов работы объекта управления. Однако в общем виде гипепроцесс не позволяет описать упорядоченные последовательности событий, необходимые для динамической верификации, поскольку переменные в гиперпроцессе не типизированные. Поэтому для целей динамической верификации было решено расширить модель гиперпроцесса типом переменных «Очередь сообщений».

X – объединенное множество событий всех компонент гиперпроцесса, а за Y – объединенное множество всех реакций компонент гиперпроцесса. Входные переменных гиперпроцесса принадлежат множеству X, выходные – множеству Y.

Изначально в гиперпроцессе переменные нетипизированные. Введем структурный тип переменных *Очередь сообщений*.

Переменная принадлежит типу *Очередь сообщений,* если выполняются следующие условия:

1) *Очередь* – это кортеж элементов из множества значений переменных гиперпроцесса Q = ($q_0, \dots, q_m$);

2) Для очереди Q определены операции добавления элементов add(*Q, γ*) и операции считывания элемента *read(Q). Q′ = add(Q, γ)* – новое значение переменной Q получается путем добавления значения *γ* в конец Q. *read(Q)* - новое значение переменной Q получается путем удаления первого значения из Q

Значения переменной типа «очередь» - это упорядоченная последовательность переменных типа «сообщение». Сообщение – это кортеж:

$$m = (ID, Type, Param) \qquad (2)$$

где *ID* – у.никальный идентификатор сообщения;
Type – тип передаваемого параметра в сообщении;
Param – параметр передаваемого сообщения.

1.3. Четырехкомпонентная формальная модель динамической верификации процесс-ориентированных программ управления КФС

Определение:

Модель киберфизической системы (*MCPS*) – Это система, состоящая из двух гиперпроцессов: алгоритма управления (Controller) и виртуального объекта управления (Plant):

$$MCPS = (Controller, Plant) \quad (3)$$

где Controller – расширенный гиперпроцесс, который описывает поведение алгоритма управления. Алгоритм управления получает управляющие команды от окружения через переменные типа Очередь, принадлежащие множеству $X_{Controller}$. Controller реагирует на команды от окружения и возвращает данные о состоянии объекта, а также о результатах выполнения управляющих команд оператора, используя для этого переменные типа очередь, принадлежащие множеству $Y_{Controller}$.

Plant – гиперпроцесс, который моделирует поведение объекта управления. Режимы работы объекта управления могут быть различными для одного и того же объекта (например, на объекте управления может что-то сломаться), а также различными могут быть значения физических параметров Plant. Plant получает настроечные команды от внешней среды через переменные типа Очередь, принадлежащие множеству X_{Plant}. Настроечные команды задают режимы работы на объекте управления (к примеру, штатная работа или имитация поломок), а также изменение физических параметров объекта управления.

Также Plant сообщает окружающей среде о своем текущем состоянии в ответ на запрос или по возникновению какого-либо события – например, при возникновении поломки, используя для этого переменные типа очередь, принадлежащие множеству Y_{Plant}.

Также гиперпроцессы Controller и Plant воздействуют друг на друга, имитируя передачу цифровых управляющих сигналов от алгоритма управления и передачу данных с датчиков от объекта управления: реакции Controller являются событиями

для Plant, и наоборот. В этом взаимодействии не участвуют переменные типа Очередь.

Определение:

Модель настраиваемой киберфизической системы **(tuned CPS – TCPS)** это кортеж, который состоит из трех взаимодействующих гиперпроцессов: алгоритма управления, объекта управления и блока управления сценариями (Dispatcher), который моделирует воздействия на Controller и Plant.

$$TCPS = (Dispatcher, Plant, Controller) \quad (4)$$

Как и в MCPS, Controller и Plant моделируют поведение алгоритма управления и объекта управления соответственно, а также их взаимодействие. Dispahcter берет на себя роль настройки КФС: моделирует внешние воздействия на Controller и Plant, задавая или набор строго определенных сценариев работы, или реализуя систему генерации сценариев работы, или же совмещая обе эти роли. В частности, Dispatcher:

• Задает последовательность управляющих команд от оператора КФС – имитирует присутствие оператора, воздействующего на систему;

• Задает последовательность настроечных команд для объекта управления: передает значения физических параметров для ВОУ и задает режимы работы ВОУ;

В случае с MCPS эту роль брало на себя окружение КФС.

Воздействие Dispatcher на Controller и Plant происходит с использованием переменных типа Очередь.

Определение:

Четырёхкомпонентная формальная модель динамической верификации КФС **–** это кортеж *DV*.

$$DV = (Controller, Plant, Dispatcher, Verifier, AR, TS, Qm, rez(ts)) \quad (5)$$

DV включает четыре расширенных гиперпроцесса (рис. 2):

Controller (1) - гиперопроцесс, моделирующий алгоритм управления. Входные данные Controller – управляющие команды от оператора КФС (5), а также сигналы от окружающей среды (сигналы датчиков); выходные данные – управляющие воздействия на физическую компоненту КФС и диагностические сообщения для оператора (6).

Plant (2) – гиперопроцесс, моделирующий виртуальный объект управления (ВОУ). Входные данные Plant – управляющие воздействия Controller, а также конфигурационные сообщения, которые задают режимы работы объекта и его физические параметры (7). Выходные данные – значения сигналов датчиков;

Dispatcher (3) – гиперопроцесс, моделирующий управление сценариями тестирования. Выходные данные Dispatcher - управляющие сообщения, имитирующие сообщения от оператора КФС для Controller (5) и настроечные сообщения Plant (7). Dispatcher запускает и останавливает выполнение тестовых сценариев;

Verifier (4) – гиперопроцесс, обеспечивающий проверку реакций верифицируемого АУ в соответствии с требованиями. Входные данные – управляющие воздействия от Controller на Plant, значения сигналов датчиков, передаваемые от Plant в Controller, и диагностические сообщения для оператора от Controller (6). Выходные данные – отчет о результатах верификации (8).

$AR \subseteq P_V$ – множество требований к верифицируемому ПО, где P_V - все множество процессов Verifier.

A – множество весов требований, где каждый элемент α_i этого множества $\alpha_i \in [0, 1]$.

TS – множество тестовых сценариев.

Qm – вектор результатов исполнения тестовых сценариев.

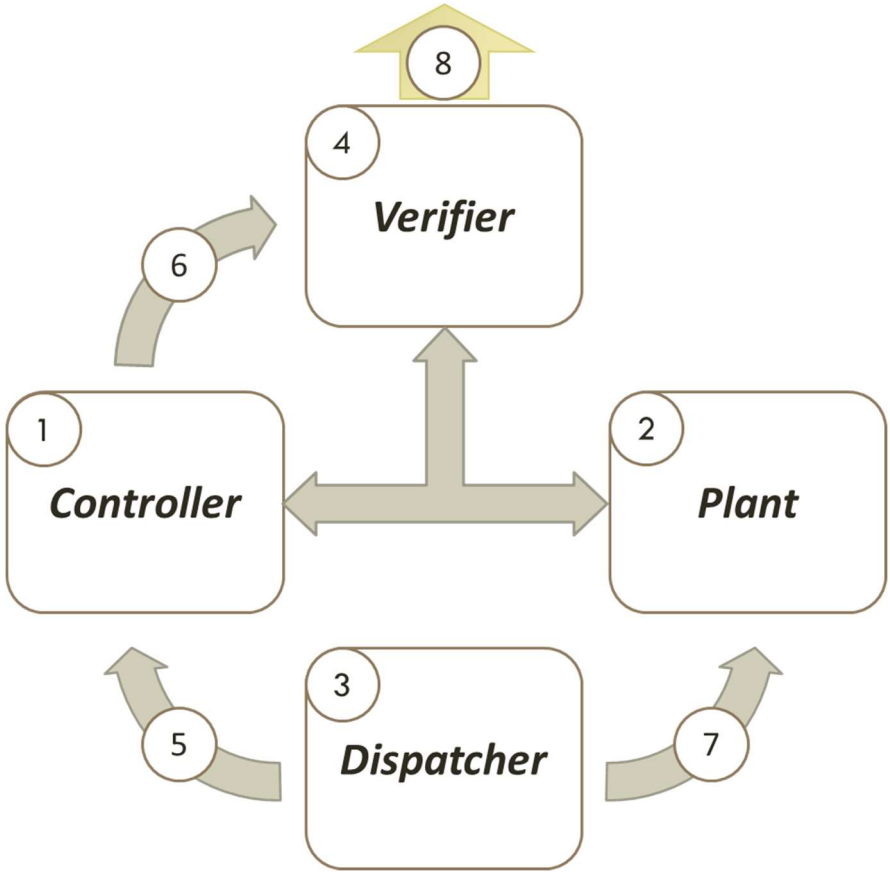


Рис. 2. Общая схема четырёхкомпонентной формальной модели динамической верификации программ управления КФС.

res(ts) – функция, вычисляющая компоненту вектора Qm для тестового сценария ts.

В двух предыдущих системах задача проверки корректности реакций алгоритма управления делегировалась окружению системы гиперпроцессов. В DV проверку корректности реакций Controller при заданных условиях выполняет блок верификации (Verifier). Поскольку реакции Controller зависят от текущего состояния Plant, и от поступавших от Dispatcher настроечных команд, для оценки корректности реакций Controller, блок верификации выносит решения на основании:

1) последовательности реакций Controller, в которые входят диагностическое выходные сообщения, а также управляющие сигналы Plant;

2) последовательности реакций Plant (выходных значений датчиков);

3) последовательности настоечных команд от Dispatcher – в зависимости от ожидаемого сценария, реакция Controller также отличается.

Первые два пункта означают, что для вынесения вердикта о корректности Controller выходные реакции Controller, а также выходные реакции Plant являются входными событиями для Verifier. Передача от Dispatcher информации о том, какие тестовые сценарии сейчас исполняются, происходит с использованием переменных типа Очередь.

Для моделирования перечисленных задач было предложено воспользоваться математической моделью гиперпроцеса, которая была расширена типом данных «Очередь сообщений».

Тестовый сценарий – это упорядоченная последовательность внешних команд для объекта управления и объекта управления, воздействующих на алгоритм управления (к примеру, поступление команды от оператора), а также управляющих режимами работы объекта управления.

Каждый тестовый сценарий $ts \subseteq TS$ – это кортеж

$$ts = \left(M_{D\to C}, M_{D\to P}, ar'\right) \qquad (6)$$

где $M_{D\to C}$ – кортеж сообщений, который получает Controller от Dispatcher при проверке конкретного сценария.

$M_{D\to P}$ – кортеж сообщений, который получает Plant от Dispatcher при проверке конкретного сценария.

$ar'_{TS} \subseteq AR$ – эти процессы моделируют проверяемые на текущем шаге воздействия на управляющее ПО и ВОУ. По запуску текущей тестовой ситуации эти процессы переходят из пассивной функции-состояния в начальную активную функцию.

При завершении верификации текущего тестового сценария критерием успешности прохождения выступает функция *rez(ts),* вычисляемая по завершению проверки каждого тестового сценария.

Вектор результатов исполнения тестовых сценариев определяется как:

$$Qm = (res(ts_1), \dots, res(ts_m)) \quad (7)$$

При исполнении конкретного тестового сценария алгоритм генерирует реакцию: упорядоченную последовательность управляющих воздействий на объект управления, а также упорядоченную последовательность диагностических выходных сообщений для оператора КФС. Выполнение алгоритма управления на текущем тестовом сценарии считается корректным, если алгоритм генерирует корректную последовательность воздействий и диагностических сообщений. В противном случае, алгоритм управления считается некорректным.

1.4. Численный метод определения вектора результатов исполнения тестовых сценариев

Численный метод определения вектора результатов исполнения тестовых сценариев Qm представлен на рис. 3-5 и заключается в последовательной активации гиперопроцессов Dispatcher, Plant, Controller и Verifier (рис. 3).

Dispatcher (рис. 4) управляет порядком прохождения тестов через механизм обмена сообщениями, а блоки Plant, Controller и Verifier последовательно активируются, взаимодействуя через выделенную область данных. Dispatcher моделирует внешнее воздействия на Controller и Plant, задавая или набор строго определенных сценариев работы, или реализуя систему генерации сценариев работы, или же совмещая обе эти роли. Роли Dispatcher:

• Задание последовательности управляющих команд от оператора КФС – имитирует присутствие оператора, воздействующего на систему;

• Задание последовательности настроечных команд для объекта управления: передает значения физических параметров для Plant и задает режимы работы Plant.

Когда Dispatcher исполняет тестовый сценарий ts, он последовательно отправляет сообщения $M_{D\to C}$ и $M_{D\to P}$, соответствующие этому сценарию, тем самым имитируя присутствие оператора и настраивая Plant. Передаются значения физических параметров для Plant и команды на переключение режимов его работы – например, если в тестовом сценарии исследуются реакции ПО при штатной работе или режиме неисправности.

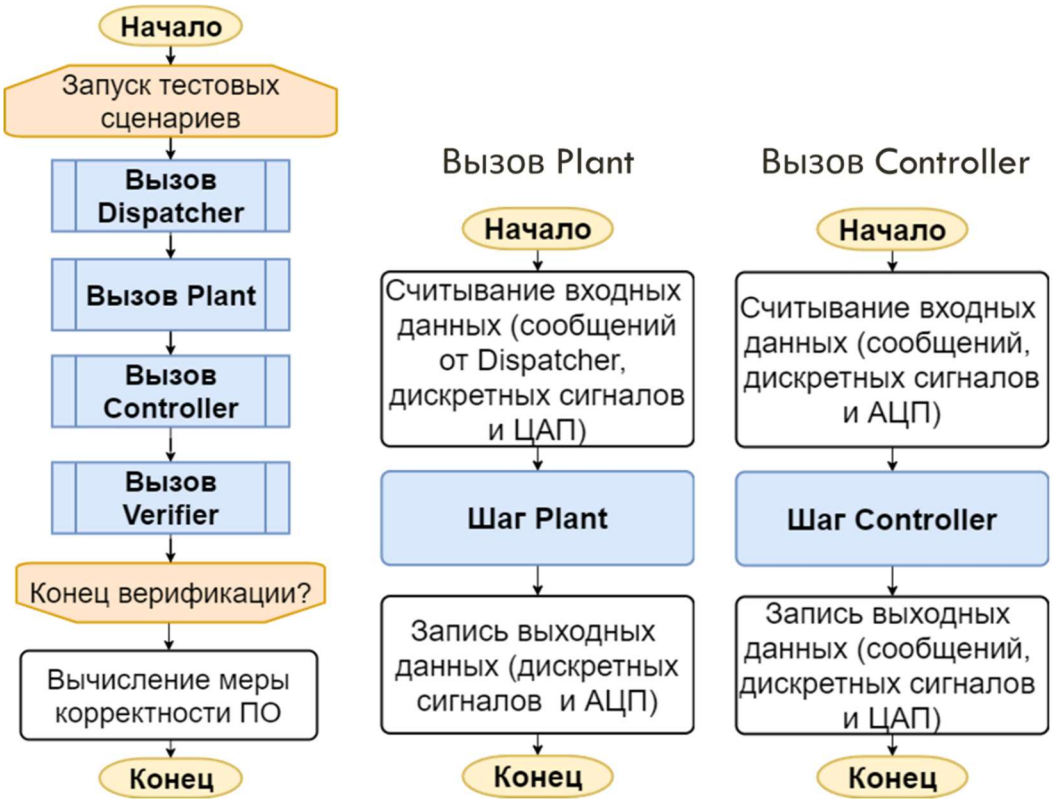


Рис. 3. Численный метод определения вектора результатов исполнения тестовых сценариев.

При исполнении конкретного тестового сценария алгоритм генерирует реакцию: упорядоченную последовательность управляющих воздействий на объект управления, а также упорядоченную последовательность диагностических выходных сообщений для оператора КФС. Выполнение алгоритма управления на текущем тестовом сценарии считается корректным, если алгоритм генерирует

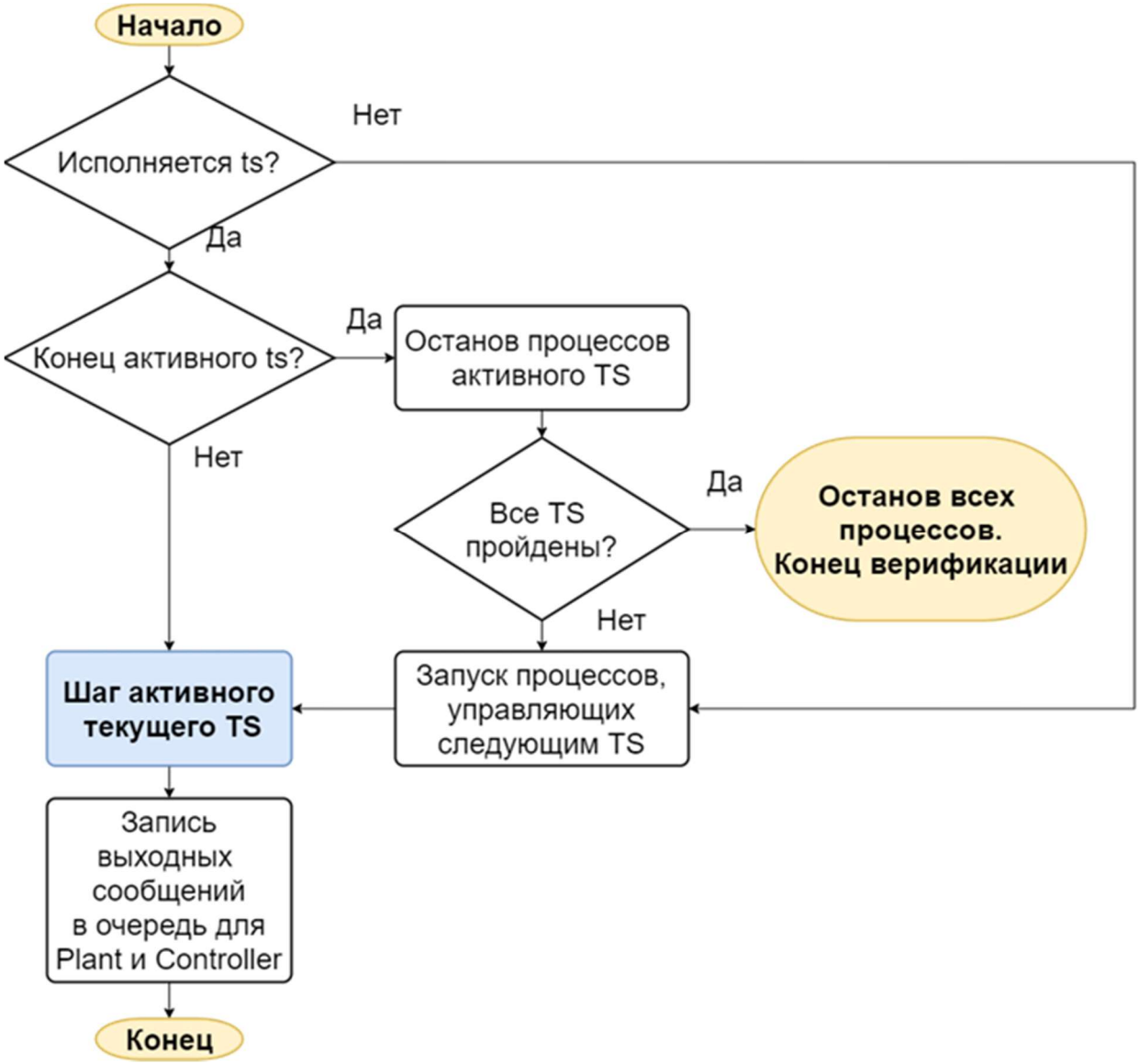


Рис. 4. Численный метод определения бинарного вектора результатов исполнения тестовых сценариев: шаг Dispatcher

корректную последовательность воздействий и диагностических сообщений. В противном случае, алгоритм управления считается некорректным.

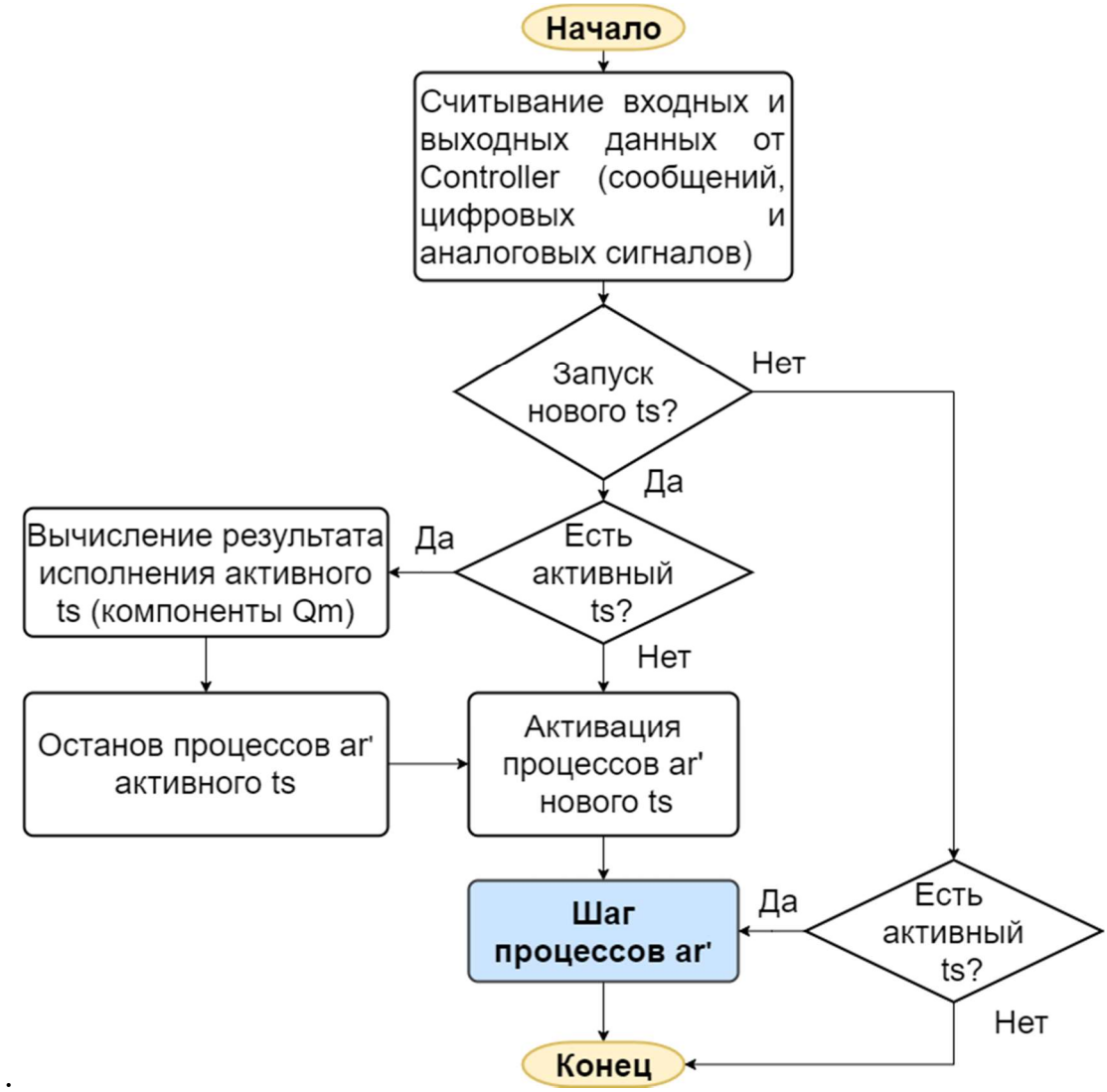
Verifier (рис. 5) по срезу данных между Controller и Plant на каждом цикле активации DV определяет корректность реакций Controller при заданных условиях. Поскольку реакции Controller зависят от текущего состояния Plant, и от поступавших от Dispatcher настроечных команд, для оценки корректности реакций Controller, блок верификации выносит решения на основании:

• Последовательности реакций Controller, в которые входят диагностическое выходные сообщения, а также управляющие сигналы Plant;

• Последовательности реакций Plant (выходных значений датчиков);

• Последовательности настоечных команд от Dispatcher – в зависимости от ожидаемого сценария, реакция Controller также отличается.

Выходные реакции Controller, а также выходные реакции Plant являются входными событиями для Verifier. Передача от Dispatcher информации о том, какие тестовые сценарии сейчас исполняются, происходит с использованием переменных типа *Очередь сообщений*.



.

Рис. 5. Численный метод определения бинарного вектора результатов исполнения тестовых сценариев: шаг Verifier.

По завершению тестового сценария вычисляется результат исполнения тестового сценария:

$$res(ts) = \frac{\sum_i a_i * (curr\_state(p_i) != s^{ERROR})}{\sum_i a_i} \quad (8)$$

$p_n \in AR \ \forall$ n $\in$ [1, R], R – мощность множества ar'_{TC}.

curr_state(proc) – функция, возвращающая текущую функцию-состояние от процесса *proc*.

s^{ERROR} – выделенная пассивная функция-состояние «останов по ошибке».

$\alpha_i \in [0, 1]$ – вес каждого требования.

В случае соблюдения всех требований тестового сценария rez(ts) равен 1. Вектор S = (res (ts_1) , … , res(ts_m)) формируется на основании результатов исполнения тестовых сценариев.

Выводы главы

При динамической верификации программ управления КФС исследуется поведение алгоритма управления и его реакции на различные события, возникающие в системе, такие как: изменения значений сигналов датчиков, управляющие команды от оператора. События эти могут происходить в различной последовательности и с различными временными задержками. Таким образом, при динамической верификации КФС проверяется не только поведение программы, но и учитывается поведение физической составляющей КФС и поведение оператора КФС. Отслеживается реакция алгоритма управления на различные сценарии событий на объекте управления.

В главе предложена четырёхкомпонентная формальная модель динамической верификации процесс-ориентированных программ управления на программной модели объекта управления. Поведение объекта управления, модель алгоритма управления, диспетчеризация тестовых сценариев и проверка реакций управляющего алгоритма моделируется унифицировано математической моделью гиперпроцесса, который был расширен типом переменных Очередь сообщений (гиперпроцессы Controller, Plant, Verifier и Dispatcher).

Поскольку модель гиперпроцесса является исполняемой моделью, динамическая верификация, использующая результаты исполнения программы, может быть автоматизирована. Численный метод определения вектора результатов исполнения тестовых сценариев позволяет автоматизировать проверку корректности разрабатываемой процесс-ориентированной программы управления КФС.

Четырехкомпонентная формальная модель динамической верификации процесс-ориентированных программ и численный метод определения результатов верификации могут быть использованы в итерационной разработке, поскольку все четыре гиперпроцесса могут разрабатываться и изменяться независимо друг от друга. Расширяется не только функциональность алгоритма и объекта управления, но также может быть расширен и кортеж тестовых сценариев вместе с множеством проверяемых требований.

Использование программной модели объекта управления позволяет минимизировать задержки реального объекта управления. При моделировании тестовых ситуаций учитывается возможность управления через Dispatcher режимами работы объекта управления Plant и передачи ему настроченных параметров. Также формальной модели динамической верификации учитывается и диагностические сообщения от алгоритма управления, отправляемые оператору, по которым также можно определить корректность функционирования объекта управления. Таким образом, разработанные методы и модели динамической верификации удовлетворяют сформулированным по результатам исследований требованиям.

Глава 3. Программные комплексы автоматизированной динамической верификации и автоматической динамической верификации программ управления КФС

В третьей главе описана реализация программных комплексов автоматизированной динамической верификации и автоматической динамической верификации программ управления КФС, описанных на процесс-ориентированном языке Reflex, в виде LabVIEW-приложения. Приведен механизм получения исполняемого кода из языка Reflex для встраивания его в автоматизированный и автоматический комплексы динамической верификации, реализованные на базе среды LabVIEW. Приведена апробация предлагаемого подхода динамической верификации процесс-ориентированных программ управления КФС.

3.1. Виртуальные лабораторные стенды

Ранее схема, приведенная на рис. 1, применялась в проекте «Виртуальные лабораторные стенды». Студенты Факультета информационных технологий НГУ использовали эти стенды как тренажер для разработки управляющих программ на языке Reflex. Схема верификации была реализована в упрощенном виде [102]. Виртуальный объект управления создавался в среде LabVIEW. Также средствами LabVIEW описывалась графическая анимация ВОУ и упрощенный интерфейс оператора. Преподаватель визуально проверял корректность написанной студентом программы.

Reflex-код транслировался в код на языке Python. Лабораторный стенд вызывал транслятор Python, который и исполнял программу.

Недостатки:

• Для загрузки на целевые платформы код на языке Reflex транслировался в Си-код. Использование Python могло привести к неожиданным эффектам при запуске кода на реальной системе;

• Использование LabVIEW для описания логики имитатора объекта управления обеспечивало визуализацию, но усложняло разработку объектов управления и их последующую модификацию;

• отсутствие унификации при описании АУ и ВОУ;

• отсутствие механизма обмена сообщениями не позволяло отправлять верифицируемой программе управляющие команды оператора;

• визуальный контроль ошибок

Подход показал свою эффективность в учебном процессе, однако не нашел практического применения в реальных проектах.

3.2. Автоматизированный комплекс динамической верификации

3.2.1. Архитектура комплекса автоматизированной верификации программ управления КФС на языке Reflex

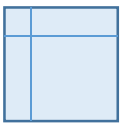
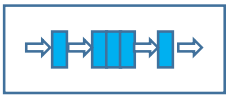
Предложенные в диссертации подходы к динамической верификации процесс-ориентированных программ управления КФС были затем реализованы в автоматизированном комплексе верификации программ на языке Reflex.

В комплексе автоматизированной динамической верификации (рис. 6, табл. 1) оператор управляет тестовыми сценариями вручную и визуально проверяет корректность реакций верифицируемой процесс-ориентированной программы

через графический интерфейс GUI (1). С помощью графического интерфейса оператор подает проверяемой программе (Controller, 6) управляющие команды через очередь сообщений (2). Диагностические сообщения от Controller поступают в интерфейс через очередь сообщений (4). По ним оператор определяет корректность реакций Controller на задаваемые тестовые сценарии и изменения на виртуальном объекте управления Plant (9). Через очередь сообщений (3) оператор может управлять режимами работы Plant и отслеживать его состояние через очередь диагностических сообщений от имитатора объекта управления Plant (5).

По срезу данных между Controller и Plant – входных дискретных сигналов от датчиков (7) и выходных дискретных управляющих сигналов (8) – которые отображаются на GUI, оператор также делает вывод о корректности функционирования Controller. Очередь (10) используется как канал связи между Plant и Controller, позволяя имитировать передачу входных аналоговых сигналов для Controller от датчиков/АЦП. Очередь (11) имитирует передачу выходных аналоговых сигналов Controller (ЦАП).

Таблица 1 – Используемые условные обозначения

	Исполняемый модуль на языке Reflex
	Цифровые порты модуля (I – входные, O – выходные)
	Буфер значений дискретных цифровых сигналов
	Очередь сообщений
	Цифровые данные
	Сообщения

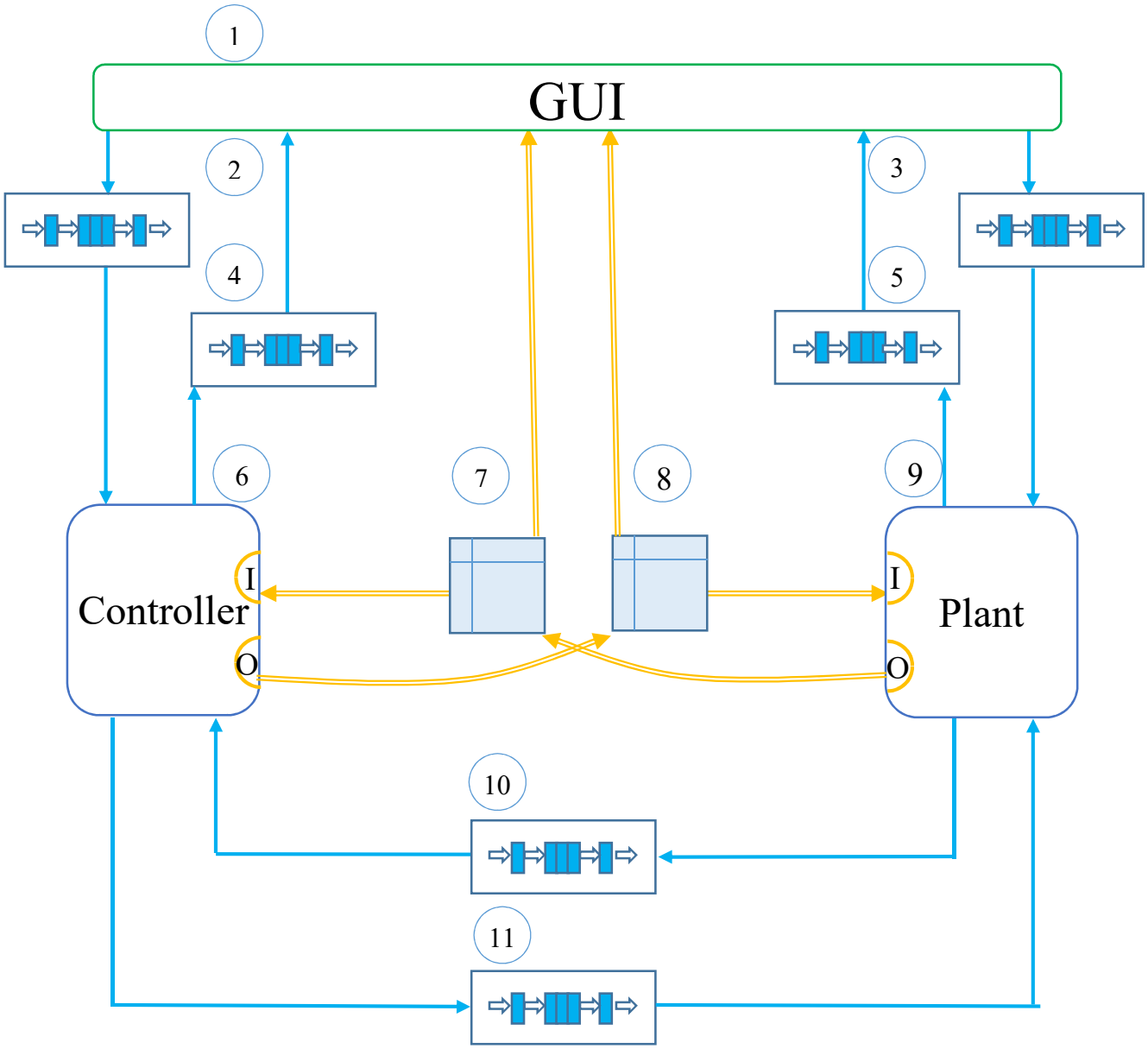


Рис. 6. Архитектура комплекса автоматизированной динамической верификации процесс-ориентированных программ управления КФС на Reflex: 1 – GUI; 2, 3 –очередь входных сообщений для GUI; 4, 5 – очереди выходных сообщений от GUI, 6 – Controller, 7, 8 – буферы входных\выходных дискретных сигналов Controller, 9 – Plant, 10, 11 – очереди для имитации ЦАП\АЦП.

Взаимодействие алгоритмических модулей происходит по следующей схеме:

1. Извлечение входных сообщений от Plant для Controller из очереди сообщений (10), куда помещаются значения фактических параметров на объекте

управления. В этих сообщениях поступает информация о показаниях аналоговых датчиков на объекте управления.

2. Запуск модуля Controller. Управляющая программа: а) извлекает значения своих входных цифровых портов из кэша входных портов Controller (7); б) извлекает сообщения из очереди входных сообщений от интерфейса оператора (2); в) извлекает поступившие на шаге №1 значения фактических параметров. По завершению цикла управления значения выходных цифровых портов Controller заносятся в кэш значений выходных портов (8). Также Conroller формирует очередь сообщений для интерфейса оператора (3), в которых сообщает о текущем состоянии объекта управления и формирует отклики на команды оператора.

3. Запуск модуля Plant. Plant а) извлекает значения своих входных цифровых портов из кэша выходных портов Controller (8); б) извлекает сообщения из очереди входных сообщений от интерфейса оператора (5). Эти сообщения задают: (1) состояние окружающей среды (2) режимы работы Plant (штатный режим работы, спонтанное включение-выключение отдельных элементов объекта управления или авария на объекте). Plant имитирует изменения окружающей среды и ее воздействие на имитируемый объект управления. Также Plant отслеживает, что действия Controller не привели объект управления к недопустимым состояниям. По завершению цикла работы Plant значения его выходных цифровых портов заносятся в кэш входных портов Controller (7). Plant формирует отчет для оператора отладочного интерфейса, в котором сообщает о текущих режимах работы объекта управления, о значении его аналоговых сигналов, об авариях и о недопустимых командах Controller.

Вся информация об объекте управления и об алгоритме управления поступает на графический образ объекта управления (GUI, 1). GUI – это отладочный интерфейс, позволяющий оператору:

- Наблюдать за текущим состоянием Plant и Controller (получать информацию о текущих состояниях, в которых находятся гиперпроцессы, описывающие Controller и Plant);
- Отслеживать кэши значений портов (7, 8);

• Отслеживать очереди сообщений (3, 4), поступающих от Plant и Controller;

• Помещать в очередь сообщений для Controller (2) сообщения, имитируя команды от интерфейса оператора системы управления (задавать команды и настроечные параметры);

• Помещать в очередь сообщений для Plant (2). Этими сообщениями оператор задает: (1) режимы работы Plant (штатный режим работы, имитация поломок и отказов различного оборудования), (2) настроечные параметры Plant (значения температуры, давления и т. д.), (3) внешне параметры окружающей среды. Таким образом оператор способен задавать сценарии работы на объекте управления.

Оператор имеет возможность в любой момент остановить симуляцию и скорректировать сценарий работы на объекте управления – например, на любом шаге симуляции отправить объекту сообщение с командой имитировать поломку или отказ оборудования (например, спонтанное включение или выключение оборудования). В задачу оператора отладочного интерфейса так же входит необходимость визуальной верификации реакции проверяемой программы на команды оператора или на различные ситуации на объекте. Часть задачи по верификации Controller берет на себя Plant, сообщая оператору, если команды от Controller привели Plant в аварийное состояние – или если команды от Controller содержат в себе противоречия. Однако большая часть работы по проверке соответствия поведения управляющей программы наложенным на нее требованиям лежит на операторе отладочного интерфейса: он отслеживает, что после отправления команд управляющая программа корректно управляет объектом управления и формирует соответствующие реакции в зависимости от различных ситуаций на объекте управления.

Недостатки подхода:

• Оператор может пропустить ошибку в работе проверяемой программы управления, поскольку контроль за реакцией алгоритма ведется визуально и это и усложняет анализ отображаемой информации;

• Большое количество ручных операций (на каждой итерации разработки поведение программы должно быть проверено заново по всем тестовым

ситуациям). Это повышает вероятность пропустить тестовую ситуацию и допустить ошибку верификации.

Необходимо было усовершенствовать существующий комплекс динамической верификации программ управления КФС и исключить рутинные операции по запуску тестовых сценариев и контролю реакции программы.

Программный комплекс автоматизированной динамической верификации программ управления КФС был реализован средствами LabVIEW (рис. 7). Модуль загрузки конфигурационных файлов (Tuner) (1) настраивает графический интерфейс GUI (2) комплекса. Tuner получает данные о проверяемой программе и выводит их на GUI. Эти данные позволяют управляют тестовыми сценариями в терминах переменных и констант, описанных в программе на языке Reflex. GUI взаимодействует с модулями Controller(3) и Plant (4). Tuner и GUI, реализованы на языке G, средствами LabVIEW. Исполняемые модули Controller и Plant описываются на языке Reflex, автоматически транслируются в код на языке Си, компилируются в DLL [103] и загружаются в комплекс. В отличие использованного ранее подхода при разработке виртуальных лабораторных стендов, комплекс позволяет вести итерационную разработку Plant и Controller.

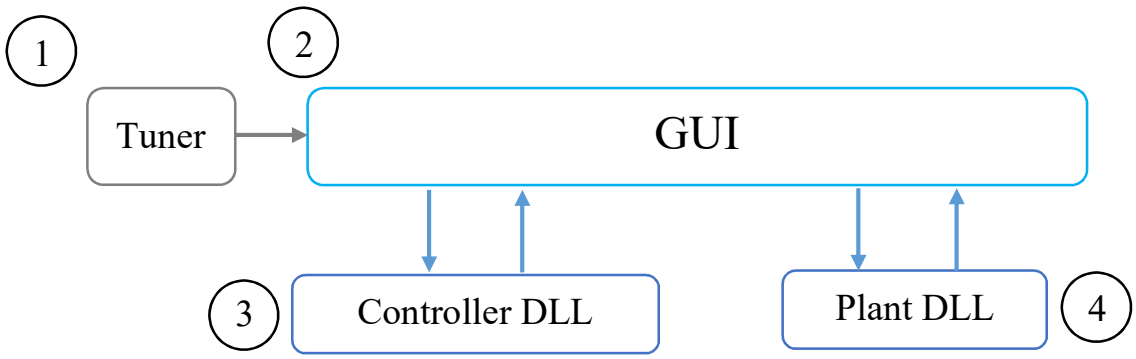


Рис. 7. Архитектура программного комплекса автоматизированной динамической верификации программ управления КФС: 1 – Tuner, 2 – GUI, 3, 4 – алгоритмические модули Controller и Plant (из описания на языке Reflex).

Через GUI оператор управляет тестовыми сценариями и проверяет корректность реакций программы (рис. 8).

Структура GUI:

• панель управления динамической верификацией (3): запуск и останов верификации, переключение пошагового и непрерывного режимов верификации;

• вкладка генерации входных сообщений (рис. 8): установка входных сообщений для Controller (1) и Plant (2);

• вкладки переменных Controller / Plant (рис. 9). Во вкладках находятся таблицы входных портов Controller (или Plant соответственно) (1). В таблице указывается идентификатор порта. Оператор имеет возможность управлять значениями порта вручную (3), отключив сигнал, идущий от Plant (РУЧН), или же подключив у порту Controller соответствующий выходной сигнал от Plant (ВОУ). В битовом представлении порт отображается цветом. На внутренней панели (2) расположена таблица значений глобальных переменных модуля;

• вкладки внутреннего состояния Controller / Plant (рис. 10) отвечают за логирование выходных сообщений модуля (1) и отображение его отладочной информации (2) (таблица процессов и их состояний);

• руководство пользователя «Помощь».

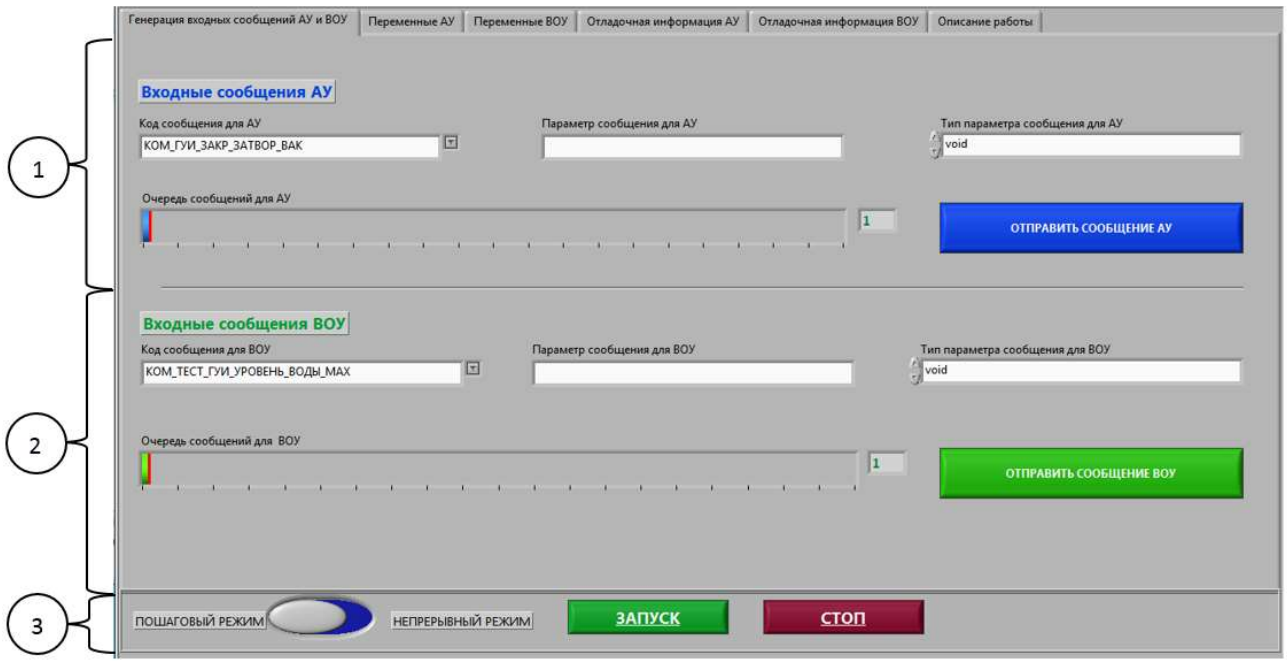


Рис. 8. GUI, вкладка «Генерация входных сообщений для АУ и ВОУ»: 1 – панель генерации входных сообщений для Controller, 2 – панель генерации входных сообщений для Plant, 3 – панель управления динамической верификацией.

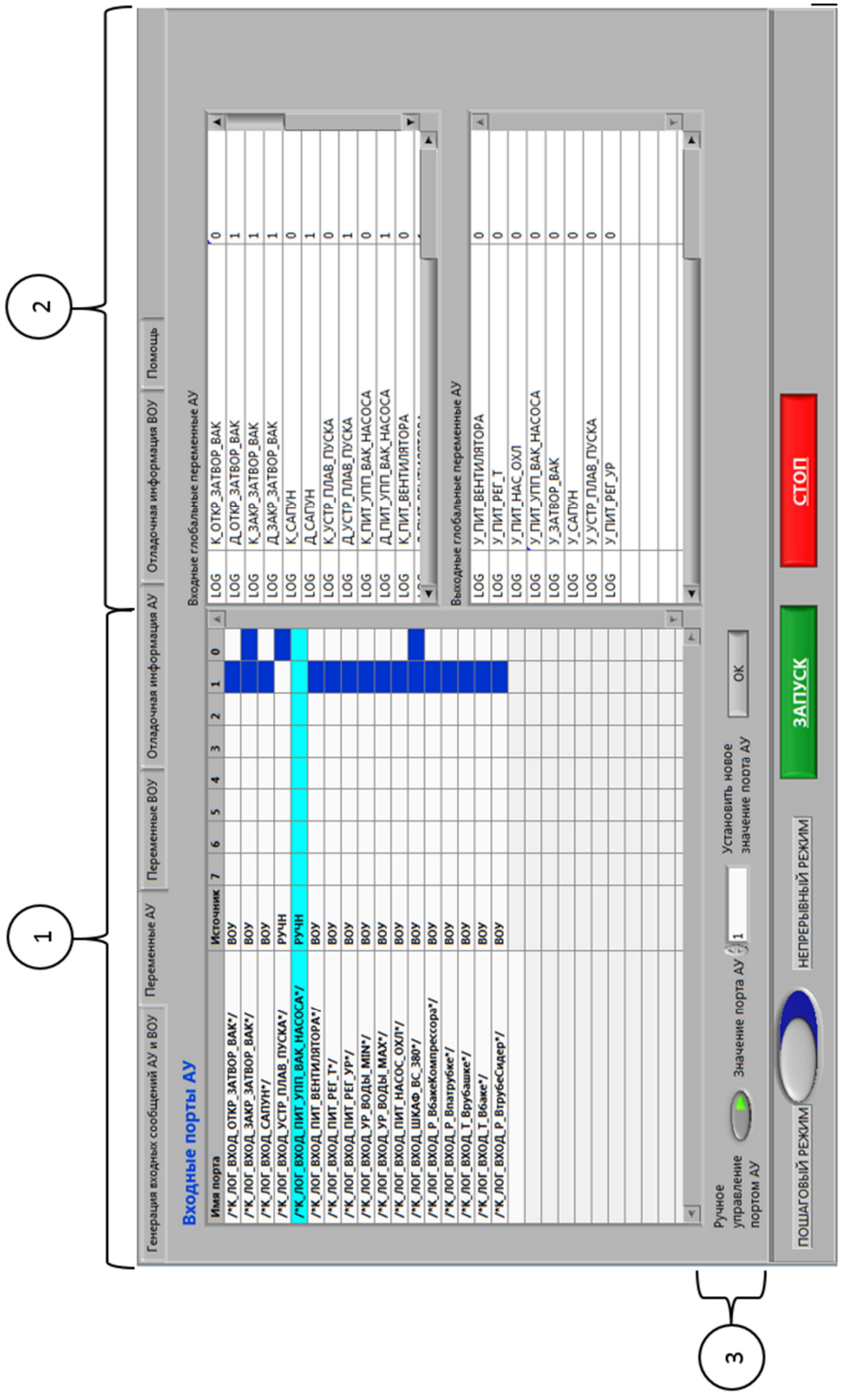


Рис. 9. GUI, вкладка «Переменные АУ»: 1 – панель входных портов Conrtoller, 2 – панель значений глобальных переменных Controler, 3 – панель управления входным портом Controller.

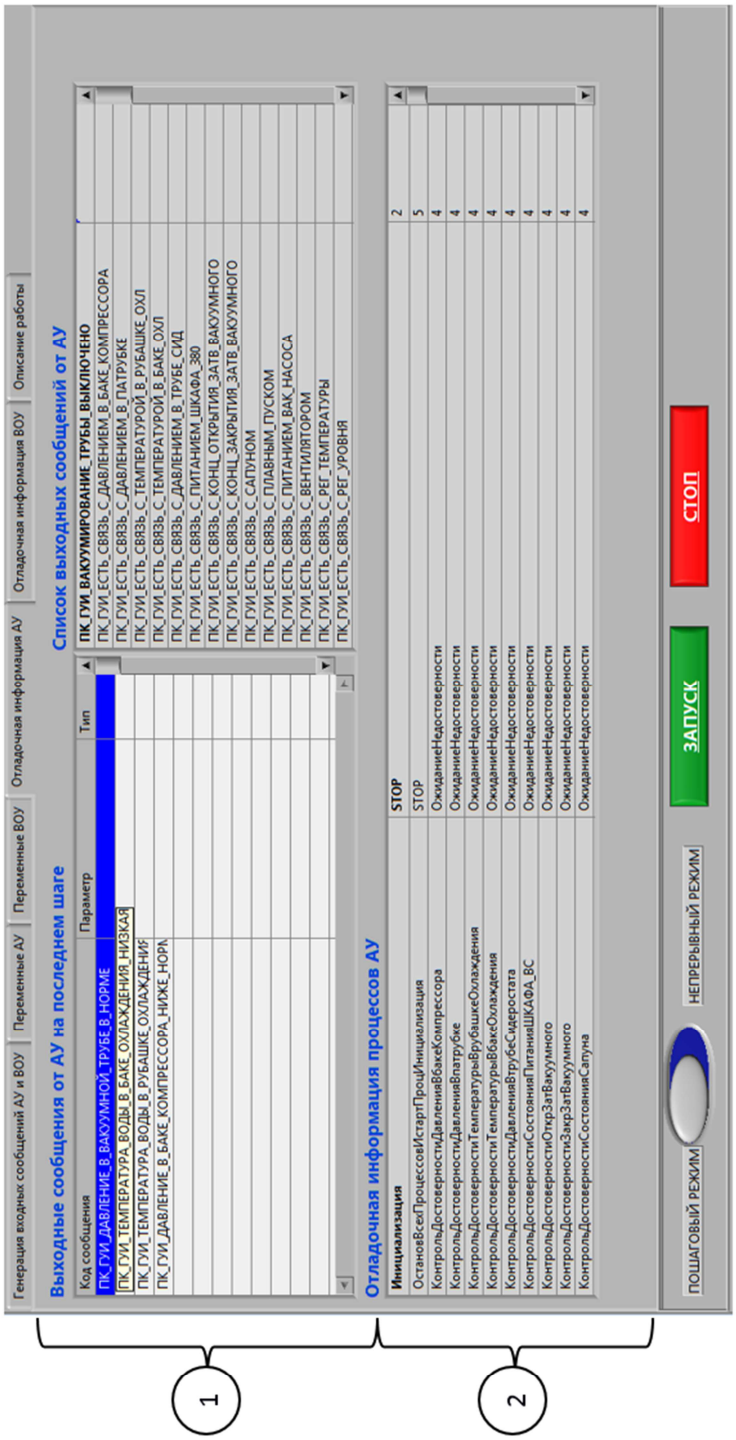


Рис. 10. GUI, «Отладочная информация АУ». 1 – панель выходных сообщений Controller, 2 – панель состояний процессов Controller.

При запуске комплекса Tuner извлекает информацию из конфигурационных файлов, полученных из кода на языке Reflex, и настраивает GUI: устанавливает

идентификаторы портов, процессов, переменных, входных и выходных сообщений. Затем GUI выделяет память, основываясь на данных о Controller и Plant, под буферы входных и выходных переменных для них, очереди сообщений, состояния процессов.

Динамическая верификация запускается по нажатию кнопки «Запуск». GUI циклически активирует Plant и Controller (сначала Plant, затем Controller) и передает им указатели на выделенную память под порты, процессы и очереди сообщений. Оператор создает тестовые сценарии, пользуясь вкладками «Переменные ВОУ», «Переменные АУ», и «Генерация входных сообщений АУ и ВОУ», а также визуально отслеживает корректность реакции программы управления. Разработка программы управления КФС ведется в соответствии с общей схемой итерационной разработки (рис. 1).

3.2.2. Алгоритм работы комплекса автоматизированной динамической верификации

Алгоритм запуска комплекса:

1) Извлечение из конфигурационых файлов информации о портах Controller и Plant, сохранение их имен в строковый массив;

2) Извлечение из конфигурационых файлов информации о входных и выходных сообщениях Controller и Plant. Для каждого алгоблока формируется два массива сообщений: массив идентификаторов сообщений и массив кодов сообщений. В массивах совпадают индексы имени сообщения и соответствующего значения его кода;

3) Извлечение из конфигурационных файлов информации о процессах в Plant и Controller. Для каждого процесса извлекается его имя и список состояний процесса.

Имя процесса сохраняется в строковый массив, список состояний – в двумерный строковый массив по тому же индексу, что и имя;

4) Извлечение из конфигурационных файлов информации о глобальных переменных Controller и Plant. Для каждого типа переменных (входных и выходных) создается четыре массива: массив типов переменных (типа Перечисление), строковый массив имен переменных, массив индексов портов, к которым привязаны переменные, и массив пар типа «беззнаковое целое», хранящих индекс начального и конечного битов, соответствующих переменной в указанном порте.

По команде оператора (нажатие кнопки на графическом интерфейсе) запускается *алгоритм основного цикла динамической верификации*:

1) запуск виртуального объекта управления. По запуску ему передается:
   - Указатель на заголовок массивов входных портов Plant (выходных портов Controller);
   - Указатель на заголовок массива выходных портов Plant (входных портов Controller);
   - Указатель на заголовок массива входных и выходных сообщений Plant;
   - Указатель на заголовок массива объектов структур, описывающих внутреннее состояние Plant (текущие состояния портов и значения тактов).
2) запуск алгоритма управления. На запуске ему предается:
   - Указатель на заголовок массивов входных портов Controller (выходных портов Plant);
   - Указатель на заголовок массива выходных портов Controller (входных портов Plant);
   - Указатель на заголовок массива входных и выходных сообщений Controller;
   - Указатель на заголовок массива объектов структур, описывающих внутреннее состояние Controller (текущие состояния портов и значения тактов).

3) преобразование выходных значений алгоритмических блоков и отображение полученных данных на GUI.

Был разработан C++ интерфейс для обмена данными между алгоритмическими блоками, сгенерированными из описания на языке Reflex, и GUI, реализованном средствами среды LabVIEW. При передаче данных в формате внутреннего представления LabVIEW в алгоритмический блок происходит их преобразование во внутренний формат данных алгоритмический блок. Аналогично, при выводе данных из алгоритмический блок, происходит обратное преобразование выходных данных.

3.3. Автоматический комплекс динамической верификации процесс-ориентированных программ управления КФС

3.3.1. Архитектура комплекса автоматической верификации КФС

Архитектура комплекса автоматической динамической верификации программ управления КФС представлена на рис. 11, табл. 1. Был добавлен алгоритмический модуль управления тестовыми сценариями (Dispatcher) (9). Блок Verifier автоматически контролирует реакцию проверяемой программы (2). Dispatcher через очередь сообщений (8) передает управляющие команды для Controller (4), тем самым имитируя команды от оператора системы. Dispatcher управляет режимами работы Plant (7) через очередь сообщений для настроечных команд (10). Симулятор Plant позволяет имитировать различные режимы работы объекта управления – например, режим штатной работы, аварии или отказы оборудования. Dispatcher оповещает Verifier о запусках и остановах тестовых сценариев через очередь

сообщений (3). Verifier получает информацию о запущенном сценарии – это позволяет ему предсказать, какую реакцию от проверяемой программы ожидать. Verifier определяет корректность Controller на основании буферов входных/выходных сигналов Controller (5, 6) и диагностических сообщений от Controller, передаваемых через очередь сообщений (1).

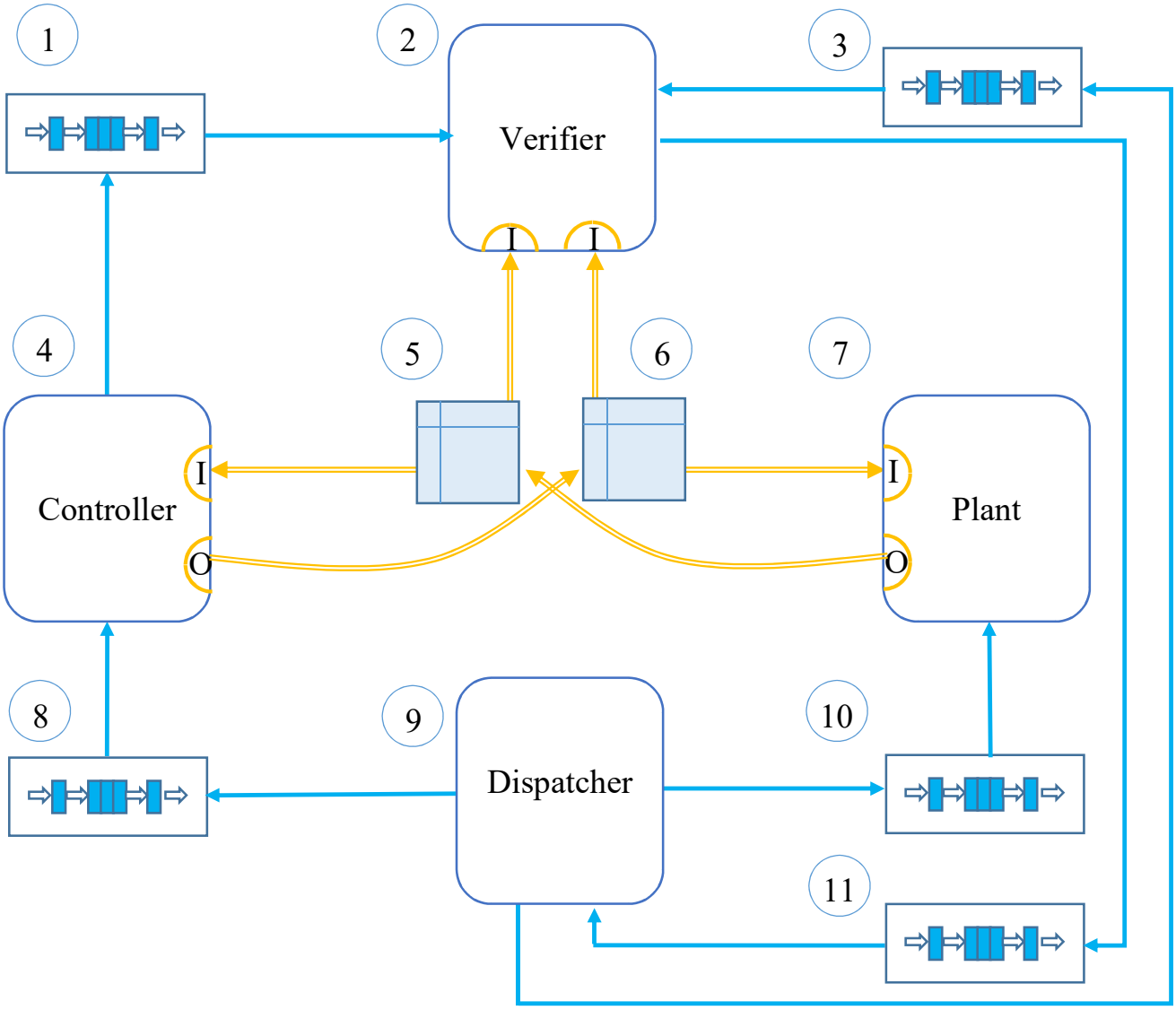


Рис. 11. Архитектура комплекса автоматической динамической верификации программ управления КФС на языке Reflex: 1 – очередь выходных сообщений от Controller, 2 – модуль верификации (Verifier), 3 – очередь сообщений для Verifier от модуля диспетчеризации тестовых сценариев (Dispatcher), 4 – Controller, 5 – буфер входных сигналов Controller, 6 – буфер выходных сигналов Controller. Используется для имитации выходных аналоговых сигналов (ЦАП), 7 –

виртуальный объект управления (Plant), 8 – очередь сообщений для передачи штатных команд от Dispatcher к Controller, 9 – модуль управления сценариями (Dispatcher), 10 – очередь сообщений от Dispatcher к Plant, 11 – очередь сообщений от Verifier к Dispatcher.

Для верификации управляющей программы используются четыре взаимодействующих алгоритмических модуля (гиперпроцесса) – четыре программы, описанные на языке Reflex, которые запускаются в заданном порядке и формируют очереди сообщений друг для друга, а также значения входных портов и фактических параметров:

• Controller – алгоритмический модуль, реализующий логику работы разработанного алгоритма управления.

• Виртуальный объект управления (Plant) – алгоритмический модуль, реализующий логику работы объекта управления. Plant имитирует данные, поступающие алгоритму управления от АЦП, а также данные, поступающие алгоритму от цифровых устройств ввода (DI).

• Verifier – алгоритмический модуль, отвечающий за проверку выполнения конкретного набора требований при заданном сценарии работы Controller и Plant. БВ поочередно проверяет наборы непротиворечащих друг другу темпоральных требований и сообщает об ошибках выполнения при работе алгоритма. Verifier верифицирует исключительно алгоритмический блок.

• Модуль диспетчеризации тестовых сценариев (Dispatcher) – алгоритмический блок, задача которого – устанавливать Controller, Plant и Verifier в соответствующие режимы работы и «проигрывать» предопределённые сценарии работы. Для Controller Dispatcher имитирует команды, поступающие от интерфейса оператора, тем самым имитируя ручное управление. Для Plant Dispatcher формирует очередь входных сообщений, в котором сообщает Plant, в каком режиме на данный момент происходит верификация (штатный режим или имитация поломок). Также Dispatcher сообщает объекту об изменениях в окружающей среде и проводит начальную настройку Plant. Для Verifier Dispatcher формирует сообщения, которые оповещают Verifier о том, по какому сценарию сейчас

работает Controller и Plant, и, соответственно, какие требования в данный момент необходимо верифицировать.

Взаимодействие алгоритмических модулей происходит по следующей схеме:

1. Запуск модуля управления сценариями (9, Dispatcher). Dispatcher заполняет три очереди сообщений: очередь сообщений для Controller (8), для Plant (10) и для Verifier (3). Для Controller в очередь сообщений помещаются сообщения, имитирующие действия оператора системы управления. Для Plant Dispatcher формирует сообщение о том, какой сценарий в данный момент проверяется.

Dispatcher может сообщить Plant о том, что сейчас ему необходимо сымитировать отказ оборудования, нештатную ситуацию или аварию. Dispatcher также сообщает Plant об изменениях условий окружающей среды, оставляя на отступ самому Plant решать, как реагировать на эту ситуацию. В третью очередь сообщений, которая предназначается для Verifier, Dispatcher также помебщает сообщение о том, какой сценарий отыгрывается в данный момент. По этому сообщению Verifier делает вывод, какой реакции следует ожидать от алгоритма управления в случае корректной работы. По завершению тестового сценария Dispatcher отправляет Verifier служебное сообщение;

2. Запуск модуля виртуального объекта управления (7, Plant). Plant принимает сообщения и очереди сообщений от Dispatcher (10), анализирует их и переходит в требуемый режим верификации. Также из этих сообщений Plant получает данные об окружающей среде и соответственно изменяет свои внутренние параметры. В значения своих входных портов Dispatcher считывает данные из кэша выходных портов Controller (6). Это имитирует обмен данными между реальным объектом управления и алгоритмом управления, когда все управляющие дискретные сигналы объект получает в виде входных цифровых сигналов. Изменения настроечных параметров Controller передает Plant также через порты. По завершению цикла работы Plant помещает значения своих выходных портов в кэш входных портов Controller (5). Значения фактических параметров Plant возвращает для Controller также через свои выходные порты

3. Запуск управляющей программы (4, Controller). В этом модуле исполняется управляющий код системы управления. Controller считывает данные своих входных из кэша входных значений портов алгоритма (5), извлекает значения фактических параметров и считывает сообщения, пришедшие от Dispatcher из очереди сообщений (8). По завершению цикла работы блока, Controller формирует значения выходных портов и записывает их в кэш значений выходных портов алгоритма (6). Выходные сообщения он помещает в очередь выходных сообщений для Verifier (1);

4. Запуск блока верификации (2, Verifier). Verifier анализирует все данные, поступающие от алгоритма управления: очередь его выходных сообщений (1) и значения входных и выходных портов, извлеченных из кэшей значений портов (5, 6). Обладая этими данными, Verifier в состоянии проверить, насколько корректно алгоритм реагирует на команды оператора (входные сообщения, имитируемые блоком Dispatcher) и на значения входных портов, которые формирует для Controller блок Plant. Очередь выходных сообщений Verifier предназначена только для отладочного интерфейса оператора, которому Plant сообщает о результатах своей проверки;.

Разработка программ управления КФС на языке Reflex происходит по итерационной схеме:

1. Создается функционально обособленная часть кода Controller на языке Reflex, т.е. набор процессов, отвечающих за выполнение определенной задачи;

2. Создается функционально обособленная часть кода Plant на языке Reflex. Этот код имитирует работу тех элементов объекта управления, которые управляются описанным в П.1. алгоритмическим блоком.

3. Описывается блок Dispatcher. В Dispatcher задаются три типа сообщений:

- сообщения, которые поступают в созданный блок Controller и имитируют действия оператора;
- сообщения, которые поступают в блок Plant и задают сценарии его работы;

• сообщения для Verifier – эти сообщения оповещают Verifier о том, в каком режиме сейчас ведется проверка. По этим сообщениям Verifier определяет, какой сценарий исполняется в данный момент и какой реакции ожидать от Controller.

4. Описывается блок Verifier. В блок Verifier добавляется реакция на сообщения от Dispatcher, описанные в П.3. Также описываются требования, накладываемые на созданный алгоритмический блок Controller. Так как работа ведется итерационно, эти требования также могут проверять работу частей алгоритма, созданных на предыдущих шагах – в таком случае, проверяется совместное взаимодействие различных частей алгоритма.

При таком подходе сложность управляющей программы постепенно увеличивается, усложняется и сам симулятор объекта управления. На начальных циклах итерации описываются простые компоненты объекта управления, тогда как в последующих создаются компоненты, описывающие взаимодействие созданных ранее простых частей имитатора – так имитируется взаимодействие различных устройств на объекте управления. При этом, код алгоритма управления не зависит от того, что в данный момент происходит имитация и верификация, а не работа с реальным объектом управления. Верифицированный таким методом код встраивается в результирующую систему управления

Программный комплекс автоматической динамической верификации процесс-ориентированных программ на языке Reflex (рис. 3) реализован как LabVIEW-приложение. Графический интерфейс оператора (GUI) (2) комплекса конфигурируется модулем загрузки конфигурационных файлов (Tuner) (1). Модуль активации и синхронизации алгоритмических модулей (DV Core) (3) вызывается из GUI и управляет исполнением алгоритмических модулей Controller (4), Plant (5), Dispatcher (6) и Verifier (7). Tuner и GUI реализованы на языке G LabVIEW. DV Core на языке Си++ и встроен в среду LabVIEW как DLL. Такое решение было принято чтобы отделить управление и процессом динамической верификации от средств LabVIEW – при таком подходе загрузка исполняемых модулей происходит независимо от LabVIEW, которая отвечает только за интерфейс пользователя и его настройку. Исполняемые модули Controller, Plant, Dispatcher и Verifier

транслируются из описания на языке Reflex в код на языке Си и собираются в DLL. DV Core загружает и выгружает их автоматически. Такой подход позволяет не перезагружать комплекс, когда собирается новая версия исполняемых модулей.

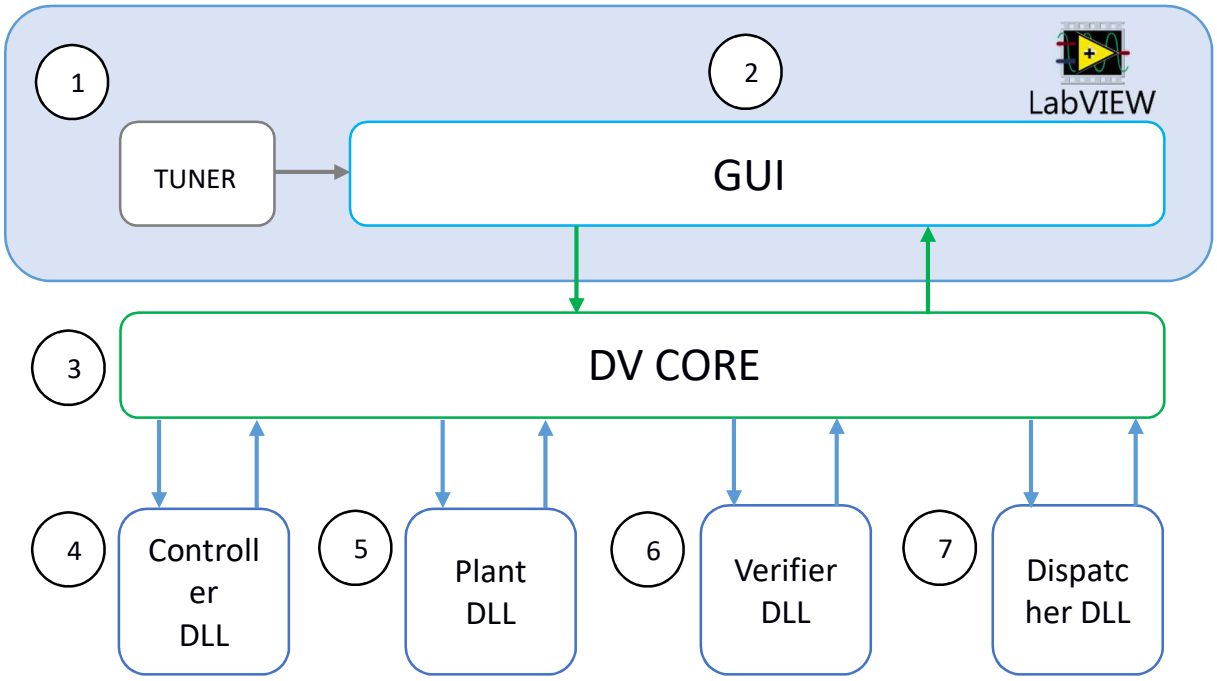


Рис. 12. Архитектура программного комплекса автоматической динамической верификации программ управления КФС: 1 – модуль загрузки конфигурационных файлов (Tuner), 2 – GUI, 3 – модуль активации и синхронизации алгоритмических модулей DV Core, 4–7 – исполняемые модули Controller, Plant, Verifier, Dispatcher, сгенерированные из описания на языке Reflex.

Проект состоит из программ на языке Reflex, описывающих алгоритмические модули Controller, Plant, Verifier, Dispatcher, схемы объекта управления (графический файл) и конфигурационного XML-файла, который описывает активные графические индикаторы на схеме объекта. Файлы проекта находятся в отдельной директории.

На запуске открывается окно верификации (рис. 13):

1. Меню управления проектом.

2. Панель управления верификацией.

3. Графическое представление объекта управления.

Через меню управления проектом пользователь загружает проект (или создает новый).

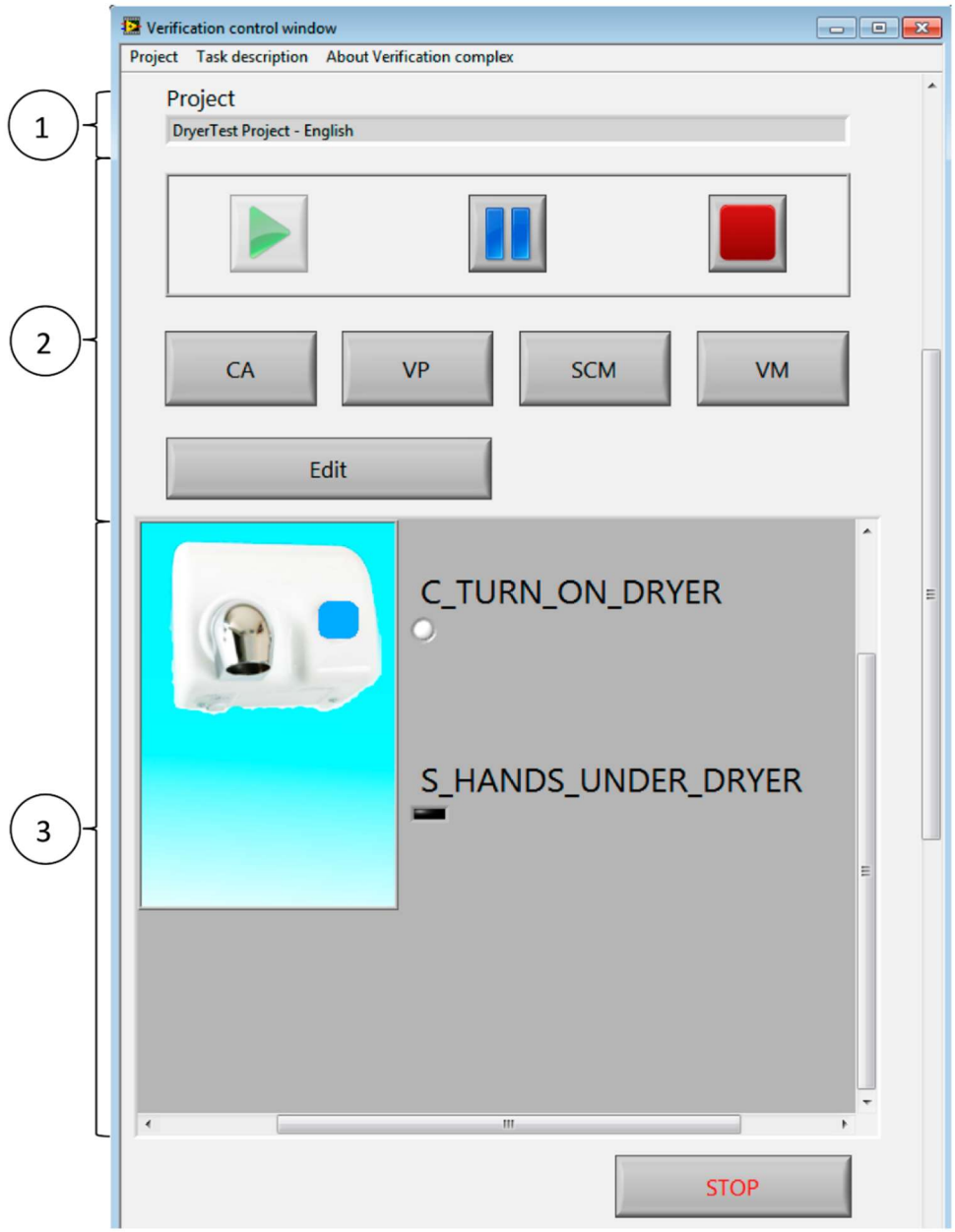


Рис. 13. GUI комплекса автоматической динамической верификации: окно верификации. 1 – меню управления проектом, 2 – панель управления верификацией, 3 – графическое представление объекта управления.

Панель (2) предоставляет возможность:

- запустить редактор кода на языке Reflex;
- начать, прервать и остановить верификацию;
- открыть панели Controller, Plant, Dispatcher и Verifier, на которых отображается информация о внутреннем состоянии алгоритмических модулей.

На GUI алгоритмического модуля (рис. 13) отображаются:

1. Значения внутренних глобальных переменных.
2. Лог входных и выходных сообщений.
3. Отладочная информация модуля (состояние процессов).

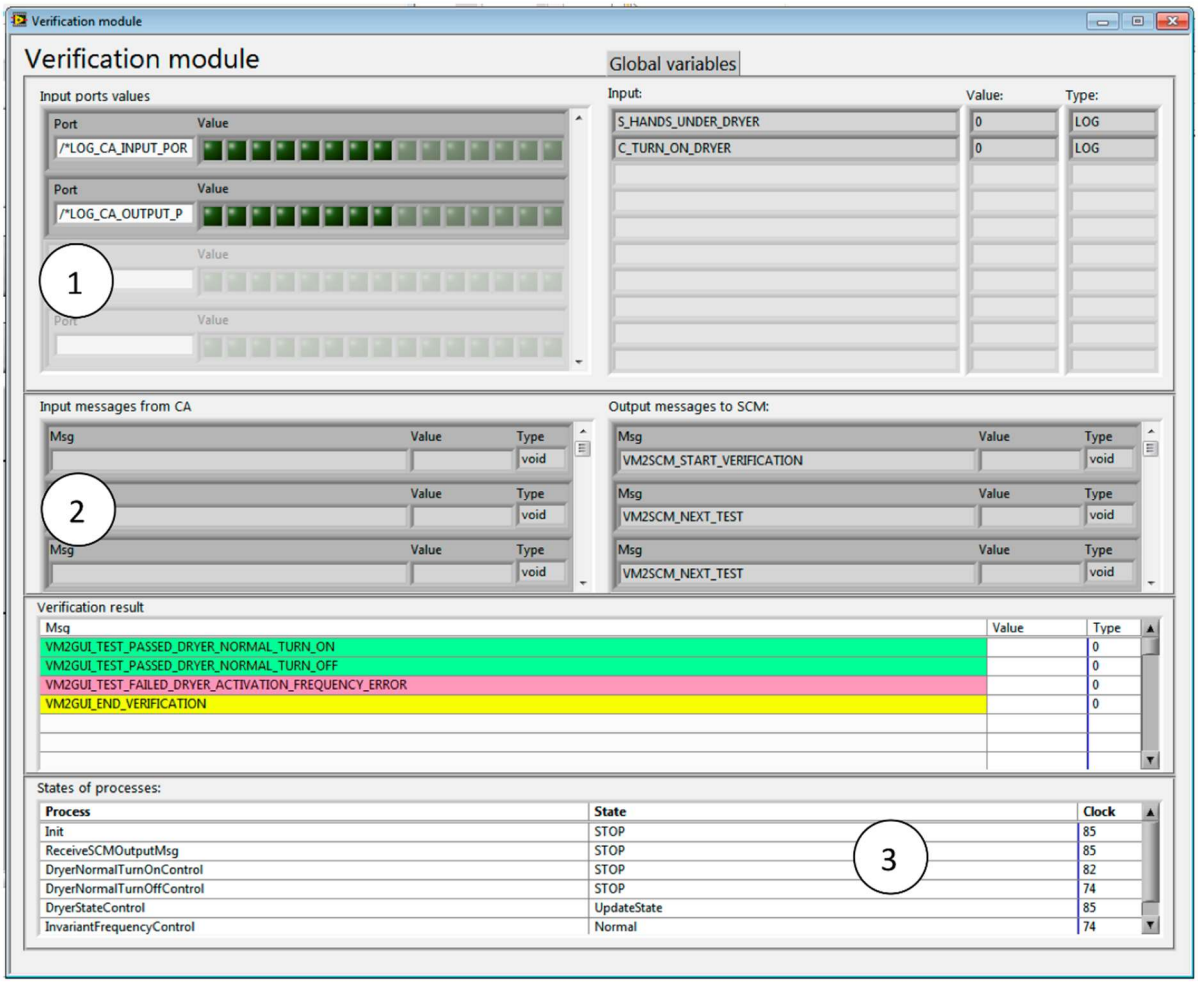


Рис. 13. GUI: панель Verifier. 1 – значения внутренних глобальных переменных, 2 – лог входных и выходных сообщений, 3 – отладочная информация модуля.

По запуску динамической верификации GUI вызывает DV Core и передает ему запрос на подключение исполняемых модулей. DV Core делает попытку загрузить алгоритмические модули. Если DV Core не находит нужных DLL или в процессе загрузки возникает ошибка (например, произошла попытка загрузки модулей из некорректного проекта), на GUI отображается диагностическое сообщение. Если загрузка DLL проходит успешно, на GUI отображается диагностическое сообщение об успехе загрузки.

Загруженные модули обновляют каждый свои панели из GUI, основываясь на своих внутренних параметрах: резервируют необходимое количество памяти под буферы переменных, очереди сообщений и отладочную информацию. Если загрузка прошла успешно, GUI инициализирует область графического представления объекта управления (рис. 13 (3)).

DV Core циклически вызывает модули в порядке: Dispatcher, Plant, Controller, Verifier. На панели Verifier логируются результаты прохождения тестовых сценариев. Комплекс предполагает разработку программ по итерационной схеме (рис. 1).

Комплекс работает в двух режимах: режим соответствия модели DV физическим характеристикам системы и режим изменения масштаба времени. В первом режиме частота запуска модулей соответствует частоте из активации на реальной системе. Во втором режиме убирается задержка между запусками модулей. Это режим был введен, чтобы предоставить оператору возможность уменьшить время верификации программы (до двух порядков).

3.3.2. Алгоритм работы комплекса автоматической динамической верификации процесс-ориентированных программ

Алгоритм запуска комплекса автоматической динамической верификации:

1) если комплекс запускается в первый раз – в п. 5. Иначе – в следующий шаг;

2) извлечение данных из конфигурационных файлов настойки комплекса. Из конфигурационного файла извлекаются: файловый адрес последнего открытого проекта, файловый адрес графического представления объекта управления проекта и адрес конфигурационного файла XML для описания графического представления объекта;

3) отображение графского представления объекта (т.е. некой графической схемы объекта управления с расположенными на нем сигнальными индикаторами), генерация по конфигурационному файлу индикаторов для отображения состояния объекта, привязка их ко внутреннему представлению портов алгоритма управления;

4) запуск рабочего цикла комплекса;

5) если нажата кнопка запуска верификации – переход в режим верификации алгоритма управления. Иначе – холостой режим.

Алгоритм работы в режиме динамической верификации программ управления КФС:

1) по запуску динамической верификации в блок DV Core передаются:

a. Указатели на заголовки на входной и выходной массивы образов портов для Controller и Plant;
b. Указатели на заголовки очередей сообщений.
   Для Controller:
   - Указатель на заголовок очереди входных сообщений от Dispatcher;
   - Указатель на заголовок очереди выходных сообщений от Controller;

   Для Plant:
   - Указатель на заголовок очереди входных сообщений от Dispatcher;
   - Указатель на заголовок очереди выходных сообщений для GUI;

   Для Verifier:
   - Указатель на заголовок очереди входных сообщений от Dispatcher;
   - Указатель на заголовок очереди выходных сообщений для Dispatcher;

– Указатель на заголовок очереди выходных сообщений для GUI.

Для Dispatcher:

– Указатель на заголовок очереди выходных сообщений для Controller;

– Указатель на заголовок очереди выходных сообщений для Plant;

– Указатель на заголовок очереди выходных сообщений для GUI.

c. Указатели на массивы с отладочной информацией для Controller, Plant Dispatcher и Verifier

2) в DV Core проходит полный цикл верификации;

3) DV Core возвращает обновленные данные о значениях массивов портов, заголовков очередей сообщений и массивов с отладочной информацией для Controller, Plant Dispatcher и Verifier;

4) обновление графического представления объекта: извлечение данных из массива образа портов и изменение значений индикаторов на графическом представлении в соответствии со значениями входных и выходных портов Controller.

5) если в очереди выходных сообщений от Verifier есть сообщение о конце верификации – в п. 6. Иначе – П. 1.

6) отображение диагностического сообщения о конце верификации.

7) останов верификации.

Алгоритм работы DV Core.

1) загрузка алгоритмических модулей в память комплекса;

2) проход цикла верификации. Модули активируются в порядке: Dispatcher, Plant, Controller, Verifier;

3) передача обновленной по результатам цикла верификации информации на интерфейс пользователя;

4) если от интерфейса пользователя есть сообщение об остановке симуляции – переход в П. 5. Иначе- в П.2;

5) выгрузка алгоритмических модулей из памяти комплекса. Останов верификации.

Алгоритм динамической загрузки алгоритмических модулей в модуле DV Core.

1) по нажатию на кнопку запуска верификации на пользовательском интерфейсе происходит вызов метода динамической загрузки алгоритмических модулей из модуля DV Core. В метод передается строковое представление адреса директории проекта;

2) по запуску DV Core происходит выгрузка из памяти комплекса уже загруженных DLL Controller, Plant, Dispatcher и Verifier. Если верификация запускается первый раз и загруженных библиотек еще нет, переход на шаг 4;

3) очистка (обнуление) информации, описывающей выгруженные DLL: обнуление указателей на загруженные DLL, обнуление указателей на вызываемые из DLL алгоритмических модулей методы, очистка памяти с адресом директории модуля в файловой системе ОС, удаление строковых представлений имен библиотек;

4) формирование текстовых представления для адресов DLL модулей;

5) поочередная загрузка модулей всех алгоритмических модулей состоит из следующих шагов:

   a. Загрузка модуля DLL по сформированному в п.4 адресу директории DLL. Если произошла ошибка загрузки – прерывание процесса загрузки, сохранение индекса ошибки, запись сообщения об ошибке в файл. Переход в П. 6.
   b. Получение от модуля DLL адреса основой функции, которая будет вызываться при каждом шаге верификации. Если произошла ошибка– прерывание процесса загрузки, сохранение индекса ошибки, запись сообщения об ошибке в файл. Переход в П. 6

6) завершение загрузки. Возврат информации о результатах загрузки алгоритмических модулей в память LabVIEW;

7) в LabVIEW-приложении происходит анализ возвращаемого из DV Core значения. В случае детектирования сообщения о ошибке – будет передано диагностическое сообщение.

После того как алгоритмические модули загружены в память комплекса, запускается цикл верификации. Цикл верификации состоит из последовательной активации алгоритмических модулей и вызова у них метода, отвечающего за исполнение одного шага конкретного модуля.

Проход цикла верификации происходит по следующему универсальному алгоритму (П. 2.)

1) запуск модуля DV Core. Из комплекса верификации в DV Core передается на каждом шаге верификации следующая информация:

a. Заголовки массивов образов входных и выходных портов Controller, Plant, Dispatcher и Verifier.

– Заголовки следующих очередей сообщений:
  – очередь выходных сообщений от Controller;
  – очередь сообщений для Verifier от Dispatcher;
  – очередь сообщений для передачи штатных команд от Dispatcher к Controller;
  – очередь сообщений от Dispatcher к Plant;
  – очередь сообщений от Verifier к Dispatcher;

b. Заголовки массивов с отладочной информацией процессов модуля (состояние каждого процесса и текущий такт)

Необходимость передачи этих данных на каждом шаге верификации связана с тем, что DV Core не имеет возможности один раз сохранить заголовок массивов с данными, чтобы затем передавать их конкретным модулям. В LabVIEW существует свой менеджер управления памятью, который занимается упорядочиванием памяти и перевыделением памяти под ресурсы. Это перевыделение может привести к тому, что хранящиеся в DV Core заголовки могут стать некорректными. Поэтому необходимо на каждом шаге передавать корректные заголовки.

2) если алгоритмические модули уже загружены в память комплекса – в следующий шаг. Иначе – в П. 11.;

3) из DV Core в алгоритмический модуль передается полученная от комплекса информация: заголовки массивов образов портов, заголовки очередей сообщений и заголовки массивов с отладочной информацией;

4) если это первый запуск алгоритмического модуля – следующий шаг. Иначе – п. 11;

5) инициализация внутреннего массива с отладочной информацией алгоритмического модуля;

6) инициализация внутренней очереди сообщений алгоритмического модуля;

7) перевыделение памяти для массива образов портов, заголовок которого был передан из комплекса. Таким образом, после завершения первого шага алгоритмического модуля, комплекс сможет оперировать с массивами корректной длины. Такой подход связан с тем, что в различных задачах количество портов алгоритма может быть различным. Таким образом, на первом запуске алгоритмический модуль, используя стандартные библиотеки для работы с внутренней памятью, перевыделяет память под данные, которыми он обменивается с комплексом. Перевыделение памяти позволяет сэкономить используемую память и происходит только один раз, после загрузки алгоритмического модуля в память комплекса – потому не приводит к накладным расходам производительности;

8) перевыделение памяти для очередей сообщений, заголовки которых были переданы из комплекса (аналогично п 7);

9) перевыделение памяти для массива с отладочной информацией (аналогично П. 7);

10) обновление внутренних указателей алгоритмического модуля на в соответствии с полученным от DV Core заголовком;

11) конвертация массивов входных сообщений, полученных от DV Core в виде заголовка во внутреннем формате данных LabVIEW вон внутренний формат очередей сообщений алгоритмического модуля. Это необходимо для дальнейшего переиспользования алгоритмического модуля независимо от среды LabVIEW;

12) запуск по очереди всех процессов гиперпроцесса;

13) конвертация массивов выходных сообщений, полученных от DV Core в виде заголовков во внутреннем формате данных LabVIEW, во внутренний формат очередей сообщений алгоритмического модуля. Это необходимо для дальнейшего переиспользования алгоритмического модуля независимо от среды LabVIEW;

14) конвертация массива с отладочной информацией из внутреннего формата алгоритмического модуля во внутренний формат данных LabVIEW.

3.4. Генерация исполняемых алгоритмических модулей из описания на языке Reflex

Комплексы автоматической динамической верификации и автоматизированной динамической верификации интегрируют в себя алгоритмические модули, полученные из описания модулей на языке Reflex. Для того чтобы удовлетворять нуждам динамической верификации, в рамках данной работы транслятор из языка Reflex в язык Си был расширен набором команд (консольных ключей), указывающих на необходимость генерации конфигурационных файлов.

На вход транслятор получает текстовый файл с расширением rcs. Изначально по результатам своей работы транслятор генерировал файлы:

- .c – файлы Си-кода, автоматически сгенерированного из описания на языке Reflex.
- .map – файл, содержащий перечисление всех процессов и их состояний.

При разработки комплексов динамической верификации к транслятору были добавлены ключи:

- P – ключ, указывающий на то, что транслятор генерирует .cpp файлы кода;
- G – ключ, указывающий на то, что транслятор генерирует файл-описание портов (ports.h);

• T – ключ, указывающий на то, что транслятор генерирует файл-описание глобальных переменных, объявленных в коде на языке Reflex (.var)

• N – ключ, указывающий на то, что транслятор генерирует конфигурационный файл .cfg, который содержит таблицу идентификаторов и значений констант, объявленных в коде на языке Reflex (в том числе и константы, полученные из перечислений)

Извлеченная таким образом информация упрощает анализ реакций исполняемых модулей при динамической верификации.

Сгенерированный транслятором код на языке Си\C++ передается на вход компилятору языка C\C++ (MinGW). Компилятор генерирует DLL, которая затем может быть загружена в среду LabVIEW комплексами динамической верификации.

Выводы главы

Разработанные в Главе №2 модели и методы динамической верификации процесс-ориентированных программ были реализованы в комплексе автоматизированной динамической верификации и в комплексе автоматической динамической верификации программ на языке Reflex.

В комплексе автоматической верификации программ на языке Reflex описывается проверяемая программа управления и имитатор объекта управления. Комплекс предоставляет оператору возможность проводить динамическую верификацию исследуемой программы на имитаторе объекта управления. Контроль за поведением управляющей программы и управление порядком прохождения тестовых сценариев лежат на операторе.

В комплексе автоматизированной верификации программ были устранены недостатки комплекса автоматизированной верификации: диспетчеризация

тестовых сценариев и верификация поведения управляющей программы проходит автоматически.

Глава 4. Апробация подходов к динамической верификации процесс-ориентированных программ управления КФС, описанных на языке Reflex

4.1. Общие рекомендации к описанию алгоритмических модулей при динамической верификации программ КФС

Для работы в комплексе автоматизированной динамической верификации программа управления описывается без оглядки на наличие имитатора объекта управления. Это значит, что от программиста не требуется прописывать дополнительные интерфейсы взаимодействия Controller с Plant для динамической автоматизированной верификации. Для передачи сообщений между Controller, Plant и GUI, требуется определить идентификаторы входных и выходных сообщений. Идентификаторы сообщений, которые будут отправляться оператору КФС, а также для входных сообщений для Controller от оператора, рекомендуется объявлять в типе ПЕРЕЧИСЛЕНИЕ (ENUM). Для корректной работы комплекса автоматизированной верификации необходимо, чтобы идентификаторы входных сообщений имели одинаковые префиксы. Аналогично для идентификаторов выходных сообщений. Префиксы идентификаторов входных и выходных сообщений должны быть различными. Также это упрощает дальнейшее внедрение кода, сгенерированного из описания на языке Reflex, в целевую КФС.

При описании Plant уже приходится учитывать, что он используется для динамической верификации. Для передачи входных сообщений для Plant также, как и для Controller, требуется задать идентификаторы входных сообщений (через тип ПЕРЕЧИСЛЕНИЕ). Plant может получать сообщения как от Controller, так и от GUI. От Controller передаются данные, имитирующие ЦАП. Выходные же

сообщения Plant делятся на два типа – сообщения, предназначенные для передачи непосредственно ГУИ, и сообщения, предназначенные для передачи в Controller (для имитации АЦП). Они также реализовываются через ПЕРЕЧИСЛЕНИЕ, и идентификаторы выходных сообщений должны иметь одинаковые префиксы. Однако при использовании такого подхода пользователю необходимо вручную на языке Си описать функцию анализа поступающих сообщений и извлечения их них аргументов, чтобы поместить их во внутренние переменные Controller.

Также для имитации ЦАП-АЦП в Plant и Controller пользователь может воспользоваться портами. Однако при таком подходе, при передаче аналоговых данных, в случае преобразования целочисленного значения Reflex-переменной, привязанной к порту требуется прописать Си-библиотеку, преобразующую биты портов в значения с плавающей точкой.

Для проведения автоматизированной динамической верификации, при реализации Plant предлагается определить следующие процессы:

1) Процессы, отвечающие за инициализацию внутреннего состояния объекта управления (к этим процессам относится обязательный процесс Инициализация (Init)), а также процессы, а также за останов объекта (процессы, возвращающие объект к начальному состоянию);

2) Процессы, отвечающие за прием входящих сообщений от Controller и GUI. В этих сообщениях также рекомендуется передавать настроечные параметры для Plant (значения физических характеристик объекта управления). Также рекомендуется передавать через эти сообщения команды на изменение состояние Plant (к примеру, имитацию отказов);

3) Высокоуровневые процессы, отвечающие за мониторинг состояния текущего объекта управления. Также этими процессами задается, в каком состоянии находится объект управления и его части – исправны они или нет;

4) Низкоуровневые процессы, отвечающие за имитацию конкретных элементов объекта управления. По запросу состояний процессов из П.3. эти процессы имеют возможность скорректировать поведение Plant;

Далее приведены рекомендации к описанию алгоритмических модулей на языке Reflex для комплекса автоматической верификации. Для блока Dispatcher рекомендации аналогичны тем, что приведены для автоматизированной верификации.

Блок управления тестовыми сценариями (Dispatcher):

Сообщения:

1) Набор идентификаторов выходных сообщений от Dispatcher для Plant, Verifier и Controller. По идентификатору сообщения Verifier определяет, какой тестовый сценарий сейчас исполняется и, соответственно, какую реакцию ожидать от Controller.

2) Набор идентификаторов входных сообщений для Verifier от Dispatcher. Рекомендуется выделять как минимум два сообщения: команду начала верификации и команду на запуск следующего теста.

Процессы:

1) процесс, отвечающий за прием сообщений от Verifier. При запуске Dispatcher он переводится в активное состояние и слушает очередь входных сообщений от Verifier. При получении команды о запуске верификации он запускает процесс, управляющий переключением тестовых сценариев;

2) процесс, отвечающий за поочередный запуск тестовых сценариев. Процесс по очереди запускает тестовые сценарии, а по завершению тестовых сценариев (срабатыванию таймера тестового сценария) отправляет Verifier сообщение об запуске следующего тестового процесса, и запускает процессы, ответственные за следующий тестовый сценарий;

3) процессы, отвечающие за исполнение тестового сценария. Активируются и останавливаются процессом, отвечающим за запуск тестовых сценариев.

Блок верификации (Verifier):

Входные порты для Verifier – это конкатенация входных и выходных портов Controller. Выходных портов у Verifier нет.

Процессы:

1) процессы, отвечающие за обработку входных сообщений от Dispatcher. При получении сообщения с индексом нового тестового сценария, запускают соответствующие этому сценарию процессы-мониторы и процесс, отвечающий за проверку результатов верификации.

2) процессы-мониторы отвечают за проверку реакций Controller. Мониторы останавливаются только в случае детектирования ошибки и переходят в состояние «ошибка». Процессы-мониторы делятся на два типа:

- процессы, отвечающие за мониторинг инвариантные требования: требования, актуальные для любого тестового сценария. Переходят в активное состояние по запуску Verifier.
- процессы, отвечающие за мониторинг требований, специфичных для текущего тестового сценария

3) Процесс, отвечающий за проверку состояний процессов мониторов. При получении команды от Dispatcher о запуске следующего тестового сценария\останове верификации он проверяет состояние процессов-мониторов. Если монитор находится в состояние «активное» – требование выполнено, «ошибка» – требование не выполнено, «нормальный останов» – требование не проверялось на текущем тестовом сценарии, и формирует отчет для оператора в виде сообщений и переходит в состояние «нормальный останов».

Общий алгоритм цикла динамической верификации:

1. Перед началом верификации требуется привести систему в корректное состояние. Например, Plant и Verifier может требоваться провести некоторую начальную настройку: Plant устанавливает начальные параметры, Verifier запускает процессы, отвечающий за мониторинг очереди сообщений, а также процессы-инварианты. На это может уйти несколько тактов.

2. По завершению своей настойки, Verifier отправляет Dispatcher команду о запуске верификации.

3. Dispatcher получает сообщение о запуске верификации от Verifier и запускает процесс, отвечающий за последовательное исполнение тестовых сценариев.

4. Процесс Dispatcher, отвечающий за последовательное исполнение тестовых сценариев, активирует процессы этого сценария.

5. Запущенные процессы управления тестовым сценарием отправляют в Verifier сообщение, идентификатор которого точно определяет номер исполняющегося сценария. Также для Controller и Plant отправляются сообщения с настроечными параметрами и команды, соответствующие проверяемому тестовому сценарию.

6. Verifier, получив сообщение о запуске тестового сценария, переводит в активное состояние процессы, отвечающие за мониторинг реакций Controller, ожидаемых при текущем тестовом сценарии. Также в активное состояние переводится процесс, отвечающий за проверку состояний мониторящих процессов.

7. Цикл верификации: алгоритмические модули активируются в порядке: Dispatcher, Plant, Controller, а затем Verifier.

8. Если процесс-монитор Verifier детектирует некорректную реакцию алгоритма управления, он переходит в состояние ОШИБКА.

9. По прошествии заданного количества тактов, необходимого для проверки стабильности поведения Dispatcher, отправляет Verifier команду о запуске следующего тестового сценария\останове верификации.

10. В Verifier активируется процесс, отвечающий за проверку мониторов, проводит анализ их состояний на момент завершения верификации. Если процесс-монитор находится в активном состоянии – для оператора формируется сообщение об удачном завершении верификации. Если в состоянии ошибка – сообщение об неудачном завершении верификации. После формирования отчета для оператора, процесс контроля состояний мониторов переводит процессы-мониторы в состояние «нормальный останов». Процессы-инварианты переводятся в активное состояние.

11. После того, как Verifier вычислил результат верификации на текущем тестовом сценарии, он отправляет оператору диагностическое сообщение о результатах верификации, завершает свои активные процессы, которые были связаны с этим тестовым сценарием.

12. Dispatcher запускает следующий тестовый сценарий. Если сценарии закончились, Dispatcher отправляет Verifier сообщение о завершении верификации.

13. Verifier получает сообщение о завершении верификации и формирует диагностическое сообщение пользователю о завершении верификации.

Библиотека работы с входными и выходными очередями сообщений приведены в Приложении А.

4.2. Апробация комплекса автоматизированной динамической верификации на задаче управления вакуумной подсистемой Большого солнечного вакуумного телескопа

Большой солнечный вакуумный телескоп (БСВТ) [104] был создан в 1982 г. для изучения тонкой структуры явлений на поверхности Солнца (Институт солнечно-земной физики СО РАН, пос. Листвянка, Иркутская обл.). На БСВТ установлена оптическая система с пространственным разрешением 0.2". Длина трубы телескопа – сорок метров. Для телескопов такого класса (крупных телескопов) важно избежать тепловой конвекции воздушных потоков в трубе, поскольку они могут привести к оптическим искажениям. Потому на телескопе была установлена подсистема вакуумирования трубы, которая позволяет повысить точность получаемых данных. Однако подсистема вакуумирования телескопа относится к классу систем повышенной надежности и ошибки в управления как в штатном режиме, так и при возможных отказах оборудования могут привести к серьезным последствиям. Например, ранее исследователи были вынуждены прервать свою работу на телескопе на несколько недель после того, на телескопе из-за сбоя электропитания вся точная оптика была загрязнена машинным маслом из насоса.

Изначально телескоп управлялся вручную. Это доставляло неудобства исследователям, потому что им приходилось перемещаться по всему зданию

сорокаметрового телескопа в процессе экспериментов. Кроме того, поскольку на телескопе не было предусмотрено автоматического контроля ошибок ни подсистемы вакуумирования, ни проверки действий оператора, на телескопе постоянно должен был находиться хотя бы один оператор и следить за его состоянием. Это существенно увеличивало эксплуатационные расходы.

Это привело к тому, что была начата разработка комплекса автоматического управления телескопом (2014 г.). Первая версия этой системы была полностью реализована на Qt [105]. Qt обеспечивало кросс-платформенность управляющей программы, и предоставляло широкий набор стандартных библиотек для создания графики, управления памятью и работы с сетевыми интерфейсами. Такой подход показал себя успешным при автоматизации малых телескопов, однако попытка перенести этот опыт на БСВТ привел к ряду трудностей.

При автоматизации малых телескопов объектно-ориентированными средствами программная архитектура системы управления представляла из себя набор взаимодействующих объектов-одиночек (Singleton) – абстрактных драйверов. Каждый из этих объектов предоставлял уникальный интерфейс взаимодействия с элементами автоматизируемой системы. В случае БСВТ подсистема вакуумирования должна обеспечивать круглогодичное бесперебойное функционирование, поэтому на телескопе была установлена водяная система климат-контроля. Это привело к увеличению количества абстрактных драйверов и усложнению связей между ними до такой степени, что было чрезвычайно трудно обеспечить требуемый уровень корректности, верифицируемости, устойчивости и сопровождаемости системы управления телескопом.

Было предложено описать алгоритм управления подсистемой вакуумирования на языке Reflex и интегрировать этот код в систему управления телескопом на Qt.

Подсистема вакуумирования БСВТ

Вакуум в трубе телескопа создается подсистемой вакуумирования БСВТ (рис. 14) (1), поршневым насосом с водяным охлаждением (9). Насос подключается к трубе телескопа через отсечной клапан (4). Отсечной клапан (6) (сапун) открывается при выключении насоса и соединяет насос с атмосферой. В случае аварии электропитания клапан закрывается автоматически (4) с помощью компрессора (2). На насосе установлена водяная рубашка охлаждения (насос 11), в которой система климат-контроля обеспечивает мониторинг уровня воды (датчик 12) нужной температуры (датчик 10). В случае возникновения опасности замерзания вода нагревается нагревателем (14). В рубашке охлаждения вакуумного насоса установлен датчик температуры(7). Вентилятор (8) откачивает масляные пары, когда насос работает. Датчик давления в вакуумируемой трубе (3) отслеживает глубину вакуума в трубе телескопа. Давление в патрубке вакуумного насоса определяет датчик (5).

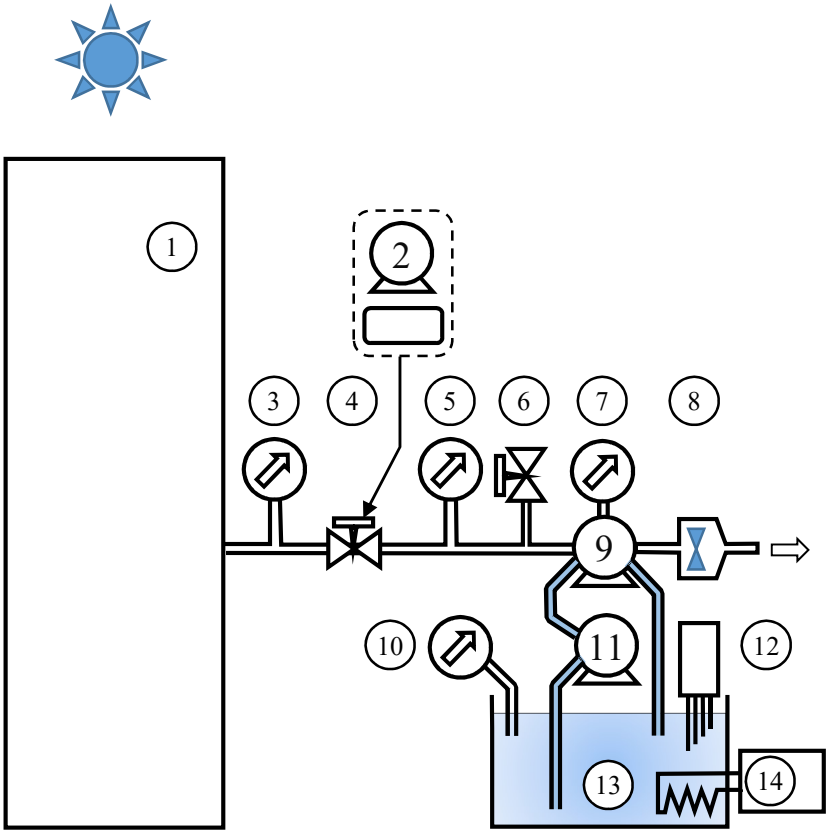


Рис. 14. Подсистема вакуумирования БСВТ: 1 – труба телескопа, 2 – пневмоустройство, 3 – датчик давления в трубе телескопа, 4 – клапан подключения вакуумного насоса к трубе телескопа, 5 – датчик давления в

патрубке вакуумного насоса, 6 – отсечной клапан соединения вакуумного насоса с атмосферой (сапун), 7 – датчик температуры воды в рубашке охлаждения вакуумного насоса, 8 – вентилятор, 9 – вакуумный насос, 10 – датчик температуры воды в системе климат-контроля, 11 – насос охлаждения, 12 – датчики уровня воды в системе климат-контроля, 13 – система климат-контроля, 14 – нагреватель воды в системе климат-контроля.

Программная архитектура автоматизированной системы управления БСВТ

На рис. 2. описана программная архитектура системы управления малыми телескопами, основанная на объектно-ориентированной концепции абстрактных драйверов. При усложнении алгоритма управления, которое требовалось для автоматизации подсистемы вакуумирования БСВТ, эта объектно-ориентированная архитектура дала сбой и привела к серьезной запутанности кода. Попытка внедрить объект-контроллер в управляющую систему привело к проблеме «разбухания» кода контроллера. Реализация контроллера средствами C++ требует большого количества затрачиваемых усилий. В результате код контроллера ненадежен, возникают значительные трудности при его модификации и верификации в силу его чрезвычайной запутанности.

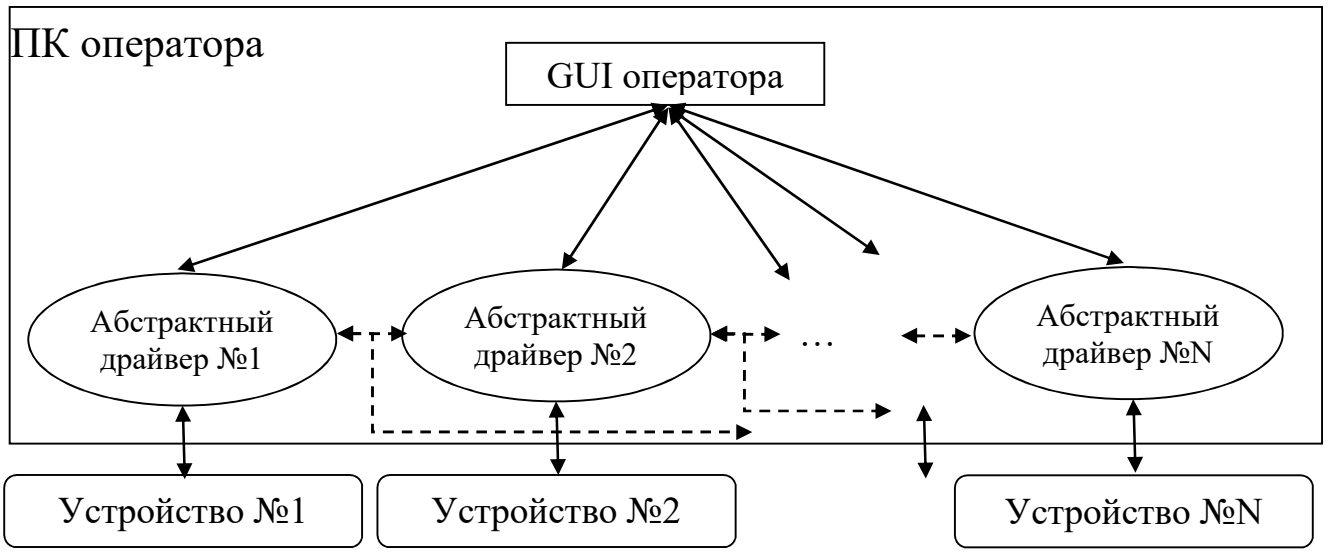


Рис. 15. Архитектура системы управления малыми телескопами

Поэтому было предложено описывать алгоритм управления вакуумной подсистемой телескопа (рис. 16) средствами процесс-ориентированных языков. Код на языке Reflex было решено оформить в событийно-управляемый алгоритмический блок (СУАБ) и встроить в архитектуру подсистемы управления телескопом. Код СУАБ на языке Reflex транслируется автоматически в код на языке Си, а затем встраивается в систему управления на Qt.

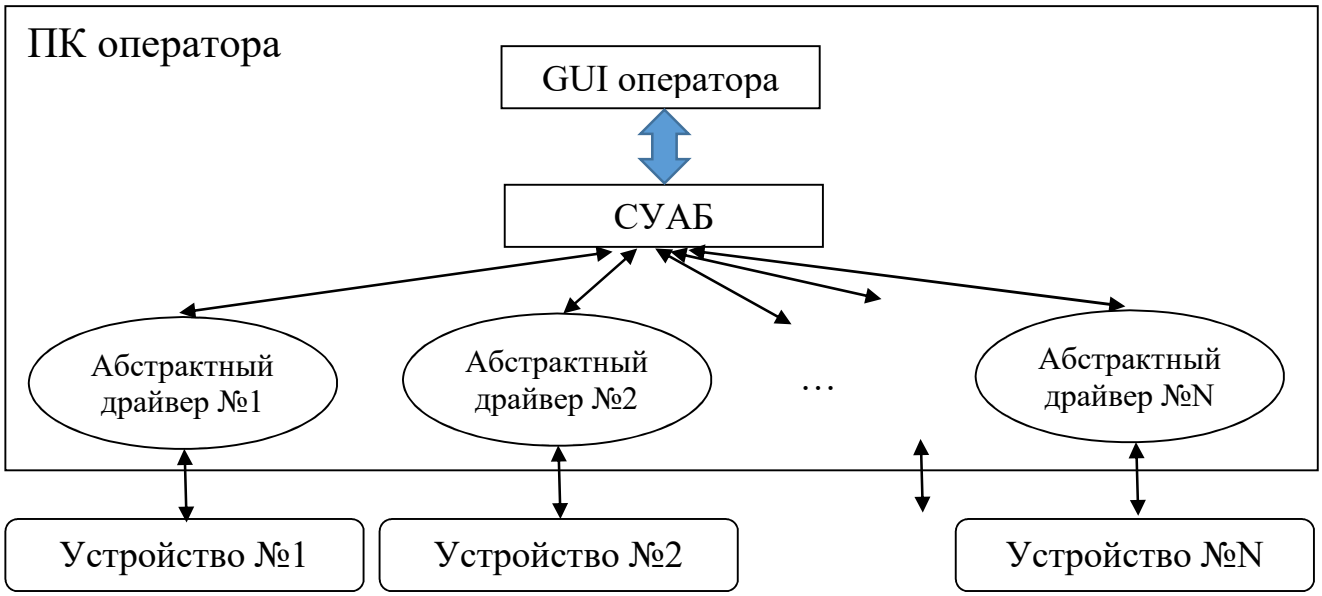


Рис. 16. Программная архитектура системы управления вакуумной подсистемой БСВТ

Интерфейс взаимодействия СУАБ и системы управления БСВТ

Был разработан интерфейс между СУАБ и GUI оператора на Qt, который позволяет:

– записывать и читывать данные во входные и выходные порты СУАБ,
– обмениваться сообщениями с GUI оператора,
– передавать значения глобальных скалярных переменных.

Абстрактные драйвера получают информацию с датчиков, установленных на телескопе, и помещают их во входные порты СУАБ, которые реализованы как статический целочисленный массив. Аналогично были реализованы выходные порты СУАБ – абстрактные драйвера считывают их значения и передают на

управляющие устройства телескопа. Также в портах были зарезервированы служебные биты, которые показывали наличие связи с элементами телескопа. СУАБ считывает записанные во входные порты данные, запускает цикл управления вакуумной подсистемой и заносит управляющие сигналы в выходные порты, которые затем считываются виртуальными драйверами.

Аналоговые сигналы от датчиков давления и температуры записываются в глобальные переменные, к которым СУАБ имеет прямой доступ. Сообщения между СУАБ и GUI передаются через два кольцевых буфера.

Таким образом, для задачи управления вакуумной подсистемой БСВТ были разработаны механизмы бесшовной интеграции кода на языке Reflex в управляющие системы на Qt.

Алгоритм управления вакуумной подсистемой БСВТ

Программа управления вакуумной подсистемой на языке Reflex содержит 97 процессов:

- процесс начальной инициализации, который отвечает за запуск процессов мониторинга и процессов считывания входных сообщений от оператора системы.
- процесс аварийного останова;
- высокоуровневые управляющие процессы (автоматическое вакуумирование, автоматический останов). Эти процессы, если происходит сбой, переводят подсистему вакуумирования в безопасное состояние и перезапускают СУАБ. Управление подсистемой вакуумирования они осуществляют через низкоуровневые процессы;
- низкоуровневые процессы управления, отвечающие за управление элементами вакуумной подсистемы. У оператора есть возможность запускать их вручную;
- процессы мониторинга состояния объекта (сигналов с датчиков), детектирования аварийных ситуаций и связи с УСО. Эти процессы отслеживают состояние объекта управления и повещают оператора в случае неполадок;

– процессы, отвечающие за считывание команд от оператора и отправку выходных сообщений.

Автоматическое вакуммирование может или запускаться автоматически (если давление в трубе упало ниже определенного значения), или вручную оператором.

Алгоритм работы процесса автоматического вакуумирования:

1) включение в системе климат-контроля регулятора уровня воды;
2) проверка наличия воды по датчику уровня воды;
3) запуск регулятора температуры;
4) включение вентилятора и питания насоса охлаждения;
5) проверка, что вакуумный затвор закрыт;
6) открытие сапуна.

Если температура воды в рубашке охлаждения в норме, то включается устройство плавного пуска насоса:

1) пауза чтобы вакуумный насос прогрелся;
2) закрытие сапуна;
3) пауза на откачку воздуха из входного патрубка;
4) датчик давления в патрубке контролирует процесс откачки - открывается клапан между насосом и трубой телескопа.

Если давление в трубе телескопа упало ниже заданного значения, насос отключается и СУАБ переходит в состояние поддержания вакуума. СУАБ проверяет периодически давления в трубе. Если давление растет выше заданного значения – процедура вакуумирования повторяется.

В нештатной ситуации (например, при аварии оборудования) СУАБ отправляет оператору диагностическое сообщение и запускает безопасный останов подсистемы вакуумирования.

Тестирование и отладка кода СУАБ

Предлагаемые в работе методы и модели динамической верификации процесс-ориентированных программ управления были апробированы при отладке алгоритма управления подсистемой вакуумирования БСВТ. Верификация велась с помощью комплекса автоматизированной динамической верификации программ на языке Reflex. Было проверено 33 тестовых сценария и 38 требований (16 требований безопасности, 22 требования прогресса). Отлаженные ситуации:

- сбой питания;
- нештатное открытие вакуумного затвора при сбое питания;
- потеря связи с выносными УСО;
- отказ исполнительных органов;
- самопроизвольное включение / выключение оборудования;
- падение уровня воды в системе охлаждения вакуумного насоса ниже допустимого;
- перегрев и замерзание воды.

Пусконаладочные работы на БСВТ заняли две недели. В процессе работы были обнаружены ошибки в виртуальных драйверах объектов и GUI оператора. После проведенной верификации на комплексе автоматизированной верификации Reflex-модуль управления вакуумной подсистемой не требовал изменений в процессе пусконаладочных работ.

4.3. Апробация комплекса автоматической динамической верификации

В полной версии метод динамической верификации процесс-ориентированных программ управления КФС и реализованный инструментарий автоматической верификации был апробирован при модернизации виртуальных лабораторных стендов. Эти стенды использовались в практикуме по дисциплине «Процесс-

ориентированное программирование», читаемой в магистратуре Факультета информационных технологий Новосибирского государственного университета.

Разработанные методы и модели динамической верификации процесс-ориентированных программ были проверены на алгоритме управления тепловентилятором для сушки свежеокрашенных изделий и алгоритме управления системой контроля уровня воды в резервуаре.

4.3.1. Задача управления тепловентилятором для сушки свежеокрашенных изделий

На тепловентилятор для сушки окрашенных изделий установлен IR-датчик. Датчик детектирует, что изделие помещено в камеру тепловентилятора. Также на тепловентиляторе установлен нагревательный элемент с феном. Программа управления тепловентилятором на вход получает значение датчика наличия изделия и управляет нагревательным элементом. Управляющая программа должна включать нагревательный элемент, если изделие появляется, и автоматически выключать, когда исчезает. Когда изделие находится в камере тепловентилятора, показания датчика не постоянны: изделие вращается и из-за этого датчик временами показывает, что изделия нет, хотя на самом деле оно все еще находится под тепловентилятором.

Если управляющая программа не будет учитывать подобные особенности системы, тепловентилятор будет постоянно переключаться. Кроме того, тепловентилятор не должна работать «вхолостую», т.е. программа должна выключать прибор, когда изделие убирают. У управляющей программы, которая должна включать и выключать тепловентилятор, есть только текущее значение сигнала датчика. Также программа может отслеживать продолжительность, как долго был датчик показывал отсутствие изделия. Если изделие отсутствуют

достаточно долго – тепловентилятор можно выключать. В случае тепловентилятора было предложено выключать тепловентилятор после одной секунды отсутствия изделия. Controller тепловентилятора на языке Reflex представлен в листинге 1.

```
PROGR FanHeaterController {
    PROC Init {

      STATE Waiting {
         IF (I_PRODUCT_ON == ON) {
            O_TURN_ON_HEATER = ON;
            SET NEXT;
         } ELSE {
              O_TURN_ON_HEATER = OFF;
          }
      }
      STATE Drying {
          IF (I_PRODUCT_ON == ON) RESET TIMEOUT;
          TIMEOUT 10 { SET STATE Waiting; }
      }
   }
}
```

Листинг 1. Реализация Controller «Тепловентилятор» на языке Reflex.

Модель «Тепловентилятора» (код Plant) на языке Reflex представлен в Листинге 2. Модель имитирует исчезновение сигнала датчика в диапазоне от 100 мс до 1 с., когда изделие появляются в камере тепловентилятора. Сигнал датчика наличия изделия является для Plant выходным.

```
PROGR Plant {
   PROC InsertProduct {
      STATE Init {
         O_PRODUCT = ON;
         OFF_counter = 1;
         SET NEXT;
```

```
    }
    STATE Off {
      O_PRODUCT = OFF;
      TIMEOUT OFF_counter {
        OFF_counter++;
        SET NEXT;
      }
    }
    STATE On {
      O_PRODUCT = ON;
      IF (OFF_counter >= MAX_LIMIT) SET STATE Init;
      TIMEOUT OFF_counter SET STATE Off;
    }
  }
}
```

Листинг 2. Фрагмент кода Plant «Тепловентилятор» на языке Reflex.

Требования к программе управления тепловентилятором в терминах метрической темпоральной логики [61]:

Таблица 2 – Требования к программе управления тепловентилятором

№	MTL, clock = 100 ms	Пояснение
1	□ (product→ ◊ (1, 2) heater)	При появлении изделия тепловентилятор включается не позднее 0.2 с
2	□ ((¬product ∧ heater) U {10} ¬ heater)	Если изделие убрали, то через 1 с тепловентилятор выключится
3	□ ((¬product ∧ ¬ heater) W (heater ∧ ¬ product))	Тепловентилятор спонтанно не включается

Каждому требованию соответствует отдельный процесс Verifier. Например, требование 1 соответствует процессу HeaterNormalTurnOnControl (Листинг 3). Процесс ожидает появления сигнала от датчика, а затем либо детектирует включение тепловентилятора и перейдет в состояние контроля стабильности управляющего сигнала (проверяя, что управляющая программа не выключит спонтанно тепловентилятор), либо детектирует ошибку в программе, если по прошествии тайм-аута тепловентилятор так и не включился, и перейдет в состояние Error.

Процесс Terminator вычисляет результаты динамической верификации. По завершению тестового сценария, этот процесс опрашивает состояния всех активных процессов, которые отвечали за проверку требований. Если процесс в активном состоянии – значит, требование выполнено, и Terminator отправляет сообщение GUI о том, что требование выполнено. Если процесс находится в состоянии «Error», Terminator посылает сообщение GUI об ошибке верификации.

Когда опрошены все процессы требований, Terminator их останавливает и переходит в состояние нормального останова. Для Verifier входные сигналы – это сигналы от датчика и нагревателя.

```
PROC HeaterNormalTurnOnControl {
  STATE WaitingOfProduct {
        IF(I_PRODUCT_UNDER_DRYER == ON)
              SET STATE WaitingOnLaunch;
        TIMEOUT PlantDelay ERROR;
  }
  STATE DryerError {
        IF (O_PRODUCT_UNDER_DRYER == ON)
        {
              SET NEXT;
        }
        TIMEOUT AlgorithmDelay ERROR;
```

```
    }
     STATE StabilityControl {
          IF (I_PRODUCT_UNDER_DRYER == OFF)
                    SET STATE WaitingOfProduct;
          ELSE IF(O_PRODUCT_UNDER_DRYER == OFF)
                ERROR;
     }
}
```

Листинг 3. Процесс проверки требования на задержку включения тепловентилятора

(фрагмент кода Verifier на языке Reflex)

Код Dispatcher в последовательно отправляет команды Plant (появление и исчезновение изделия) и Verifier (запуск нового теста или останов верификации) теста. После заверения последнего тестового сценария Dispatcher останавливает все свои процессы.

В результате проведённой динамической верификации кода тепловентилятора не было выявлено ограничений на использование разработанного подхода для динамической верификации процесс-ориентированных программ управления КФС.

4.3.2. Задача управления системой контроля уровня в водяном резервуаре

Система контроля уровня в водяном резервуаре является прототипом ряда промышленных управляющих систем [106]. Задача системы – отслеживать уровень воды в резервуаре, управляя втекающим в резервуар потоком воды при различных влияющих факторах. Кроме того, система контроля уровня с одним, двумя или тремя резервуарами часто используется в качестве эталона для диагностики неисправностей и изоляции, а также отказоустойчивого управления.

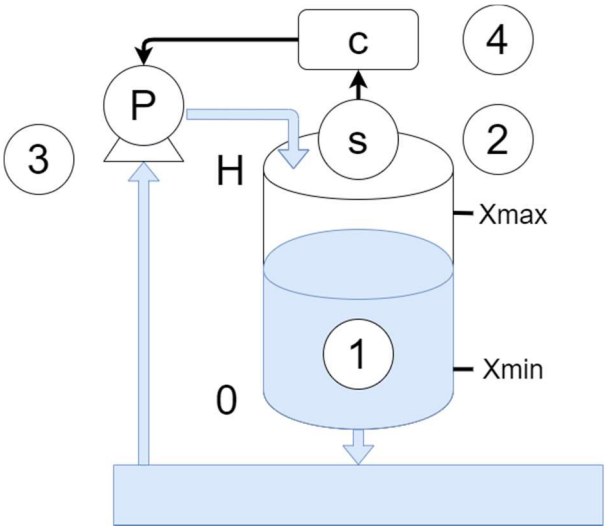


Рис. 17. Схема системы контроля уровня водяного резервуара

Система контроля уровня водяного резервуара представлена на рис. 6. Эта система состоит из резервуара с водой (1), датчика уровня воды (2), насоса (3) и контроллера насоса (4). На дне резервуара имеется сливное отверстие.

Насос отвечает за подачу воды в резервуар, а сливное отверстие отвечает за слив воды из резервуара. Уровень воды в резервуаре переделяется через датчик уровня воды. Контроллер управляет интенсивность заполнения резервуара, изменяя уровень входного напряжения на насосе. Уровень воды должен находиться в определенном диапазоне. Контроллер не должен допускать сильного опустошения или перелива резервуара. От оператора системы контроллер получает сигнал о желаемом уроне воды.

Программа управления на основе ПИ-контроллера задает уровень напряжения в насосе в зависимости от показаний датчика уровня воды. Основные требование к программе управления:

1. Не допускать перелива воды;

2. Не допускать понижения уровня воды ниже определенной отметки;

3. Уровень управляющего напряжения должен находиться в определенном диапазоне;

Не допускать резких скачков величины управляющего сигнала, что может привести к преждевременному выходу оборудования из строя.

Код программы управления системы контроля уровня водяного резервуара на языке Reflex представлен на листинге 4.

В коде программы управления описан ПИ-регулятор и вычисляется новое значение управляющего напряжения насоса, которое зависит от того, как сильно реальный уровень воды отличается от требуемого уровня. При этом в случае перелива управляющий процесс переходит в состояние Owerflow и управляющее напряжение становится равным нулю. В случае падения уровня воды ниже установленной отметки, насос включается на максимальную мощность.

```
/*Tank PI controller*/
/*X – уровень воды*/
/*XD – требуемый уровень воды */
/*U – управляющий сигнал*/
/*U_MAX – максимальное значение управляющего сигнала */
PROC CONTROLLER {
    STATE Init {
        Sum = 0.0;
        E = 0.0;
        SET NEXT;
    }
    STATE NormalPIControl {
        E = XD - X;
        Sum += E;
        U = K_P * E + K_I * Sum * T_H;
```

```
            IF (U > U_MAX)
            { U = U_MAX; }
            IF( U < 0 ) U = 0.0;
            IF (X >= X_MAX) { SET STATE Owerflow; }
            ELSE IF ( X < X_MIN) { SET STATE LowLevel;       }
        }
        STATE Owerflow {
            /* Слишком высокий уровень воды. Останов насоса */
            U = 0;
            IF (X < X_MAX) SET STATE Init;
        }
        STATE LowLevel {
        /* Слишком низкий уровень воды. Макс. мощность насоса */
            U = U_MAX;
            IF (X > X_MIN) SET STATE Init;
        }
}
```

Листинг 4. Код программы управления системы контроля уровня водяного резервуара (фрагмент кода Controller на языке Reflex)

Не приведены в листинге, но присутствуют в результирующем коде программы, процессы:

Процесс Filter, который сохраняет значения управляющего напряжения на предыдущем шаге алгоритма и не допускает резких скачков управляющего напряжения.

Процесс ReadInputMsgs, который обрабатывает очередь входных сообщений для АУ и извлекает из сообщений значения требуемого уровня воды.

Значения управляющего напряжения и значения уровня воды передаются в алгоритм через механизм портов. В коде Controller определены два процесса,

которые преобразуют на каждом тексте целочисленные значения, полученные из портов, в значения типа FLOAT обратно.

Модель объекта управления «Системы управления уровнем воды в баке» (код Plant), специфицированная на языке Reflex, представлена в Листинге 5.

```
PROC TANK {
FROM  PROC INIT X, U;
   STATE Normal {
              X += (-GAMMA * SQRT(X)  * T_H
              + BETA * U  *          T_H )/SIMULATOR_DELAY ;
              IF (X >= H) SET STATE Owerflow;
              IF ( X < X_MIN ) SET STATE Underfill;
}
    STATE Owerflow{
              X = H;
              X += (-GAMMA * T_H * SQRT(X)
                   + BETA * T_H * U) / SIMULATOR_DELAY;
               IF (X < H) SET STATE Normal;
}
  STATE Underfill{
           X += ( -GAMMA * X * SQRT(X)  * T_H  / X_MIN  + BETA * U *
T_H ) / SIMULATOR_DELAY ;
              IF (X < 0) X = 0;
              IF ( X > X_MIN ) SET STATE Normal;
}
}
```

Листинг 5. Модель объекта управления «Системы управления уровнем воды в баке» (код Plant на языке Reflex)

Процесс, моделирующий изменение уровня воды, содержит три состояния: состояние перелива, состояние падения уровня воды ниже максимальной отметки,

и состояние, когда уровень воды находится в норме. В случае нормального уровня воды, вычисляется численное решение дифференциального уравнения для определения уровня воды в зависимости от поданного значения управляющего напряжения. В случае перелива имитируется то, что максимальный уровень воды будет равен высоте резервуара. В случае падения высоты водяного столба ниже минимального значения имитируется, что скорость вытекания также изменится. Во всех состояниях имитируется инертность объекта управления.

В Dispatcher задается три сценария работы:

1. Требуемый уровень воды устанавливается в среднее значение;
2. Требуемый уровень воды установлен на нулевую отметку;
3. Требуемый уровень воды установлен на высоту всего бака.

```
PROC TestNormalDesiredLevelControl{
    STATE InitialSetup    {
        XD = (X_MIN+X_MAX)/2;
        SendMsgFloatParamToControlAlgorythm(COM2CA_XD, XD);
    SendMsgFloatParamToVerificationModule(SCM2VM_StartTestNewDesiredLe
vel, XD);
        SET NEXT;
    }
    STATE Idiling    {
        LOOP;
    }
}
PROC TestMinimumDesiredLevelControl{
    STATE InitialSetup    {
        SendMsgFloatParamToControlAlgorythm(COM2CA_XD, 0);
    SendMsgFloatParamToVerificationModule(SCM2VM_StartTestNewDesiredLe
vel, 0);
        SET NEXT;
```

```
        }
        STATE Idiling     {
                LOOP;
        }
}
PROC TestMaximumDesiredLevelControl{
        STATE InitialSetup
        {
                SendMsgFloatParamToControlAlgorythm(COM2CA_XD, H);
                SendMsgFloatParamToVerificationModule
                                (SCM2VM_StartTestNewDesiredLevel, H);
                SET NEXT;
        }
        STATE Idiling     {
                LOOP;
        }
}
```

Листинг 6: Код Dispatcher «Системы управления уровнем воды в баке»

В Verifier (Листинг № 7) отслеживаемые требования делятся на два типа:

Требования к программе управления тепловентилятором.

Инварианты – требования, которые проверяются при любом тестовом сценарии. В случае системы контроля уровня воды эти требования формулируются следующим образом:

1. Уровень воды всегда остается в заданном диапазоне.
2. Управляющее напряжение всегда остается в заданном диапазоне.
3. Управляющее напряжение всегда изменяется плавно.

Требование «живости» (Liveness), которое формулируется как «значение уровня воды всегда стремится к значению, заданному оператором системы». В

формулах темпоральной логики требования к алгоритму управления описываются в Таблице 3:

Таблица 3 – Требования к программе системы управления уровнем воды

№	MTL, clock = 100 ms	Пояснение
1	□ (X > 0 ∧ X < H)	АУ не должен опускать падения уровня воды до нулевой отметки, а также перелива
2	□ (U >= 0 ∧ U <= Umax)	АУ не должен удерживать управляющее напряжение в диапазаоне [0, Umax]
3	□ (|U_Prev – U| >= DELTA_U)	Разница между значением управляющего напряжения на предыдущем шаге (U_Prev) и значением на текущем шаге не должна превышать DELTA_U
4	□ (|X – XD| > DELTA → ◊((1, N)(□(|X – XD| <= DELTA)))	Уровень воды стремится к заданному уровню

```
PROC XValueRestrictionsInvariant{
FROM PROC INIT X;
    STATE Normal {  IF ( X > H || X < 0 ) ERROR;   }
}
PROC UValueRestrictionsInvariant{
    STATE Normal    {IF ( U > U_MAX || U < 0)  ERROR; }
}
PROC VoltageDerivativeRestrictionsInvariant{
    STATE Setup      { U_Prev = U;  SET NEXT;    }
    STATE Normal    {
        IF ( ABS(U_Prev - U) >= DELTA_U)  ERROR;
        U_Prev = U;
    }
```

```
}
PROC XLevelTendsToDesired{
        STATE Idiling      {
                TIMEOUT IDILING_DELAY SET NEXT;
        }
        STATE LevelControl     {
                        IF (ABS(X - XD) > DELTA_X) ERROR;
                        TIMEOUT CONTROLLER_DELAY SET NEXT;
        }
        STATE Progress   { LOOP; }
}
```

Листинг 7: Код Dispatcher «Системы управления уровнем воды в баке»

Как и в случае задачи управления тепловентилятором, выделенный процесс GUI определяет, была ли верификация успешна или же нет, и оповещает GUI о результатах.

Выводы главы

Предложенные методы и модели к динамической верификации процесс-ориентированных программ управления КФС был апробированы в ряде задач. Получено два акта о внедрении полученных в работе результатов и два свидетельства о регистрации программ для ЭВМ (Приложение Б.).

Апробация предлагаемого подхода динамической верификации процесс-ориентированных алгоритмов управления КФС была проведена на следующих задачах:

1. Программа управления подсистемой вакуумирования Большого солнечного вакуумного телескопа БСВТ.

2. Тестовая задача автоматизации тепловентилятора для свежеокрашенных изделий.

3. Тестовая задача автоматизации системы управления уровнем воды в баке.

В результате проведённой апробации не было выявлено ограничений на использование разработанного подхода для динамической верификации процесс-ориентированных программ управления КФС. Заказчики системы управления вакуумной подсистемой БСВТ отметили сокращение времени пусконаладочных работ за счет бесперебойной работы подсистемы вакуумирования.

Заключение

В диссертации получены следующие основные научные результаты:

1. Предложена четырехкомпонентная формальная модель динамической верификации процесс-ориентированных программ управления КФС на имитаторе объекта управления.

2. Предложен численный метод динамической верификации процесс-ориентированных программ управления КФС.

3. Разработаны программные решения, использованные при проектировании комплексов автоматизированной и автоматической динамической верификации программ на языке Reflex.

4. Подход апробирован на задаче верификации вакуумной подсистемой Большого солнечного телескопа (БСВТ). Исследованы существующие подходы к верификации процесс-ориентированных программ управления КФС.

Разработанные методы динамической верификации могут использоваться при создании процесс-ориентированных программ управления КФС, а также для подготовки специалистов в области разработки КФС.

В заключение автор выражает благодарность и большую признательность научному руководителю Зюбину В. Е. за поддержку, помощь, обсуждение результатов и научное руководство. Также автор благодарит коллектив Лаборатории №19 и Лаборатории №16, особенно: Ануреева И. С. за обсуждение диссертации, Гаранину Н. О. как соавтора работ, Кугаевских А. В. и Розова А. С. за мотивацию. Автор выражает благодарность Лубкову А. А., Петухову А. Д., Будникову К. И., Котову В. Н., Курочкину А. В. и заведующему Лабораторией №16 Кирьянову А. В.. Кроме того, автор благодарит Лабораторию экспериментальной физики Солнца и астрофизического приборостроения Института солнечно-земной физики СО РАН за помощь в проведении экспериментов на Большом солнечном вакуумном телескопе.

Список сокращений и условных обозначений

КФС киберфизические системы

АУ алгоритм управления

ВОУ виртуальный объект управления

ПО программное обеспечение

MDD model-driven development

MBD model-based design

TDD test-driven developnet

MBT model-based testing

БСВТ Большой солнечный вакуумный телескоп

DSL domain specific language

ТС тестируемая система

MIL model-in-the-loop

SIL software-in-the-loop

PIL process-in-the-loop

HIL hardware-in-the-loop

RV runtime verification

MCPS модель киберфизической системы

TCPS Модель настраиваемой киберфизической системы (tuned CPS)

DV модель динамической верификации КФС

AR множество требований к верифицируемому ПО

A множество весов требований

TS множество тестовых сценариев

Qm	вектор результатов исполнения тестовых сценариев
res(ts)	функция, вычисляющая компоненту вектора Qm
СУАБ	событийно-управляемый алгоритмический блок

Список литературы

1. .Platzer A. Logical Foundations of Cyber-Physical Systems. – Switzerland, Cham : Springer, 2018. – С. 639.
2. Nunes D. A Practical Introduction to Human-in-the-Loop Cyber-Physical Systems / D. Nunes, J. Sa Silva, F. Boavida. – UK, Chichester : John Wiley & Sons Ltd, 2018. – С. 320.
3. Закревский А. Д. Параллельные алгоритмы логического управления. – М. Эдиториал УРСС, 2003. – С. 200.
4. Modeling software with finite state machines: a practical approach / F. Wagner [и др.]. – US, Florida : Auerbach Publications, 2006. – С. 390.
5. Harel D. Statecharts: A visual formalism for complex systems // Science of computer programming. – 1987. – Т. 8, № 3, – С. 231-274.
6. Zyubin V. E. Hyper-automaton: A Model of Control Algorithms // Материалы международной научной конференции IEEE International Siberian Conference on Control and Communications (SIBCON-2007). – Россия, Томск IEEE, 2007. – С. 51–57.
7. Barraquand J. Robot Motion Planning: A Distributed Representation Approach. / J. Barraquand, J. C. Latombe // Claude International Journal of Robotic Research. – 1991. – Т. 10, № 6. – С. 628-649.
8. Blume C. PasRo: Pascal for Robots / C. Blume, J. Wilfried. – Berlin, Heidelberg Springer Science & Business Media, 2012. – С. 239.
9. Supplemental Guide for ROBOTC Programming [Электронный ресурс]. – Режим доступа: http://engineering.nyu.edu/gk12/amps-cbri/pdf/RobotC%20FTC%20Books/ROBOTC_Training_Guide.pdf, свободный. – Загл. с экрана.

10. Testing embedded software: A survey of the literature / V. Garousi, M. Felderer, M. Felderer, C. M. Karapicak, U. Yılmaz // Information and Software Technology. – 2018. – Т. 104, № 12. – С. 14-45.
11. Lee. E. A. The Past, Present and Future of Cyber-Physical Systems: A Focus on Models // Sensors. – 2015. – Т. 15, №3. – С. 4837-4869.
12. Stroustrup B. The C++ Programming Language. – United States, Boston : Addison-Wesley Professional, 2013. – С. 1360.
13. Barr M. Programming embedded systems in C and C++. – Newton, Massachusetts, United States : O'Reilly Media, 1999. – С. 200.
14. Lee. E. A. Cyber Physical Systems: Design Challenges // Материалы международной научной конференции Object Oriented Real-Time Distributed Computing (ISORC). – Orlando, Florida, USA : IEEE Xplore, 2008. – С. 363-369.
15. The Java Language Specification [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/javase/specs/jls/se12/html/index.html, свободный. – Загл. с экрана.
16. Robot Operating System (ROS) / под ред. A. Koubaa. – Switzerland, Cham: Springer International Publishing – С. 728.
17. Meyer B. Eiffel: The Language. – United States, New Jersey: Prentice Hall, 1991. – С. 300.
18. Programming Embedded Systems in Functional Languages [Электронный ресурс]. – Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.163.4146&rep=rep1&type=pdf, свободный. – Загл. с экрана.
19. Development of a cyber-physical system for mobile robot control using Erlang / S. Szomiński [и др.]. // Proceedings of Federated Conference on Computer Science and Information Systems. – 2013. – Poland, Krakow: IEEE. – С. 1441-1448.

20. Aronsson M. Hardware Software Co-Design in Haskell / M. Aronsson, M. Sheeran // Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. – 2017. UK, Oxford : ACM. – С. 162–173.
21. Murphy J. C. Real-time capabilities in functional languages / B. Shivkumar, L. Ziarek // Proceedings of CPSWeek Workshop on Declarative Cyber-Physical Systems (DCPS). – 2016. – Austria, Vienna : IEEE. – С. 1-10.
22. Fowler M. Domain specific languages. – United States, Boston: Addison-Wesley Professional, 2010. – С. 640.
23. Kring J. LabVIEW for Everyone. – United States, New Jersey : Prentice Hall, 2006. – С. 1032.
24. Пароннджанов В. Д. Графический синтаксис языка ДРАКОН // Программирование. – 1995. – Т. 3. – С. 45-62.
25. Tyagi A. K. MATLAB and Simulink for Engineers. – United Kingdom, Oxford : Oxford University Press, 2012. – С. 492.
26. IEC 61131-3. Programmable controllers. Part 3: Programming languages, 2nd Ed. // IEC 65B/373/CD, International Electrotechnic Commission. – 1998.
27. Basile F. On the implementation of industrial automation systems based on PLC / P. Chiacchio, D. Gerbasio // Trans. on automation science and engineering. – 2013. – Т. 10. № 4. С. 990-1003.
28. Zyubin V. E. Programming of PLC: languages IEC 61131-3 and possible alternatives // Industrial control systems and controllers. – 2005. № 11. – С. 31–35.
29. Zyubin V. E. To the fifth anniversary of the IEC 1131-3 standard. Results and forecasts // Devices and systems. Management, control, diagnostics. – 1999. – Т. 1. – С. 64–71.
30. 5 языков программирования стандарта МЭК 6-1131/3 [Электронный ресурс]. – Режим доступа: http://www.adastra.ru/products/overview/IEC61131/, свободный. – Загл. с экрана.

31. Thramboulidis K. Towards an Object-Oriented Extension for IEC 61131 // Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012). – 2012. – Poland, Krakow: IEEE. – С. 1-8.
32. Samek M. State oriented programming / M. Samek, P. Montgomery // Embedded Systems Programming. – 2000. – Т. 13, № 8. – С. 22-43.
33. Function Blocks for Industrial-Process Measurement and Control Systems: IEC 61499 Introduction and Run-time Platforms / J. L. M. Lastra [и др.]. – Tampere University of Technology. Institute of Production Engineering, 2004.
34. Control Technology Corporation. QuickBuilder Reference Guide [Электронный ресурс]. – Режим доступа: https://controltechnologycorp.com/docs/QuickBuilder_Ref.pdf, свободный. – Загл. с экрана.
35. Liakh T.V., Rozov A.S., Zyubin V.E. Reflex Language: a Practical Notation for Cyber-Physical Systems // System Informatics. – 2018. № 12. – С. 85–104.
36. Зюбин В. Е. Анализ алгоритмов измерения диаметра выращиваемого кристалла кремния / А. С. Розов, В. Е. Зюбин // Материалы международной научно-практической конференции «Металлургический кремний-2012. Физико-химические процессы и технологии получения металлургического кремния». – 2012. – Казахстан. Караганда. – С. 103-104.
37. Базовый модуль, управляющий установкой для выращивания монокристаллов кремния / В. Е. Зюбин [и др.]. // Датчики и системы. – 2004, №12. – С.17-21.
38. Розов А. С. Адаптация процесс-ориентированного подхода к разработке встраиваемых микроконтроллерных систем / А.С. Розов, В.Е. Зюбин // Автометрия. – 2019. – Т. 55, № 2. – С. 114-122.
39. Zyubin V. E. Hyper-automaton: A Model of Control Algorithms // Proceedings of IEEE International Siberian Conference on Control and Communications (SIBCON-2007). – 2007. – Россия, Томск: IEEE. – С. 51–57.

40. Шалыто А. А. SWITCH технология — автоматный подход к созданию программного обеспечения "реактивных" систем / А. А. Шалыто, Н. И. Туккель // Программирование. – 2001. № 5. – С.45-62.
41. Zyubin V. Using Process-Oriented Programming in LabVIEW // Proceedings of the Second IASTED International Multi-Conference on Automation, control, and information technology: Control, Diagnostics, and Automation. – 2010. – Россия, Новосибирск : IASTED. – С. 35–41.
42. Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems [Электронный ресурс]. – Режим доступа: https://www.nrc.gov/docs/ML0634/ML063470583.pdf , свободный. – Загл. с экрана.
43. Systems and software engineering – Vocabulary. – ISO, 2017. – С. 522.
44. An Ontology of Specification Patterns for Verification of Concurrent Systems / N. O. Garanina [и др.]. // Proceedings of the 17th International Conference SoMeT-18. Series: Frontiers in Artificial Intelligence and Applications. – 2018. – Amsterdam: IOS Press – Т. 303. – С. 515–528.
45. Shilov N. V. Combined logics of knowledge, time, and actions for reasoning about multi-agent systems. Knowledge processing and data analysis / N. V. Shilov, N. O. Garanina // Lecture Notes in Computer Science. – 2011. – Т. 6581. – С. 48–58.
46. Шелехов В. И. Верификация и синтез программ сложения на базе правил корректности операторов // Моделирование и анализ информационных систем. – 2010. – Т. 17, № 4. – С. 101–110.
47. Clarke E. M. Model checking hybrid systems / E. M. Clarke, S. Gao // International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Lecture Notes in Computer Science. – 2014. – Т. 8803. – С. 385–386
48. Towards formal verification for cyber-physically agnostic software: A case study / D. Drozdov [и др.]. // Proceedings of the 43rd Annual Conference of the

IEEE Industrial Electronics Society (IECON 2017). – 2017. Beijing, China: IEEE. – С. 5509-5514.

49. Abran A., The Guide to the Software Engineering Body of Knowledge (SWEBOK Guide) / A. Abran, J. W. Moore. – IEEE Computer Society, 2004. – С. 206.
50. Кулямин В. В. Методы верификация программного обеспечения. – Москва : Конкурс обзорно-аналитических статей по направлению «Информационно-телекоммуникационные системы», 2008. – С. 111.
51. Kaner C. Testing Computer Software Book / C. Kaner, H. Q. Nguyen, J. Falk. – UK, Chichester : John Wiley & Sons Ltd, 1999. – С. 480.
52. PVS-Studio Analyzer [Электронный ресурс]. – Режим доступа: https://www.viva64.com/en/pvs-studio/ , свободный. – Загл. с экрана.
53. Extra Clang Tools 12 documentation [Электронный ресурс]. – Режим доступа: https://clang.llvm.org/extra/clang-tidy/ , свободный. – Загл. с экрана.
54. COVERITY [Электронный ресурс]. – Режим доступа: https://scan.coverity.com/ , свободный. – Загл. с экрана.
55. Emanuelsson P. A Comparative Study of Industrial Static Analysis Tools / P. Emanuelsson, U. Nilsson. – Linköping, Sweden : Linkoping Electronic Press, 2008. – С. 34.
56. Kumar M. Formal verification: an essential toolkit for modern VLSI design / M. Kumar, T. Schubert, E. Seligman. – United States, Burlington : Morgan Kaufmann, 2015. – С. 408.
57. Randell B. Software Engineering Techniques // Report on a conference sponsored by the NATO Science Committee. – 1970. – Rome, Italy : Scientific Affairs Division, NATO. – С. 16.
58. Карпов Ю. Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. – СПб.: БХВ-Петербург, 2010. – С. 560.

59. Clarke E. M. Handbook of Model Checking. - Switzerland, Cham : Springer International Publishing, 2018. – C. 1210.
60. Kripke. S. Semantical Considerations on Modal Logic // Acta Philosophica Fennica. – 1963. – T. 16: - C. 83-94.
61. Temporal and Modal Logic / E.A. Emerson // Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. – 1900. – United States, Cambridge : MIT Press. – C. 995–1072.
62. Aravantinos V. Linear Temporal Logic and Propositional Schemata, Back and Forth / V. Aravantinos, R. Caferra, N. Peltier. // Proceedings of the eighteenth International Symposium on Temporal Representation and Reasoning, Lubeck. – 2011. – Lubeck, Germany: IEEE. – C. 80-87.
63. Biere1 A. Bounded Model Checking / A. Biere1, A. Cimatti, E. M. Clarke. // Advances in Computers, Academic Press. – 2003. – T. 58.
64. Holzmann J. The Model Checker SPIN // IEEE transactions on software engineering. – 1997. – T. 23, №. 5. – C. 279-295.
65. Larsen, K., Pettersson, P. & Yi, W. Uppaal in a nutshell / K. Larsen, P. Pettersson, W. Yi // International Journal on Software Tools for Technology Transfer. – 1997. – T. 1. – C. 134–152.
66. Zhang P. A classification and comparison of model checking software architecture techniques / P. Zhang, H. Muccini, B. Li // Journal of Systems and Software. – 2010. – T. 83, №5. – C. 723-744.
67. Baier C. Principles of Model Checking / C. Baier, J. P. Katoen. Principles of Model Checking. – Cambridge: The MIT Press, 2008. – C.975.
68. Two-Step Deductive Verification of Control Software Using Reflex / I. Anureev [и др.]. // LNCS. – 2020. – T. 11964. – C. 50-63.
69. C. A. R. Hoare. An axiomatic basis for computer programming // Communications of the ACM. – 1969. – T. 12, № 10. – C. 576–580.
70. A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip // Electronics. – 2018. – T.7, №6. – C. 1-27.

71. J. Rushby. Theorem Proving for Verification // LNCS. – 2001. – Т. 2067. – С. 39-57.
72. Beckert B. Deductive Verification of System Software in the Verisoft XT Project / B. Beckert, M. Moskal // Künstliche Intelligenz. – 2010. – Т. 24, № 1. – С. 57-61.
73. The KeY project Homepage, [Электронный ресурс]. – Режим доступа: https://www.key-project.org, свободный. – Загл. с экрана.
74. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems, [Электронный ресурс]. – Режим доступа: http://symbolaris.com/info/KeYmaera.html, свободный. – Загл. с экрана.
75. Synthesizing Distributed Systems Orna Kupferman // Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. – 2001. – Boston, USA : IEEE. – Т. 1. – С. 389.
76. David C. Program synthesis: challenges and opportunities / C. David, K. Daniel // Philosophical transactions. Series A, Mathematical, physical, and engineering sciences. – 2017. – Т. 375, № 2104.
77. Зюбин В. Е. Итерационная разработка управляющих алгоритмов на основе имитационного моделирования объекта управления // Автоматизация в промышленности. – 2010. № 11. – С. 43-48.
78. C. Berger. Accelerating regression testing for scaled self-driving cars with lightweight virtualization – a case Study // Proceedings IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems. – 2015. Florence : IEEE. – С. 2-7.
79. Kane A. Monitor based oracles for cyber-physical system testing: practical experience report / A. Kane, T. Fuhrman, P. Koopman // Proceedings 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. – 2014. Atlanta: IEEE. – С. 148-155.
80. Holberg H. Significant quality and performance gains through fully automated back-to-back testing / H. Holberg, Dr. U. Brockmeyer. [Электронный ресурс].

– Режим доступа: https://www.btc-es.de/assets/files/whitepapers/significant-quality-and-performance-gains-through.pdf, свободный. – Загл. с экрана.

81. Osherove R. The Art of Unit Testing. – New York, United States : Manning Publications, 2013. – С. 296.
82. Broekman B. Testing Embedded Software / B. Broekman, Notenboom E.– United States, Boston : Addison-Wesley Professional, 2002. – С. 368.
83. Abbaspour Asadollah S. A survey on testing for cyber physical system / S. Abbaspour Asadollah, R. Inam, H. Hansson // Proceedings of ICTSS: IFIP International Conference on Testing Software and Systems. – 2015. – Switzerland, Cham : Springer. – Т. 9447. – С. 194-207.
84. Bringmann E. Model-Based Testing of Automotive Systems / E. Bringmann, A. Krämer // Proceedings of 2008 International Conference on Software Testing, Verification, and Validation. International Conference on Software Testing, Verification, and Validation (ICST). – 2008. – Lillehammer, Norway: IEEE. – С. 485-493.
85. Деменков Н. П. Модельно-ориентированное проектирование систем управления // Промышленные АСУ и контроллеры. – 2008. № 11. – С. 66-69.
86. Model Based Testing - An Introduction to Model-Based Testing and Spec Explorer. [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/december/model-based-testing-an-introduction-to-model-based-testing-and-spec-explorer, свободный. – Загл. с экрана.
87. Ibrahim K. El-Far. Enjoying the perks of model-based testing // Proceedings of the Software Testing, Analysis, and Review Conference (STARWEST 2001). – 2001.
88. Allen J. Managing Data and the Testing Process in the MBD Environment // SAE Technical Paper. - 2014.

89. Isermann R. Hardware-in-the-loop simulation for the design and testing of engine-control systems / R. Isermann, J. Schaffnit, S. Sinsel // Control Engineering Practice. – 1999. – Т. 7. – С. 643-653.
90. Implementation of system level control and communications in a Hardware-in-the-Loop microgrid testbed / B. Xiao [и др.]. // Proceedings of 2016 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT). – 2016. Minneapolis, USA : IEEE. – С. 1-5.
91. Опыт использования UniTESK как зеркало развития технологий тестирования на основе моделей / В. П. Иванников [и др.]. // Труды ИСП РАН. – 2013. – С. 207-218.
92. SimInTech software for programming control-system devices / F. I. Baum [и др.]. // At Energy. – 2013. №113. – С.443–446.
93. Real-Time Simulation of a Complete Electric Vehicle Based on NI VeriStand Integration Platform / S. Ciornei [и др.]. // Proceedings of 2018 International Conference and Exposition on Electrical And Power Engineering (EPE). – 2018. – С. 107-112.
94. Introduction to Runtime Verification / E. Bartocci [и др.]. // Lectures on Runtime Verification. Lecture Notes in Computer Science. – 2018. – Т. 10457. – С. 1-33.
95. Wagner F. Misunderstandings about state machines / F. Wagner, P. Wolstenholme // Computing & Control Engineering Journal. – 2004. – Т. 15, № 4. – С. 40-45.
96. Mikk E. Hierarchical automata as model for statecharts // Advances in Computing Science (ASIAN'97). – 1997. – Т. 1345. – С. 181-196.
97. Alur R., Dill D. L. A Theory of Timed Automata / R. Alur, D. L. Dill // Theoretical Computer Science. – 1994. – Т. 126. – С. 183–235.
98. Анисимов Н. А. Композиционный подход к разработке параллельных и распределенных систем на основе сетей Петри / Анисимов Н. А., Голенков Е. А. Харитонов Д. И. // Программирование. – 2001. № 6. – С. 30–43.

99. Wagner F. Going beyond the limitations of IEC 61131-3 // StateWORKS. – 2005. [Электронный ресурс]. – Режим доступа: https://www.stateworks.com/active/download/TN11-Going-Beyond-Limitations-Of-IEC-61131.pdf , свободный. – Загл. с экрана.
100. TDDM4IoTS: A Test-Driven Development Methodology for Internet of Things (IoT)-Based Systems // Applied Technologies. ICAT 2019. Communications in Computer and Information Science. – 2019. – Т. 1193. – С. 41-55.
101. Beydeda S. Model-Driven Software Development / S. Beydeda, M. Book, G. Volker. – Berlin, Heidelberg : Springer-Verlag, 2005. – С. 464.
102. Зюбин В. Е. Виртуальные лабораторные стенды: обучение программированию задач промышленной автоматизации / В. Е. Зюбин, А. А. Калугин // Промышленные АСУ и контроллеры. – 2009. № 2. – С. 39 – 44.
103. DLL, [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/windows/win32/dlls/dynamic-link-libraries?redirectedfrom=MSDN , свободный. – Загл. с экрана.
104. Skomorovsky V.I. The large solar vacuum telescope: The optical system, and first spectral observations / V.I. Skomorovsky, N.M. Firstova // Solar Physics. – 1996. – Т.163. – С. 209–222.
105. Саммерфилд М. Qt. Профессиональное программирование. Разработка кросс-платформенных приложений на C++. – М. : Символ-Плюс, 2011. – С. 560.
106. A. Kroll and H. Schulte, Benchmark problems for nonlinear systemidentification and control using soft computing methods: Need andoverview / A. Kroll and H. Schulte// Applied Soft Computing. – 2014. – Т. 25, № 12. – С. 496–513.

Приложение А. Библиотека работы с входными и выходными очередями сообщений комплексов динамической верификации программ на языке Reflex

Для комплекса автоматизированной верификации

Для работы с очередями входных\выходных сообщений была разработана библиотека, функции которой вызываются из кода на языке Reflex:

Для входной очереди сообщений:

ФУНКЦИЯ ЦЕЛ GetNextMsgGUI(ПУСТО) – извлечение следующего сообщения из очереди. При этом происходит сдвиг указателя очереди.

ФУНКЦИЯ ЦЕЛ GetMsgGUICode(ПУСТО) – функция возвращает код последнего извлеченного сообщения.

ФУНКЦИЯ ЦЕЛ GetIntParamGUI(ПУСТО);

ФУНКЦИЯ КЦЕЛ GetShortParamGUI(ПУСТО);

ФУНКЦИЯ ДЦЕЛ GetLongParamGUI(ПУСТО);

ФУНКЦИЯ ПЛАВ GetFloatParamGUI (ПУСТО);

Функции, возвращающие аргумент последнего извлеченного из очереди сообщения и преобразующие его, соответственно к типу INT/SHORT/LONG/FLOAT.

Аналогично, для выходной очереди сообщений используются функции:

ФУНКЦИЯ ЦЕЛ SendMsgGUICode (ЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgFloatParamGUI (ЦЕЛ, ПЛАВ);

ФУНКЦИЯ ЦЕЛ SendMsgShortParamGUI (ЦЕЛ, КЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgIntParamGUI (ЦЕЛ, ЦЕЛ);

ФУНКЦИЯ ЦЕЛ GetInpMsgNumberGUI (ПУСТО);

ФУНКЦИЯ ЦЕЛ GetOutMsgNumberGUI (ПУСТО);

Для комплекса автоматической верификации

Для Controller – функции совпадают с функциями, используемыми для автоматизированной верификации.

Для Plant:

ЦЕЛ GetOutMsgNumberFromDispatcher(ПУСТО);

Возвращает количество сообщений в очереди сообщений от Dispatcher.

ФУНКЦИЯ ЦЕЛ GetMessageFromDispatcher (ПУСТО);

Извлекает сообщение из очереди сообщений от Dispatcher. Извлеченное сообщение удаляется из очереди. Если функция возвращает ноль – очередь пуста.

ФУНКЦИЯ ЦЕЛ GetMessageCodeFromDispatcher (ПУСТО);

Возвращает идентификатор последнего извлеченного сообщения из очереди для Dispatcher.

ФУНКЦИЯ КЦЕЛ GetShortParamFromDispatcher(ПУСТО);

ФУНКЦИЯ ЦЕЛ GetIntParamFromDispatcher(ПУСТО);

ФУНКЦИЯ ДЦЕЛ GetLongParamFromDispatcher(ПУСТО);

ФУНКЦИЯ ПЛАВ GetFloatParamFromDispatcher (ПУСТО);

Возвращает параметр последнего извлеченного сообщения из очереди для Dispatcher и преобразовывает его к типу SHORT/INT/LONG/FLOAT соответственно.

Также присутствуют функции отправки сообщений для ГИО оператора (см. Controller). Рекомендуется использовать их для отладочных сообщений.

Для использования в Dispatcher:

Для работы с входными сообщениями от Верификатора.

ФУНКЦИЯ ЦЕЛ GetNextMsgFromVerifier(ПУСТО);

Извлекает сообщение из очереди сообщений от Verifier. Извлеченное сообщение удаляется из очереди. В случае, если очередь пуста, возвращаемое значение равно 0.

ФУНКЦИЯ ЦЕЛ GetMsgCodeFromVerifier(ПУСТО);

FUNC INT GetMsgCodeFromVerifier(VOID);

Возвращает значение идентификатора последнего извлеченного сообщения из очереди входных сообщений от Verifier.

ФУНКЦИЯ КЦЕЛ GetShortParamFromVerifier (ПУСТО);

ФУНКЦИЯ ЦЕЛ GetIntParamFromVerifier (ПУСТО);

ФУНКЦИЯ ДЦЕЛ GetLongParamFromVerifier (ПУСТО);

ФУНКЦИЯ ПЛАВ GetFloatParamFromVerifier (ПУСТО);

Возвращает параметр последнего извлеченного сообщения из очереди для Verifier и преобразовывает его к типу SHORT/INT/LONG/FLOAT соответственно.

Работа с очередями выходных сообщений:

Для Plant:

ФУНКЦИЯ ЦЕЛ SendMessageToVirtualPlantCode (ЦЕЛ);

Помещает сообщение без аргументов в очередь выходных сообщений для Plant.

ФУНКЦИЯ ЦЕЛ SendMsgShortParamToVirtualPlant (ЦЕЛ, КЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgIntParamToVirtualPlant (ЦЕЛ, ЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgLongParamToVirtualPlant (ЦЕЛ, ДЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgFloatParamToVirtualPlant (ЦЕЛ, ПЛАВ);

Помещают сообщение с аргументом типа INT/SHORT/LONG/FLOAT соответственно в очередь выходных сообщений для Plant.

Выходная очередь сообщений для Controller:

ФУНКЦИЯ ЦЕЛ SendMsgToControlAlgorythmCode (ЦЕЛ);

Помещает сообщение без аргументов в очередь выходных сообщений для Controller.

ФУНКЦИЯ ЦЕЛ SendMsgShortParamToControlAlgorythm (ЦЕЛ, КЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgIntParamToControlAlgorythm (ЦЕЛ, ЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgLongParamToControlAlgorythm (ЦЕЛ, ДЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgFloatParamToControlAlgorythm (ЦЕЛ, ПЛАВ);

Помещают сообщение с аргументом типа INT/SHORT/LONG/FLOAT соответственно в очередь выходных сообщений для Controller.

Выходная очередь сообщений для Verifier:

ФУНКЦИЯ ЦЕЛ SendMsgToVerifier (ЦЕЛ);

Помещает сообщение без аргументов в очередь выходных сообщений для Controller.

ФУНКЦИЯ ЦЕЛ SendMsgShortParamToVerifier (ЦЕЛ, КЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgIntParamToVerifier (ЦЕЛ, ЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgLongParamToVerifier (ЦЕЛ, ДЦЕЛ);

ФУНКЦИЯ ЦЕЛ SendMsgFloatParamToVerifier (ЦЕЛ, ПЛАВ);

Помещают сообщение с аргументом типа INT/SHORT/LONG/FLOAT соответственно в очередь выходных сообщений для Verifier.

Для Verifier

INT GetOutGUIMsgNumber(VOID);

Возвращает количество сообщений в очереди для ГИО.

INT GetOutSCMMsgNumber(VOID);

Возвращает количество сообщений в очереди для Dispatcher.

Для получения входных сообщений из входной очереди от Controller

ФУНКЦИЯ ЦЕЛ GetNextMsgFromAlgorithm(ПУСТО);

Извлечение сообщения из очереди сообщений от Controller. Очередь сдвигается на один элемент. Если возвращаемое значение равно нулю, то очередь пуста.

ФУНКЦИЯ ЦЕЛ GetMsgCodeFromAlgorithm (ПУСТО);

Возвращает идентификатор последнего считанного сообщения из очереди входных сообщений от Controller.

ФУНКЦИЯ КЦЕЛ GetShortParamFromAlgorithm (ПУСТО);

ФУНКЦИЯ ЦЕЛ GetIntParamFromAlgorithm (ПУСТО);

ФУНКЦИЯ ДЦЕЛ GetLongParamFromAlgorithm (ПУСТО);

ФУНКЦИЯ ПЛАВ GetFloatParamFromAlgorithm (ПУСТО);

Возвращает параметр последнего извлеченного сообщения из очереди от Controller и преобразовывает его к типу SHORT/INT/LONG/FLOAT соответственно.

Аналогичный набор функций существует для очереди входных сообщений от Dispatcher для Plant.

Приложение Б. Информация о внедрении результатов

Утверждаю
Директор ИСЗФ СО РАН
д.ф.-м.н. Медведев А.В.
«28» февраля 2019 г.
М.П.

АКТ

о внедрении результатов кандидатской диссертационной работы

Лях Татьяны Викторовны

Настоящий акт составлен о том, что научные результаты кандидатской диссертационной работы Лях Татьяны Викторовны, представленной на соискание степени кандидата технических наук, использовались при разработке автоматической системы управления Большого Солнечного Вакуумного Телескопа (БСВТ). При создании системы управления использовался разработанный Лях Т.В. метод автоматизированной верификации процесс-ориентированных алгоритмов, созданных на языке Reflex. Были разработаны:

1. Программный имитатор вакуумной подсистемы БСВТ;
2. Комплекс автоматизированной верификации процесс-ориентированных алгоритмов, созданных на языке Reflex.

Метод автоматизированной верификации процесс-ориентированных алгоритмов, созданных на языке Reflex, был апробирован при отладке алгоритма управления подсистемой вакуумирования и обеспечил верификацию алгоритма в следующих аварийных ситуациях:

- сбой питания;
- нештатное открытие вакуумного затвора при сбое питания;
- потеря связи с выносными УСО;
- отказ исполнительных устройств;
- самопроизвольное включение/выключение оборудования;
- падение уровня воды в системе охлаждения вакуумного насоса ниже допустимого;
- перегрев и замерзание воды.

Результатом работы стало сокращение этапа пусконаладочных работ и качественное повышение надежности функционирования вакуумной подсистемы БСВТ при обработке аппаратных сбоев.

Составил
к.ф.-м.н.,
заведующий Лабораторией
экспериментальной физики Солнца и
астрофизического приборостроения

Колобов Д.Ю.

Утверждаю

Директор Института автоматики и
электрометрии СО РАН

чл. -корр. РАН, д.ф.-м.н._____ Бабин С.А.

«___» _____ г.

АКТ

о внедрении результатов кандидатской диссертационной работы

Лях Татьяна Викторовны

Настоящий акт составлен о том, что научные результаты кандидатской диссертационной работы Лях Татьяны Викторовны, представленной на соискание степени кандидата технических наук, использовались при разработке автоматической системы управления углоизмерительной машиной НОНИУС. При создании системы управления использовался разработанный Лях Т.В. механизм бесшовной интеграции алгоритмических блоков, созданных из описания на языке Reflex, в среду LabVIEW. Reflex. Механизм предоставил возможность:

1) Автоматически собирать и бесшовно интегрировать алгоритмический блок, созданный из описания на языке Reflex, в среду LabVIEW, в виде DLL;
2) Обмен значениями дискретных и аналоговых сигналов с алгоритмическим блоком;
3) Обмен сообщениями с алгоритмическим блоком;
4) Получение отладочных данных о внутреннем состоянии алгоблока.

Результатом работы стало сокращение времени этапов разработки и пуско-наладочных работ.

Заведующий лабораторией интегрированных
информационных систем управления

к.т.н. Кирьянов А.В.

РОССИЙСКАЯ ФЕДЕРАЦИЯ

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

RU 2013660427

ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2013660427

Дата регистрации: **06.11.2013**

Номер и дата поступления заявки:
2013616968 02.08.2013

Дата публикации: **20.12.2013**

Контактные реквизиты:
+7 (3383)3309005, zyubin@iae.nsk.su,
Зюбин Владимир Евгеньевич

Авторы:
Зюбин Владимир Евгеньевич (RU),
Журавлева Нина Владимировна (RU),
Лях Татьяна Викторовна (RU)

Правообладатель:
Федеральное государственное бюджетное учреждение науки Институт автоматики и электрометрии Сибирского отделения Российской академии наук (ИАиЭ СО РАН) (RU)

Название программы для ЭВМ:
Программный комплекс "Набор виртуальных лабораторных стендов для изучения стратегий управления объектами автоматизации"

Реферат:
Программный комплекс предназначен для организации обучения специалистов в области промышленной автоматизации программированию управляющих алгоритмов технологического уровня (уровня программируемых логических контроллеров). Комплекс предоставляет возможность создания и отработки алгоритмов управления следующими объектами автоматизации, которые реализованы программно средствами пакета LabVIEW и оформлены в виде виртуальных лабораторных стендов: «Электросушилка для рук», «Перекресток», «Автоматизированный розлив бутылок», «Сортировочный конвейер», «Лифт пятиэтажного дома», «Турникет метрополитена», «Бетоносмесительный узел». Виртуальные лабораторные стенды обеспечивают возможность практического исследования типовых алгоритмов управления, в том числе связанных с дискретным управлением, регулированием, обработкой временных интервалов, конвергенцией и дивергенцией потоков управления.

Тип реализующей ЭВМ: IBM PC-совмест. ПК

Язык программирования: G

Вид и версия операционной системы: Windows

Объем программы для ЭВМ: 13,7 Мб

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU 2018665910

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2018665910

Дата регистрации: **11.12.2018**

Номер и дата поступления заявки:
2018662615 14.11.2018

Дата публикации и номер бюллетеня:
11.12.2018 Бюл. № 12

Контактные реквизиты:
нет

Автор(ы):
Лях Татьяна Викторовна (RU)

Правообладатель(и):
Федеральное государственное бюджетное учреждение науки Институт автоматики и электрометрии Сибирского отделения Российской академии наук (ИАиЭ СО РАН) (RU)

Название программы для ЭВМ:
Программный комплекс автоматической верификации процесс-ориентированных алгоритмов версии 1.0

Реферат:
Назначение: программный комплекс предназначен для динамической верификации алгоритмов на языке Reflex. Область применения: разработка процесс-ориентированных алгоритмов управления в области промышленной автоматизации. Функциональные возможности: комплекс представляет возможность разработки и динамической верификации алгоритмов управления, специфицированных на языке Reflex. Он позволяет запускать, приостанавливать и прерывать верификацию алгоритма управления, изменять описания алгоритма управления, виртуального объекта управления, задавать сценарии работы с объектом и задавать проверку корректности реакций алгоритмов управления автоматически. При работе комплекс предоставляет пользователю информацию о результатах верификации алгоритма управления.

Язык программирования: C/C++, G LabVIEW, Reflex

Объем программы для ЭВМ: 180 Мб