ISSN 1818–1015 (Print)
ISSN 2313–5417 (Online)

Министерство науки и высшего образования Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ

Том 26 4 (82) 2019

Основан в 1999 году
Выходит 4 раза в год

Адрес редакции: ЯрГУ, ул. Советская, 14, г. Ярославль, 150003, Россия
Website: http://mais-journal.ru, e-mail: mais@uniyar.ac.ru; телефон (4852) 79-77-73

Научные статьи в журнал принимаются по электронной почте. Статьи должны содержать УДК, аннотации на русском и английском языках и сопровождаться набором текста в редакторе LaTeX.

12+

СОДЕРЖАНИЕ

Моделирование и анализ информационных систем. Т. 26, №4. 2019

Computing methodologies and applications

Theory of data

Algorithms

Computer system organization

Свидетельство о регистрации СМИ ПИ № ФС 77 – 66186 от 20.06.2016 выдано Федеральной службой по надзору в сфере связи, информационных технологий и массовых коммуникаций. Учредитель – Федеральное государственное бюджетное образовательное учреждение высшего образования "Ярославский государственный университет им. П. Г. Демидова". Подписной индекс – 31907 в Объединенном каталоге "Пресса России". Редакторы, корректоры М. С. Каряева, М. С. Комар. Редактор перевода Э.И. Соколова. Подписано в печать 11.12.2019. Дата выхода в свет 31.12.2019. Формат 60x84$^1/_8$. Усл. печ. л. 15,0. Уч.-изд. л. 7,9. Объем 114 с. Тираж 46 экз. Свободная цена. Заказ 072/019. Адрес типографии: ул. Советская, 14, оф. 109, г. Ярославль, 150003 Россия. Адрес издателя: Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, г. Ярославль, 150003 Россия.

ISSN 1818–1015 (Print)
ISSN 2313–5417 (Online)

P.G. Demidov Yaroslavl State University

MODELING AND ANALYSIS OF INFORMATION SYSTEMS

Volume 26 No 4 (82) 2019

Founded in 1999
4 issues per year

Editorial Office Address: P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003, Russia
Website: http://mais-journal.ru, e-mail: mais@uniyar.ac.ru

Contents

Modeling and Analysis of Information Systems. Vol. 26, No 4. 2019

От редакторов выпуска

В. А. Захаров[1,2], *Н. В. Шилов*[3]

[1] ВШЭ
[2] ИСП РАН
[3] Университет Иннополис

From the Editors of the Issue

V. A. Zakharov[1,2], *N. V. Shilov*[3]

[1] HSE
[2] ISP RAS
[3] Innopolis University

1–2 июля 2019 г. в новосибирском Академгородке в рамках международного научного форума Computer Science Summer in Russia прошел десятый международный научно–исследовательский семинар "Семантика, спецификация и верификация программ: теория и приложения"(10-th Workshop on Program Semantics, Specification and Verification: Theory and Applications, PSSV-2019). Организатором семинара выступил Институт систем информатики им. А. П. Ершова СО РАН. Программа семинара PSSV-2019 включала 9 регулярных докладов, 5 лекций приглашенных докладчиков, а также рабочее совещание, посвященное вопросам спецификации и верификации стандартных функций. Участники семинара в своих выступлениях рассказали о результатах завершенных и продолжающихся исследований разнообразных задач в области математического моделирования и верификации программных систем, о развитии методов дедуктивного анализа программ, а также о применении методов теории автоматов к тестированию программ. Приглашенными докладчиками были А. Лисица (Ливерпульский университет, Великобритания), С. П. Шарый (Новосибирский гос. университет, Россия), Бин Фанг (компания Huawei), Д. А. Мордвинов (Санкт-Петербургский гос. университет, Россия). Данный выпуск журнала включает 6 статей участников семинара PSSV-2019.

Статья И. С. Ануреева посвящена процесс-ориентированному языку программирования Reflex; этот язык применяется для разработки простого в обслуживании управляющего программного обеспечения для программируемых логических контроллеров. В статье описана операционная семантика Reflex-программ, расширенных аннотациями, описывающими формальную спецификацию программных требований, как необходимый базис для применения формальных методов верификации. В операционной семантике аннотированных Reflex-программ используются глобальные и локальные таймеры, учитывается бесконечность цикла выполнения программы, логика управления переходами процессов из состояния в состояние и взаимодействие процессов между собой и с окружением. Расширение формальной операционной семантики языка Reflex на аннотации упрощает проведение дедуктивной верификации Reflex-программ. Статья написана по материалам доклада, представленного на PSSV-2019.

В статье Д. А. Кондратьева и А. В. Промского рассказывается об одном подходе к решению проблемы вычисления инвариантов цикла в рамках продолжающейся в ИСИ СО РАН разработки система C-lightVer для дедуктивной верификации C-программ с использованием системы автоматического доказательства теорем ACL2. Ранее авторами был разработан способ доказательства ложности условий корректности для системы ACL2. Необходимость в более подробных объяснениях условий корректности, содержащих операцию замены, привела к изменению алгоритмов генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных условий корректности. В статье представлены модификации данных алгоритмов.Статья написана по материалам доклада, представленного на PSSV-2019.

В статье Т. Баара и Х. Шульте предоставлена исполняемая модель для системы управления пассажирского самолета, написанная на языке Matlab/Simulink®. Эта модель учитывает программное обеспечение, помогающее пилоту предотвращать срыв воздушного потока, принимает во внимание возможность передачи неверных данных датчиками. Чтобы с помощью построенной модели можно было решать задачи верификации систем управ-

ления самолетом, авторы статьи сумели преобразовать модель Matlab/Simulink®в гибридную программу (HP), которую можно проверять с использованием системы автоматического доказательства теорем KeYmaera. Статья написана по материалам доклада, представленного на PSSV-2019.

В статье В. Тодорова, С. Таха, Ф. Буланже и А. Эрнандеса исследованы возможности применения формальных методов верификации для проверки свойств безопасности программных процедур, используемых в встроенных системах. Методику решения этой задачи авторы статьи демонстрируют на примере функции, вычисляющей квадратный корень с помощью линейной интерполяции. Статья написана по материалам доклада, представленного на PSSV-2019.

Статья Н. О. Гараниной, И. С. Ануреева, О. И. Боровиковой и В. Е. Зюбина продолжает цикл исследований, предпринятых авторами для создания такой разновидности онтологий процессов, которые можно конструировать на основании технических заданий на разработку информационных систем и в то же время использовать для формальной верификации проекта на ранних этапах его осуществления. В статье авторы предлагают два метода специализации таких онтологий, позволяющих учитывать индивидуальные особенности распределенных систем: декларативный и конструктивный. Приведен пример, иллюстрирующий применение предложенных методов специализации и наполнения онтологии процессов. Статья написана по материалам доклада, представленного на PSSV-2019.

Д. А. Мордвинов в своей статье представил новый подход к решению нелинейных систем дизъюнктов Хорна. Главная особенность этого подхода состоит в автоматическом выводе реляционных инвариантов, аппроксимирующих сверху семантику групп неинтерпретированных символов. Практическая эффективность нового метода была показана на тестовых наборах различных задач реляционной верификации. Статья написана по материалам лекции, прочитанной на PSSV-2019.

Моделирование и анализ информационных систем. Т. 26, № 4 (2019), с. 475–487
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 475–487

Computing methodologies and applications

DOI: 10.18255/1818-1015-2019-4-475-487

УДК 04.423.42, 004.434, 681.51

Операционная семантика аннотированных Reflex программ

Ануреев И. С.

Поступила в редакцию 12 сентября 2019
После доработки 15 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. Reflex — процесс-ориентированный язык, который обеспечивает разработку простого в обслуживании управляющего программного обеспечения для программируемых логических контроллеров. Язык был успешно использован в нескольких системах управления с повышенными требованиями к надежности, например, в системе управления печью для выращивания монокристаллов кремния и в комплексе контроля радиоэлектронной аппаратуры. В настоящее время основной целью языкового проекта Reflex является разработка методов формальной верификации для Reflex программ для того, чтобы гарантировать повышенную надежность создаваемого на его основе программного обеспечения. В статье представлена формальная операционная семантика Reflex программ, расширенных аннотациями, описывающими формальную спецификацию программных требований, как необходимый базис для применения таких методов. Дан краткий обзор языка Reflex и приведен простой пример его использования — управляющая программа для сушилки рук. Определены понятия окружения и переменных, разделяемых с окружением, позволяющие абстрагироваться от конкретных портов ввода/вывода. Определены типы аннотаций, задающие ограничения на значения переменных при запуске программы, ограничения на окружение (в частности, на объект управления), инварианты цикла управления, пред- и постусловия внешних функций, используемых в Reflex программах. Аннотированный Reflex также использует стандартные аннотации `assume`, `assert` и `havoc`. Операционная семантика аннотированных Reflex программ использует глобальные часы и локальные часы отдельных процессов, время которых измеряется в количестве итераций цикла управления, для моделирования временных ограничений на исполнение процессов в определенных состояниях. Она хранит полную историю изменений значений разделяемых переменных для более полного описания временных свойств программы и ее окружения. Семантика учитывает бесконечность цикла выполнения программы, логику управления переходами процессов из состояния в состояние и взаимодействие процессов между собой и с окружением. Расширение формальной операционной семантики языка Reflex на аннотации упрощает доказательство корректности разрабатываемого авторами трансформационного подхода к дедуктивной верификации Reflex программ, трансформирующего аннотированную Reflex программу к аннотированной программе на сильно ограниченном подмножестве языка C, за счет сведения сложного доказательства сохранения истинности требований к программе при трансформации к более простому доказательству эквивалентности исходной и результирующей аннотированных программ относительно их операционных семантик.

Ключевые слова: операционная семантика, язык Reflex, система управления, управляющее программное обеспечение, программируемый логический контроллер, аннотация, аннотированная программа

Для цитирования: Ануреев И. С., "Операционная семантика аннотированных Reflex программ", *Моделирование и анализ информационных систем*, **26**:4 (2019), 475–487.

Об авторах:
Ануреев Игорь Сергеевич, orcid.org/0000-0001-9574-128X, канд. физ.-мат. наук, с.н.с.,
Институт систем информатики им. А.П. Ершова СО РАН, Институт автоматики и электрометрии СО РАН,
пр. Акад. Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: anureev@iis.nsk.su

Благодарности:
Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 17-07-01600, а также в рамках темы госзадания ИАиЭ СО РАН (№ АААА-А17-11706061006-6).

Введение

Многие системы управления, в частности, в области промышленной автоматизации, основаны на промышленных программируемых логических контроллерах (ПЛК), обладающих следующими особенностями: они по своей природе являются открытыми (то есть взаимодействуют с внешней средой), реактивными (имеют управляемое событиями поведение) и параллельными (должны обрабатывать многочисленные асинхронные события). Эти особенности привели к использованию специальных языков при разработке управляющего программного обеспечения, например, языков стандарта IEC 61131-3 [1], которые являются наиболее популярными в области программирования ПЛК. Однако поскольку сложность управляющего программного обеспечения возрастает, а качество становится более приоритетным, 35-летняя технология, основанная на подходе IEC 61131-3, не в состоянии удовлетворить современные требования [2].

Reflex – это предметно-ориентированный язык с C-подобным синтаксисом для области управляющего программного обеспечения, основанный на концепциях процесса и состояния процесса как программного кода и созданный как альтернатива языкам стандарта IEC 61131-3. Программа на языке Reflex описывается как множество взаимодействующих процессов. Язык имеет специализированные инструкции для управления процессами и их состояниями и инструкции для работы с временными интервалами. Эти средства поддерживают парадигму событийно-ориентированного программирования, в которой выполнение программы определяется событиями (в том числе, временными событиями). Он также имеет инструкции, связывающие переменные программ на этом языке с физическими портами ввода/вывода. Reflex предполагает выполнение программы на основе цикла управления с фиксированным временем итерации и строгую инкапсуляцию зависимых от платформы подпрограмм ввода-вывода в библиотеку, что является широко применяемой техникой в системах управления, созданной на основе IEC 6113-3. Для обеспечения простоты поддержки и межплатформенной переносимости генерация исполняемого кода осуществляется в два этапа: транслятор для языка Reflex генерирует C-код, а затем C-компилятор создает исполняемый код для целевой платформы.

В настоящее время проект Reflex сфокусирован на средствах разработки программного обеспечения для систем управления, имеющих повышенные требования к безопасности. Благодаря платформенной независимости язык Reflex легко интегрируется с LabVIEW [3]. Это позволяет разрабатывать программное обеспечение, сочетающее событийную ориентированность с развитым графическим пользовательским интерфейсом, удаленными датчиками и актуаторами, устройствами

с поддержкой LabVIEW и т. д. Используя гибкость LabVIEW был разработан набор симуляторов объектов управления для целей обучения [4]. Основанные на LabVIEW симуляторы включают в себя 2D-анимацию, инструменты для отладки и языковую поддержку для обучения разработке управляющего программного обеспечения. Одним из результатов, полученным в этом направлении, является набор инструментов динамической верификации на основе LabVIEW для Reflex программ. Динамическая проверка рассматривает программное обеспечение как черный ящик и проверяет его соответствие требованиям, наблюдая за поведением программного обеспечения во время выполнения на наборе тестовых случаев. Хотя такая процедура может помочь обнаружить наличие ошибок в программном обеспечении, она не может гарантировать их отсутствие [5].

В отличие от методов тестирования и динамической верификации, методы формальной верификации являются единственным способом обеспечить требуемые свойства программного обеспечения. Предложенный в [6] трансформационный подход к дедуктивной верификации Reflex программ базируется на алгоритме трансформации [7], сводящем дедуктивную верификацию аннотированных Reflex программ к дедуктивной верификации очень ограниченного подмножества аннотированных C программ, и алгоритме генерации условий корректности для этого подмножества C программ, основанном на исчислении сильнейшего постусловия. Доказательство корректности этих алгоритмов требует, чтобы исходная и результирующая программы имели формальную семантику. В этой статье мы представляем формальную операционную семантику аннотированных Reflex программ.

1. Введение в язык Reflex

Синтаксис языка Reflex демонстрируется здесь на простом примере программы, управляющей сушилкой для рук (Листинг 1). Программа использует вход от ИК-датчика, указывающего на присутствие рук под сушилкой, и управляет вентилятором и обогревателем с помощью совместного выходного сигнала. Формальное определение синтаксиса языка Reflex в EBNF можно найти [8].

```
PROGR HandDryerController {

 INIT <formula 1>;

 ENVIRONMENT <formula 2>;

 INVARIANT <formula 3>;

 TACT 100;
 CONST ON 1;
 CONST OFF 0;
 /*============================*/
 /* I/O ports specification    */
 /* direction, name, address,  */
 /* offset, size of the port   */
 /*============================*/
 INPUT  SENSOR_PORT  0 0 8;
 OUTPUT ACTUATOR_PORT 1 0 8;

 /*============================*/
 /* processes definition       */
 /*============================*/
 PROC Controller {
  /*===== VARIABLES =============*/
  BOOL HANDS = {SENSOR_PORT[1]} FOR ALL;
  BOOL DRYER = {ACTUATOR_PORT[1]} FOR ALL;
```

```
  /*===== STATES ================*/
  STATE Waiting {
   IF (HANDS == ON) {
    O_DRYER = ON;
    SET NEXT;
   } ELSE DRYER = OFF;
  }
  STATE Drying {
   IF (HANDS == ON) RESET TIMEOUT;
   TIMEOUT 10;
   SET STATE Waiting;
  }
 } /* PROC */
}  /* PROGR */
```

Листинг 1. Программа управления сушилкой для рук
Listing 1. Hand Dryer Control Program

На языке Reflex программа представляется как множество взаимодействующих процессов. Определение процесса `PROC <process name> <process body>` начинается с ключевого слова `PROC`, за которым следуют имя и тело процесса.

Выполнение программы суть повторяющиеся итерации цикла управления, где каждая итерация заключается в последовательном выполнении процессов программы (в порядке их определения в программе). Каждая итерация имеет фиксированное время выполнения в миллисекундах (период выполнения), задаваемое инструкцией `TACT`.

Тело процесса состоит из объявлений переменных и определений состояний. Определение состояния процесса `STATE <state name> <state body>` состоит из имени и тела состояния. Последнее специфицирует действие, выполняемое процессом в этом состоянии. Для каждого процесса неявно определены два дополнительных пассивных состояния `STOP` и `ERROR`. Эти состояния называются пассивными, так как никаких действий в них не выполняется. Состояние `STOP` соответствует остановке или приостановке выполнения процесса. Состояние `ERROR` соответствует ошибочному выполнению процесса. Все остальные состояния процесса являются активными.

Тело состояния определяется как блок (оператор последовательной композиции), включающий операторы присваивания, условный оператор if, вложенные блоки, операторы управления состояниями процессов и операторы таймаута.

Состояние, определяемое первым в теле процесса, называется стартовым состоянием этого процесса. Первый процесс, определенный в тексте программы, является единственным активным процессом при ее запуске. Он устанавливается в свое стартовое состояние. Остальные процессы устанавливаются при запуске программы в состояние `STOP`.

Синтаксис и семантика Reflex выражений идентичны синтаксису и семантике C выражений, за исключением фиксированного порядка вычисления аргументов операций и функций (слева направо) и специфических для Reflex логических операций над состояниями, называемых предикатами активности. Операторы присваивания, условные операторы и блоки имеют синтаксис и семантику, как в языке C (несущественные отличия заключаются в том, что присваивание рассматривается в качестве оператора в языке Reflex, т.е.не допускаются вложенные присваивания, и все переменные Reflex программы глобальны). Поэтому ниже мы опишем только те инструкции, которые специфичны для Reflex.

Каждый процесс имеет локальные часы, которые отсчитывают дискретное вре-

мя, измеряемое в количестве итераций цикла управления. Один тик локальных часов соответствует одной итерации цикла управления.

Оператор `SET STATE` s; устанавливает текущий процесс в состояние s для следующей итерации цикла управления. Этот и другие операторы, изменяющие состояние процесса (даже на то же самое состояние), сбрасывают время локальных часов этого процесса в ноль. Оператор `SET NEXT`; устанавливает текущий процесс в состояние, которое следует за текущим состоянием в теле процесса. Его применение к текущему состоянию, которое определяется последним в теле процесса, является синтаксической ошибкой.

Операторы `START PROC` p; и `STOP PROC` p; запускают и останавливают процесс p. Они устанавливают p в его стартовое состояние и в состояние `STOP` соответственно.

Операторы `RESTART`;, `STOP`; и `ERROR`; запускают и останавливают (нормальным образом и с ошибкой) текущий процесс. Они устанавливают этот процесс в его стартовое состояние, состояние `STOP` и состояние `ERROR` соответственно.

Процессы могут проверять, находятся ли другие процессы в активных или пассивных состояниях, используя условные операторы совместно с предикатами активности, например:

```
IF (PROC <process name> IN STATE ACTIVE) { ... }
```

Операторы таймаута осуществляют контроль времени выполнения процесса в определенном состоянии. Оператор `RESET TIMEOUT`; обнуляет локальные часы текущего процесса. Оператор `TIMEOUT <clocks num> <statement>` обнуляет локальные часы текущего процесса и выполняет оператор `<statement>` в том случае, если время на локальных часах достигло значения `<clocks num>` и ничего не делает в противном случае. Этот оператор может использоваться только один раз в теле состояния и должен быть последним оператором в теле состояния.

Тело процесса может содержать определения переменных

```
<type> <variable name> = {<port name>[<bit number>]} <scope>;
```

задающие такие характеристики переменной как тип, имя, порт (с которым она связывается), номер бита в порту (соответствующего значению переменной) и область видимости.

Поддерживаемыми типами являются `BOOL` для булевских значений, а также целочисленные и вещественные типы `INT`, `SHORT`, `LONG`, `FLOAT`, `DOUBLE`, определяемые как в языке C. Область действия `FOR ALL` указывает на то, что эта переменная может использоваться любыми процессами в программе. Привязка к порту позволяет считывать значение из входного порта в переменную и записывать значение переменной в выходной порт.

Декларации портов, размещаемые перед определениями процессов, задают характеристики портов, используемых в программе,

```
<direction> <port name> <base address> <offset> <size in bits>;
```

такие как направление (входной и выходной порты), имя, базовый адрес, смещение и размер в битах.

Важной особенностью переменных, связанных с портами, является то, что все операции чтения и записи для этих переменных имеют двойную буферизацию. Значения портов ввода/вывода считываются один раз за итерацию цикла управления, и каждое значение сохраняется в двух экземплярах — один для операций чтения

и один для операций записи. Новые значения для выходных портов устанавливаются и отправляются на внешние устройства в конце итерации. Таким образом, все процессы читают одни и те же значения портов, даже если они изменяются в итерации.

Объявления констант `CONST <constant name> <constant value>;` определяют имена и значения констант.

Формальная спецификация требований к Reflex программе задается специальными инструкциями, называемыми аннотациями. Пусть `<formula>` — формула на языке многосортной логики первого порядка.

Аннотация `INIT <formula>;` описывает ограничения на значения переменных при запуске программы.

Аннотация `ENVIRONMENT <formula>;` описывает ограничения на значения, которые окружение (например, объект управления) может передавать в программу (записывать в переменные программы) через входные порты.

Аннотация `INVARIANT <formula>;` описывает инварианты цикла управления, т. е. свойства, которые должны выполняться на каждой итерации цикла управления. В частности, эта аннотация описывает связи между значениями, считываемыми с входных портов, и значениями, записываемыми в выходные порты после очередной итерации.

Примеры этих трех видов аннотаций для программы управления сушилкой для рук можно найти в [6]. Программа, включающая аннотации, называется аннотированной программой.

Reflex программа может использовать в выражениях вызовы внешних C функций `<function name>(<expression 1>, ..., <expression n>)`. Для каждой такой функции в программе должен быть описан ее прототип,

```
<function type> <function name>
 (<parameter type 1> <parameter name 1>, ...,
  <parameter type n> <parameter name n>) {
 REQUIRES <formula 1>;
 ENSURES <formula 2>;
};
```

задающий ее имя, тип, параметры и их типы, пред- и постусловия.

Аннотация `REQUIRES <formula 1>;` описывает ограничения на параметры функции (предусловие функции).

Аннотация `ENSURES <formula 2>;` описывает ограничения на значение, возвращаемое функцией (постусловие функции). Формула `<formula 2>` содержит переменную `<function name>`, которая ссылается на значение, возвращаемое функцией.

Reflex программа может также включать следующие виды аннотаций.

Аннотация `ASSUME <formula>;` завершает программу со значением `IGNORE`, если формула `<formula>` ложна и ничего не делает, в противном случае. Значение `IGNORE` на некотором пути выполнения программы означает, что этот путь не учитывается при доказательстве корректности программы.

Аннотация `ASSERT <formula>;` завершает программу со значением `FAIL`, если формула `<formula>` ложна и ничего не делает, в противном случае. Значение `FAIL` на некотором пути выполнения программы означает, что программа выполняется некорректно на этом пути.

Аннотация `HAVOC <variable name>;` присваивает произвольное значение переменной `<variable name>` в соответствие с ее типом. Аннотация

```
HAVOC <variable name 1>, ..., <variable name n>;
```

сводится к последовательному выполнению аннотаций

```
HAVOC <variable name 1>; ...; HAVOC <variable name n>;
```

Определим операционную семантику Reflex программ.

2. Операционная семантика аннотированных Reflex программ

Пусть N — множество натуральных чисел (включая 0). Пусть U^* — множество всех последовательностей из элементов множества U, $|u|$ и $u[i]$ обозначают длину и i-й элемент последовательности $u \in U^*$ соответственно. Пусть $< u_1, \ldots, u_m >$ обозначает последовательность элементов u_1, ..., u_m и $con(w_1, \ldots, w_n)$ — конкатенацию последовательностей w_1, ..., w_n.

Для множества A всех Reflex программ определим: множества T, H и E типов, операторов и выражений; множество $val(t)$ всех значений типа $t \in T$; значение `OK` как нормальное завершение Reflex инструкции, множество Γ всех значений такое, что $\cup_{t \in T} val(T) \cup \{\texttt{OK}, \texttt{IGNORE}, \texttt{FAIL}\} \subseteq \Gamma$.

Для каждой программы $\alpha \in A$ определим:

- ее окружение π;
- множество $P = \{p_1, \ldots, p_n\}$ процессов ($P \subseteq \Gamma$);
- множество Θ состояний процессов ($\Theta \subseteq \Gamma$);
- множество F функций ($F \subseteq \Gamma$);
- множество $V = V_s \cup V_l$ переменных ($V \subseteq \Gamma$);
- множество $V_s = V_i \cup V_o$ разделяемых переменных (они разделяются с π);
- множество V_l локальных переменных ($V_s \cap V_l = \emptyset$);
- множество V_i входных переменных (окружение π может только писать в эти переменные);
- множество V_o выходных переменных (окружение π может только читать из этих переменных);
- $V_i \cap V_o = \emptyset$;
- функцию $vt \in V \to T$, связывающую переменные с их типами;
- функцию $pso \in P \to \Theta^*$, связывающую процессы с упорядоченными последовательностями их состояний (состояния в $pso(p)$ перечисляются в порядке, в котором они определяются в p);
- упорядоченную последовательность $po \in P^n$ процессов (процессы в po перечисляются в порядке, в котором они определяются в α);

- функцию $psv \in \Theta \to H$, связывающую состояния процессов с их телами;
- функцию $ftype \in F \to T$, связывающую внешние функции с их типами;
- функцию $fpar \in F \to V_l^*$, связывающую внешние функции Reflex программы с их параметрами;
- функцию $ftpar \in F \to T^*$, связывающую внешние функции Reflex программы с типами их параметров;
- функции $fpre, fpost \in F \to H$, связывающую функции с их пред- и постусловиями;
- формулы ini, env и inv в аннотациях `INIT`, `ENVIRONMENT` и `INVARIANT` соответственно.

Для каждой программы α мы считаем, что следующие ограничения выполняются при определении операционной семантики:

1. Программа α корректна определена.
2. Информация о портах программы α и сопоставление переменных с портами не принимается в расчет, так как она относится к взаимодействию с физическими устройствами. Вместо этого программа α взаимодействует с абстрактным окружением через входные и выходные переменные.
3. Области действия переменных в α не принимаются в расчет, так как они определяют только корректный доступ к переменным, который гарантируется для корректно-определенных программ.
4. Отсутствует перегрузка имен переменных, состояний процессов, внешних функций и параметров внешних функций. Это ограничение достигается переименованием перегруженных имен. Поэтому мы считаем, что для каждой внешней функции `<function name>` параметры этой функции, задаваемые в ее прототипе, и переменная `<function name>` являются локальными переменными программы α. Типы этих переменных определяются прототипом функции.
5. Программа α не содержит деклараций констант и декларации такта. Это ограничение достигается выполнением C-подобных макроподстановок.

Операционная семантика языка Reflex определяется системой переходов $(S, S_i, \to_R, \to_C)$, где S — множество состояний, $S_i \subseteq S$ — множество начальных состояний, $\to_R$ и $\to_C$ — отношения переходов для Reflex инструкций и внешних C функций соответственно. Имеются несколько решений для описания операционной семантики языка C [9–13]. Поэтому мы фокусируемся на операционной семантике Reflex инструкций (отношении переходов $\to_R$), считая, что отношение $\to_C$ определяется с помощью одного из этих подходов (наш подход наиболее близок к подходу [12]).

Состояние $s \in S$ определяется как кортеж $(gc, cp, lc, ps, vv, ih, oh)$, который включает:

- глобальные часы $gc \in N$;

- текущий процесс $cp \in P$;
- функцию $lc \in P \to N$, связывающую процессы с их локальными часами, при этом все часы отсчитают время в тиках (один тик соответствует одной итерации по циклу управления);
- функцию $ps \in P \to \Theta$, связывающую процессы с их текущими состояниями;
- функцию $vv \in V \to \Gamma$, связывающую переменные с их значениями;
- входную историю $ih \in V_i \to \Gamma^*$ ($ih(v)[i]$ — значение переменной $v \in V_i$, записанное π во время i-го тика глобальных часов gc);
- выходную историю $oh \in V_o \to \Gamma^*$ ($oh(v)[i]$ — значение переменной $v \in V_o$, прочитанное π во время i-го тика глобальных часов gc).

Пусть $s.gc$, ..., $s.oh$ обозначает доступ к компоненте gc, ..., oh состояния s. Состояние s является начальным, если выполнены следующие свойства:

- $gc = 0$;
- $lc(p) = 0$ для каждого $p \in P$;
- $ps(po.1) = pso(po.1)$;
- $ps(po.i) = stop$ для каждого $1 < i \leq n$;
- $|ih(v)| = 0$ для каждого $v \in V_i$;
- $|oh(v)| = 0$ для каждого $v \in V_o$.

Отношение перехода $\to_R \in \Xi_R \times \Xi_R$ определяется на множестве $\Xi_R = I_R \times S$ конфигураций Reflex программ, где I_R — множество всех Reflex инструкций, обладающее следующим свойством: $T \cup H \cup E \cup \Gamma \subseteq I_R$. Таким образом, инструкциями могут быть типы (из T), операторы (из H) или выражения (из E), встречающиеся в Reflex программах, значения (из Γ), возвращаемые Reflex инструкциями в операционной семантике языка Reflex, а также вспомогательные инструкции, используемые (и определяемые) в правилах операционной семантики языка Reflex (см., например, метаприсваивание ниже).

Операционная семантика Reflex инструкций определяется правилами перехода. Многие из этих правил используют метаприсваивание $u_1.u_2.\ ...\ .u_m\ \texttt{:=}\ u;$.

Метаприсваивание. Пусть $upd(w, u_1.u_2.\ ...\ .u_m, u)$ — функция, заменяющая $w(u_1)(u_2)\ ...\ (u_m)$ на u в w. Например, если $w \in N \to (N \to N)$, то $upd(w, 3.7, 10) = w'$ означает, что $w' \in N \to (N \to N)$, для любых $x \in N$ и $y \in N$ таких, что $x \neq 3$, или $y \neq 7$, верно $w'(x)(y) = w(x)(y)$, и $w'(3)(7) = 10$. В случае операций доступа к элементам последовательностей и кортежей $u(i)$ означает $u[i]$ и $u.i$ соответственно. Пусть $val(u, s)$ обозначает значение выражения или формулы u в состоянии s. Тогда метаприсваивание определяется правилом

$$(u_1.u_2.\ ...\ .u_m\ \texttt{:=}\ u;, s) \to_R (val(u, s), upd(s, u_1.u_2.\ ...\ .u_m, val(u, s))).$$

Итератор. Итераторы используются в правилах перехода как сокращение для повторения инструкций. Пусть $\eta[v \leftarrow u]$ обозначает замену всех вхождений v в инструкции η на u. Тогда итератор определяется правилами

(FOREACH v IN $< u_1, ..., u_m >$ DO η END; $\rightarrow_R \eta[v \leftarrow u_1]$ ... $\eta[v \leftarrow u_m], s$);
Если $< k_1, ..., k_m >$ — перестановка $< 1, ..., m >$,
то (FOREACH v IN $\{u_1, ..., u_m\}$ DO η END; $\rightarrow_R \eta[v \leftarrow u_{k_1}]$ ... $\eta[v \leftarrow u_{k_m}], s$).

Аннотации ASSUME, ASSERT и HAVOC. Эти аннотации определяются правилами

Если $val(f, s) =$ TRUE, то (ASSUME $f;, s$) $\rightarrow_R$ (OK, s);
Если $val(f, s) =$ FALSE, то (ASSUME $f;, s$) $\rightarrow_R$ (IGNORE, s);
Если $val(f, s) =$ TRUE, то (ASSERT $f;, s$) $\rightarrow_R$ (OK, s);
Если $val(f, s) =$ FALSE, то (ASSERT $f;, s$) $\rightarrow_R$ (FAIL, s);
Если $\gamma \in val(vt(v))$, то (HAVOC $v;, s$) $\rightarrow_R$ ($vv.v$:= γ, s);
(HAVOC $v_1, ..., v_n;, s$) $\rightarrow_R$ (FOREACH v IN $< v_1, ..., v_n >$ DO HAVOC v; END;, s);

Программа. Программа α определяется правилами

($\alpha;, s$) $\rightarrow_R$ (ASSUME ini; CONTROL LOOP;, s);
(CONTROL LOOP;, s) $\rightarrow_R$ (INPUT; p_1; ... p_n; OUTPUT; ASSERT inv; TICK; CONTROL LOOP;, s).

Инструкция CONTROL LOOP; определяет повторяющийся цикл управления. Инструкции INPUT; и OUTPUT; взаимодействуют с окружением, записывая значения во входные переменные и читая значения из выходных переменных соответственно. Инструкция p; выполняет процесс p. Инструкция TICK; увеличивает на единицу значения глобальных и локальных часов. Эти инструкции определяются правилами

(INPUT;, s) $\rightarrow_R$
(FOREACH v IN V_i DO HAVOC v; ih := $con(ih, < vv(v) >)$; END; ASSUME env;, s);
(OUTPUT;, s) $\rightarrow_R$ (FOREACH v IN V_o DO oh := $con(oh, < vv(v) >)$; END;, s);
($p;, s$) $\rightarrow$ (cp := p; $psv(ps(p))$, s);
(TICK;, s) $\rightarrow$ (gc := $gc + 1$; $lc.p_1$:= $lc(p_1) + 1$; ...; $lc.p_n$:= $lc(p_n) + 1;, s$).

Предикаты активности. Они определяются правилами:

((PROC p IN STATE ACTIVE), s) $\rightarrow_R$ ($s.ps(p) \neq$ STOP $\wedge$ $s.ps(p) \neq$ ERROR, s);
(PROC IN STATE ACTIVE, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE ACTIVE), s);
((PROC p IN STATE INACTIVE), s) $\rightarrow_R$ ($s.ps(p) =$ STOP $\vee$ $s.ps(p) =$ ERROR, s);
(PROC IN STATE INACTIVE, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE INACTIVE), s);
((PROC p IN STATE STOP), s) $\rightarrow_R$ ($s.ps(p) = stop, s$);
(PROC IN STATE STOP, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE STOP), s);
((PROC p IN STATE ERROR), s) $\rightarrow_R$ ($s.ps(p) =$ ERROR, s);
(PROC IN STATE ERROR, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE ERROR), s).

Операторы управления состояниями процессов. Они определяются правилами

(STOP PROC $p;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= STOP;, s);
(STOP;, s) $\rightarrow_R$ (STOP PROC $s.cp;, s$);
(ERROR;, s) $\rightarrow_R$ ($lc.(s.cp)$:= 0; $ps.(s.cp)$:= ERROR;, s);
(START PROC $p;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $pso(p)[1];, s$);
(RESTART;, s) $\rightarrow_R$ (START PROC $s.cp;, s$);
(SET STATE $\theta;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $\theta;, s$);
Если $pso.p = con(..., < s.ps(s.cp), \theta >, ...)$,
то (SET NEXT, s) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $\theta;, s$).

Операторы таймаута. Они определяются правилами

(RESET TIMEOUT;, s) $\to_R$ ($lc.(s.cp)$:= 0;, s);
Если (e, s) $\to_R$ (γ, s'), и $s.cl(s.cp) \geq \gamma$, то (TIMEOUT e η, s) $\to_R$ (η, s');
Если (e, s) $\to_R$ (γ, s'), и $s.cl(s.cp) < \gamma$, то (TIMEOUT e η, s) $\to_R$ (OK, s').

Условный оператор. Он определяется правилами

Если (e, s) $\to_R$ (TRUE, s'), то (IF e η_1 ELSE η_2, s) $\to_R$ (η_1, s');
Если (e, s) $\to_R$ (FALSE, s'), то (IF e η_1 ELSE η_2, s) $\to_R$ (η_2, s');
Если (e, s) $\to_R$ (TRUE, s'), то (IF e η, s) $\to_R$ (η, s');
Если (e, s) $\to_R$ (FALSE, s'), то (IF e η, s) $\to_R$ (OK, s');

Блоки. Они определяются правилом

({η_1 ... η_m}, s) $\to_R$ (η_1 ... η_m, s).

Оператор присваивания. Он определяется правилами

Если (e, s) $\to_R$ (FAIL, s'), то (v = e, s) $\to_R$ (FAIL, s');
Если (e, s) $\to_R$ (γ, s'), и $\gamma \neq$ FAIL, то (v = e, s) $\to_R$ ($vv.v$:= γ;, s');

Последовательная композиция. Она определяется правилами

Если (η_1, s) $\to_R$ (γ, s'), и $\gamma \notin$ {FAIL, IGNORE},
то (η_1 η_2 ... η_m, s) $\to_R$ (η_2 ...η_m, s');
Если (η_1, s) $\to_R$ (FAIL, s'), то (η_1 η_2 ... η_m, s) $\to_R$ (FAIL, s').
Если (η_1, s) $\to_R$ (IGNORE, s'), то (η_1 η_2 ... η_m, s) $\to_R$ (IGNORE, s').

Вызов внешней функции. Он определяется правилами

Если $fpar(f) =< v_1, \ldots, v_m >$, FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (f(\(e_1, \ldots, e_m\);, s) $\to_R$
($vv.v_1$:= γ_1; ...; $vv.v_m$:= γ_m; ASSUME $fpre(f)$;
HAVOC f; ASSUME $fpost(f)$;, s_m);
Если $fpar(f) =< v_1, \ldots, v_m >$, FAIL $\notin \{\gamma_1, \ldots, \gamma_{k-1}\}$, γ_k = FAIL, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_k, s_{k-1}) $\to_R$ (γ_k, s_k),
то (f(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_k).

Переменные в выражениях. Правило для переменной имеет вид

(v, s) $\to_R$ ($vv.v, s$).

Операции в выражениях. Определим над множеством O операций языка Reflex предикат $odef \in O \times \Gamma^+ \to bool$, специфицирующий область определения этих операций, и функцию $oval \in O \times \Gamma^+ \to \Gamma$, специфицирующую значение для аргументов из области определения. Тогда правила для префиксной формы o(\(e_1, \ldots, e_m\); операции o местности m имеют вид:

Если FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, $odef(o, \gamma_1, \ldots, \gamma_m)$ = TRUE и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ ($oval(o, \gamma_1, \ldots, \gamma_m), s_m$);
Если FAIL $\notin \{\gamma_1, \ldots, \gamma_{k-1}\}$, γ_k = FAIL, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_k, s_{k-1}) $\to_R$ (γ_k, s_k),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_k).
Если FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, $odef(o, \gamma_1, \ldots, \gamma_m)$ = FALSE и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_m);

Заключение

Reflex имеет очень простой синтаксис. Он не имеет составных типов данных и указателей. Он не имеет операторов итерации и операторов перехода. Он имеет ограниченное количество операций в выражениях и ограниченное число простых типов данных.

Сложность семантики аннотированных Reflex программ обусловлена необходимостью учитывать бесконечность цикла выполнения программы, логику управления переходами процессов из состояния в состояние, взаимодействие процессов между собой и с окружением, ограничения на окружение и запуск программы, временные ограничения на исполнение процессов в определенных состояниях, вызовы внешних функций и инварианты цикла управления.

Предложенная в данной работе операционная семантика аннотированных Reflex программ справляется с этими трудностями, используя понятия глобальных и локальных часов, разделяя переменные на внутренние, входные и выходные, учитывая полные истории изменения значений разделяемых переменных и моделируя ограничения на запуск программы и окружения и инварианты цикла управления через аннотации `ASSUME`, `ASSERT` и `HAVOC`.

Мы планируем использовать эту семантику при доказательстве корректности трансформационного подхода к верификации Reflex программ [6], в частности, для доказательства корректности трансформационной семантики Reflex программ [7].

Список литературы / References

[1] Iec, IEC, “61131-3: Programmable Controllers–Part 3: Programming Languages”, *International Standard, Second Edition, International Electrotechnical Commission, Geneva*, **1** (2003).

[2] Basile F., Chiacchio P., Gerbasio D., “On the Implementation of Industrial Automation Systems Based on PLC”, *IEEE Transactions on Automation Science and Engineering*, **10**:4 (2012), 990–1003.

[3] Travis J., Kring J., *LabVIEW for Everyone: Graphical Programming Made Easy and Fun, Third Edition*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[4] 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. Novosibirsk*, 2010, 35–41.

[5] Buxton J. N., Randell B., *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, NATO Science Committee, Brussels, Scientific Affairs Division, NATO, Rome, Italy, 1970.

[6] Anureev I. S., Garanina N. O., Liakh T. V., Rozov A. S., Zyubin V. E., Gorlatch S., “Two-Step Deductive Verification of Control Software Using Reflex”, *Preliminary Proceedings of A. P. Ershov Informatics Conference (PSI-19). Novosibirsk, Russia, Akademgorodok, Russia, July 2–5*, 2019, 17–30.

[7] Anureev I. S., Garanina N. O., Liakh T. V., Rozov A. S., Schulte H., Zyubin V. E., “Towards Safe Cyber-Physical Systems: the Reflex Language and its Transformational Semantics”, *14th International Siberian Conference on Control and Communications (SIBCON). Tomsk State University of Control Systems and Radioelectronics, Tomsk, April 18-20*, 2019, 1–6.

[8] Zyubin V. E., Liakh T. V., Rozov A. S., “Reflex Language: a Practical Notation for Cyber-Physical Systems”, *System Informatics*, **12** (2018), 85–104.

[9] Norrish M., “C Formalised in HOL”, *Ph.D. thesis. University of Cambridge, Technical Report, UCAM-CL-TR-453*, 1998.

[10] Gurevich Y., Huggins J., “The Semantics of the C Programming Language”, *International Workshop on Computer Science Logic. Lecture Notes in Computer Science*, **702** (1992), 274–308.

[11] Blazy S., Leroy X., “Mechanized Semantics for the Clight Subset of the C Language”, *Journal of Automated Reasoning*, **43**:3 (2009), 263–288.

[12] Nepomniaschy V. A., Anureev I. S., Mikhailov I. N., Promsky A. V., “Towards Verification of C Programs. C-light Language and its Formal Semantics”, *Programming and Computer Science*, **28**:6 (2002), 314–323.

[13] Ellison C., Rosu G., “An Executable Formal Semantics of C with Applications”, *Proc. of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, **47**:1 (2012), 533–544.

Anureev I. S., "Operational Semantics of Annotated Reflex Programs", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 475–487.

DOI: 10.18255/1818-1015-2019-4-475-487

Abstract. Reflex is a process-oriented language that provides a design of easy-to-maintain control software for programmable logic controllers. The language has been successfully used in a several reliability critical control systems, e. g. control software for a silicon single crystal growth furnace and electronic equipment control system. Currently, the main goal of the Reflex language project is to develop formal verification methods for Reflex programs in order to guarantee increased reliability of the software created on its basis. The paper presents the formal operational semantics of Reflex programs extended by annotations describing the formal specification of software requirements as a necessary basis for the application of such methods. A brief overview of the Reflex language is given and a simple example of its use – a control program for a hand dryer – is provided. The concepts of environment and variables shared with the environment are defined that allows to disengage from specific input/output ports. Types of annotations that specify restrictions on the values of the variables at program launch, restrictions on the environment (in particular, on the control object), invariants of the control cycle, pre- and postconditions of external functions used in Reflex programs are defined. Annotated Reflex also uses standard annotations `assume`, `assert` and `havoc`. The operational semantics of the annotated Reflex programs uses the global clock as well as the local clocks of separate processes, the time of which is measured in the number of iterations of the control cycle, to simulate time constraints on the execution of processes at certain states. It stores a complete history of changes of the values of shared variables for a more precise description of the time properties of the program and its environment. Semantics takes into account the infinity of the program execution cycle, the logic of process transition management from state to state and the interaction of processes with each other and with the environment. Extending the formal operational semantics of the Reflex language to annotations simplifies the proof of the correctness of the transformation approach to deductive verification of Reflex programs developed by the authors, transforming an annotated Reflex program to an annotated program in a very limited subset of the C language, by reducing a complex proof of preserving the truth of program requirements during the transformation to a simpler proof of equivalence of the original and the resulting annotated programs with respect to their operational semantics.

Keywords: operational semantics, Reflex language, control system, control software, programmable logic controller, annotation, annotated program

On the authors:
Igor S. Anureev, orcid.org/0000-0001-9574-128X, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS,
6 Acad. Lavrentjev pr., Novosibirsk 630090, Russia, e-mail: anureev@iis.nsk.su

Acknowledgments:
This work was funded by the RFBR according to the research № 17-07-01600 and Funding State budget of the Russian Federation (IA&E project № AAAA-A17-11706061006-6).

Моделирование и анализ информационных систем. Т. 26, №4 (2019), с. 488–501
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 488–501

DOI: 10.18255/1818-1015-2019-4-488-501

UDC 004.052

Safety Analysis of Longitudinal Motion Controllers during Climb Flight

Baar T., Schulte H.

Received September 30, 2019
Revised November 19, 2019
Accepted November 27, 2019

Abstract. During the climb flight of big passenger airplanes, the airplane's vertical movement, i.e. its pitch angle, results from the elevator deflection angle chosen by the pilot. If the pitch angle becomes too large, the airplane is in danger of an airflow disruption at the wings, which can cause the airplane to crash. In some airplanes, the pilot is assisted by a software whose task is to prevent airflow disruptions. When the pitch angle becomes greater than a certain threshold, the software overrides the pilot's decisions with respect to the elevator deflection angle and enforces presumably safe values. While the assistance software can help to prevent human failures, the software itself is also prone to errors and is - generally - a risk to be assessed carefully. For example, if software designers have forgotten that sensors might yield wrong data, the software might cause the pitch angle to become negative. Consequently, the airplane loses height and can - eventually - crash.

In this paper, we provide an executable model written in Matlab/Simulink® for the control system of a passenger airplane. Our model takes also into account the software assisting the pilot to prevent airflow disruptions. When simulating the climb flight using our model, it is easy to see that the airplane might lose height in case the data provided by the pitch angle sensor are wrong. For the opposite case of correct sensor data, the simulation suggests that the control system works correctly and is able to prevent airflow disruptions effectively.

The simulation, however, is not a guarantee for the control system to be safe. For this reason, we translate the Matlab/Simulink®-model into a hybrid program (HP), i.e. into the input syntax of the theorem prover KeYmaera. This paves the way to formally verify safety properties of control systems modelled in Matlab/Simulink®. As an additional contribution of this paper, we discuss the current limitations of our transformation. For example, it turns out that simple proportional (P) controllers can be easily represented by HP programs, but more advanced PD (proportional-derivative) or PID (proportional-integral-derivative) controllers can be represented as HP programs only in exceptional cases.

Keywords: Cyber-Physical System (CPS), Formal Safety Analysis, Hybrid Automaton

For citation: Baar T., Schulte H., "Safety Analysis of Longitudinal Motion Controllers during Climb Flight", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 488–501.

On the authors:
Thomas Baar, orcid.org/0000-0002-8443-1558,
Hochschule für Technik und Wirtschaft (HTW) Berlin, Germany,
Campus Wilhelminenhof, Wilhelminenhofstraße 75A, 12459 Berlin, e-mail: thomas.baar@htw-berlin.de

Horst Schulte, orcid.org/0000-0001-5851-3616,
Hochschule für Technik und Wirtschaft (HTW) Berlin, Germany,
Campus Wilhelminenhof, Wilhelminenhofstraße 75A, 12459 Berlin, e-mail: horst.schulte@htw-berlin.de

1. Flight Control Model of Longitudinal Motion

For a complete description of the airplane motion in the three dimensional space, six variables are needed that denote the degrees of freedom of a rigid body [1]. The airplane motion is calculable by six nonlinear ordinary differential equations (ODEs) of these variables. However, under certain assumptions, the ODEs can be decoupled and linearized into longitudinal and lateral equations. It is common practice to describe the longitudinal motion by a third order state space model [1], [2]:

$$x = [\,\alpha \;\; q \;\; \theta\,]^T \tag{1}$$

The state vector (1) contains the *angle of attack* α, *pitch rate* q, and *pitch angle* θ (cmp. Figure 1).

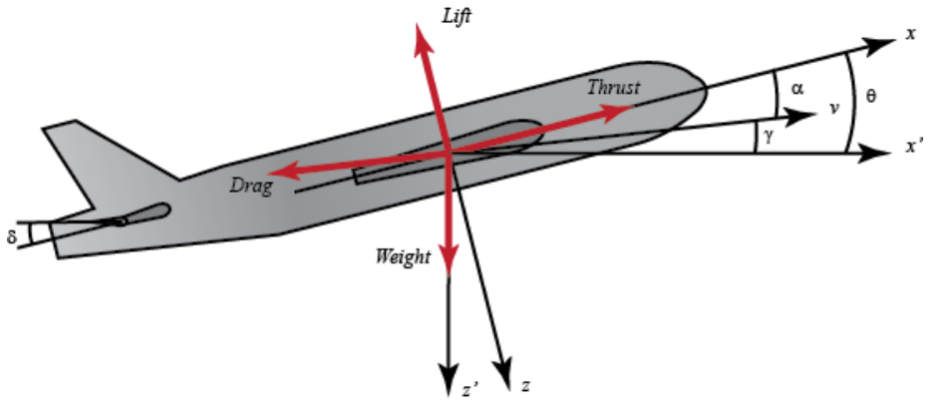

Fig 1. Important parameters of flight model
(Source: *http://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch &section=SystemModeling*)

Based on the assumption that the airplane is in steady-cruise at constant velocity, the longitudinal equations of motion for the airplane in state space form $\dot{x} = f(x,u)$ with the state vector given in (1) and the input $u := \delta$ can be written as

$$\begin{aligned}
\dot{\alpha} &= \mu\Omega\sigma\big[-(C_L + C_D)\alpha + \frac{1}{(\mu - C_L)}q - (C_W \sin\gamma)\theta + C_L\big] \\
\dot{q} &= \frac{\mu\Omega}{2I_{yy}}\Big(\big[C_M - \eta(C_L + C_D)\big]\alpha + \big[C_M + \sigma C_M(1 - \mu C_L)\big]q + (\eta C_W \sin\gamma)\delta\Big) \\
\dot{\theta} &= \Omega q
\end{aligned} \tag{2}$$

where

$$\Omega = \frac{2U}{\bar{c}}, \qquad \mu = \frac{\rho S\bar{c}}{4m}, \qquad \sigma = \frac{1}{1 + \mu C_L}, \qquad \eta = \mu\sigma C_M, \tag{3}$$

with the equilibrium flight speed U and γ as the flight path angle. The parameter ρ denotes the density of air, S denotes the platform area of the wing, $\bar{c}$ denotes the average

chord length and m denotes the mass of the airplane, C_W denotes the coefficient of weight, C_M denotes coefficient of pitch moment, and I_{yy} denotes the normalized moment of inertia. The aerodynamic coefficients of thrust, drag and lift are C_T, C_D, C_L. Based on the above assumptions, the dynamics of the airplane around a stationary operating point $p_c = (\alpha_c, q_c, \theta_c, \delta_c)$ for an equilibrium flight speed is obtained by Taylor linearization of $f(x, u)$

$$A = \frac{\partial f}{\partial x}|_{p_c} = \begin{pmatrix} -0.313 & 56.7 & 0 \\ -0.0139 & -0.426 & 0 \\ 0 & 56.7 & 0 \end{pmatrix}, \qquad B = \frac{\partial f}{\partial u}|_{p_c} = \begin{pmatrix} 0.232 \\ 0.0203 \\ 0 \end{pmatrix} \tag{4}$$

and can be described as follows

$$\begin{aligned} \dot{\alpha} &= -0.313\,\alpha & +56.7\,q & & +0.232\,\delta \\ \dot{q} &= -0.0139\,\alpha & -0.426\,q & & +0.0203\,\delta \\ \dot{\theta} &= & 56.7\,q & & \end{aligned} \tag{5}$$

2. Control loop designed in Matlab/Simulink®

2.1. Description of the designed structure

For the flight model introduced above, we have developed a series of controllers using Matlab/Simulink® whose aim is to keep the pitch angle θ below a maximum value θ_{max} to prevent airflow disruption at the wings. We have selected the period of a climb flight and assume, that the pilot selects a constant deflection angle δ_{man} as manual input, what might cause pitch angle θ to increase. If θ becomes greater than an upper bound, the anti-stall mode is activated and the controller computes a corrective δ_{corr}. When – as a consequence – θ falls again below a lower bound (due to hysteresis, the lower bound is slightly different from upper bound), the anti-stall mode is switched off again and the pilot's δ_{man} become again the input for the airplane. An overview of the entire control system is shown in Figure 2.

Our closed loop control system follows the classical approach and can be split into a plant model and a controller system. The plant model is depicted in Figure 2 by the orange block (1) with input *delta* and outputs q and *theta*. This block realizes the linearized airplane model specified by equation Eq.(5).

The controller system consists of a total of three blocks (cmp. Figure 2) including the following functions: (2) Computation of the anti-stall mode, (3) computation of the delta correction value δ_{corr}, and (4) selection of the delta value δ as actuating input of the airplane model block (1).

The decision making in block (2) is based on a comparison between the currently measured pitch angle θ and the maximum value θ_{max}. To avoid chattering, a distinction is made between a lower $\theta_{max,low}$ and an upper threshold $\theta_{max,up}$. Thereby the anti-stall mode is activated if $\theta > \theta_{max,up}$ is valid and is deactivated again if $\theta < \theta_{max,low}$ is fulfilled.

Block (3) realizes the computation of the delta correction value δ_{corr} by applying the simple proportional (P) control law

$$\delta_{corr} = k_p\, e \tag{6}$$

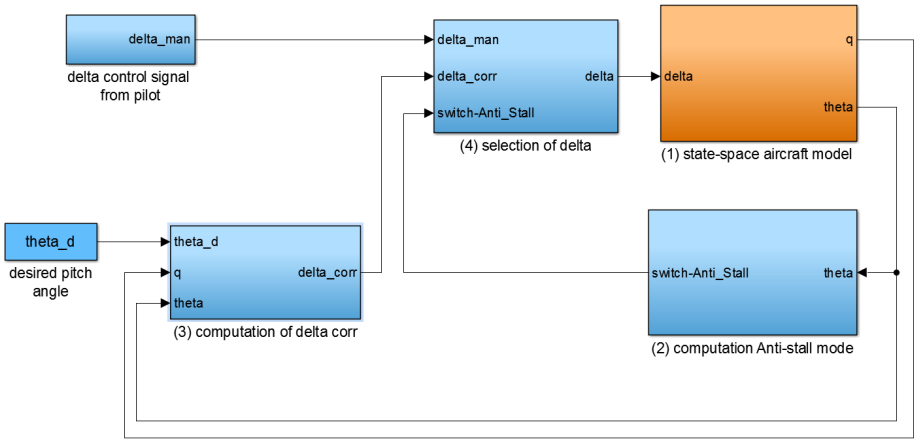

Fig 2. Overview of the entire control system designed in Matlab/Simulink®

where $e = \theta_d - \theta$ denotes the control error and $k_p > 0$ denotes the real-valued controller gain.

In many cases, a system behaves smoother and has more performance if instead the simple P-control law the PD (proportional-derivative) or the PID (proportional-integral-derivative) control law is applied. While the implementation of the PD / PID control law within block (3) in Matlab/Simulink® can be done easily, the transformation of this new Matlab/Simulink®-model into a hybrid program (HP) as input for the theorem prover KeYmaera as described in Section 3. faces some serious problems. These problems are discussed in Section 4.

In block (4), the delivery of the actuating signal either from the manual demand of the pilot δ_{man} or from the anti-stall controller δ_{corr} takes place. The implementation of block (4) is illustrated in detail in Figure 3, where the switching between δ_{man} and δ_{corr} is realized by a crisp mode-dependent function.

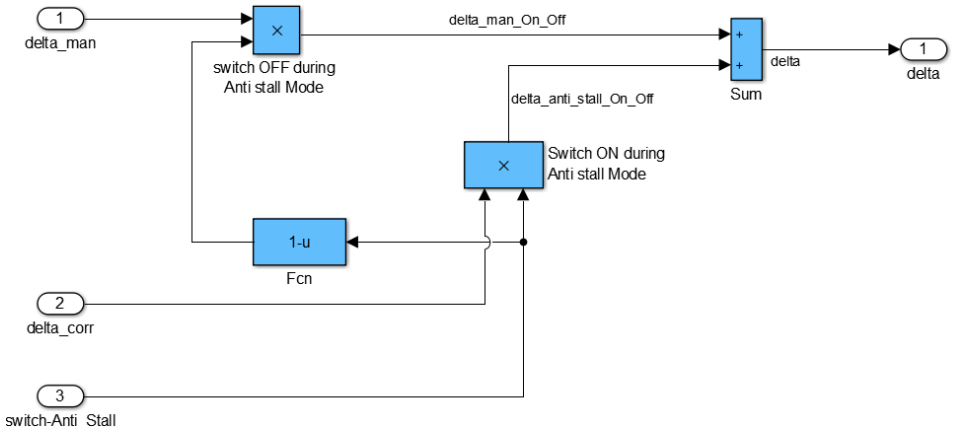

Fig 3. Control block (4) of Figure 2: Selection of the delta value by crisp function

2.2. System analysis by simulation

The standard technique to analyse Matlab/Simulink®-models is *simulation*. For a simulation, concrete values for constants (e.g. k_p) and for input parameters (δ_{man}, θ_d) are chosen and the system is executed. The system behaviour can be depicted by function graphs showing the values of system variables over time (cmp. Figure 4, Figure 5).

We actually simulate two versions of the system: the first version assumes correct measurement of the output θ of block (1), as described above and as depicted in Figure 2.

In the second simulation, we assume that output θ of block (1) is not measured correctly by sensors. This is modelled by applying an error function on θ before it becomes the input for block (2) and block (3). Applying error functions is a standard modelling technique in Matlab/Simulink®.

2.2.1. Assuming correct sensor measuring for θ

The pitch angle θ is one of the outputs of the plant model realized by block (1) and the input for the control loop consisting of blocks (2-4). Based on θ, the input δ (pitch elevator angle) is computed for the next cycle.

Assuming that the angle θ is correctly measured by sensors, the simulation of the system as shown in Figure 4 does not detect any point in time in which pitch angle θ becomes negative. Note that if pitch angle θ would become negative, the airplane would lose height and might eventually crash.

Observing only non-negative values for θ in a simulation is not a guarantee that θ never becomes negative, but it is already a good starting point for formal verification of the system's safety (cmp. Section 3.).

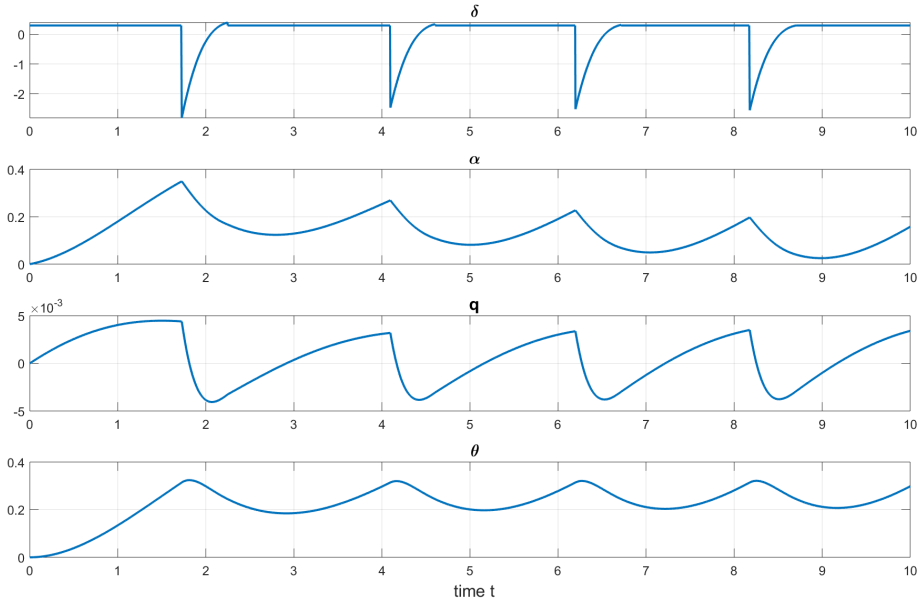

Fig 4. Simulation results for correct (fault free) sensor measuring of θ

2.2.2. Assuming incorrect sensor measuring for θ

When building safety critical systems, engineers should always take into account that sensors might provide wrong data. We have modelled in a second Matlab/Simulink®-model a faulty sensor just by substituting the output θ of the plant model in block (1) by $\theta + \theta_{\text{offset}}$, where θ_{offset} is a predefined constant. When simulating this second model, it can be immediately seen that θ becomes soon negative, i.e. the airplane can lose height. This simulation is shown in Figure 5.

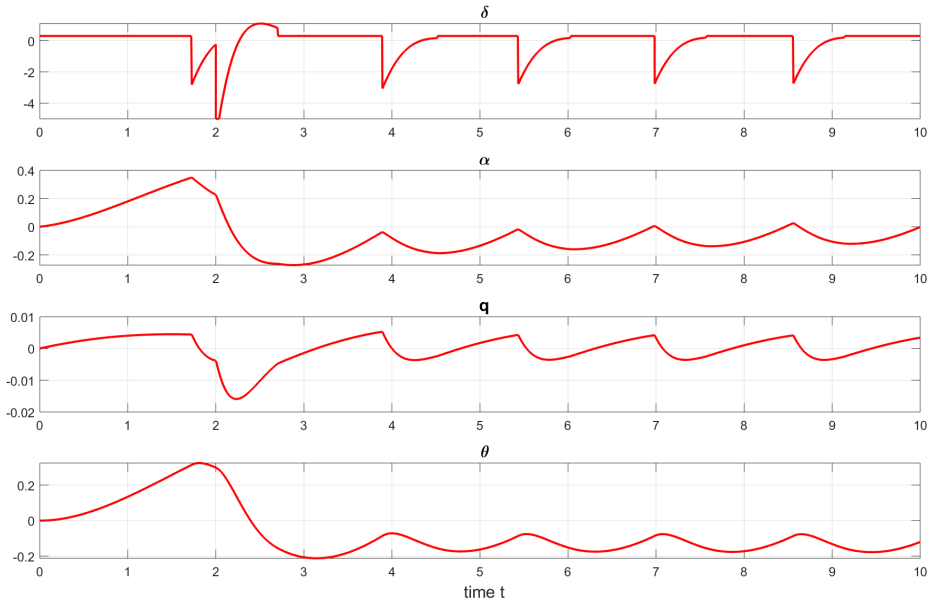

Fig 5. Simulation results for incorrect sensor measuring of θ

3. Logical Analysis of Flight Control Models

As detailed in the previous section, the Matlab/Simulink® toolkit is able to simulate the modelled system. As one can easily see, the airplane might lose height when sensors for measuring pitch angle θ provide wrong data. However, for the opposite case of having a (presumably) correct system, simulation is not a sufficient technique to ensure the correct system behaviour under all possible circumstances. In our case, the correct behaviour means that pitch angle θ remains always positive (recall that we model the phase of climb flight).

In this section, we present a translation of the Matlab/Simulink® model into a hybrid program (HP), a notion similar to well-known hybrid automata [7]. Since the notion of HP is supported by the theorem prover KeYmaera [4, 3], our transformation paves the way to formally verify safety properties of hybrid systems [6, 5].

3.1. KeYmaera

A proof task for KeYmaera has to be formulated in *differential dynamic logic* (DDL), which is an extension of classical *dynamic logic* (DL) [9].

Classical DL allows to prove pre-/post-condition contracts (known from Design-by-Contract) for programs written in a simple while-language, i.e. a programming language supporting the classical concepts of imperative programming, such as *assignment to a variable*, *sequential execution*, *iterative execution*, and *case distinction*. The programming language handled by the theorem prover KeYmaera - known as *Hybrid Program* (HP) - also supports to a certain degree nondeterministic execution, as realized by the statements *nondeterministic assignment* (`x = *`), *nondeterministic choice* ($\alpha \cup \beta$), and *nondeterministic iteration* (`(`α`)*`).

Technically, classical DL is a modal logic with modalities *box* ($[\alpha]\psi$) and *diamond* ($< \alpha > \psi$), where α is a program and ψ a logical formula expressing a property on the state of the machine, on which the program is executed. The state of the machine is defined as the value vector of all program variables occurring in program α. Note that in HP all variables are of type Real and represent only one single scalar value. This is an important difference to Matlab/Simulink®, where a variable can be of type vector or matrix and can represent a list of scalar values or even a (numerically represented) function.

For the rest of the paper, only the box-modality is applied; the formula $[\alpha]\psi$ states that in each possible post-state after program α has terminated the formula ψ holds. Please note that termination of α is not claimed! Please further note that since α can contain nondeterministic statements the execution of α can indeed result in multiple post-states.

The main difference of DDL and DL is that the former supports *continuous evolution* statements. When a continuous evolution statement is executed, the variables $x_1, x_2, \ldots, x_n$ selected by the statement change their values synchronously according to coupled differential equations. The value change is restricted by an optional *evolution constraint* H. Table 1 summarizes the basic statements of KeYmaera's programming language HP.

Table 1. Statements of HP (adapted from [6])

HP Notation	Operation	Effect
$x_1 := \theta_1, \ldots, x_n := \theta_n$	discrete jump	simultaneously assign θ_i to variable x_i
$x := *$	nondet. jump	assign any value to variable x
$x_1' := \theta_1, x_2' := \theta_2, \ldots$ $\ldots, x_n' := \theta_n \ \& \ H$	continuous evo.	differential equations for x_i within evolution domain H (first-order formula)
$?H$	state test	test first-order formula H at current state
$\alpha; \beta$	seq. composition	HP β starts after HP α finishes
$\alpha \cup \beta$	nondet. choice	choice between alternatives HP α or HP β
$\alpha*$	nondet. repetition	repeats HP α n-times for any $n \in \mathbb{N}_0$

For a detailed introduction to DDL, the reader is referred to [6]. In [8], the limitations of HP with respect to expressiveness and maintainability is investigated.

3.2. Flight model as KeYmaera-input

Table 2. AirplaneClimbControl (ACC)

$ACC \equiv$	$(ctrl; plant)*$	(7)
$ctrl \equiv$	$block_2; block_3; block_4$	(8)
$block_2 \equiv$	$(?\theta > \theta_{max,up}; switch_{AntiStall} := 1) \cup$	(9)
	$(?\theta < \theta_{max,low}; switch_{AntiStall} := 0) \cup$	(10)
	$(?(\theta \leq \theta_{max,up} \wedge \theta \geq \theta_{max,low}))$	(11)
$block_3 \equiv$	$\delta_{corr} = k_p * (\theta_d - \theta)$	(12)
$block_4 \equiv$	$\delta = switch_{AntiStall} * \delta_{corr} + \delta_{man} * (1 - switch_{AntiStall})$	(13)
$plant \equiv$	$t := 0;$	(14)
	$(t' = 1,$	(15)
	$\alpha' = -0.313 * \alpha + 56.7 * q + 0.232 * \delta,$	(16)
	$q' = -0.0139 * \alpha - 0.426 * q + 0.0203 * \delta,$	(17)
	$\theta' = 56.7 * q$	(18)
	$\& \; t <= \epsilon)$	(19)

The Matlab/Simulink®-model shown in Figure 2 can be translated into a hybrid program ACC as shown in Table 2. In line (7), the overall structure of ACC is shown as the nondeterministic iteration (operator $*$) over the sequential composition (operator $;$) of subsystems $ctrl$ and $plant$. The control part $ctrl$ in line (8) is sequentially composed of $block_2 - block_4$, while these HP programs tightly correspond to the blocks (2), (3), (4) in the Matlab/Simulink®-model shown in Figure 2.

In $block_2$, the flag $switch_{AntiStall}$ is either switched on (line (9)) or switched off (line (10)) after comparing the current value of pitch angle θ with predefined threshold values $\theta_{max,up}, \theta_{max,low}$. The program structure of $block_2$ is basically an if-then-else and therefore we have to specify in line (11) that in all remaining cases the value of $switch_{AntiStall}$ remains the same.

In $block_3$, the correction value δ_{corr} is computed in terms of a simple proportional (P) controller (line (12)).

In $block_4$, the actual plant input δ is computed by switching between δ_{corr} and δ_{man} depending on the value of $switch_{AntiStall}$ (line (13)).

Subprogram $plant$ is the sequential composition of the reset of auxiliary variable t to 0 (line (14)) and a continuous evolution statement (lines (15) - (19)). This continuous evolution statement encodes exactly the linearized airplane model specified in equation Eq.(5). The domain constraint $t < \epsilon$ (line (19)) together with the ODE $t' = 1$ (line (15)) has the effect that the system remains at longest time ϵ in the evolution state.

3.3. Proof task for correct behaviour

We can now formally formulate the safety property we would like to show for our hybrid program ACC:

$$\theta > 0 \rightarrow [ACC]\,\theta > 0 \tag{20}$$

In words, (20) reads as: whenever the system (including its controller) is started in a situation in which the pitch angle is positive, then after every control loop (which takes mostly time ϵ), the pitch angle remains positive.

Note that it is not the goal of this paper to actually establish a formal proof for (20) using the theorem prover KeYmaera. This task has been postponed to future work.

4. Lessons Learned

In Section 3.2. we presented based on an example a transformation of Matlab/Simulink®-models into input artefacts for the theorem prover KeYmaera, which are written in HP

Source and target notation of this transformation have a semantic gap that cannot always be bridged by a clever encoding in the target notation HP. The most striking difference are the allowed types for variables and the predefined operations on these types. In KeYmaera, every variable is of type *Real*, for which only common arithmetic operation such as $+, -, *, /$ and comparison operation $<, \leq, >, \geq, =$ are provided. In Matlab, variables can be of a numerical type or of an n-dimensional list (aka. vector, matrix, grid) or even of a function type. The list of operations supported by these types is much longer than in HP: arithmetic operation, trigonometric function, matrix multiplication, and many more.

Another important difference is that in HP the derivation operator ' can only be applied in a continuous evolution statement, whose ODEs are restricted to the form $x_i' = \theta_i$, while θ_i must not contain any derivative term. In Matlab, the derivation operator can be applied much more freely, e.g. in the plant model as well as in the controller system.

Let us illustrate these problems with a concrete example. In our Matlab/Simulink®-model presented above, the computation of δ_{corr} is done by applying the P control law $\delta_{corr} = k_p\, e$ (cmp. Eq. (6)). This law is transformed to $block_3$ with implementation $\delta_{corr} = k_p * (\theta_d - \theta)$ (comp. line (12) in Table 2).

Suppose, we would like to substitute in the Matlab/Simulink®-model the P by a PD control law, which would be implemented by

$$\delta_{corr} = k_p\, e + k_d\, \dot{e} \tag{21}$$

where $e = \theta_d - \theta$ denotes the control error and $k_p > 0$ and $k_d > 0$ denote real-valued controller gains. For such a system definition, our transformation would yield for $block_3$ the implementation

$$\delta_{corr} = k_p * (\theta_d - \theta) + k_d * (\theta_d - \theta)^{\cdot} \tag{22}$$

The problem now is that Eq. (22) cannot be directly transformed into a HP since the derivation operator ' must only occur in form var' = <exp> within a continuous evolution statement and this form is not met by Eq. (22).

Fortunately, since θ_d is a predefined constant, we can now rewrite within Eq. (22) subterm $(\theta_d - \theta)^{\cdot}$ by $-\dot{\theta}$ and get

$$\delta_{corr} = k_p * (\theta_d - \theta) - k_d * \dot{\theta} \tag{23}$$

Now we remember the last line of Eq.(5) $\dot{\theta} = 56.7\, q$ and get finally

$$\delta_{corr} = k_p * (\theta_d - \theta) - k_d * 56.7 * q \tag{24}$$

If the PD control law is formulated in this form, it can be directly transformed into HP since not derivation operator is applied any longer.

5. Related Work

Safety analysis of flight control systems has a long tradition in the academic control community. Based on linear models which describe the dynamics of the aircraft for individual degrees of freedom, formal methods of control theory such as stability analysis, model-based fault detection and isolation (FDI) and fault tolerant control (FTC) are applied [16],[29]. Due to the nonlinear behaviour of aircraft dynamics, the assumption of an approximate description with linear models is only valid in a small working range. In the case of an fault, however, this range is left. For this reason, model classes such as quasi-LPV (linear parameter variable) [30] and Takagi-Sugeno systems have been investigated in recent years. These techniques allow to describe nonlinear dynamics in such a way that formal safety system analysis is feasible [31]. In order to be able to compare the different approaches in an objective way, benchmarks have been established in cooperation with industry [20], [17], [18]. For this purpose, generic flight models or models that describe common target aircraft are used.

For the safety analysis of a longitudinal motion controller, a logic based verification technique has been chosen in this paper. The core of our approach is based on differential dynamic logic (DDL) [3], for which the prover KeYmaera has been developed. Numerous case studies from different domains have been conducted, in which KeYmaera proved to be powerful enough to verify real-world cyber-physical systems [22], [23], [24].

An alternative logic-based verification backend would be HyComp [27], which is also able to verify hybrid automata. HyComp is an SMT-based model checker and expects a system description in terms of the HyDI symbolic language.

One of the bottlenecks for applying KeYmaera is the necessity to describe the system (both plant and controller) in form of a hybrid program, which is a rather archaic, text oriented notation. In this paper, we have provided – based on an example – a translation from the notation Matlab/Simulink® into a hybrid program. While this translation is for many constructs straightforward, special attention is needed to translate how the controller computes the corrective input for the plant. In practice, this is often done by applying a P-, PD-, or PID-controller law.

In [6], the integration of a PD-controller is illustrated in Example 9b (page 18), but the PD-controller becomes part of the plant and not of the controller. Thus, to our knowledge, it is still an unsolved problem how to model PD- and PID-controller in terms of hybrid programs.

Instead of using logic-based verification methods, tools such as SpaceEx [25] and Flow* [26] apply *reachability analysis* as verification method. The goal of reachability analysis is to obtain a verification whether a hybrid automaton never reaches an unsafe state configuration [11]. In contrast to logic analysis, the verification of the reachability for hybrid systems is based on the level set techniques [12], which determines an implicit representation of the boundary of this reachable set.

For systems with complex dynamics, some approximation methods are needed for reachability computations [13]: A group of methods seeks an efficient over-approximation of the reachable set for hybrid systems, whose continuous dynamics is defined by linear differential equations. Such systems can be implemented using tools like *d/dt* [14] or the MATLAB-based tool *CheckMate* [28]. Here, sets are represented as convex polyhedra. The propagation of these polyhedra under linear dynamics could result in over-approximations of nonlinear dynamics along each surface of the polyhedra.

Conclusion

In this paper, we have investigated safety critical software for controlling the flight of modern airplanes. Such control software is usually developed using tools such as Matlab/Simulink®. We present a possible controller for the computation of the pitch elevator angle, but this controller has been completely designed by ourselves. The sole purpose of our model is to provide an example at which quality assurance techniques can be applied and demonstrated.

For the controller of our example, we review two main safety properties: Does the controller effectively prevent airflow disruption, which is the main purpose of the controller. However, there is another safety property, which can be easily overlooked when airplane software is hastily developed: Is it possible that the controller software could cause the airplane to lose height, which - eventually - might cause the airplane to crash.

The simulations of our controller suggest, that both safety properties are met. However and not surprisingly, the controller can cause airplane crashes when the sensor measuring the pitch angle θ does not provide correct data.

For the case that the sensor works correctly, we have translated the Matlab/Simulink®-model successfully into a HP model. This paves the way to verify using the theorem prover KeYmaera that for all possible situations the system is safe (not only for the few situations captured by the simulation). Finding an actual proof for the described safety properties was not the goal of this paper.

As discussed in Section 4., a successful transformation is only possible if the source model meets some restrictions. Whenever functions are used in the source model for which there is no corresponding function in HP, the generated target model cannot be parsed by KeYmaera. Likewise, the derivation operator should be used in the source model only at locations, which are mapped to continuous evolutions statements in HP. As the example given in Section 4. illustrates, all other occurrences of the derivation operator have to be manually substituted by equivalent terms prior to transformation, but this seems to be not always possible.

References

[1] Stengel R. F., *Flight Dynamics*, Princeton University Press, 2004.

[2] Yechout T. R., *Introduction to Aircraft Flight Mechanics*, American Institute of Aeronautics & Astronautics, 2003.

[3] Platzer A., *Logical Foundations of Cyber-Physical Systems*, Springer, Heidelberg, 2018.

[4] Platzer A., *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*, Springer, Heidelberg, 2010.

[5] Platzer A., "Logic and Compositional Verification of Hybrid Systems (Invited Tutorial)", *In: Gopalakrishnan G., Qadeer S. (eds) Computer Aided Verification. CAV 2011. LNCS. Springer, Berlin, Heidelberg*, **6806** (2011), 28–43.

[6] Quesel J. D., Mitsch S., Loos S., Aréchiga N., Platzer A., "How to Model and Prove Hybrid Systems with KeYmaera: A Tutorial on Safety", *International Journal on Software Tools for Technology Transfer*, **18**:1 (2016), 67–91.

[7] Henzinger T. A., "The Theory of Hybrid Automata", *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA*, 1996, 278–292.

[8] Baar T., "A Metamodel-Based Approach for Adding Modularization to KeYmaera's Input-Syntax", *Proceedings, 11th A. P. Ershov Informatics Conference, Akademgorodok, Novosibirsk, Russia. 2019.*

[9] Harel D., Kozen D., Tiuryn J., *Dynamic Logic*, MIT Press Cambridge, 2000.

[10] Frehse G., Kekatos N., Nickovic D., Oehlerking J., Schuler S., Walsch A., Woehrle M., "A Toolchain for Verifying Safety Properties of Hybrid Automata via Pattern Templates", *Proceedings, Annual American Control Conference (ACC), Milwaukee, USA*, 2018, 2384–2391.

[11] Alur R., Courcoubetis C., Henzinger T. A., Ho P. H., "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems", *International Hybrid Systems Workshop. LNCS, Springer, Berlin, Heidelberg*, **736** (1991), 209–229.

[12] Osher S., Sethian J. A., "Fronts Propagating with Curvature-dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations", *Journal of Computational Physics*, **79**:1 (1988), 12–49.

[13] Tomlin C. J., Mitchell I., Bayen A. M., Oishi M., "Computational Techniques for the Verification of Hybrid Systems", *Proceedings of the IEEE*, **91**:7 (2003), 986–1001.

[14] Asarin E., Bournez O., Dang T., Maler O., "Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems", *In: Lynch N., Krogh B.H. (eds) Hybrid Systems: Computation and Control. HSCC 2000. LNCS, Springer, Berlin, Heidelberg*, **1790** (2000), 20–31.

[15] Clarke Jr. E. M., Grumberg O., Kroening D., Peled D., Veith H., *Model Checking (second edition)*, MIT Press, 2018.

[16] Chen J., Patton R. J., *Robust Model-Based Fault Diagnosis for Dynamic Systems*, Kluwer Academic Publishers Norwell, MA, USA, 1999.

[17] Goupil P., Marcos A., *Advanced Diagnosis for Sustainable Flight Guidance and Control: The European ADDSAFE Project*, SAE technical paper 2011-01-2804, 2011.

[18] Goupil P., Marcos A., "Industrial Benchmarking and Evaluation of ADDSAFE FDD Design", *In Proc. of 8th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, **45**:20 (2012), 1131–1136.

[19] Grenaille S., Henry D., Zolghadri A., "A method for designing fault diagnosis filters for LPV polytopic systems", *Journal of Control Science and Engineering*, 2008.

[20] Christopher E., Thomas L., Hafid S., "Fault Tolerant Flight Control: A Benchmark Challenge", *Lecture Notes in Control and Information Sciences*, **399** (2010).

[21] Witczak M., "Fault Diagnosis and Fault-Tolerant Control Strategies for Non-Linear Systems", *Lecture Notes in Electrical Engineering, Springer*, **266** (2014), 375–392.

[22] Mitsch S., Loos S. M., Platzer A., "Towards Formal Verification of Freeway Traffic Control", *In Proc. of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems(ICCPS)*, 2012, 171–180.

[23] Jeannin J. B., Ghorbal K., Kouskoulas Y., Gardner R., Schmidt A., Zawadzki E., Platzer A., "A Formally Verified Hybrid System for the Next-Generation Airborne Collision Avoidance System", *In Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015, 21–36.

[24] Platzer A., Quesel J. D., "European Train Control System: A Case Study in Formal Verification", *In: Breitman K., Cavalcanti A. (eds) Formal Methods and Software Engineering. ICFEM 2009. LNCS, Springer*, **5885** (2009), 246–265.

[25] Frehse G., Le Guernic C., Donzé A.,Cotton S., Ray R., Lebeltel O., Ripado R., Girard A., Dang Th. Maler O., "SpaceEx: Scalable Verification of Hybrid Systems", *In: Gopalakrishnan G., Qadeer S. (eds) Computer Aided Verification. CAV 2011. LNCS, Springer*, **6806** (2011), 379–395.

[26] Chen X., Ábrahám E., Sankaranarayanan S., "Flow*: An Analyzer for Non-linear Hybrid Systems", *In: Sharygina N., Veith H. (eds) Computer Aided Verification. CAV 2013. LNCS, Springer*, **8044** (2013), 258–263.

[27] Cimatti A., Griggio A., Mover S., Tonetta S., "HyComp: An SMT-Based Model Checker for Hybrid Systems", *In: Baier C., Tinelli C. (eds) Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer*, **9035** (2015), 52–67.

[28] "Formal Verification of Hybrid Systems Using CheckMate: A Case Study", *Proceedings of the 2000 American Control Conference*, **3** (2000), 1679–1683.

[29] Zolghadri A., "Advanced Model-Based FDIR Techniques for Aerospace Systems: Today Challenges and Opportunities", *Progress in Aerospace Sciences, Elsevier*, **53** (2012), 18–29.

[30] Grenaille S., Henry D., Zolghadri A., "A Method for Designing Fault Diagnosis Filters for LPV Polytopic Systems", *Journal of Control Science and Engineering*, 2008, 1–11.

[31] Witczak M., Dziekan L., Puig V., Korbicz J., "Design of a Fault-Tolerant Control Scheme for Takagi-Sugeno Fuzzy Systems", *In Proc. 16th Mediterranean Conference on Control and Automation*, 2008, 280–285.

Баар Т., Шульте Х., "Анализ безопасности контроллеров продольного движения во время набора высоты", *Моделирование и анализ информационных систем*, **26**:4 (2019), 488–501.

DOI: 10.18255/1818-1015-2019-4-488-501

Аннотация. Во время набора высоты на больших пассажирских самолетах вертикальное движение самолета, то есть его угол наклона, зависит от угла отклонения руля высоты, выбранного пилотом. Если угол наклона становится слишком большим, самолет рискует нарушить воздушный поток на крыльях, что может привести к его падению. В некоторых самолетах пилоту помогает программное обеспечение, задачей которого является предотвращение нарушения воздушного потока. Когда угол наклона становится больше определенного порога, программное обеспечение отменяет решения пилота относительно угла отклонения руля высоты и обеспечивает предположительно безопасные значения. Хотя вспомогательное программное обеспечение может помочь предотвратить человеческие сбои, само программное обеспечение также подвержено ошибкам и, как правило, представляет собой риск для тщательной оценки. Например, если разработчики программного обеспечения забыли, что датчики могут давать неправильные данные, программное обеспечение может привести к тому, что угол наклона станет отрицательным. Следовательно, самолет теряет высоту и может – в конечном итоге – разбиться.

В этой статье мы представляем исполняемую модель, написанную на Matlab/Simulink® для системы управления пассажирским самолетом. Наша модель также учитывает программное обеспечение, помогающее пилоту предотвращать нарушение воздушного потока. При моделировании

набора высоты с использованием нашей модели легко увидеть, что самолет может потерять высоту, если данные, предоставленные датчиком угла наклона, неверны. Для противоположного случая правильных данных датчика, моделирование предполагает, что система управления работает правильно и способна эффективно предотвращать нарушение воздушного потока.
Однако симуляция не является гарантией безопасности системы управления. По этой причине мы переводим Matlab/Simulink®-модель в гибридную программу (HP), т. е. во входной синтаксис средства доказательства теорем KeYmaera. Это открывает путь для формальной проверки свойств безопасности систем управления, смоделированных в Matlab/Simulink®. В качестве дополнительного вклада в эту статью мы обсудим текущие ограничения нашей трансформации. Например, оказывается, что простые пропорциональные (P) контроллеры могут быть легко представлены программами HP, но более продвинутые контроллеры PD (пропорционально-производные) или PID (пропорционально-интегрально-производные) могут быть представлены как программы HP только в исключительных случаях.

Ключевые слова: киберфизическая система (CPS), анализ формальной безопасности, гибридный автомат

Об авторах:
Томас Баар, orcid.org/0000-0002-8443-1558,
Hochschule für Technik und Wirtschaft (HTW) Berlin, Germany,
Кампус Wilhelminenhof, Wilhelminenhofstraße 75A, 12459 Берлин, e-mail: thomas.baar@htw-berlin.de

Хорст Шульте, orcid.org/0000-0001-5851-3616,
Hochschule für Technik und Wirtschaft (HTW) Berlin, Germany,
Кампус Wilhelminenhof, Wilhelminenhofstraße 75A, 12459 Берлин, e-mail: horst.schulte@htw-berlin.de

Моделирование и анализ информационных систем. Т. 26, № 4 (2019), с. 502–519
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 502–519

DOI: 10.18255/1818-1015-2019-4-502-519

УДК 004.052.42

Комплексный подход системы C-lightVer к автоматизированной локализации ошибок в C-программах

Кондратьев Д. А., Промский А. В.

Поступила в редакцию 23 сентября 2019
После доработки 25 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. В ИСИ СО РАН разрабатывается система C-lightVer для дедуктивной верификации C-программ. Исходя из двухуровневой архитектуры системы, входной язык C-light транслируется в промежуточный язык C-kernel. Метагенератор условий корректности принимает на вход C-kernel программу и логику Хоара для C-kernel. Для решения известной проблемы задания инвариантов циклов выбран подход финитных итераций. Тело цикла финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности, а правило вывода для них использует операцию замены rep, выражающую действие цикла в символической форме. Также в нашем метагенераторе внедрен и расширен метод семантической разметки условий корректности. Он позволяет порождать пояснения для недоказанных условий и упрощает локализацию ошибок. Наконец, если система ACL2 не справляется с установлением истинности условия, можно сосредоточиться на доказательстве его ложности. Ранее нами был разработан способ доказательства ложности условий корректности для системы ACL2. Необходимость в более подробных объяснениях условий корректности, содержащих операцию замены rep, привела к изменению алгоритмов генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных условий корректности. В статье представлены модификации данных алгоритмов. Эти изменения позволяют пометить исходный код функции rep семантическими метками, извлекать семантические метки из определения rep, а также генерировать описание условия исполнения инструкции break.

Ключевые слова: дедуктивная верификация, семантическая метка, локализация ошибок, C-lightVer, ACL2, метагенератор условий корректности, финитная итерация, стратегия доказательства

Для цитирования: Кондратьев Д. А., Промский А. В., "Комплексный подход системы C-lightVer к автоматизированной локализации ошибок в C-программах", *Моделирование и анализ информационных систем*, **26**:4 (2019), 502–519.

Об авторах:
Кондратьев Дмитрий Александрович, orcid.org/0000-0002-9387-6735, аспирант,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: apple-66@mail.ru

Промский Алексей Владимирович, orcid.org/0000-0002-5963-2390, канд. физ.-мат. наук,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: promsky@iis.nsk.su

Благодарности:
Исследование выполнено при частичной финансовой поддержке РФФИ в рамках научного проекта № 17-01-00789.

Введение

Автоматизация верификации С-программ – актуальная проблема современного программирования. В ИСИ СО РАН разрабатывается система C-lightVer [17] для автоматизированной дедуктивной верификации С-программ [16]. Входной язык C-light – значительное подмножество стандарта С99. В языке C-light было выделено ограниченное ядро – язык C-kernel, обладающий непротиворечивой аксиоматической семантикой. Процесс верификации разбивается на три этапа. На первом этапе аннотированная C-light программа транслируется в эквивалентную программу на языке C-kernel. Далее, с помощью аксиоматической семантики выводятся условия корректности (УК) [2]. Затем происходит доказательство полученных УК с помощью системы автоматизированного доказательства теорем. При разработке системы C-light возникла задача расширения генератора УК. Для решения задачи была использована концепция метагенерации УК [26]. Правила вывода аксиоматической семантики поступают на вход метагенератору и сопоставляются с программными конструкциями для вывода УК [16].

Правило вывода условий корректности для циклов языка C-kernel основано на использовании инварианта [24]. Инвариант цикла – это утверждение, которое истинно как перед исполнением цикла, так и для каждой итерации цикла, и обеспечивает корректность на выходе из цикла. Но в общем случае генерация инварианта цикла – алгоритмически неразрешимая задача. В проекте C-light мы реализовали обработку циклов специального вида – финитных итераций над структурами данных [27]. Тело цикла финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности. Чтобы избежать задания инвариантов для таких циклов, в системе C-lightVer реализован символический метод верификации финитных итераций [14]. В данном методе для таких итераций используется специальное правило вывода УК, основанное на функции *rep*, называемой операцией замены и выражающей действие цикла в символической форме.

Функция *rep* определяется рекурсивно по номеру итерации. В ходе нашей работы был разработан алгоритм автоматической генерации операции замены [15]. Метод метагенерации позволил дополнить систему C-lightVer правилом вывода для финитной итерации [14]. Если C-kernel программа содержит финитную итерацию, то метагенератор порождает для нее условия корректности, основанные на операции замены. В системе C-lightVer используется система автоматизированного доказательства теорем ACL2 [25]. Но использование классической индукции в системе ACL2 недостаточно для автоматического доказательства содержащих *rep* УК. Ранее нами были разработаны стратегии доказательства [15, 18], основанные и на индукции, и на структуре УК, и на структуре финитной итерации.

Для решения проблемы автоматизации дедуктивной верификации необходимо автоматизировать аннотирование циклов инвариантами, доказательство УК и локализацию ошибок в случае ложных УК [9]. Система C-lightVer использует комплексный подход к автоматизированной дедуктивной верификации С-программ. Данный подход включает символический метод верификации финитных итераций [27], который позволяет избежать задания инвариантов циклов, реализующих финитные итерации, стратегии доказательства для проверки УК на истинность [15, 18] и метод семантической разметки Денни и Фишера для локализации ошибок [5].

Денни и Фишер предложили добавить в правила вывода УК семантическую разметку для объяснения результата применения правила. Система C-lightVer была дополнена такими правилами с помощью метода метагенерации [14]. Ключевая особенность подхода Денни и Фишера состоит том, что различные подформулы располагаются на специальных позициях в правилах вывода, и, исходя из этого, генератор УК добавляет соответствующие метки к ним. Для генерации объяснений УК метки извлекаются из них, сортируются по номерам строк и переводятся в текст на естественном языке. Чтобы в таком тексте номера строк были указаны именно для C-light программы, был создан протокол, позволяющий от конструкций C-kernel программы вернуться к конструкциям исходной C-light программы. Протокол реализован с помощью транслятора из C-light в C-kernel и обратного транслятора [24]. Транслятор добавляет в код метаинформацию, которая используется обратным транслятором. Он устанавливает соответствие между диапазонами номеров строк обеих программ. Такие диапазоны являются результатом анализа семантических меток из УК.

Описанные методы анализа и верификации программ были реализованы в системе C-light ранее. Но появляется задача разбора случаев, если системе ACL2 не удалось доказать истинность УК. Отметим, что доказательство ложности УК гарантирует наличие ошибки в программе или в ее спецификациях. В таком случае модуль локализации ошибок может сообщать о несоответствии программы и спецификации. Поэтому для более точного анализа УК необходимо разработать стратегии проверки их ложности. Так как переменные ACL2 находятся под неявным квантором всеобщности, то доказательство отрицания УК приводит к появлению квантора существования. Поэтому наши стратегии [15, 18] не подходят для доказательства отрицания УК. Проблема состоит также в том, чтобы стратегия проверки ложности работала в случае цикла с оператором `break`. Ранее нами был разработан способ доказательства ложности условий корректности для системы ACL2 [17]. Другой проблемой является использование семантических меток для объяснения УК, содержащего функцию *rep*. Согласно методу Денни и Фишера меткой помечается вызов функции *rep*, который не содержит информации о структуре цикла. Для генерации более подробных объяснений условий корректности, содержащих операцию замены, в статье представлен разработанный нами способ помечать семантическими метками код функции *rep*. Использование семантических меток в определении функции *rep* привело к изменению алгоритмов генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных условий корректности. В статье представлены разработанные нами модификации данных алгоритмов.

Данная статья имеет следующую структуру. В разделе 1. описаны методы, реализованные ранее в системе C-lightVer и составляющие основу комплексного подхода к дедуктивной верификации C-программ. В разделах 2. и 3. приведены разработанные нами ранее стратегии доказательства. В разделе 2. описана вспомогательная стратегия проверки формулы на истинность, а в разделе 3. – стратегия проверки ложности УК. В разделе 4. представлены наши новые результаты, расширяющие комплексный подход системы C-lightVer. В разделе 4. приведены модифицированные алгоритмы генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных УК. В разделе 5. показан пример, де-

монстрирующий применение наших новых результатов. В заключении дан список наших новых результатов и описаны наши планы.

Обзор литературы. Существуют различные методы автоматизации дедуктивной верификации программ с циклами. Но в отличие от символического метода верификации финитных итераций, данные методы основаны на генерации инвариантов циклов в определенных случаях. Ли и др. [22] разработали обучающийся алгоритм генерации инвариантов циклов, но их метод не допускает наличие оператора `break` в теле цикла, а также операций над массивами. Туэрк [30] предложил использовать пред- и постусловия для циклов типа `while`, однако их должны задавать пользователи. Галеотти и др. [8] улучшили известный метод изменения постусловия комбинацией генерации тестов и динамическим нахождением инварианта. Тем не менее с помощью этого подхода им не удалось вывести полный инвариант в программах сортировки. Кроме того, Галеотти и др. [8] не доказывали свойство, что отсортированный массив является перестановкой исходного. Сривастава и др. [29] предложили метод, основанный на шаблонах инвариантов, предоставляемых пользователями. Но этот метод не позволил использовать в постусловии свойство перестановочности, необходимое для задания свойства отсортированности. В экспериментах по верификации программ сортировки Сривастава и др. задавали постусловие с использованием более слабого свойства, чем свойство перестановочности. Ковач [20] разработала метод автоматической генерации инвариантов цикла для P-разрешимых циклов. Главные отличия данного метода от символического метода верификации финитных итераций состоят в ограничениях на тело цикла: 1) правые операнды инструкций присваивания должны иметь вид полиномов, 2) в методе [20] не рассматривается инструкция `break`.

Для сравнения с комплексным подходом, применяемым в системе C-lightVer, рассмотрим два класса исследований, основанных на стратегиях автоматизации доказательства по индукции.

В обзорной работе [11] рассматривается класс исследований, основанный на стратегиях генерации таких вспомогательных лемм, которые могут помочь доказать исходную теорему. Свойство полноты для таких методов невозможно гарантировать, а вопрос корректности сводится только к вопросу корректности работы применяемых систем автоматизированного доказательства. Примером реализации такого подхода является система ACL2(ml) [10]. Генерация вспомогательных лемм в данной системе основана на комбинации двух методов. Первым методом является распознавание шаблонов доказательств с помощью статистического машинного обучения. Вторым методом служит символический метод нахождения аналогичной леммы. Но теории предметной области могут серьезно отличаться для разных программ. Поэтому методы машинного обучения плохо подходят для доказательства УК. Отметим, что Регеру и др. для проекта по расширению возможностей системы Vampire [28] пришлось разрабатывать новые стратегии автоматизации доказательства по индукции.

Другой класс исследований основан на узкоспециализированных стратегиях, ориентированных на упрощение формальной верификации [32]. Отметим, что наличие циклов приводит к необходимости автоматизации доказательства по индукции [23]. Так как для решения этой проблемы нами был разработан комплексный подход, основанный на стратегиях доказательства, то применяемый в системе

C-lightVer подход можно отнести к данному классу. Кроме него в качестве примера можно рассмотреть метод формальной верификации CLP [1]. Он основан на моделировании программы логическими конструкциями. Для обработки такой модели необходимо использовать специальные стратегии. Но стратегии проекта CLPT [1] ориентированы только на обработку предикатов. Также в качестве примера можно рассмотреть систему AstraVer [6] и используемый в ней метод лемма-функций [31]. Данная стратегия основана на задании спецификаций определенного вида, что облегчает задачу по сравнению с заданием инвариантов, но не позволяет достигнуть полной автоматизации. Подобная стратегия применена и в проекте по расширению возможностей системы Frama-C [3].

Дедуктивная верификация программ основывается на построении и выводе УК. Но для локализации ошибок построения и вывода УК недостаточно, необходимо произвести анализ УК, объясняющий структуру УК. Актуальность проблемы автоматизации локализации ошибок при использовании дедуктивной верификации была продемонстрирована на примере обучения студентов применению системы AutoProof для верификации Eiffel-программ [4, 12]. Кенигхофер и др. [19] реализовали в системе Frama-C автоматизированную локализацию ошибок в C-программах с помощью дедуктивной верификации. Их подход основан на внесении изменений в выражения верифицируемой программы. Но в их подходе не поддерживается полная автоматизация, так как инварианты циклов задает пользователь системы верификации. Рассмотрим два других проекта, основанных на формальном подходе к локализации ошибок. В проекте Centaur [7] УК анализируются для поиска условных выражений из исходных условных операторов и циклов. В проекте Лейно [21] базовая логика расширена метками, предоставляющими пригодную для объяснения семантическую информацию. В проекте Centaur [7] используются некоторые алгоритмы из области отладки программ. В проекте Лейно [21] метки вводятся на этапе трансляции в промежуточный язык, к которому применяется стандартный генератор. Также в этих двух проектах используются более простые, чем C, входные языки.

1. Методы комплексного подхода C-lightVer

Для описания основы комплексного подхода к дедуктивной верификации C-программ рассмотрим методы, реализованные ранее в системе C-lightVer.

Метод метагенерации условий корректности. Так как аксиоматическая семантика представляет собой набор правил вывода и аксиом для всех конструкций языка программирования, то для вывода УК метагенератор принимает в качестве аргумента правила вывода и аксиомы аксиоматической семантики. Они задают собой шаблоны для сопоставления с программными конструкциями [16]. Основой языка шаблонов является логика первого порядка и грамматика языка C. В выражениях этого языка сохраняются нетерминальные символы (неинтерпретированные предикатные символы и “фрагментные переменные” [26] для обозначения фрагментов кода). Принадлежность метаданных определенному классу в языке шаблонов

задается в явном виде [16]. Например, конструкция $any\_code(S)$ может соответствовать любой последовательности (включая пустую) конструкций языка программирования. Для описания конструкции $vector\_substitution$ введем обозначения. Пусть вектор $v = \langle v_1 \dots v_k \rangle$ и для каждого $j (1 \le j \le k)$ выражение $expr_j$ является результатом замены всех вхождений терма $vector\_element$ в выражении $expr$ на v_j. Тогда $vector\_substitution(T,\ vec,\ expr)$ обозначает одновременную замену для каждого $j (1 \le j \le k)$ всех вхождений v_j в формуле T на выражение $expr_j$.

Метод семантической разметки. Денни и Фишер [5] предложили добавить в правила Хоара семантическую разметку для объяснения результата применения правила. Будем использовать для помеченных термов $\lceil t \rceil^l$, означающее, что терму t сопоставляется метка l. Метки имеют вид $c(o, n)$, где c – концепция (тип метки), o – диапазон строк, n – дополнительная информация.

Для каждого типа метки пользователь системы верификации задает шаблон текста. Такие текстовые шаблоны подаются на вход метагенератору. Они подобны форматным строкам в языке Си, так как при их задании можно использовать специальные символы для обозначения диапазона строк.

Для поддержки произвольных концепций меток в проекте C-light язык описания правил вывода был расширен специальной конструкцией $label$ [14], используемой для описания меток. Конструкция $label$ имеет вид $(label\ t\ c)$, где t — терм, к которому приписана метка, а c – строка (тип метки).

Денни и Фишер предложили извлекать метки из УК в порядке увеличения номеров соответствующих метке строк [5]. Так формируется список меток, используемый для генерации объяснения УК. В системе C-lightVer реализован алгоритм извлечения меток из УК [14], отличающийся от подхода Денни и Фишера. Этот алгоритм основан на представлении УК в виде дерева, которое обходится в глубину. Метки добавляются в список, используемый для генерации объяснения УК, в порядке такого обхода.

Таким образом, результатом работы алгоритма извлечения меток из УК является список меток, используемый для генерации объяснения УК. После извлечения меток из УК происходит генерация объяснения УК. Алгоритм генерации объяснения УК основан на последовательном обходе полученного списка меток. Для каждой посещенной при обходе метки текст ее заполненного номерами строк шаблона добавляется к тексту, объясняющему УК.

Символический метод верификации финитных итераций. Рассмотрим инструкцию вида $for\ x\ in\ S\ do\ v := body(v, x)\ end$, где S – структура данных, x – переменная типа “элемент S”, v – вектор переменных цикла, который не содержит x, и $body$ представляет тело цикла, которое не изменяет x и которое завершается для каждого $x \in S$. Как описано ниже структура S может быть изменена. Тело цикла может содержать только инструкции присваивания, инструкции if (возможно вложенные) и инструкции $break$. Такие инструкции называются допустимыми конструкциями тела цикла [15]. Такие циклы for называются финитными итерациями. Пусть v_0 – вектор значений переменных v до исполнения цикла. Чтобы выразить эффект финитной итерации определим операцию замены $rep(n, v, S, body)$, где $rep(0, v, S, body) = v_0$, $rep(i, v, S, body) = body(rep(i-1, v, S, body), s_i)$ для каждого

$i = 1, 2, \ldots, n$. Если оператор выхода из цикла сработал на итерации i $(1 \le i \le n)$, то финитная итерация продолжает свое исполнение, но вектор v не изменяется: $\forall j (i \le j \le n)\ rep(i, v, S, body) = rep(j, v, S, body)$.

Рассмотрим правило вывода, предложенное для финитной итерации и расширенное семантическими метками, на языке описания шаблонов:

```
{P} prog {(vector_substitution(Q, v,
                  (label rep_iter rep(n, v, S, body).vector_element))}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

где конструкция $admissible\_construct(i, n, v, S, body)$ соответствует допустимой конструкции тела цикла, int_var соответствует целочисленной переменной. Конструкция $vector\_substitution(Q,\ v,\ rep(n,\ v,\ S,\ body).vector\_element)$ обозначает одновременную заменую для каждого $t (1 \le t \le length(v))$ всех вхождений v_t в формуле Q на $rep(n,\ v,\ S,\ body).v_t$. Рекурсивное определение допустимой конструкции описано в [15]. Алгоритмы сопоставления этих шаблонов и программных конструкций были реализованы в системе C-lightVer [14–16]. Правило вывода для обратной итерации определено подобным образом.

Автоматическая генерация операции замены. Генерация операции замены основана на трансляции [15] допустимых конструкций тела цикла в конструкции ACL2 [25]. Рассмотрим конструкцию $(b* \ (\ldots (var\ expr) \ldots)\ result)$.

Конструкция вида $(var\ expr)$ означает связывание переменной var со значением выражения $expr$. Выражение $expr$ может зависеть от связанных ранее переменных. Значением $b*$ является значение $result$, которое может зависеть от связанных переменных. Значения переменных вектора v соответствуют значениям полей структуры fr типа $frame$. Поэтому для моделирования изменения значения переменной вектора v мы связываем объект fr с новым объектом, который отличается от старого значением соответствующего поля. Для моделирования выхода из цикла мы используем булевское поле *loop-break* объекта fr. Это поле истинно только после срабатывания `break`. Для моделирования *break* мы используем связывание вида $((when\ t)\ fr)$. Так как в этом случае условием $when$ является t, т.е. "истина", то такое связывание прекращает исполнение текущего блока $b*$ и возвращает fr.

2. Стратегия доказательства формул с операцией замены

Для описания автоматизации локализации ошибок в системе C-lightVer рассмотрим разработанную нами ранее и приведенную в предыдущей работе [17] стратегию доказательства формул с операцией замены.

Входными аргументами данной стратегии являются импликация ψ, содержащая применение функции $rep(n, \ldots)$, и ее посылка ϕ. Попытаемся доказать формулу

$$\phi \to rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}1)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-1*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации ψ описывает один из случаев, когда при исполнении цикла происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-1*) и индукцию по n.

Если система ACL2 не доказала (ψ-*lemma-1*), то попытаемся доказать формулу

$$\phi \to \neg rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}2)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-2*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации ψ описывает один из случаев, когда при исполнении цикла не происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-2*) и индукцию по n.

Отметим, что стратегия доказательства формул с операцией замены похожа на описанную в предыдущей работе [15] стратегию доказательства УК для программ с инструкцией выхода из цикла. Во-первых, обе стратегии основаны на использовании значения поля *loop-break*. Во-вторых, обе стратегии полностью автоматизированы, так как при применении данных стратегий от пользователя системы верификации не требуется предоставлять дополнительные данные. В-третьих, обе стратегии являются эвристическими методами доказательства, т.е. применение данных стратегий к истинной формуле может не привести к успешному доказательству.

Поэтому опишем различия между стратегиями. Во-первых, стратегия доказательства формул с операцией замены принимает в качестве аргумента любую импликацию, содержащую *rep*. Стратегия доказательства УК для программ с инструкцией выхода из цикла [15] анализирует только постусловие программы. Во-вторых, стратегия доказательства УК для программ с инструкцией выхода из цикла [15] генерирует лемму в виде конъюнкции. Первый элемент такой конъюнкции – это УК, второй – равенство посылки импликации из постусловия и значения поля *loop-break*. Такая структура генерируемой леммы обусловлена тем, что стратегия доказательства УК для программ с инструкцией выхода из цикла [15] применяется только тогда, когда постусловие программы представляет собой конъюнкцию импликаций, т.е. представляет собой разбор случаев, где каждый случай описывается посылкой импликации. Поэтому стратегия доказательства УК для программ с инструкцией выхода из цикла [15] генерирует лемму более сложной структуры, чем стратегия доказательства формул с операцией замены.

3. Стратегия проверки ложности недоказанного УК

Для описания автоматизации локализации ошибок в системе C-lightVer рассмотрим разработанную нами ранее и описанную в предыдущей работе [17] стратегию доказательства ложности недоказанного УК.

Входным аргументом данной стратегии является УК (формула ω), содержащая применение функции $rep(n, \ldots)$ и имеющая следующую структуру:

$$\forall x_1 \ldots x_n \ (\phi_1(x_1 \ldots x_n) \to \psi_1(x_1 \ldots x_n)) \wedge \ldots \wedge (\phi_m(x_1 \ldots x_n) \to \psi_m(x_1 \ldots x_n)).$$

Все переменные формулы находятся под квантором всеобщности из-за ориентированности данной стратегии на систему ACL2. Отметим, что в определении любой

из формул ϕ_i или ψ_i могут использоваться не все переменные из набора $x_1 \dots x_n$. Но мы обозначили в качестве параметров весь набор переменных, так как в определении любой формулы можно ввести использование любой новой переменной, записав формулу в конъюнкции с этой переменной. Такое преобразование возможно, так как весь набор переменных находится под квантором всеобщности. Поэтому без ограничения общности можно обозначать зависимость от всего набора переменных. Известно, что формула ложна тогда и только тогда, когда ее отрицание истинно. Поэтому докажем формулу $\neg\omega$:

$$(\exists x_1 \dots x_n\ (\phi_1(x_1 \dots x_n) \wedge \neg\psi_1(x_1 \dots x_n))) \vee \dots \vee (\exists x_1 \dots x_n\ (\phi_m(x_1 \dots x_n) \wedge \neg\psi_m(x_1 \dots x_n))).$$

Для каждого $i(1 \le i \le m)$ подформулу $\exists x_1 \dots x_n\ (\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$ обозначим как T_i. Для доказательства формулы $\neg\omega$ достаточно доказать любую формулу $T_i(1 \le i \le m)$.

Поэтому наша стратегия проверки ложности УК состоит в том, чтобы применять специальную процедуру доказательства к формулам T_i. Если удалось доказать любую формулу T_i, то результатом стратегии будет сообщение о ложности УК `Verification condition is false`. Если не удалось доказать ни одну формулу T_i, то результатом стратегии будет сообщение о неизвестном результате `Unknown`.

Опишем специальную процедуру доказательства. Рассмотрим применение данной процедуры к формуле $T_i(1 \le i \le m)$. Для доказательства формулы $\exists x_1 \dots x_n$ $(\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$ достаточно доказать формулу $(\exists x_1 \dots x_n\ \phi_i(x_1 \dots x_n))$ $\wedge(\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n)))$ (формула T_i'). Обозначим как U_i формулу $\exists x_1 \dots x_n\ \phi_i(x_1 \dots x_n)$. Обозначим как V_i $\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n))$. Специальная процедура доказательства сводится к проверке истинности U_i и к доказательству V_i.

Как мы описали ранее в предыдущей работе [17], истинность формулы U_i проверяется пользователем. Таким образом, наша стратегия проверки ложности УК является не автоматической, а автоматизированной. Мы отказываемся от автоматизации процесса доказательства U_i по следующим причинам. Во-первых, для генерации УК мы вычисляем слабейшее предусловие [15]. Поэтому формула U_i не может содержать операцию замены. Значит, формула U_i может содержать только известные пользователю функции из спецификации. Во-вторых, практически все известные нам системы доказательства имеют проблемы при автоматическом доказательстве формулы с квантором существования. В-третьих, наша эвристика основана на гипотезе о простой посылке и сложном заключении импликаций, сгенерированных в результате вычисления слабейшего предусловия. По нашей гипотезе мы автоматизируем более сложную задачу доказательства V_i. Если пользователь не сообщил об истинности U_i, то мы полагаем, что формулу T_i не удалось доказать.

Если пользователь сообщил об истинности U_i, то мы пытаемся доказать V_i [17]. Сначала рассмотрим случай, когда формула V_i не содержит применения функции *rep*. Так как система ACL2 ориентирована на доказательство формул, где все переменные находятся под квантором всеобщности, то попытаемся автоматически доказать формулу V_i в системе ACL2. Теперь рассмотрим случай, когда формула V_i содержит применение функции $rep(n, \dots)$. Так как пользователь не знает кода

функции rep, то в этом случае возможно только автоматическое доказательство. Также проблемой является возможный выход из цикла, усложняющий код функции rep. Поэтому попытаемся доказать V_i, используя стратегию автоматического доказательства содержащей операцию замены формулы из раздела 2.

4. Дополнение семантическими метками определения операции замены

В данном разделе представим наши новые результаты, которыми являются модифицированный алгоритм генерации операции замены, модифицированный алгоритм извлечения семантических меток и модифицированный алгоритм генерации объяснений недоказанных условий корректности.

Описанное в разделе 1. расширение метода семантической разметки позволило пользователю системы верификации задавать свои концепции семантических меток и снабжать ими правила вывода [14]. Но возникла необходимость еще в одном расширении метода семантической разметки.

Рассмотрим случай, когда УК содержит применение операции замены. Рассмотренное правило вывода для операции замены помечает применение функции rep в УК специальной семантической меткой. Но такая метка не может предоставить информацию о структуре финитной итерации, которую она помечает. Объяснение УК в таком случае будет содержать текст о том, что применение функции rep выражает действие цикла. Но такой текст не содержит информацию о структуре цикла.

Денни и Фишер предложили способ снабжать семантическими метками формулы, но они не предложили способа снабжать семантическими метками определения применяемых в формулах функций. В случае применения операции замены возникла необходимость исправить этот недостаток метода семантической разметки. Поэтому нами было предложено решение расширить метод семантической разметки, снабжая семантическими метками определение операции замены. Для реализации этого подхода нами был предложен способ генерации снабженного метками определения функции rep, способ извлечения меток из определения функции rep и способ генерации текста для списка извлеченных меток. Рассмотрим эти способы.

Модифицированный алгоритм генерации операции замены. Для дополнения метками определения функции нужно найти такой способ, который не нарушает семантику языка задания этого определения, и который позволяет упростить извлечение меток из этого определения. Поэтому предложенный нами способ генерации снабженного метками определения функции rep основан на возможностях, предоставляемых языком системы ACL2.

Наш способ представляет собой модификацию алгоритма автоматической генерации операции замены [15] из раздела 1. Тот алгоритм основан на моделировании последовательности конструкций цикла с помощью последовательности связываний переменной fr с новыми значениями. Но конструкция $b*$ позволяет задать не только последовательное связывание переменных со значением, но и одновременное связы-

вание переменных со значениями. Для одновременного связывания двух переменных с двумя значениями удобно использовать конструкцию *cons*.

Введем переменную *label*, соответствующую семантической метке. Каждое связывание переменной *label* со значением соответствует своей конструкции цикла и происходит одновременно со связыванием переменной fr с новым значением. Когда мы генерируем связывание, моделирующее очередную конструкцию цикла, мы связываем с новым значением не только переменную fr, но и переменную *label*.

Рассмотрим, что представляет собой значение, связываемое с переменной *label*. Оно представляет собой список следующего вида

$$(list\ 'concept\ begin\ end\ 'break\_path),$$

где

- concept – одна из концепций меток. Для помечающих код меток заранее задан определенный набор концепций (типов) меток. Рассмотрим эти концепции:
  - $empty\_stmt$ – концепция метки, соответствующей пустому оператору;
  - $break\_stmt$ – концепция метки, соответствующей инструкции `break`;
  - $assign\_stmt$ – концепция метки, соответствующей присваиванию;
  - if_stmt – концепция метки, соответствующей сокращенному `if`;
  - $full\_if\_stmt$ – концепция метки, соответствующей инструкции `if`;
  - $block\_stmt$ – концепция метки, соответствующей блоку.

  Отметим, что концепции меток однозначно соответствуют видам допустимых конструкций тела финитной итерации;
- begin – начало диапазона строк кода соответствующей метке конструкции;
- end – конец диапазона строк кода соответствующей метке конструкции;
- $'break\_path$ – опциональный элемент списка, который присутствует в списке тогда и только тогда, когда соответствующая метке конструкция лежит на пути к одному из операторов выхода из цикла.

Такая конструкция представляет собой запись семантической метки.

Также нами был предложен способ помечать специальной семантической меткой условие инструкции `if`. Для этого выражение e, соответствующее такому условию, записывается как результат конструкции $b*$, но переменная fr связывается только своим значением, так как при вычислении условия инструкции `if` значения переменных не изменяются. В такой конструкции происходит связывание только переменной *label*, а значением этой конструкции является значение e. Переменная *label* связывается в такой конструкции со списком, реализующим семантическую метку. Такой список соответствует введенному ранее формату, но в качестве задающего концепцию первого элемента списка используется if_cond.

Модифицированный алгоритм генерации операции замены – это алгоритм из предыдущей работы [15], в котором использование функции $generate\_rep$ заменено

на использование функции *gen_rep*. Эти функции принимают в качестве аргумента допустимую конструкцию и транслируют ее на язык системы ACL2. Функция *c_kernel_translator* [15] осуществляет трансляцию C-kernel выражений на язык системы ACL2.

Определим функцию *gen_rep* для каждого вида допустимой конструкции *op*, обозначив как ... поставляемые номера строк и опциональный элемент *'break_path*:

- если *op* – это пустой оператор, то результатом *gen_rep*(*op*) будет

$$((cons\ ?!label\ fr)\ (cons\ (list\ 'empty\_stmt\ \ldots)\ fr));$$

- если *op* — это **break**;, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\\ (list\ \ 'break\_stmt\ \ \ldots)\\ (change\text{-}frame\ \ fr\ \ :loop\text{-}break\ \ t)))\\ ((when\ \ t)\ \ fr);\end{gathered}$$

- если *op* — это **if** (**a**) **b**, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ \ (list\ \ 'if\_stmt\ \ \ldots)\\ (if\ \ (b*\ \ ((cons\ \ ?!label\ \ ?!fr)\ \ (cons\\ (list\ \ 'if\_cond\ \ \ldots)\ \ fr))\ \ c\_kernel\_translator(a))\\ (b*\ (gen\_rep(b))\ fr))))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr);\end{gathered}$$

- если *op* — это **if** (**a**) **b else c**, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ \ (list\ \ 'full\_if\_stmt\ \ \ldots)\\ (if\ \ (b*\ \ ((cons\ \ ?!label\ \ ?!fr)\ \ (cons\\ (list\ \ 'if\_cond\ \ \ldots)\ \ fr)\ c\_kernel\_translator(a))\\ (b*(gen\_rep(b))\ fr))\ (b*(gen\_rep(c))\ fr))))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr);\end{gathered}$$

- если *op* — это {$\mathbf{a_1}$ $\mathbf{a_2}$ ... $\mathbf{a_{k-1}}$ $\mathbf{a_k}$}, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ (list\ \ 'block\_stmt\ \ \ldots)\\ (b*(gen\_rep(a\_1)\ gen\_rep(a\_2)\ \ldots gen\_rep(a\_k\text{-}1)\ \ gen\_rep(a\_k))\ fr)))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr).\end{gathered}$$

Модифицированный алгоритм генерации операции замены отличается от исходного тем, что он снабжает все виды допустимых конструкций семантическими метками.

Модифицированный алгоритм извлечения семантических меток. Модифицированный алгоритм извлечения меток – это алгоритм из раздела 1., дополненный способом извлечения семантических меток из специального представления функции *rep* в виде дерева кода. Опишем способ извлечения семантических меток из специального представления функции *rep* в виде дерева кода.

Представим определение функции *rep* в виде дерева. Построим его по процедуре построения дерева кода функции. Примененная к связываниям, которые соответствуют пустым операторам, инструкциям `break` и присваиваниям тела цикла, данная процедура возвращает вырожденные деревья в виде соответствующих листьев. Примененная к связыванию, которое соответствует инструкции `if`, данная процедура возвращает дерево, корнем которого является это связывание, а потомки (поддеревья) получены в результате применения данной процедуры к блокам, соответствующим ветвям этой инструкции `if`. Примененная к блоку $b*$, которая соответствует C-блоку, данная процедура возвращает дерево, корнем которого является этот блок, а потомки (поддеревья) получены в результате применения этой процедуры к последовательности связываний, составляющих этот блок. Таким образом, процедура построения дерева кода функции определена рекурсивно.

Для извлечения меток из определения функции *rep* для него строится дерево с помощью применения процедуры построения дерева кода функции к блоку $b*$, соответствующему телу цикла. Затем это дерево обходится в глубину, и метки извлекаются из вершин дерева в порядке обхода. Таким образом представление в виде дерева определения функции *rep* используется для реализации обхода в глубину.

Модифицированный алгоритм генерации объяснений недоказанных условий корректности. Модифицированный алгоритм генерации объяснений недоказанных условий корректности – это алгоритм из раздела 1., дополненный способом генерации текста для меток, извлеченных из специального представления функции *rep* в виде дерева кода. Опишем способ генерации текста для меток, извлеченных из определения функции *rep*.

Каждой концепции меток, помечающих определение функции *rep*, соответствует текстовый шаблон. Такие шаблоны похожи на форматные строки языка C со специальными управляющими конструкциями для вставки номера первой строки диапазона строк кода и номера последней строки диапазона строк кода.

При генерации текста список извлеченных меток обрабатывается последовательно. Для каждой метки происходит извлечение диапазона строк, затем с помощью обратного транслятора [24] данный диапазон строк C-kernel программы переводится в диапазон строк C-light программы. Потом для данной метки генерируется текст с помощью подстановки полученных номеров строк C-light программы. Таким образом, общий текст получается из набора текстов для каждой метки.

Если в верифицируемой программе содержится финитная итерация с инструкцией `break`, то нами был предложен способ генерации дополнительного поясняющего текста. УК такой программы представляет собой импликацию, содержащую применение функции *rep*. Рассмотрим посылку такого УК. Проанализируем случай, описываемый рассматриваемой посылкой, используя результат применения стратегии для формул с операцией замены из раздела 2.

Если удалось доказать лемму, что в описываемом рассматриваемой посылкой

случае исполнилась инструкция `break`, то в данном случае рассмотрим связывания, соответствующие инструкциям `if` на пути к данной инструкции. Именно для этой цели список, соответствующий метке, содержит опциональный элемент $'break\_path$. Используя все условия условных операторов, вычислим символически с помощью подстановок общее условие, при выполнении которого исполняется связывание, соответствующее инструкции `break`. Обозначим такое условие как θ. Так как в рассматриваемом случае инструкция `break` исполнилась, то хотя бы для одной итерации условие θ истинно. Значит, пользователю системы верификации имеет смысл проверить условие θ на предмет наличия ошибки. Поэтому об этом генерируется дополнительный поясняющий текст. Этот текст содержит: 1)информацию о исполнении инструкции *break*, заданной на определенной строке программы, 2) информацию о существовании такого номера итерации, что условие θ выполняется на этой итерации, и 3)условие θ.

5. Пример

В данном разделе описываются результаты эксперимента с примером из работы [17], чтобы проиллюстрировать применение разработанных и добавленных в систему C-lightVer инструментов для автоматизированной локализации ошибок. Эксперимент состоит в применении системы C-lightVer к содержащей ошибку функции `grt_eql_key`:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==> \result == 0) &&
3.             (grt-eql-cnt(n, key, a) > 0 ==> \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] < key){result = 1; break;}}
9.    return result;}
```

Спецификации функции заданы на языке ACSL. Определение функции *grt-eql-cnt* на языке ACL2 приведено в репозитории [13]. В соответствии со спецификацией функция `grt_eql_key` проверяет существование в массиве `a` элемента, который больше ключа `key` или равен ключу `key`. Если такой элемент найден, то по спецификации функция должна возвращать `1`, иначе – `0`. Таким образом, ошибка состоит в использовании оператора `<` вместо оператора `>=` в инструкции `if` тела цикла.

Полученная C-kernel программа, ее УК, описание применения стратегий и вспомогательные леммы приведены в статье [17]. Заданные на языке системы ACL2 УК, определение *rep* и вспомогательные леммы приведены в репозитории [13].

В ходе проведенного ранее эксперимента с этим примером [17] применение стратегии проверки ложности из раздела 3. позволило доказать ложность УК, используя его первый конъюнкт. Поэтому результатом проведенного ранее эксперимента является следующее объяснение [17], сгенерированное для этого конъюнкта:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that postcondition goal from line 2
    with substitution loop effect from lines 7-8 by rep
does not hold.
```

Отметим, что системой C-lighVer генерируются именно такие объяснения. Единственное отличие объяснений, генерируемых системой C-lightVer, от объяснений, приведенных в данном разделе, состоит в наличие отступов, которые для улучшения читабельности текста добавляются вручную.

В отличие от предыдущей работы [17] для проведения эксперимента мы использовали модифицированный алгоритм генерации операции замены, модифицированный алгоритм извлечения меток и модифицированный алгоритм генерации объяснений недоказанных условий корректности. Поэтому в результате проведения эксперимента было сгенерировано более подробное объяснение:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that
    - postcondition goal from line 2
      with substitution loop effect from lines 7-8 by rep
        that corresponds to the following loop body
        sequence of the following statements
        from line 8 to line 8
          if statement from line 8
          with condition "a[i] < key" from line 8
          and with the following positive branch
          sequence of the following statements
          from line 8 to line 8
            assignment statement "result = 1" from line 8
            break statement from line 8
      does not hold
    - break statement execution in the loop body from line 8
      holds
    - exists such i (0 <= i < n)
      that condition
      a[i] < key
      holds
```

В отличие от объяснения, полученного в результате проведенного ранее эксперимента [17], данное объяснение содержит текст, соответствующий телу цикла,

и текст, описывающий содержащее ошибку условие. Данный текст позволяет обратить внимание пользователя системы верификации на содержащее ошибку условие инструкции `if`, в ветви которой находится `break`.

6. Заключение

Эта статья посвящена работе подсистемы локализации ошибок системы C-lightVer. Данная подсистема работает в том случае, если не удалось доказать истинность УК. В таком случае подсистема локализации ошибок пытается проверить ложность УК. Если недоказанное УК основано на функции *rep*, то подсистема объяснения УК должна давать подробные объяснения о структуре цикла. Если посылка ложного УК описывает случай исполнения инструкции `break`, то подсистема объяснений должна обратить внимание пользователя на условие исполнения `break`.

В данной статье описаны разработанные авторами модификации алгоритмов:

1. Модифицированный алгоритм генерации операции замены. Новая модификация алгоритма позволяет пометить исходный код функции *rep* семантическими метками.

2. Модифицированный алгоритм извлечения семантических меток. Новая модификация алгоритма позволяет извлекать семантические метки не только из УК, но и из специального представления функции *rep* в виде дерева кода.

3. Модифицированный алгоритм генерации объяснений недоказанных условий корректности. Если посылка ложного УК описывает случай исполнения инструкции `break`, то новая модификация алгоритма позволяет подсистеме локализации ошибок обратить внимание на условие исполнения `break`.

Данные методы позволили нам провести эксперимент по локализации ошибки в функции `grt_eql_key`. Результат данного эксперимента приведен в этой статье.

Ранее мы провели успешные эксперименты [14] по дедуктивной верификации функции `asum` [15] из интерфейса BLAS. Так как функции интерфейса BLAS совершают итерации над векторами и матрицами, то планы нашей дальнейшей работы состоят в применении к данным функциям комплексного подхода системы C-lightVer, основанного на символическом методе верификации финитных итераций.

Список литературы / References

[1] De Angelis E., Fioravanti F., Pettorossi A., Proietti M., “Lemma Generation for Horn Clause Satisfiability: A Preliminary Study”, VPT 2019, EPTCS, **299**, 2019, 4–18.

[2] Apt K. R., Olderog E.-R., “Fifty years of Hoare’s logic”, *Formal Aspects of Computing*, **31**:6 (2019), 751–807.

[3] Blanchard A., Loulergue F., Kosmatov N., “Towards Full Proof Automation in Frama-C Using Auto-active Verification”, NFM 2019, LNCS, **11460**, 2019, 88–105.

[4] De Carvalho D. et al., “Teaching Programming and Design-by-Contract”, ICL 2018, AISC, **916**, 2019, 68–76.

[5] Denney E., Fischer B., "Explaining Verification Conditions", AMAST 2008, LNCS, **5140**, 2008, 145–159.

[6] Efremov D., Mandrykin M., Khoroshilov A., "Deductive Verification of Unmodified Linux Kernel Library Functions", ISoLA 2018, LNCS, **11245**, 2018, 216–234.

[7] Fraer R., "Tracing the Origins of Verification Conditions", AMAST 1996, LNCS, **1101**, 1996, 241–255.

[8] Galeotti J. P., Furia C. A., May E., Fraser G., Zeller A., "Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking", *IEEE Transactions on Software Engineering*, **41**:10 (2015), 1019–1037.

[9] Hähnle R., Huisman M., "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools", *Computing and Software Science*, LNCS, **10000**, 2019, 345–373.

[10] Heras J., Komendantskaya E., Johansson M., Maclean E., "Proof-Pattern Recognition and Lemma Discovery in ACL2", LPAR 2013, LNCS, **8312**, 2013, 389–406.

[11] Johansson M., "Lemma Discovery for Induction", CICM 2019, LNCS, **11617**, 2019, 125–139.

[12] Khazeev M., Mazzara M., De Carvalho D., Aslam H., "Towards A Broader Acceptance of Formal Verification Tools: The Role of Education", 2019, arXiv:abs/1906.01430.

[13] Kondratyev D. A., "Automated Error Localization in C Programs.", bitbucket.org/Kondratyev/verify-c-light.

[14] Kondratyev D., "Implementing the Symbolic Method of Verification in the C-Light Project", PSI 2017, LNCS, **10742**, 2018, 227–240.

[15] Кондратьев Д. А., Марьясов И. В., Непомнящий В. А., "Автоматизация верификации С-программ с использованием символического метода элиминации инвариантов циклов", *Моделирование и анализ информационных систем*, **25**:5 (2018), 491–505; [Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A., "The Automation of C Program Verification by Symbolic Method of Loop Invariants Elimination", *Modeling and Analysis of Information Systems*, **25**:5 (2018), 491–505, (in Russian).]

[16] Kondratyev D. A., Promsky A. V., "Developing a Self-Applicable Verification System. Theory and Practice", *Automatic Control and Computer Sciences*, **49**:7 (2015), 445–452.

[17] Kondratyev D. A., Promsky A. V., "Towards Automated Error Localization in C Programs with Loops", *System Informatics*, 2019, № 14, 31–44.

[18] Kondratyev D., Promsky A., "Proof Strategy for Automated Sisal Program Verification", TOOLS 2019, LNCS, **11771**, 2019, 113-120.

[19] Könighofer R., Toegl R., Bloem R., "Automatic Error Localization for Software Using Deductive Verification", HVC 2014, LNCS, **8855**, 2014, 92–98.

[20] Kovács L., "Symbolic Computation and Automated Reasoning for Program Analysis", IFM 2016, LNCS, **9681**, 2016, 20–27.

[21] Leino K. R. M., Millstein T., Saxe J. B., "Generating Error Traces from Verification-Condition Counterexamples", *Science of Computer Programming*, **55**:1–3 (2005), 209–226.

[22] Li J., Sun J., Li L., Loc Le Q., Lin S-W., "Automatic Loop Invariant Generation and Refinement through Selective Sampling", ASE 2017, 2017, 782–792.

[23] Lin Y., Bundy A., Grov G., Maclean E., "Automating Event-B invariant proofs by rippling and proof patching", *Formal Aspects of Computing*, **31**:1 (2019), 95–129.

[24] Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A., "Automatic C Program Verification Based on Mixed Axiomatic Semantics", *Automatic Control and Computer Sciences*, **48**:7 (2014), 407–414.

[25] Moore J. S., "Milestones from the Pure Lisp Theorem Prover to ACL2", *Formal Aspects of Computing*, **31**:6 (2019), 699–732.

[26] Moriconi M., Schwarts R. L., "Automatic Construction of Verification Condition Generators From Hoare Logics", ICALP 1981, LNCS, **115**, 1981, 363–377.

[27] Nepomniaschy V. A., "Symbolic Method of Verification of Definite Iterations over Altered Data Structures", *Programming and Computer Software*, **31**:1 (2005), 1–9.

[28] Reger G., Voronkov A., “Induction in Saturation-Based Proof Search”, CADE 2019, LNCS, **11716**, 2019, 477–494.

[29] Srivastava S., Gulwani S., Foster J. S., “Template-Based Program Verification and Program Synthesis”, *International Journal on Software Tools for Technology Transfer*, **15**:5–6 (2013), 497–518.

[30] Tuerk T., “Local Reasoning about While-Loops”, VSTTE 2010. Workshop Proceedings, 2010, 29–39.

[31] Volkov G., Mandrykin M., Efremov D., “Lemma Functions for Frama-C: C Programs as Proofs”, 2018 Ivannikov Ispras Open Conference (ISPRAS), 2018, 31–38.

[32] Yang W., Fedyukovich G., Gupta A., “Lemma Synthesis for Automating Induction over Algebraic Data Types”, CP 2019, LNCS, **11802**, 2019, 600-617.

Kondratyev D. A., Promsky A. V., "The Complex Approach of the C-lightVer System to the Automated Error Localization in C-programs", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 502–519.

DOI: 10.18255/1818-1015-2019-4-502-519

Abstract. The C-lightVer system for the deductive verification of C programs is being developed at the IIS SB RAS. Based on the two-level architecture of the system, the C-light input language is translated into the intermediate C-kernel language. The meta generator of the correctness conditions receives the C-kernel program and Hoare logic for the C-kernel as input. To solve the well-known problem of determining loop invariants, the definite iteration approach was chosen. The body of the definite iteration loop is executed once for each element of the finite dimensional data structure, and the inference rule for them uses the substitution operation rep, which represents the action of the cycle in symbolic form. Also, in our meta generator, the method of semantic markup of correctness conditions has been implemented and expanded. It allows to generate explanations for unproven conditions and simplifies the errors localization. Finally, if the theorem prover fails to determine the truth of the condition, we can focus on proving its falsity. Thus a method of proving the falsity of the correctness conditions in the ACL2 system was developed. The need for more detailed explanations of the correctness conditions containing the replacement operation rep has led to a change of the algorithms for generating the replacement operation, and the generation of explanations for unproven correctness conditions. Modifications of these algorithms are presented in the article. They allow marking rep definition with semantic labels, extracting semantic labels from rep definition and generating description of break execution condition.

Keywords: deductive verification, semantic label, error localization, C-lightVer, ACL2, MetaVCG, definite iteration, proof strategy

On the authors:
Dmitry A. Kondratyev, orcid.org/0000-0002-9387-6735, postgraduate student,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: apple-66@mail.ru

Alexei V. Promsky, orcid.org/0000-0002-5963-2390, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: promsky@iis.nsk.su

Acknowledgments:
This work was partially funded by the RFBR according to the research № 17-01-00789.

Моделирование и анализ информационных систем. Т. 26, №4 (2019), с. 520–533
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 520–533

DOI: 10.18255/1818-1015-2019-4-520-533

UDC 519.987

Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root

Todorov V., Taha S., Boulanger F., Hernandez A.

Received September 17, 2019
Revised November 19, 2019
Accepted November 27, 2019

Abstract. For many years, automotive embedded systems have been validated only by testing. In the near future, Advanced Driver Assistance Systems (ADAS) will take a greater part in the car's software design and development. Furthermore, their increasing critical level may lead authorities to require a certification for those systems. We think that bringing formal proof in their development can help establishing safety properties and get an efficient certification process. Other industries (e.g. aerospace, railway, nuclear) that produce critical systems requiring certification also took the path of formal verification techniques. One of these techniques is *deductive proof*. It can give a higher level of confidence in proving critical safety properties and even avoid unit testing.

In this paper, we chose a production use case: a function calculating a square root by linear interpolation. We use deductive proof to prove its correctness and show the limitations we encountered with the off-the-shelf tools. We propose approaches to overcome some limitations of these tools and succeed with the proof. These approaches can be applied to similar problems, which are frequent in the automotive embedded software.

Keywords: formal methods, deductive proof, proving discrete-valued functions

For citation: Todorov V., Taha S., Boulanger F., Hernandez A., "Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root ", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 520–533.

On the authors:
Vassil Todorov, orcid.org/0000-0002-2739-499X,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, France, e-mail: vassil.todorov@lri.fr

Safouan Taha, orcid.org/0000-0003-3950-6415, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, France, e-mail: safouan.taha@lri.fr

Frédéric Boulanger, orcid.org/0000-0003-3185-2807, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, France, e-mail: frederic.boulanger@lri.fr

Armando Hernandez, orcid.org/0000-0003-4555-4616,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, France, e-mail: armando.hernandez@mpsa.com

Acknowledgments:
This work was supported by the Groupe PSA, a French multinational manufacturer of automobiles and motorcycles sold under the Peugeot, Citroën, DS, Opel and Vauxhall brands.

Introduction and Motivation

Today, the automotive industry relies mostly on a model-based approach for developing embedded software. It consists in connecting common library blocks (operators) to design and simulate a model of the behavior to be produced. It uses a higher level of abstraction than the code. Code with the behavior of the model is then produced automatically. The most commonly used tools for software design are Simulink, from the MathWorks, and Scade, from ANSYS.

The main advantage of this approach is that models can be simulated and debugged before code generation. Thus, some of the errors are found and fixed earlier in the design process. On the other hand, simulation shares many common points with testing and cannot prove that the calculation is correct. Furthermore, the implementation of a model on a specific hardware can bring behaviors that have not been seen before at design stage.

For the rest of our study we take as example a function calculating a square root. During the design stage, the simulation can use a standard implementation of this function. However, in the implementation, we replace it with an optimized version because of hardware constraints. Fig. 1 shows this approach. Our example is a discrete-valued function implementing the square root calculation, which uses a linear interpolation table. In automotive applications, as on-board computers have limited power, discrete-valued functions are frequently used in the implementation to avoid complex calculations.

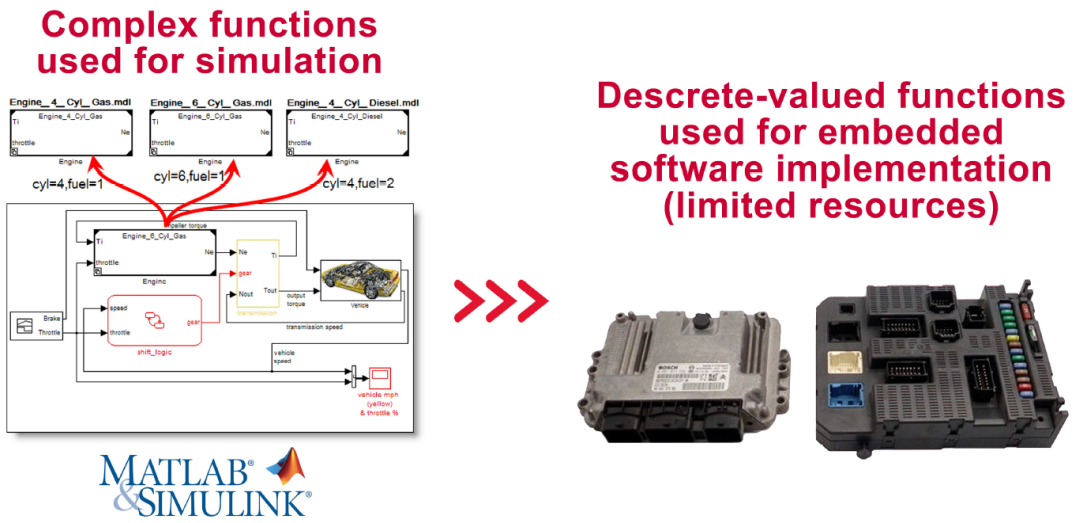

Figure 1. Complex functions for simulation vs. discrete-valued ones for implementation

In the near future, we expect that authorities will require a certification for highly critical software in self-driving cars. Our motivation is to provide proofs of correctness for production code using formal methods.

In a previous paper [29], we summarized some experiments about applying tools that use formal methods to industrial software. In this paper, we give details about the application of deductive proof to production code, the problems we encountered with off-the-shelf tools, and some approaches to solve this type of problems. Our function has been implemented in C and we used Frama-C WP [16] for proving its correctness. As some of the goals were impossible to prove with Frama-C and its solvers we implemented it in

SPARK (based on Ada) to prove it with GNATprove [7]. We discuss the results as well as how other methods such as Abstract interpretation can be combined with deductive proof.

1. Deductive methods

1.1. Preliminaries

The foundations of the proof of logical properties on an imperative language program were put forward by C. A. R. Hoare [15] in 1969. Based on the precise semantic of a computer program, Hoare proposed to prove certain properties by mathematical deductive reasoning, generally at the end of the program.

He introduced a notation called the *Hoare triple*, which associates a program Q, start hypotheses P, and expected output properties R:

$$P\ \{Q\}\ R$$

The logical meaning of this triple corresponds to: if P is true, then after executing program Q, R will be true if Q terminates. The calculus of Hoare's triples is, in general, undecidable.

The proving by application of Hoare's rules is an intellectual process and is not tool driven. It is up to the author of the proof to define the correct properties between each instruction of the program and to establish its demonstration by applying the different theorems. This activity is not adapted to process thousands of lines of code in an acceptable time.

An initial automation of the process of proving programs was brought by the calculation of the WP (*Weakest Precondition*) from Dijkstra [10]. The principle consists in automatically calculating the most general property *WP(S,P)* holding before a statement S such that property P holds after the execution of S:

$$WP(S,P)\ \{S\}\ P$$

The calculus of WP is defined for each instruction. The proof process consists in calculating WP by going backward from the end of the program for which we want to prove P, up to the beginning. For full correctness, S must terminate.

The returned predicate from the WP calculation can rapidly become rather complex. Efficient (quadratic instead of exponential) verification condition generation (including WP generation) were proposed in the following papers [27, 3, 13]. In order to automate the process, all modern tools based on WP are using automatic theorem provers as back-end. We can cite, for example Alt-Ergo [8], Colibri[1], CVC4 [4], Yices2 [11], Z3 [9].

1.2. Tools for deductive reasoning

As we are interested in tools used by the industry, we present here only those that are mostly used today: Atelier B[2], Caveat [22], Frama-C WP and GNATprove.

[1]Colibri: http://smtcomp.sourceforge.net/2018/systemDescriptions/COLIBRI.pdf
[2]Atelier B: https://www.atelierb.eu

1.2.1. Atelier B

Atelier B is a tool supporting the B method, which is a formal methodology to specify, build and implement software systems. The B method was originally developed in the 1980s by Jean-Raymond Abrial [2] and is based on first-order logic, set theory, abstract machine theory and refinement theory. This method is suitable for a new development. As we reused existing C code, we did not use this method.

1.2.2. Caveat and Frama-C WP

Caveat and Frama-C WP are tools for deductive reasoning on C programs. Caveat was introduced at Airbus in 2002 to replace unit tests by unit proof and thus obtain a cost reduction and quality improvement over this part of the development process. The tool with its back-end Alt-Ergo were certified and recognized by the aviation certification authorities. Caveat analyzed C programs (with some restrictions in terms of language constructs) and had its own specification language based on a first order logic.

Frama-C is the academic open source tool developed by the same team as Caveat. Its WP module verifies properties written in the ACSL[3] language in a deductive manner. It implements the Weakest Precondition calculus and targets multiple automatic solvers via the Why3 platform[4].

1.2.3. GNATprove

GNATprove is a tool for deductive reasoning over SPARK (based on Ada) programs. Like Frama-C, it uses the Why3 platform but SPARK supports bit-vector data types. A bit-vector is an array data type for compactly storing bits. Most modern SMT-solvers support a theory of bit-vectors, which can help solving problems using this data type. Furthermore, for properties that are not valid, GNATprove can obtain a counterexample from the SMT solver.

2. Environment

We present in Fig. 2 our environment. We have C code, which is annotated with contracts using the ACSL language. We use two different features of Frama-C WP. First, we use it to parse and then transform the C code together with the contracts into verification conditions (VCs) that are directly sent to the SMT solvers. Second, we also use Frama-C WP to transform the C code together with the contracts into WhyML language files. The Why3 framework then transforms the WhyML files into VCs and addresses the SMT solvers. The main difference between these two approaches is that the direct SMT-LIB output was initially developed for the Colibri SMT solver, which does not support quantifiers. Thus, the direct SMT-LIB output provides a set of quantifier-free formulas. The other way, through WhyML, allows for richer theories and supports quantified formulas even within the specification contracts.

[3]ACSL specification language: https://frama-c.com/acsl.html
[4]Why3: http://why3.lri.fr/

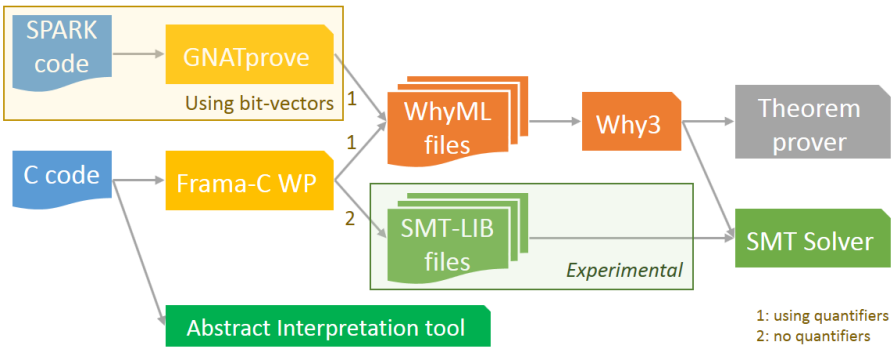

Figure 2. Environment for deductive proof on C and SPARK code

We used GNATprove to prove the equivalent code written in SPARK. This approach is similar to using Frama-C with WhyML and quantified formulas. The advantage of SPARK for our use case is that we can use bit-vector types for modular arithmetic and thus facilitate the proof.

Because we experienced some difficulties with the analysis of our C code, we also analyzed it with an Abstract interpretation tool to get additional confidence.

3. Experiment

We took the C code implemented in an on-board computer to prove its correctness using deductive proof. The function calculates the square root Y of X by linear integer interpolation between two known points (X_a, Y_a) and (X_b, Y_b) using the following formula:

$$Y = Y_a + (X - X_a)\frac{(Y_b - Y_a)}{(X_b - X_a)}$$

This code is used in an implementation on an on-board computer, which cannot use floating-point numbers. We calculate the square root for numbers between 0.00 and 100.00 using an integer representation. We consider it as a fix-point number (multiplied by 100 to have a precision of 2 digits after the decimal separator), thus the input range is between 0 and 10000 (representing 0 and 100.00) and the returned result is a linearly interpolated value between 0 and 1000 (to be interpreted as a number between 0 and 10.00). We want to prove that the calculation is correct for a given precision.

We proceeded in two steps. First, we proved a simplified version of the code using only eight values in the interpolation table (Fig. 3) and limited to the range $[0, 1.00]$. These values were a subset of the full table present in the code, which contains 41 values. Then, we added the other values in the table and updated the contracts to take into account the new bounds. To our surprise, this did not scale up with Frama-C. We worked with the developers of Frama-C to understand why (we explain it in Section 4.).

The code of our main function is given in Fig. 4. This function takes a number and returns its square root using a table for some known values or interpolates a value when the number is between two known points. Using the ACSL annotation language, we define two behaviors for this function: whenever the number is less than 10000 the function is defined, otherwise it returns the maximum value i.e. 1000. For more readability, we removed some intermediate values from the two tables.

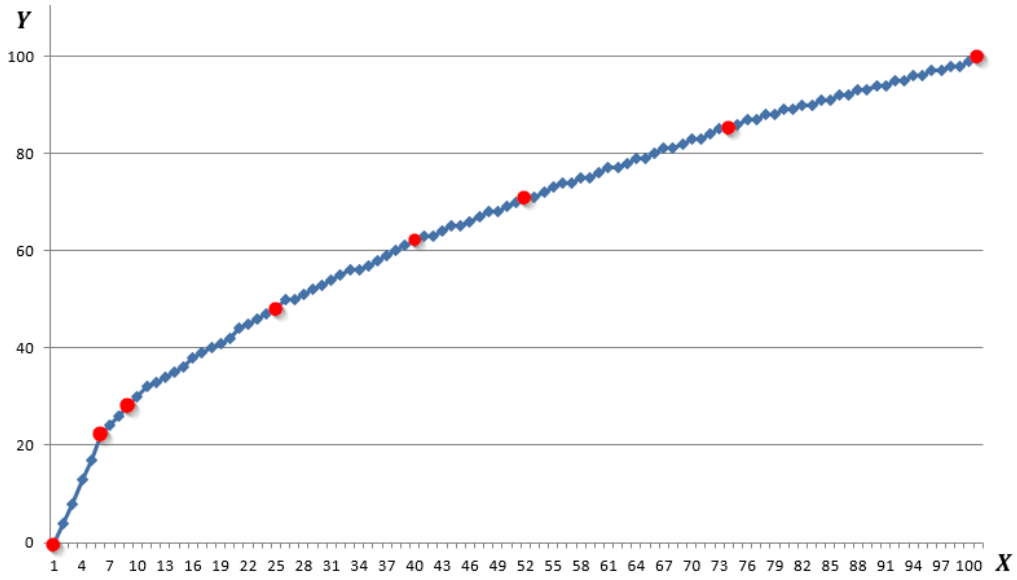

Figure 3. Square root calculation in [0, 1.00] by linear interpolation from eight known values

As we have a loop, in formal verification we need to specify a loop invariant for it. A loop invariant is a predicate (condition) that holds for every iteration of the loop (before and after the iteration). This predicate should be strong enough and its automatic generation is generally a difficult problem.

```
/*@ assigns \nothing;
 behavior in_range:
   assumes number <= 10000;
   ensures number-30 <= (\result)*(\result)/100 <= number+10;
 behavior out_of_range:
   assumes number > 10000;
   ensures \result == 1000;
 complete behaviors in_range, out_of_range;
 disjoint behaviors in_range, out_of_range;
*/
uint16 IntSqrt(uint16 number) {
   uint8 i = 0;
   uint16 TabX[41] = {0,5,10,25,40,...,7500,8000,8600,9200,10000};
   uint16 TabY[41] = {0,22,32,50,63,...,866,894,927,959,1000};
   /*@ loop invariant 0 <= i <= 40 && number >= TabX[i];
    loop assigns i;
    loop variant 40-i; */
   for (i = 0 ; i < 40 ; i++) {
      if ((number >= TabX[i]) && (number <= TabX[i+1])) {
         return(LinearInterpolation(TabX[i], TabY[i], TabX[i+1], TabY[i+1], number));
      }
   }
   return TabY[40];
}
```

Figure 4. Annotated square root function for Frama-C WP automatic proof

With Frama-C we also need to define precisely which variables are modified (assigned) during the loop. In our example, i is incremented on each iteration.

For the loop to be proved, we also need to write a variant function which is a function whose value is monotonically decreased with respect to a (strict) well-founded relation by the iteration of the loop. It is used to ensure the termination of the loop.

Then we rewrote the function in SPARK[5] to see whether it would scale better. Fig. 5 presents the SPARK code. The main difference between C and SPARK is that we can specify a `bit-vector` data type in SPARK, which is then communicated to the SMT solver via Why3. For our use case, it helped the solver to reason using modular arithmetic. Most SMT solvers used as back-end of Why3 have a theory of bit-vectors. If we do not use bit-vectors, the SMT solver is reasoning by default using non-modular arithmetic.

```
type Unsigned is mod 2**32;
subtype uint16 is Unsigned range 0 .. 65535;
type UINT16_ARR is array (Positive range <>) of uint16;
Max : constant := 10_000;
function LinearInterpolation(Xa, Ya, Xb, Yb, X : uint16) return uint16 is
   Result : uint16;
begin
   if Xa /= Xb then
      Result := Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa);
   else
      Result := Ya;
   end if;
   return Result;
end LinearInterpolation;
function IntSqrt(number : uint16) return uint16
   with Global => null, Contract_Cases =>
   (number <= Max => IntSqrt'Result * IntSqrt'Result / 100 + 30 >= number and
       number+10 >= IntSqrt'Result * IntSqrt'Result / 100, number > Max =>
       IntSqrt'Result = 1000) is
   TabX : UINT16_ARR(1 .. 41) := (0,5,10,25,40,...,8000,8600,9200,10000);
   TabY : UINT16_ARR(1 .. 41) := (0,22,32,50,63,...,894,927,959,1000);
begin
   for I in 1 .. 40 loop
      pragma Loop_Invariant (for all J in 1 .. I => number >= TabX(J));
      if number in TabX(I) .. TabX(I+1) then
         return LinearInterpolation (TabX(I), TabY(I), TabX(I+1), TabY(I+1), number);
      end if;
   end loop;
   return TabY(41);
end IntSqrt;
```

Figure 5. SPARK code for automatic proof with GNATprove

The proof of the simplified code succeeded on both Frama-C and SPARK. However, when using the full table of 41 values, Frama-C failed where only SPARK succeeded.

We also analyzed our complete C code with Astrée [18] from AbsInt, a static analysis tool using abstract interpretation, to prove some difficult goals. The abstract interpretation results can be used as assumptions for Frama-C WP or bring more confidence

[5]Special thanks to Yannick Moy from AdaCore

for certification if Frama-C can reason on them. We discuss the results in the next section.

4. Results

In this section, we explain the results and why Frama-C failed to scale-up from 8 to 41 values, and what should be done to cope with this type of problems.

4.1. From Frama-C to the SMT solver

To understand the reason why automatic proof failed for the full table, we have to detail the transformations between the C code through Frama-C, Why3 and the solvers. First, Frama-C transforms the C code and its ACSL contracts using the weakest precondition calculus into verification conditions (VC) in the WhyML language. It also introduces additional goals to verify the absence of runtime errors such as overflows. The WhyML output contains all the theories necessary for the proof and is sent to Why3. Then Why3 transforms it into the language of the chosen prover. For our use case, the WhyML transformation contained quantified formulas and had redefined some operators such as `division` using uninterpreted functions.

4.2. The difficult goal

There were 51 goals (verification conditions) to be proved and two of them were not proven. The most difficult goal was about proving that the contract of the post condition in the linear interpolation function had the same behavior as the code. We show it in Fig. 6.

```
typedef unsigned short uint16;
typedef unsigned char uint8;
/*@
   requires 0 <= Xa <= 10000 && 0 <= Xb <= 10000;
   requires 0 <= Ya <= 1000 && 0 <= Yb <= 1000;
   requires Yb > Ya && Xb >= Xa;
   requires Xa <= X <= Xb;
   ensures Xa != Xb ==> \result == (Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
   ensures Xa == Xb ==> \result == Ya;
   assigns \nothing;
*/
uint16 LinearInterpolation(uint16 Xa, uint16 Ya, uint16 Xb, uint16 Yb, uint16 X)
{
   if (Xa != Xb) {
      return(Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
   } else {
      return(Ya);
   }
}
```

Figure 6. Annotated interpolation function for Frama-C WP automatic proof

Actually, contracts use mathematical arithmetic (without overflow), but code uses modular arithmetic, where overflows may occur. For our use case, we used a 16-bit unsigned integer to store the returned value of the interpolation.

4.3. Direct proof with SMT-LIB

Since 2 goals were not proven with the official Frama-C version, we obtained a new version that could address directly SMT solvers using the SMT-LIB standard [5]. We proved our goals with Colibri, CVC4 and Yices2. We remarked that the SMT-LIB file did not contain quantifiers and did not redefine operators such as `division`. We concluded that this approach scaled and worked better for problems with nonlinear arithmetic such as interpolation functions. Furthermore, some SMT solvers such as Yices2 do not support quantification.

4.4. Experience with the Why3 SMT output files

We wanted to understand what was the impact of the redefined division using uninterpreted functions and of quantified formulas, so we modified manually the SMT request sent to the solver. First, we removed the specific functions about division and used the standard SMT-LIB `div` operator. Then, the proof succeeded with CVC4 but only if using nonlinear logic containing bit-vectors. Disabling bit-vectors from that logic resulted in a failure to prove the formula. On the other hand, the quantifier-free SMT output did not need bit-vector logic to be proved.

4.5. Abstract interpretation

Because it is difficult to understand how the SMT solvers proved the difficult goal, we used Astrée to prove the absence of overflow in the returned value of the linear interpolation function. This proof can then be used as hypothesis in Frama-C WP. Astrée could find the dependency between Y_b and Y_a and estimate a precise interval for $(Y_b - Y_a)$. The same was done for $(X_b - X_a)$ and $(X - X_a)$. Thus a precise interval was calculated for Y in [0, 10000], which fits in a 16-bit unsigned integer without overflow.

5. Methodology

In this section, we propose a methodology based on our experience to solve problems using discrete-valued functions such as linear interpolation. Our use case is a simple one and we could have tested it for each value in the domain of validity of the function. However, in practice, there are more complex discrete-valued functions implemented with linear interpolation tables called lookup tables. These functions are often called by other discrete-valued functions. The number of cases to test can be the product of the cardinalities of the domains of the individual functions. We propose to use the methodology shown below in Fig. 7 in order to prove those functions.

First, we need to isolate all the functions we want to prove together and annotate the code with contracts specifying the behavior expected from each function. Then, we can try to prove it in Frama-C via Why3. If the proof succeeds, we can stop. Otherwise, we

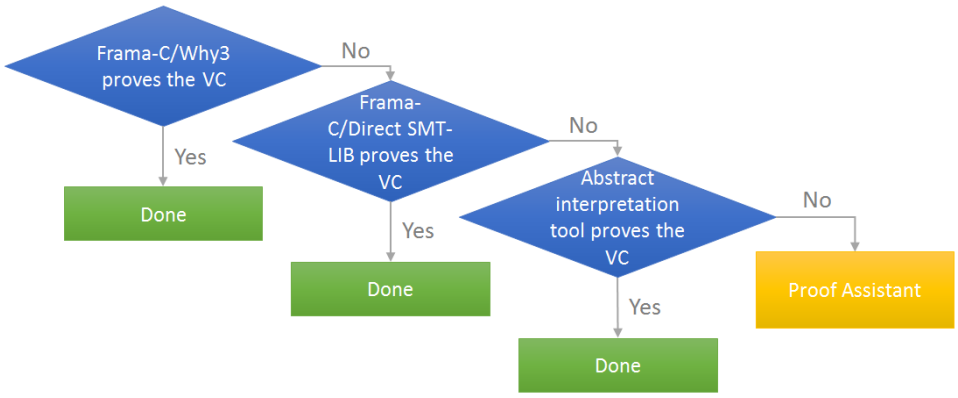

Figure 7. Methodology for proving Discrete-Valued Functions

can try to use the direct SMT-LIB output of Frama-C WP with the SMT solvers. As we have seen, this approach removes quantifiers and uses native mathematical operators. If it does not succeed, for some goals (VCs) we can try to prove them using abstract interpretation tools. If this method does not succeed, we need to use a proof assistant to prove the difficult goals.

6. Related work

Our work concerns the formal verification of the square root function used for embedded automotive applications and using fixed-point numbers. Embedded software generally needs to be optimized because of the limited power of the micro-controllers. We used deductive proof engines (Frama-C and GNATprove) that create verification obligations discharged mostly automatically by SMT solvers. For our application, we only need to calculate square root for a predefined interval and the precision of our lookup table is enough to satisfy the requirements. We could use other methods such as Newton method but it would be more resources consuming. To the best of our knowledge, a linearly-interpolated fixed-point square root algorithm has not been the subject of formal verification work. In this section, we give a survey on some related work about the correctness proof of square root algorithms for machine representation and for standard mathematical functions in general.

The problem of the specification and validation of standard functions is also discussed in [17]. Even if a standard for representing floating-point numbers has been defined (IEEE 754), this standard does not provide requirements for the specification of standard functions. This paper is a systematic presentation of ideas from other studies about the formal specification and testing of standard mathematical functions. The author does not use automatic proof assistance.

We think that the first floating-point algorithms verifications were motivated by some hardware bugs such as the Pentium FDIV bug discovered in 1994. For example, in 1998 Russinoff used the ACL2 theorem prover to verify the square root algorithm in the K7 microprocessor [24]. Later, in 1999 he also verified the square root microcode of the K5 microprocessor [23]. In 2000, researchers from Intel Corporation verified the square root algorithm used in an Intel processor with the Forte system that combines symbolic trajec-

tory evaluation and theorem proving [1]. In 2002, IBM presented a research paper about the formal verification of the IBM Power4 processor that uses Chebyshev polynomials to calculate square root [25]. The team used the ACL2 theorem prover to mechanically verify the square root algorithm. In 2002, Bertot et al. verified the divide-and-conquer part of GMP's square root using the Coq proof assistant [6]. In 2003, Harrison published his work about a square root algorithm verification using HOL Light [14]. This particular algorithm used for floating-point numbers was provided by Intel for a new 64-bit architecture called Itanium to replace some less efficient generic libraries. The main benefits of using theorem proving for the verification of this algorithm were reliability and re-usability. Actually, its proof involved Diophantine equations that were very tedious and error-prone to do by hand. The author argues that all the proof process should be done in the same tool – the proof assistant – because it uses a strict logical deduction process. In 2011, Shelekhov proposed a specification and verification of square root using PVS [26]. The paper concludes that synthesis of programs of the standard functions such as *floor*, *isqrt*, and *ilog2* is found to be less tedious than the deductive verification of these programs. In 2016, Oracle presented a research work about the formal verification of a square root implementation [21]. They used ACL2 and interval arithmetic to verify the low-level Verilog descriptions of the floating-point division and square root implementations in the SPARC ISA, and discovered new optimizations (lookup table reductions) while doing so. In 2018, Intel Corporation presented a research paper about the proof of correctness of square root using a digit serial method (DSM) and a theorem prover (HOL-Light) [12]. A DSM is an algorithm that determines the digits of a real number serially, starting with the leading digit. In 2019, Melquiond et al. presented a paper about the formal verification of the GMP library's algorithm for calculating the square root of a 64-bit integer using Why3 [19]. This algorithm can be seen as a fixed-point arithmetic algorithm that implements Newton method. The authors used the WhyML modeling language to implement GMP's algorithm together with its specification and then the Why3 tool to prove its correctness automatically. The resulting proved WhyML model was then extracted to correct-by-construction C code, which was binary compatible to the one from GMP. The authors reported that this work took a few days. They also used ghost code in WhyML to simplify the verification conditions.

The studies about standard mathematical functions and in particular square root specification and validation cited above are all platform-dependent. A new approach proposed by Shilov et al. consisted in a platform-independent verification of standard mathematical functions. In [28], this approach was applied to the square root function and combines a manual (pen-and-paper) verification of a base case that proves the algorithm's correctness with real numbers to provide a proof-outline for the verification of the algorithm for machine numbers. The function implements Newton method and uses a lookup table for initial approximations. The specification is done in terms of total correctness assertions with use of precise arithmetic and the mathematical square root and the verification is done in Floyd-Hoare style. A proof of correctness of the algorithm is given for a fixed-point arithmetic and for a floating-point arithmetic. The primary purpose of the paper is to make explicit the properties of the machine arithmetic that are sufficient to perform the verification presented in the paper. Computer-aided implementation and validation of the proof using ACL2 was partially done, the complete ACL2 implementation was left for future studies.

Conclusions

In this paper, we have presented our experiments with automatic deductive proof of correctness of a discrete-valued function calculating a square root by interpolation. We used Frama-C WP and GNATprove to prove the correctness of the function, but we encountered some difficulties with the nonlinear formula of the linear interpolation. Three non-standard approaches worked well for us: the use of bit-vectors in SPARK, the direct SMT-LIB quantifier-free output of Frama-C and the static analysis with Astrée. Bit-vectors are well supported in most modern SMT solvers and are well suited for problems that involve modular arithmetic, but scaling is sometimes difficult. For our use case, SMT requests without quantifiers performed and scaled better because there was no need for bit-vectors. Abstract Interpretation analysis gave more confidence in proving that there was no overflow in the linear interpolation calculus. We have proposed a methodology to use a combination of these different methods until the proof is done. We also show that using industrial use cases with off-the-shelf tools does not always scale, but if we work with researchers, we can find a solution and improve the tools.

Using deductive methods is very promising in an industrial context for safety-critical applications. It can replace unit tests as shown in [20] and thus decrease cost while increasing quality. It is also an intellectual activity that brings more satisfaction for engineers compared to testing.

References

[1] Aagaard M. D., Jones R. B., Kaivola R., Kohatsu K. R., Seger C.-J. H., "Formal Verification of Iterative Algorithms in Microprocessors", *Proceedings of the 37th Annual Design Automation Conference (DAC 2000)*, 2000, 201–206.

[2] Hoare A., Chapron P., Abrial J. R., *The B-book: Assigning Programs to Meanings*, Cambridge University Press, New York, NY, USA, 1996.

[3] Barnett M., Leino K. R. M., "Weakest-Precondition of Unstructured Programs", *Softw. Eng. Notes*, **31**:1 (2005), 82–87.

[4] Barrett C., Conway C. L., Deters M., Hadarean L., Jovanovi'c D., King T., Reynolds A., Tinelli C., "Cvc4.", *Computer Aided Verification. CAV 2011. LNCS*, **6806** (2011), 171–177.

[5] Barrett C. et al., "The SMT-LIB Standard: Version 2.0.", *Tech. rep.*, 2010.

[6] Bertot Y., Magaud N., Zimmermann P., "A Proof of GMP Square Root", *Journal of Automated Reasoning*, **29**:3-4 (2002), 225–252.

[7] Chapman R., "Industrial Experience with SPARK", *ACM SIGAda Ada Letters*, **20**:4 (2000), 64–68.

[8] Conchon S., Coquereau A., Iguernlala M., Mebsout A., "Alt-Ergo 2.2", *SMT Workshop: International Workshop on SMT. Oxford, United Kingdom*, 2018.

[9] De Moura L., Bjørner N., "Z3: An Efficient SMT Solver", *TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg*, **4963** (2008), 337–340.

[10] Dijkstra E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *ACM*, **18**:8 (1975), 453–457.

[11] Dutertre B., "Yices 2.2", *International Conference on Computer Aided Verification. Springer, Cham*, 2014, 737–744.

[12] Ferguson W. E., Bingham J., Erkök L., Harrison J. R., Leslie-Hurd J., "Digit Serial Methods with Applications to Division and Square Root", *IEEE Transactions on Computers*, **67**:3 (2017), 449–456.

[13] Flanagan C., Flanagan C., Saxe J. B., "Avoiding Exponential Explosion: Generating Compact Verification Conditions", *ACM SIGPLAN Not.*, **36**:3 (2001), 193–205.

[14] Harrison J., "Formal Verification of Square Root Algorithms", *Formal Methods in System Design*, **22**:2 (2003), 143–153.

[15] Hoare C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, **12**:10 (1969), 576–580.

[16] Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B., "Frama-C: A Software Analysis Perspective", *Formal Aspects of Computing*, **27**:3 (2015), 573–609.

[17] Kuliamin V. V., "Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers", *Programming and Computer Software*, **33**:3 (2007), 154–173.

[18] Mauborgne L., "Astrée: Verification of Absence of Runtime Error", *In: Jacquart R. (eds) Building the Information Society. IFIP International Federation for Information Processing*, **156** (2004), 385–392.

[19] Melquiond G., Rieu-Helft R., "Formal Verification of a State-of-the-Art Integer Square Root", *IEEE 26th Symposium on Computer Arithmetic (ARITH), Kyoto, Japan*, 2019, 183–186.

[20] Moy Y., Ledinot E., Delseny H., Wiels V., Monate B., "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience", *IEEE Soft*, **30**:3 (2013), 50-57.

[21] Rager D. L., Ebergen J., Nadezhin D., Lee A., Chau C. K., Selfridge B., "Formal Verification of Division and Square Root Implementations, an Oracle Report.", *Formal Methods in Computer-Aided Design (FMCAD)*, 2016, 149–152.

[22] Randimbivololona F., Souyris J., Baudin P., Pacalet A., Raguideau J., Schoen D., "Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach", *In: Wing J. M., Woodcock J., Davies J. (eds) — Formal Methods. FM 1999*, **1709** (1999), 1798–1815.

[23] Russinoff D. M., "A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode", *Formal Methods in System Design*, **14**:1 (1999), 75–125.

[24] Russinoff D. M., "A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7$^{\text{TM}}$ Processor", *LMS J. Comput. Math. (UK)*, **1** (1998), 148–200.

[25] Sawada J., Gamboa R., "Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem", *LNCS. Formal Methods in Computer-Aided Design. FMCAD 2002.*, **2517** (2002), 274–291.

[26] Shelekhov V. I., "Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements", *Automatic Control and Computer Sciences*, **45**:7 (2011), 421–427.

[27] Shilov N. V., Anureev I. S., Bodin E. V., "Generation of Correctness Conditions for Imperative Programs", *Programming and Computer Software*, **34**:6 (2008), 307–321.

[28] Шилов Н. В., Кондратьев Д. А., Ануреев И. С., Бодин Е. В., Промский А. В., "Платформенно-независимая спецификация и верификация стандартной математической функции квадратного корня", *Моделирование и анализ информационных систем*, **25**:6 (2018), 637-666; [Shilov N. V., Kondratyev D. A., Anureev I. S., Bodin E. V., Promsky A. V., "Platform-Independent Specification and Verification of the Standard Mathematical Square Root Function", *Modeling and Analysis of Information Systems*, **25**:6 (2018), 637-666, (in Russian).]

[29] Todorov V., Boulanger F., Taha S., "Formal Verification of Automotive Embedded Software", *Proceedings of the 6th Conference on Formal Methods in Software Engineering. ACM, New York, USA*, 2018, 84–87.

Тодоров В., Таха С., Буланже Ф., Эрнандес А., "Доказательство свойств дискретных функций с помощью дедуктивного доказательства: приложение к квадратному корню", *Моделирование и анализ информационных систем*, **26**:4 (2019), 520–533.

DOI: 10.18255/1818-1015-2019-4-520-533

Аннотация. В течение многих лет автомобильные встраиваемые системы проверялись только тестированием. В ближайшем будущем усовершенствованные системы помощи водителю (ADAS) будут играть большую роль в дизайне и разработке программного обеспечения автомобиля. Кроме того, увеличение их критического уровня может привести к тому, что власти потребуют сертификации этих систем. Мы думаем, что привнесение формальных доказательств в их развитие может помочь обеспечить выполнение свойств безопасности и получить эффективный процесс сертификации. Другие отрасли (например, аэрокосмическая, железнодорожная, ядерная), которые создают критические системы, требующие сертификации, также могут быть заинтересованы в развитии формальных методов проверки. Одним из этих методов является дедуктивное доказательство. Это может дать более высокий уровень уверенности в доказательстве критических свойств безопасности и даже избежать модульное тестирование. В этой статье мы выбрали вариант прикладного использования: функцию, вычисляющую квадратный корень с помощью линейной интерполяции. Мы используем дедуктивное доказательство, чтобы доказать его правильность и показать ограничения, с которыми мы сталкиваемся при работе с готовыми инструментами. Мы предлагаем подходы для преодоления некоторых ограничений, связанных с этими инструментами,чтобы преуспеть с доказательством. Эти подходы могут быть применены к аналогичным проблемам, которые часто встречаются в автомобильном встроенном программном обеспечении.

Ключевые слова: формальные методы, дедуктивное доказательство, доказательство дискретных функций

Об авторах:
Васил Тодоров, orcid.org/0000-0002-2739-499X,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, Франция, e-mail: vassil.todorov@lri.fr

Сафуан Таха, orcid.org/0000-0003-3950-6415, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, Франция, e-mail: safouan.taha@lri.fr

Фредерик Буланже, orcid.org/0000-0003-3185-2807, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, Франция, e-mail: frederic.boulanger@lri.fr

Армандо Эрнандес, orcid.org/0000-0003-4555-4616,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, Франция, e-mail: armando.hernandez@mpsa.com

Благодарности:
Эта работа была поддержана Groupe PSA, французским многонациональным производителем автомобилей и мотоциклов, которые продаются под брендами Peugeot, Citroën, DS, Opel и Vauxhall.

Моделирование и анализ информационных систем. Т. 26, №4 (2019), с. 534–549
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 534–549

Theory of data

DOI: 10.18255/1818-1015-2019-4-534-549

УДК 004.822, 681.51

Методы специализации онтологии процессов, ориентированной на верификацию

Гаранина Н. О., Анureев И. С., Боровикова О. И., Зюбин В. Е.

Поступила в редакцию 11 сентября 2019
После доработки 16 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. Удобная для пользователя формальная спецификация и верификация параллельных и распределённых систем, принадлежащих различным предметным областям, таким как системы автоматического управления, телекоммуникации, бизнес-процессы, являются активными темами исследований в силу их практической значимости. В этой статье мы представляем методы разработки специализированных ориентированных на верификацию онтологий процессов, которые используются для описания параллельных и распределенных систем предметных областей. Одним из преимуществ таких онтологий является их формальная семантика, которая делает возможной формальную верификацию описанных систем. Наш метод основан на абстрактной онтологии процессов, ориентированной на верификацию. Мы используем два метода специализации абстрактной онтологии процессов. Декларативный метод с помощью специализации классов исходной онтологии, введения новых декларативных классов, а также системы аксиом задаёт ограничения для классов и отношений абстрактной онтологии. Конструктивный метод использует техники семантической разметки и сопоставления с образцом, чтобы связать понятия предметной области с классами абстрактной онтологии процессов. Мы даём подробные онтологические спецификации этих техник. Наши методы сохраняют формальную семантику исходной онтологии процессов и, следовательно, возможность применения формальных методов верификации к специализированным онтологиям процессов. Мы показываем, что конструктивный метод является уточнением декларативного метода. Построение онтологии типовых элементов систем автоматического управления иллюстрирует наши методы: разработано декларативное описание классов и ограничений специализированной онтологии в системе Protégé на языке OWL с использованием правил вывода на языке SWRL и построена система шаблонов семантической разметки, которая реализует типовые элементы систем автоматического управления.

Ключевые слова: онтология процессов, специализация, аксиомы онтологии, сопоставление с образцом, семантическая разметка, системы автоматического управления, верификация

Для цитирования: Гаранина Н. О., Анureев И. С., Боровикова О. И., Зюбин В. Е., "Методы специализации онтологии процессов, ориентированной на верификацию", *Моделирование и анализ информационных систем*, **26**:4 (2019), 534–549.

Об авторах:
Гаранина Наталья Олеговна, orcid.org/0000-0001-9734-3808, канд. физ.-мат. наук, с.н.с.,
Институт систем информатики им. А.П. Ершова СО РАН, Институт автоматики и электрометрии СО РАН,
пр. Акад. Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: garanina@iis.nsk.su

Анureев Игорь Сергеевич, orcid.org/0000-0001-9574-128X, канд. физ.-мат. наук, с.н.с.,
Институт систем информатики им. А.П. Ершова СО РАН, Институт автоматики и электрометрии СО РАН,
пр. Акад. Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: anureev@iis.nsk.su

Боровикова Олеся Игнатьевна, orcid.org/0000-0001-5456-8513, м.н.с.
Институт систем информатики им. А.П. Ершова СО РАН,
пр. акад. Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: olesya@iis.nsk.su

Зюбин Владимир Евгеньевич, orcid.org/0000-0002-8198-3197, доктор тех. наук, зав. лаб.,
Институт автоматики и электрометрии СО РАН,
пр. Акад. Коптюга, 1, г. Новосибирск, 630090 Россия, e-mail: zyubin@iae.nsk.su

Благодарности:
Исследование выполнено при финансовой поддержке РФФИ в рамках научных проектов № 17-07-01600 и № 19-07-00762, а также в рамках темы госзадания ИАиЭ СО РАН № АААА-А17-11706061006-6.

Введение

Наша долгосрочная цель – разработка комплексного подхода к поддержке формальной верификации параллельных систем для обеспечения их качества. Решение включает методы извлечения формальных моделей и свойств параллельных систем из текстов технической документации, а также инструменты для ручного исправления извлеченной информации и обогащения ее новыми сущностями.

Проектируемый нами интеллектуальный комплекс для поддержки формальной верификации параллельных систем автоматически извлекает и генерирует системные требования. Мы разработали онтологию требований *SO* как первый шаг к созданию этого комплекса [3]. Другим ключевым компонентом комплекса является онтология процессов *PO* для параллельных/распределенных систем [4]. Содержание этих онтологий, то есть множества экземпляров их классов, является онтологическим описанием некоторой параллельной/распределенной системы и требований к ней. Эти описания могут быть получены из технической документации с помощью нашей системы извлечения информации, которая использует структуру онтологий в правилах анализа документации и для хранения результата извлечения [5]. В силу неоднозначности естественного языка и недостаточной формализованности технической документации, может потребоваться корректирование извлечённой информации специальными редакторами. С помощью этих редакторов также можно, независимо от модуля извлечения информации, разрабатывать онтологические описания систем параллельных/распределенных процессов и требований. Эти описания являются основой для формальной верификации таких систем процессов, поскольку для онтологии требований и онтологии процессов можно задать формальную семантику. В частности, в работе [3] семантика онтологии *SO* задана как формулы темпоральной логики LTL [6], а в работе [4] семантика онтологии *PO* задана как помеченная система переходов. Для верификации системы процессов необходимо определить подходящий верификатор (например, инструмент проверки моделей) с учетом формальной семантики представления требований и процессов. Если он существует, мы транслируем онтологическое описание системы в язык спецификации модели выбранного верификатора, а описание требований переводится на язык спецификации свойств (обычно этот язык является темпоральной логикой). Работа с требованиями в нашей системе, кроме использования формальной семантики, включает также их представление на естественном языке и в графической форме.

Методы, предложенные в этой статье, связаны как с задачей извлечения информации о параллельной/распределенной системе процессов из технической документации, так и с разработкой редактора для построения и корректирования онтологического описания таких систем. В этих задачах используется онтология

процессов PO, которая описывает параллельные системы как состоящие из взаимодействующих параллельных процессов, характеризуемых локальными и разделяемыми переменными, каналами для обмена сообщениями, действиями обработки переменных и содержания каналов. Однако при рассмотрении конкретных предметных областей выразительные возможности онтологии PO оказываются слишком абстрактными, чтобы, с одной стороны, задавать корректные правила извлечения информации из технической документации, и с другой – чтобы описания системы процессов были исчерпывающими и понятными инженерам требований и разработчикам программных систем в заданных предметных областях.

Поскольку наш комплекс поддержки верификации может быть использован в разных предметных областях, необходимо разработать методы специализации абстрактной онтологии процессов PO для конкретных предметных областей, чтобы создавать экземпляры процессов, специфичные для предметной области, которые имеют переменные, каналы и действия, соответствующие их предметной специализации. Например, в системе автоматического управления процесс, задающий датчик, должен быть обязательно связан каналом связи по меньшей мере с одним процессом, задающим контроллер. В этой статье мы предлагаем два метода специализации абстрактной онтологии процессов PO (раздел 1.).

Декларативный метод, описанный в разделе 2., основан на добавлении к онтологии PO классов, специфических для выбранной предметной области, которые наследуют классы исходной онтологии, новых декларативных классов, не наследующих классы исходной онтологии, но необходимых для задания специфических ограничений отношений между наследующими классами, и аксиом, ограничивающих значения атрибутов этих классов и отношения между ними. Наследование здесь определено как расширение исходных классов декларативными предметно-специфическими атрибутами (к которым также относится и имя нового класса). В силу декларативности такого наследования, формальная семантика системы процессов, описанных новыми классами, не изменяется, поскольку она задаётся только в терминах классов онтологии PO, а новые декларативные классы не используются. Новые аксиомы ограничивают как значения наследуемых, так и новых декларативных атрибутов, что также не влияет на формальную семантику. Поэтому такая специализация PO для заданной предметной области сохраняет возможность формальной верификации системы процессов специализированной онтологии. Итак, специализированная онтология процессов отличается от исходной онтологии процессов PO декларативными предметно-специфическими атрибутами классов (включая ненаследующие классы) и набором аксиом и правил, которые определяют специфичные ограничения на атрибуты классов и отношения. Аксиомы и правила могут быть использованы как для настройки на предметную область системы извлечения информации из технической документации, так и для проверки целостности и согласованности содержания онтологии. В нашем случае онтология служит для представления параллельной/распределенной системы, поэтому целостность и согласованность означают, что экземпляры процессов соответствуют процессам предметной области, то есть имеют все необходимые переменные, каналы и действия.

Однако декларативный метод не даёт способа задания содержания специализированной онтологии. Поэтому мы предлагаем конструктивный метод, который может быть использован для создания предметно-ориентированного содержания он-

тологии процессов с использованием новой онтологии предметно-ориентированных шаблонов. Построение этого содержания включает в себя несколько этапов. На первом этапе мы используем абстрактную онтологию процессов PO для создания новой онтологии семантически размеченных процессов, за счёт обогащения её классов атрибутами семантической разметки, содержащими строковое описание терминов предметной области. Эта новая онтология дана в разделе 3. Затем множество экземпляров предметно-ориентированных процессов может быть определено с использованием ориентированной на процессы онтологии семантической разметки, описанной в разделе 4. Экземпляры этой онтологии определяют правила, с помощью которых задается набор экземпляров специализированных процессов. Декларативный метод специализированной онтологии процессов подходит для настройки процесса извлечения данных о параллельной системе и для проверки корректности описаний уже созданных или извлеченных систем. Для создания новой специализированной онтологии процессов и ее наполнения лучше использовать конструктивный метод, поскольку он позволяет на основе шаблонов строить экземпляры новой онтологии.

Мы иллюстрируем наши методы на примере предметной области систем автоматического управления (САУ). Для иллюстрации декларативного метода разработаны описания классов типовых элементов САУ, а также их свойств и аксиом, средствами языка OWL (Web Ontology Language [11]) в редакторе Protégé [12] с использованием языка правил SWRL (Semantic Web Rule Language [13]). Эти правила задают условия, которые обеспечивают в Protégé проверку корректности онтологического описания специализированных процессов с помощью машины логического вывода Hermit [9]. Разработка онтологии процессов для этой области особенно важна, потому что удобная для пользователя формальная спецификация и верификация систем автоматического управления и, в целом, кибер-физических систем, имеют важное практическое значение.

1. Онтология процессов

Мы рассматриваем *онтологию* как структуру, которая включает в себя следующие элементы: (1) конечное непустое множество *классов*, (2) конечный непустой набор *атрибутов данных и атрибутов отношений*, и (3) конечный непустой набор *доменов атрибутов данных*. Каждый класс определяется набором атрибутов. *Класс c является подклассом* другого класса C ($c < C$ наследует родительский класс), если все атрибуты класса-родителя принадлежат также классу-наследнику. Атрибуты данных получают значения из доменов, а значения атрибутов отношений являются экземплярами классов. *Экземпляр класса* определяется набором значений атрибутов для этого класса. *Содержание* онтологии — это набор экземпляров ее классов. *Аксиомы и правила* ограничивают значения атрибутов классов.

Содержание онтологии процессов PO представляет онтологическое описание параллельной/распределенной системы. Мы рассматриваем такую систему как множество взаимодействующих процессов. Процессы (описываемые классом $Process$) характеризуются набором локальных и разделяемых переменных, списком действий над этими переменными, которые меняют их значения, списком каналов для взаимодействия, а также списком коммуникационных действий по отправке сообщений.

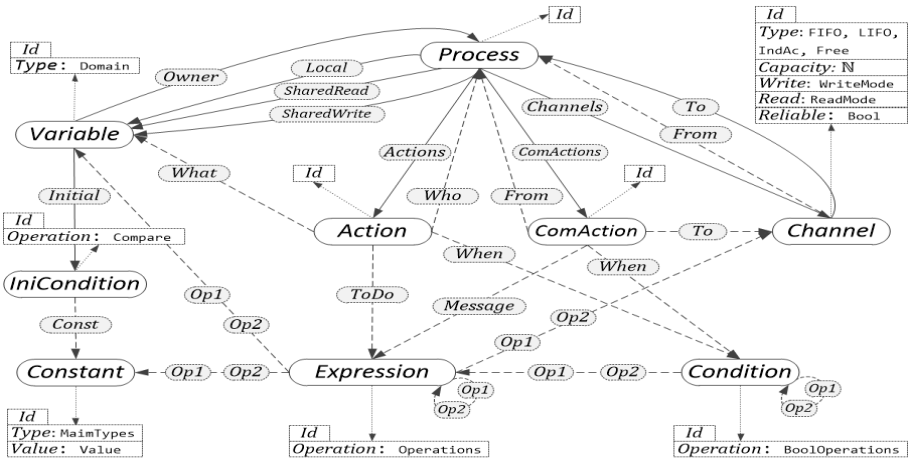

Рис. 1. Онтология процессов
Fig. 1. The Process Ontology

Переменные (класс *Variable*) и константы (класс *Constant*) процессов принимают значения в областях базовых типов. В зависимости от формальной семантики, приписываемой онтологии процессов, базовые типы и операции, определённые для переменных этих типов, могут различаться. В частности, при задании семантики как конечной системы помеченных переходов [4] в качестве базовых типов мы рассматриваем булевский тип, конечные подмножества целых и строк для перечислимых типов, а также конечные типы, производные из перечисленных. Далее в статье мы считаем, что предметные области таковы, что их системы процессов допускают упомянутую формальную семантику, т.е. не используют, например, бесконечные множества действительных чисел. Начальные значения переменных могут определяться сравнением с константами. Действия процессов (класс *Action*) изменяют значения переменных. Условие срабатывания для каждого действия определяется как охранное условие (класс *Condition*) относительно значений переменных и содержимого каналов. Процессы могут посылать сообщения через каналы (класс *Channel*) при выполнении охранных условий (класс *Condition*). Каналы коммуникации характеризуются типом чтения сообщений, ёмкостью, режимами записи/чтения и надежностью. Отметим, что при выборе формальной семантики онтологии как конечной системы помеченных переходов, ёмкость каналов должна быть ограничена. Рисунок 1 представляет онтологию процессов. Классы отображаются как белые овалы. Отношения между классами показаны пунктирными стрелками с именами в серых овалах. Эти стрелки являются сплошными, если отношение “один ко многим”, и пунктирными, если отношение “один к одному”. Атрибуты данных классов, помещённые в штрих-пунктирные прямоугольники, связаны с соответствующими классами штрих-пунктирными стрелками. Универсальные классы онтологии *PO* абстрагируются от особенностей предметной области. В следующем разделе мы определим метод специализации и расширения онтологии посредством добавления специализирующих классов и ограничивающих аксиом для предметной области.

2. Специализированная онтология процессов

Специализация онтологии – это построение иерархии классов [7]. В случае специализации онтологии процессов мы строим новую специализированную онтологию, расширяя исходную онтологию следующими элементами. Во-первых, необходимо задать множество классов для описания специфических для предметной области процессов, переменных, констант, каналов, действий процессов и т.п., которые наследуют классы исходной онтологии. При этом дополнительные атрибуты класса-наследника являются чисто декларативными, т.е. связывая абстрактные понятия онтологии процессов с понятиями предметной области, они не оказывают влияния на формальную семантику (в большинстве случаев дополнительные атрибуты отсутствуют). Например, для предметной области автоматических систем управления новый атрибут $SpecSen$ в описании процесса-датчика служит для спецификации вида и диапазона значений наблюдаемой им величины. Во-вторых, определяется множество классов, не наследующих классы онтологии PO, но необходимых для описания онтологических ограничений предметной области. Например, экземпляры класса $PhysicalQuantity$ является значениями атрибута $SpecSen$ процесса-датчика и могут служить для проверки соответствия типа его наблюдаемой величины. И, наконец, необходимо описать аксиомы и правила, ограничивающие значения атрибутов новых классов. В частности, явным образом нужно задать, что спецификация вида и диапазона значений процесса-датчика и спецификация вида и диапазона значений наблюдаемой им величины должны совпадать.

Формально специализация онтологии процессов PO описывается следующим образом. Обозначим специализированную онтологию как $SpecPO$, множество аксиом онтологии O как $Axioms(O)$, множество классов онтологии O как $Classes(O)$, множество атрибутов класса c как Atr_c, где $c \in O$ и $O \in \{PO, SpecPO\}$. Для онтологии O множество атрибутов классов, значения которых используются при описании её формальной семантики, обозначим как $FormSem(O)$. Отметим, что $FormSem(PO) = \cup_{c \in Classes(O)}(Atr_c \setminus Id)$, т.е. идентификаторы не влияют на формальную семантику (см. рис. 1 и [4]). Тогда:

1) $Classes(SpecPO) = InhCls \cup SpecCls$, где $\forall c \in InhCls\ \exists C \in Classes(PO) : c < C$ и $\forall c \in SpecCls\ \nexists C \in Classes(PO) : c < C$. При этом $FormSem(SpecPO) \subseteq FormSem(PO)$.
2) $Axioms(SpecPO) = Axioms(PO) \cup SpecAxs$, при этом $\forall c \in Classes(SpecPO)\ \exists a \in SpecAxs : a(uses)c$, где $a(uses)c$ означает, что в описании аксиомы a упоминается класс c. Аксиомы могут накладывать ограничения на количество значений атрибутов и на их возможный диапазон (включая точное соответствие). В случае атрибутов отношений этот диапазон определяет возможные классы (включая ограничения на их атрибуты), с которым данный класс связан отношениями.

Таким образом, классы исходной онтологии служат паттернами онтологического проектирования [7] для создания специализированных онтологий процессов, что иллюстрируется примером онтологии для типовых элементов систем автоматического управления в разделе 5. Далее мы опишем конструктивный подход к специализации абстрактной онтологии процессов для предметных областей.

3. Семантически-размеченная онтология процессов

В этом разделе мы формально определим наш метод семантической разметки онтологии процессов. Эта разметка используется для сопоставления абстрактных процессов PO со специфическими процессами выбранной предметной области. Разметка заключается в добавлении к классам онтологии PO специальных семантических меток и атрибутов, связывающих их с понятиями предметной области. Обогащённые классы вместе с несколькими вспомогательными классами образуют новую семантически-размеченную онтологию (онтологию $SMPO$). Экземпляры предметно-ориентированных процессов могут конструироваться, используя онтологию $SMPO$ и процесс-ориентированную онтологию шаблонов семантической разметки (онтологию $POSMPO$), описанную в следующем разделе.

Онтология $SMPO$ содержит домены $Classes$, $Domains$, $Types$, $Values$, $SLabel$ и $SAttribute$, классы $AValue$, $Element$, T и $Element\_T$ для каждого $T \in Domains$.

Домены $Classes$ и $Domains$ включают имена классов и доменов онтологии PO, соответственно. Домен $Types = Classes \cup Domains$ включает все имена онтологии PO. Домен $Values$ включает все значения атрибутов из онтологии PO: $Values = \cup_{T \in Types} Val(T)$, где $Val(T)$ — множество значений для T, которые являются экземплярами $T \in Classes$ или соответствующими значениями для $T \in Domains$.

Домен $SLabel$ — это конечное множество семантических меток, представленных строками. Метки специфицируют информацию, связанную с экземплярами классов онтологии PO. Эта информация может быть о предметной области (“sensor” или “pressure”) или особенностях моделирующих процессов (“periodic start”).

Домен $SAttribute$ — это конечное множество семантических атрибутов (S-атрибутов), представленных строками. Подобно меткам, S-атрибуты специфицируют информацию о предметной области, связанную с экземплярами классов онтологии PO. Разница состоит в том, что семантические атрибуты имеют значения (“100”, “$true$” или “экземпляр класса $Controller$”).

Класс $AValue$ имеет два однозначных атрибута: $Attribute$ со значениями в домене $SAttribute$ и $Value$ со значениями в $Values$, которые специфицируют имя S-атрибута и его значение. Его экземпляры называются атрибутными значениями.

Класс T онтологии $SMPO$ — это класс T онтологии PO, обогащённый двумя многозначными атрибутами: $SLabels$ со значениями в $SLabel$ и $SAattributes$ со значениями в $AValue$. Эти атрибуты семантически размечают экземпляры классов онтологии PO, специализируя их для конкретной предметной области, и называются размечающими атрибутами (M-атрибутами). Атрибут $SLabels$ специфицирует множество семантических меток. Атрибут $SAttributes$ специфицирует множество S-атрибутов с их значениями с естественным ограничением, что любой S-атрибут может входить в его значение не более одного раза. Атрибуты класса T, не являющиеся M-атрибутами, называются *базовыми атрибутами*.

Класс $Element\_T$ имеет только M-атрибуты и однозначный атрибут $Value$ со значением в T, специфицирующий значение из домена T. Таким образом, в онтологии $SMPO$ значения доменов онтологии PO могут включать информацию о предметной области, задаваемую M-атрибутами.

Класс $Element$ имеет только M-атрибуты. Экземпляры этого класса описывают

информацию о предметной области, которая не выражается естественным образом как специализация классов онтологии PO.

Проиллюстрируем добавление информации о предметной области к элементам онтологии PO на примере датчика, измеряющего температуру в градусах Цельсия в диапазоне от 0 до 1000. Такой датчик задаётся следующим экземпляром класса `Process` онтологии $SMPO$:

```
Process(BAVs, SLabels:{"sensor"},
SAttributes:{AValue (Attribute:"Dimension", Value:"temperature"),
AValue(Attribute:"unit", Value:"Celsius"),
AValue(Attribute:"range",
Value:Element(SLabels:{"range"},
SAttributes:{AValue(Attribute:"left", Value:"0"),
AValue(Attribute:"right", Value:"1000")})})
```

Листинг 1. Экземпляр датчика
Listing 1. The Instance of a Sensor

Здесь элемент $T(A_1 : V_1, \ldots, A_n : V_n)$ обозначает экземпляр класса T со значениями V_1, ..., V_n атрибутов A_1, ..., A_n, множество $\{V_1, \ldots, V_n\}$ перечисляет значения многозначного атрибута, и $BAVs$ — базовые атрибуты из онтологии PO.

Таким образом, онтология $SMPO$ позволяет описывать экземпляры понятий предметных областей в контексте онтологии PO. Однако, она не позволяет описывать ограничения на эти экземпляры. В следующем разделе мы определим онтологию $POSMPO$, которая накладывает ограничения на семантическую разметку, и тем самым на экземпляры понятий предметной области, специфицируемые с помощью этой разметки. Эта онтология определяет понятия предметной области, используя шаблоны для накладывания ограничений в экземплярах онтологии $SMPO$ на местность и значения их атрибутов.

4. Процесс-ориентированная онтология шаблонов семантической разметки

Онтология $POSMPO$ включает все домены и классы онтологии $SMPO$, дополнительные домены $AMatchSizes$ и $AMatchOperations$, а также класс $AMatch$.

Пусть n, m — неотрицательные целые числа. Домен $AMatchArities = \{$"m", "m|0","$m-k$","$m-k|0$","$m-$","$m-|0$","$-m$"$\}$ описывает ограничения на число значений атрибутов классов онтологии $SMPO$.

Домен $AMatchOperations = \{$" = "," < "," <= ","! = "," > "," => ","in", "$oneof$","all"$\}$ описывает множество операций сопоставления. Они сопоставляют значение атрибута экземпляра класса онтологии $SMPO$ со значением соответствующего атрибута экземпляра класса онтологии $POSMPO$. Множество значений этого домена может расширяться для конкретной предметной области.

Экземпляры классов онтологии $POSMPO$ называются шаблонами семантической разметки. Каждый шаблон специфицирует множество экземпляров классов онтологии $SMPO$, сопоставляемых с этим шаблоном. Класс T онтологии $POSMPO$ имеет те же самые атрибуты, как и класс T онтологии $SMPO$, но при этом их значения содержатся в $AMatch$. Класс $AMatch$ специфицирует правила для сопоставления значений атрибутов онтологии $SMPO$ с шаблонами для них и имеет

три атрибута. Однозначный атрибут Ar со значениями в $AMatchArities$ ограничивает число сопоставляемых значений. Однозначный атрибут Op со значениями в $AMatchOperations$ определяет операцию сопоставления. Многозначный атрибут Pat со значениями в $Values[SMPO]$ задаёт шаблоны для значений атрибутов.

Пусть $V.A$ обозначает значение атрибута A экземпляра V и $|S|$ обозначает мощность множества S. Экземпляр V класса T онтологии $SMPO$ сопоставляется с шаблоном P класса T онтологии $POSMPO$ тогда и только тогда, когда для каждого атрибута A шаблона P такого, что $P.A = AMatch(Ar : R, Op : O, Pat : \{V_1, \ldots, V_n\})$, выполнены следующие свойства:

– Если $R =$ "m", то $|V.A| = m$.
– Если $R =$ "$m-$", то $|V.A| \geq m$.
– Если $R =$ "$-m$", то $|V.A| \leq m$.
– Если $R =$ "$m-k$", то $k \leq |V.A| \leq m$.
– Если $R =$ "$m|0$", то $|V.A| = m$, или $|V.A| = 0$.
– Если $R =$ "$m-|0$", то $|V.A| \geq m$, или $|V.A| = 0$.
– Если $R =$ "$m-k|0$", то $n \leq |V.A| \leq m$, или $|V.A| = 0$.
– Если $O =$ " = ", то $n = 1$, и $V' = V_1$ для каждого $V' \in V.A$. Случаи, когда $O \in \{$" = "," < "," <= ","! = "," > "," > "$\}$ определяются аналогично. Эти случаи ограничивают сравниваемые значения атрибутов.
– Если $O =$ "in", то $n = 1$, и $V' \in V_1$ для каждого $V' \in V.A$. Этот случай определяет принадлежность значений атрибутов.
– Если $O =$ "$oneof$", то V сопоставляется с $upd(P, A, V_i)$ для некоторого $1 \leq i \leq n$, где $upd(Q, B, U)$ обозначает результат присваивания значения U атрибуту B экземпляра Q. Этот случай выбирает некоторое значение шаблона для атрибутов.
– Если $O =$ "all", то имеются S_1, ..., S_n такие, что $V.A = \{S_1, \ldots, S_n\}$, $S_i \cap S_j = \emptyset$ для $S_i \neq \emptyset$, $S_j \neq \emptyset$, и $upd(V, A, S_i)$ сопоставляется с $upd(P, A, V_i)$ для каждого $1 \leq i \leq n$. Этот случай выбирает все значения шаблона для атрибутов.
– Если O не определено, и $A \neq SLabels$, или $T = AValue$, и $A \neq Attribute$, то V' сопоставляется с $P.A$ для каждого $V' \in V.A$. Этот случай сводит сопоставление множества значений атрибутов к сопоставлению отдельных значений атрибутов. Оставшиеся случаи — специальные случаи для классов $SLabels$, $AValue$ и $SAttributes$.
– Если O не определено и $A = SLabels$ то $P.SLabels \subseteq V.SLabels$.
– Если O не определено, $T = AValue$, и $A = Attribute$ то $n = 1$, и $V.A = V_1$.
– Если O не определено и $A = SAttributes$ и $sattributes(P) = sattributes(V)$, где $attributes(i)$ — множество имен S-атрибутов в экземпляре i некоторого класса.

Мы определили онтологию $POSMPO$, которая комбинирует онтологию PO с описаниями понятий предметной области. Конкретное множество экземпляров этой онтологии задает правила для построения соответствующей предметно-ориентированной онтологии процессов. Классы и домены онтологии $POSMPO$ задают язык для конструктивного использования аксиом, которые ограничивают абстрактные классы онтологии PO относительно предметной области, так как эти аксиомы могут специфицировать только число значений атрибутов и их области изменения.

Конструктивный метод, как и декларативный, сохраняет формальную семантику исходной онтологии PO, поскольку он не изменяет базовых атрибутов классов этой онтологии, в терминах которых описывается формальная операционная семан-

тика онтологии PO [4], а только добавляет новые семантические атрибуты, отображающие экземпляры классов онтологии PO на понятия предметной области.

Декларативный и конструктивный методы специализации связаны следующим образом: в декларативном методе имена новых классов, наследующих классы онтологии процессов, а также имена их новых атрибутов согласуются со строковыми значениями доменов $SLabel$ и $SAttribute$. Поскольку декларативный метод задаёт семейство систем процессов, а конструктивный позволяет строить конкретную систему, то значения атрибутов, соответствующие строкам из домена $SAttributes$, согласуются только с типичными для этого семейства ограничениями на значения атрибутов. Классы декларативной специализации, не наследующие классы исходной онтологии, согласуются с экземплярами класса $Element$ в той мере, в которой значения их семантических меток описывают общие ограничения семейства систем. Декларативные ограничения специализации на значения атрибутов, т.е. аксиомы и правила, согласуются с экземплярами класса $AMatch$ в той мере, в которой значения их атрибутов описывают общие ограничения семейства систем.

В следующем разделе мы опишем типовые элементы систем автоматического управления (САУ) посредством метода декларативной специализации онтологии процессов и сконструируем эти элементы, используя классы онтологии $POSMPO$.

5. Специализированная онтология процессов для типовых элементов систем автоматического управления

В этом разделе мы определим классы, аксиомы и шаблоны семантической разметки для типовых элементов систем автоматического управления, таких как датчики, контроллеры, исполнительные устройства (актуаторы) и объект управления.

Простые и сложные датчики характеризуются следующими ограничениями. Датчики должны считывать наблюдаемые значения из переменных, разделяемых с объектом управления, и не могут их изменять. Для датчиков должны быть заданы коммуникационные действия для отправки сообщений контроллерам через исходящие каналы. У каждого датчика должна быть хотя бы одна разделяемая переменная и связь с хотя бы одним контроллером. У простых датчиков нет локальных переменных и действий. Они могут наблюдать ровно одну разделяемую переменную и отправлять наблюдаемое значение контроллерам без изменений. Сложные датчики могут трансформировать наблюдаемые значения нескольких разделяемых переменных для передачи контроллеру, используя локальные переменные и действия. Для датчиков должны быть определены физические величины, которые они обрабатывают. Эти величины характеризуются размерами (`"temperature"`, `"pressure"`, `"density"`, etc.), единицами измерения (`"centimeter"`, `"kilogram"`, `"volt"`, etc.) и диапазонами. Эти ограничения и сопутствующие понятия онтологии определяются декларативно классами и аксиомами из листинга 2 и конструктивно — шаблонами из листинга 3.

```
NewClass Sensor: Process
NewAttribute:
SpecSen: PhQuantity;

NewClass SimSensor: Sensor;

NewClass PhQuantity
Attributes:
Dimension: String;
Unit: String;
Range: {Int,Int};

SimSensor: local   exactly 0 Variable
shared  exactly 1 Variable
actions exactly 0 Action
specSen exactly 1 PhQuantity

Sensor(?s)^Variable(?v)^Shared(?s,?v) -> ControlledObject(?o)^Shared(?o,?v)
Sensor(?s)^Channel(?ch)^Channels(?s,?ch)^ComAction(?ca)^From(?ca,?s)^To(?ca,?ch) ->
    Controller(?c)^Channels(?c,?ch)
Sensor(?s)^Variable(?v)^Shared(?s,?v)^PhQuantity(?s,?q) -> Type(?v,?q)
```

Листинг 2. Декларативное описание датчиков
Listing 2. Declarative Description of Sensors

В листинге выше описаны три новых класса, два из которых наследуют класс *Process* исходной онтологии. Здесь и далее декларативные ограничения на количество значений атрибутов приведены в той форме, в какой они отображаются в редакторе онтологий Protégé, так как формальное описание на языке OWL выглядит громоздким. Правила, ограничивающие использование каналов, описаны более формально на языке задания правил SWRL.

```
Process( // Simple sensor
Local:AMatch("0"),
SharedRead:AMatch("1", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Actions:AMatch("0"),
Channels:AMatch("1-",
Channel(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending observed value from simple sensor"})),
SLabels:{"Simple sensor"},
SAttributes: {AValue("Physical quantity", Element(SLabels:{"Physical quantity"})})

Element( // Physical quantity
SLabels:{"Physical quantity"},
SAttributes: {AValue("Dimension", AMatch(Op:"in", Pat:Dimension)),
AValue("Unit", AMatch(Op:"in", Pat:Unit)),
AValue("Range", Element{SLabels:{"Range"}})})

Element( // Range
SLabels:{"Range"},
SAttributes: {AValue("Left", AMatch(Op:"in", Pat:Float)),
AValue("Right", AMatch(Op:"in", Pat:Float))})

Variable( // Observed value
Users:AMatch(Op:"all",
Pat:{AMatch("1", Process(SLabels:{"Controlled Object"})),
AMatch("1-", Process(SLabels:{"Simple sensor"}))}),
SLabels:{"Observed value"},
SAttributes: {AValue("Physical quantity", Element(SLabels:{"Physical quantity"})})

Channel( // Channel from sensor to controller
From:AMatch("1", "oneof", {Process(Slabels:{"Simple sensor"}),
Process(Slabels:{"Complex sensor"})}),
To:AMatch("1-", Process(SLabels:{"Controller"})),
Type:AMatch("1", "=", "FIFO"),
Capacity:AMatch("1", "=", 1),
Write:AMatch("1", "=", "Old"),
Read:AMatch("1", "=", "Keep"),
Reliable:AMatch("1", "=", "true"),
SLabels:{"Channel from sensor to controller"})

ComAction( // Sending observed value from simple sensor
From:AMatch("1", Process(Slabels:{"Simple sensor"})),
```

```
To:AMatch("1-", Channel(SLabels:{"Controller"})),
Message:AMatch("1", Expression(Op1:AMatch("1", Variable(SLabels:{"Observed value"}))))
SLabels:{"Sending observed value from simple sensor"})

Process( // Complex sensor
SharedRead:AMatch("1-", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Channels:AMatch("1-",
Variable(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from complex sensor"})),
SLabels:{"Complex sensor"},
SAttributes: {AValue("Physical quantity", Element(SLabels:{"Physical quantity"})})

ComAction( // Sending message from complex sensor
From:AMatch("1", Process(Slabels:{"Complex sensor"})),
To:AMatch("1-", Channel(SLabels:{"Controller"})),
SLabels:{"Sending message from complex sensor"})
```

Листинг 3. Конструктивное описание датчиков
Listing 3. Constructive Description of Sensors

В этом и следующих листингах мы используем следующие сокращения: `SLabels:S` для `SLabels:AMatch(Value:S)`, `AMatch(R, O, P)` для `AMatch(Ar:R, op:O, Pat:P)`, где `R`, `O`, или `P` могут опускаться, `AMatch(R, O, P)` для `{AMatch(R, O, {P})}`, `AValue(A, V)` для `AValue(Attribute:A, Value:V)` и `T` для `AMatch(Op:"in", Pat:T)`.

Ограничения на *контроллеры*, *актуаторы* и *объекты управления* состоят в следующем. Контроллеры и актуаторы не должны иметь общих переменных. Контроллеры должны иметь выходные каналы, соединяющие их с другими контроллерами и актуаторами, и входные каналы, соединяющие их с датчиками и актуаторами. Актуаторы должны иметь выходные каналы, соединяющие их с контроллерами и объектом управления, и входные каналы, соединяющие их с контроллерами. Должен быть, по меньшей мере, один датчик и, по меньшей мере, один актуатор, соединенный с контроллером через входной и выходной каналы соответственно. Должен быть хотя бы один контроллер и единственный объект управления, связанный с актуатором через входной и выходной каналы соответственно. Объект управления должен быть связан с актуаторами входными каналами. С объектом управления должны быть связаны по крайней мере одна общая переменная, один датчик и один актуатор. Эти ограничения описываются декларативно классами и аксиомами из листинга 4 и конструктивно — шаблонами из листинга 5.

```
NewClass Controller: Process
NewClass Actuator: Process
NewClass Object: Process

Controller: shared exactly 0 Variable

Actuator: shared exactly 0 Variable

Object: shared  min 1 Variable
actions min 1 Action

Controller(?c)^Channel(?ch)^Channels(?c,?ch)^ComAction(?ca)^From(?ca,?c)^To(?ca,?ch)
-> Controller(?d)^Channels(?d,?ch)
Controller(?c)^Channel(?ch)^Channels(?c,?ch)^ComAction(?ca)^From(?ca,?c)^To(?ca,?ch)
-> Actuator(?a)^Channels(?a,?ch)
Controller(?c)^Channel(?ch)^Channels(?c,?ch)^ComAction(?ca)^From(?ca,?c)^To(?ca,?ch)^
Actuator(?a)^Channels(?a,?ch)
-> Channels(?a,?ch1)^ComAction(?ca1)^From(?ca1,?a)^To(?ca1,?ch1)^ Channels(?c,?ch1)
Controller(?c) -> Sensor(?s)^Channel(?ch)^Channels(?s,?ch)^
ComAction(?ca)^From(?ca,?s)^To(?ca,?ch)^Channels(?c,?ch)

Actuator(?a)^Channel(?ch)^Channels(?a,?ch)^ComAction(?ca)^From(?ca,?a)^To(?ca,?ch)
-> Controller(?c)^Channels(?c,?ch)
Actuator(?a)^Channel(?ch)^Channels(?a,?ch)^ComAction(?ca)^From(?ca,?a)^To(?ca,?ch)^
Controller(?c)^Channels(?c,?ch)
```

```
-> Channels(?c,?ch1)^ComAction(?ca1)^From(?ca1,?c)^To(?ca1,?ch1)^ Channels(?a,?ch1)
Actuator(?a) -> Object(?o)^Channel(?ch)^
Channels(?a,?ch)^Channels(?o,?ch)^ComAction(?ca)^From(?ca,?a)^To(?ca,?ch)

Object(?o)^Channel(?ch)^Channels(?o,?ch)
-> Actuator(?a)^Channels(?a,?ch)^ ComAction(?ca)^From(?ca,?a)^To(?ca,?ch)
Object(?o)^Variable(?v)^Shared(?o,?v) -> Sensor(?s)^Shared(?s,?v)
```

Листинг 4. Декларативное описание контроллеров, актуаторов и объектов
Listing 4. Declarative Description of Controllers, Actuators and Objects

В листинге выше перечислены три новых класса, наследующих класс *Process* исходной онтологии. Правила, задающие ограничения на каналы сообщений, оказываются довольно громоздкими, поскольку должны описывать как входные, так и выходные каналы. Декларативное описание классов и ограничений, специализирующих абстрактную онтологию процессов, даёт возможность уточнить детали методов извлечения информации, использующих онтологию для сохранения результата (например, [5]), а также проверить корректность данных о системах процессов, извлечённых из технической документации.

```
Process( // Controller
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from sensor to controller"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"},
AMatch("0-", Channel(SLabels:{"Channel from controller to controller"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from  controller"})),
SLabels:{"Controller"})

Process( // Actuator
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1", Channel(SLabels:{"Channel from actuator to controlled object"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from actuator"})),
SLabels:{"Actuator"})

Process( // Controlled object
SharedRead:AMatch("0"),
SharedWrite:AMatch("1", Variable(SLabels:{"Observed value"})),
Channels:AMatch("1-", Channel(SLabels:{"Channel from actuator to controlled object"}),
ComActs:AMatch("0"),
SLabels:{"Controlled object"})
```

Листинг 5. Конструктивное описание контроллеров, актуаторов и объектов
Listing 5. Constructive Description of Controllers, Actuators and Objects

На рисунке 2 представлена иерархия классов специализированной онтологии процессов для типовых элементов САУ, описанная с помощью редактора *Protégé*. Фрагмент описания класса *SimSensor* содержит ограничения на количество значений его атрибутов, перечисленных в листинге 2.

Каждый шаблон дает правила для определения элемента САУ в контексте онтологии процессов *PO*. По набору таких шаблонов можно задать систему параллельных процессов, которые реализуют типичные элементы САУ. Таким образом, мы показали, как конструктивный метод специализации может использоваться для определения специфических процессов в заданной предметной области.

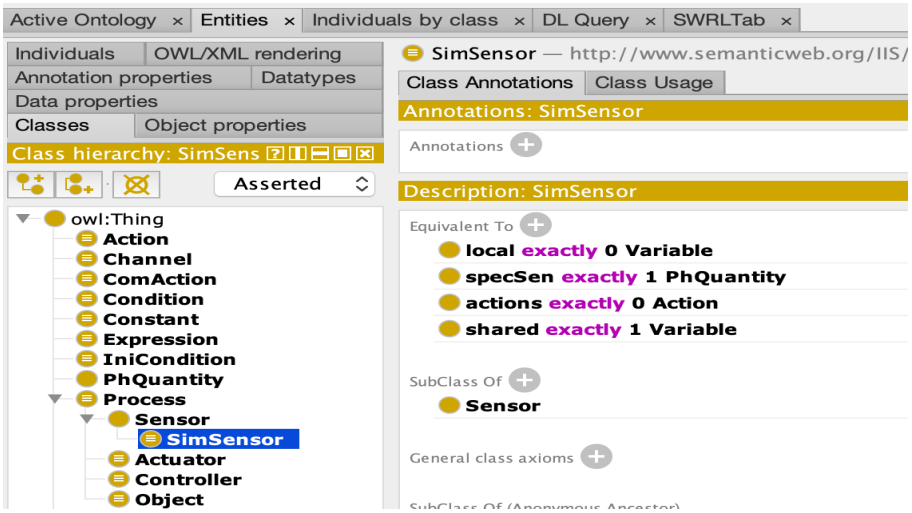

Рис. 2. Фрагмент описания специализированной онтологии процессов
Fig. 2. A fragment of the Specialized Process Ontology

Заключение

Методы разработки специализированных онтологий, основанные на трех базовых онтологиях [8], имеет несколько полезных свойств. Ориентированная на верификацию онтология процессов специфицирует компактную универсальную модель процессов с возможностью определения формальной семантики как помеченной системы переходов, что сделано в работе [4], так и других формализмов, например, машин абстрактных состояний [1] и др. Эта семантика обеспечивает возможность применения формальных методов верификации, в частности, дедуктивную верификацию и проверку моделей. Декларативная специализация необходима для задания правил извлечения информации из технической документации методами, основанными на онтологиях. Онтология $SMPO$ обеспечивает возможность связывания абстрактных процессов и их элементов с понятиями предметной области и описания новых предметно-ориентированных классов. Кроме того, разметка значений доменов онтологии PO даёт возможность описывать семантически-нагруженные значения, например, именованные числа (числа с указанием единицы измерения). Онтология $POSMPO$ шаблонов семантической разметки задаёт ограничения на семантическую разметку экземпляров онтологии $SMPO$, определяя понятия предметной области, связанные с этими экземплярами. В отличие от декларативного подхода, описывающего предметно-ориентированную онтологию процессов множеством новых классов, аксиом и правил, этот подход задаёт онтологию как множество шаблонов (экземпляров онтологии $POSMPO$) для определения предметно-ориентированных процессов конструктивно как экземпляров шаблонов из этого множества. Все три онтологии базируются на простых понятиях, которые могут быть использованы как онтологические шаблоны проектирования [2, 7, 10].

В будущем мы планируем уточнить онтологию PO и развить методы специализаций как декларативный, так и конструктивный, для построения различных предметно-ориентированных онтологий процессов. Также мы добавим новые виды операций сопоставления в конструктивном методе (например, текущее множество

операций не позволяет нам выражать свойство, что различные атрибуты имеют один и тот же экземпляр в качестве значения).

Список литературы / References

[1] Börger E., Stärk R., *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.

[2] Gangemi A., Presutti V., "Ontology Design Patterns", *Handbook on Ontologies, Springer, Second Edition*, 2009, 221–243.

[3] Garanina N. O., Zubin V., Lyakh T., Gorlatch S., "An Ontology of Specification Patterns for Verification of Concurrent Systems", *Proceedings of the 17th International Conference SoMeT-18, New Trends in Intelligent Software Methodologies, Tools and Techniques, Series: Frontiers in Artificial Intelligence and Applications*, **303** (2018), 515-528.

[4] Garanina N., Anureev I., "Verification Oriented Process Ontology", *Proceedings of the 9th Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV 2018*, 2018, 58–67.

[5] Garanina N., Sidorova E., Bodin E., "A Multi-agent Text Analysis Based on Ontology of Subject Domain", *Perspectives of System Informatics. PSI 2014. Lecture Notes in Computer Science*, **8974** (2015), 102–110.

[6] Clarke E. M., Henzinger Th. A., Veith H., Bloem R., *Handbook of Model Checking*, **10**, Springer, 2018.

[7] Staab S., Studer R., *Handbook on Ontologies*, Springer Science & Business Media, 2010.

[8] Scherp. A., Saathoff C., Franz T., Staab S, "Designing Core Ontologies", *Applied Ontology, IOS Press*, **6**:3 (2011), 177–221.

[9] HermiT OWL Reasoner, www.hermit-reasoner.com, (29.09.2019).

[10] Ontology design patterns, www.ontologydesignpatterns.org, (29.09.2019).

[11] OWL Web Ontology Language Overview: W3C Recommendation 10 February 2004, www.w3.org/TR/owl-features/, (29.09.2019).

[12] Protégé. A Free, Open-source Ontology Editor and Framework for Building Intelligent Systems, protege.stanford.edu, (29.09.2019).

[13] Horrocks I. et al., "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", www.w3.org/Submission/SWRL, (29.09.2019).

Garanina N. O., Anureev I. S., Borovikova O. I., Zyubin E. V., "Methods for Domain Specification of Verification-Oriented Process Ontology", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 534–549.

DOI: 10.18255/1818-1015-2019-4-534-549

Abstract. User-friendly formal specifications and verification of parallel and distributed systems from various subject fields, such as automatic control, telecommunications, business processes, are active research topics due to its practical significance. In this paper, we present methods for the development of verification-oriented domain-specific process ontologies which are used to describe parallel and distributed systems of subject fields. One of the advantages of such ontologies is their formal semantics which make possible formal verification of the described systems. Our method is based on the abstract verification-oriented process ontology. We use two methods of specialization of the abstract process ontology. The declarative method uses the specialization of the classes of the original ontology, introduction of new declarative classes, as well as use of new axioms system, which restrict the classes and relations of the abstract ontology. The constructive method uses semantic markup and pattern matching techniques to link sublect fields with classes of the abstract process ontology. We provide detailed ontological specifications for these techniques. Our methods preserve the formal semantics of the original process ontology and, therefore, the possibility of applying formal verification methods to the specialized

process ontologies. We show that the constructive method is a refinement of the declarative method. The construction of ontology of the typical elements of automatic control systems illustrates our methods: we develop a declarative description of the classes and restrictions for the specialized ontology in the Protégé system in the OWL language using the deriving rules written in the SWRL language and we construct the system of semantic markup templates which implements typical elements of automatic control systems.

Keywords: process ontology, specialization, ontology axioms, pattern matching, semantic markup, automatic control system, formal verification

On the authors:
Natalia O. Garanina, orcid.org/0000-0001-9734-3808, PhD, senior researcher
A.P. Ershov Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS,
6 Lavrent'eva ave., Novosibirsk 630060, Russia, e-mail: garanina@iis.nsk.su
Igor S. Anureev, orcid.org/0000-0001-9574-128X, PhD, senior researcher
A.P. Ershov Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS,
6 Lavrent'eva ave., Novosibirsk 630060, Russia, e-mail: anureev@iis.nsk.su
Olesya I. Borovikova, orcid.org/0000-0001-5456-8513, junior researcher
A.P. Ershov Institute of Informatics Systems SB RAS,
6 Lavrent'eva ave., Novosibirsk 630060, Russia, e-mail: olesya@iis.nsk.su
Vladimir E. Zyubin, orcid.org/0000-0002-8198-3197, DSci, head of laboratory,
Institute of Automation and Electrometry SB RAS,
1 Academician Koptyug ave., Novosibirsk 630090, Russia, e-mail: zyubin@iae.nsk.su

Acknowledgments:
This work was funded by the RFBR according to the research № 17-07-01600, № 19-07-00762 and Funding State budget of the Russian Federation (IA&E project № AAAA-A17-11706061006-6).

Моделирование и анализ информационных систем. Т. 26, № 4 (2019), с. 550–571
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 550–571

Algorithms

DOI: 10.18255/1818-1015-2019-4-550-571

УДК 004.05

Направляемый свойством поиск реляционных инвариантов

Мордвинов Д. А.

Поступила в редакцию 1 октября 2019
После доработки 18 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. Достижимость, направляемая свойством, (Property Directed Reachability, PDR) — эффективный и масштабируемый подход к решению систем символьных ограничений, известных как дизъюнкты Хорна с ограничениями (Constrained Horn Clauses, CHC). В случае нелинейных систем дизъюнктов, которые могут возникнуть, к примеру, из задач реляционной верификации, PDR выводит индуктивные инварианты для каждого неинтерпретированного предикатного символа. Тем не менее на практике автоматический вывод таких решений не удаётся, т.к. инварианты должны выводиться для групп предикатных символов вместо индивидуальных предикатных символов. В статье описан новый алгоритм, автоматически определяющий такие группы и обобщающий существующий подход PDR. Ключевая особенность алгоритма состоит в том, что он не требует потенциально дорогой синхронизирующей трансформации системы дизъюнктов Хорна. Алгоритм был реализован над современным решателем дизъюнктов Хорна Spacer. Эксперименты показывают, что полученная реализация успешно выводит реляционные инварианты для некоторых систем дизъюнктов, на которых существующие решатели не завершаются.

Ключевые слова: реляционная верификация, дизъюнкты Хорна с ограничениями, направляемая свойством достижимость, реляционные инварианты

Для цитирования: Мордвинов Д. А., "Направляемый свойством поиск реляционных инвариантов", *Моделирование и анализ информационных систем*, **26**:4 (2019), 550–571.

Об авторах:
Мордвинов Дмитрий Александрович, orcid.org/0000-0002-6437-3020, стар. преп.,
Санкт-Петербургский государственный Университет, JetBrains Research
Университетский пр., 28, г. Петродворец, 198504 Россия, e-mail: dmitry.mordvinov@jetbrains.com

Благодарности:
Работа выполнена при финансовой поддержке JetBrains Research.

Введение

С развитием подходов к автоматической формальной верификации программ относительно функциональных спецификаций [1–8], растет потребность применения этих подходов к верификации свойств, описывающих отношения между входами-выходами различных программ [9–12]. Эта дисциплина, называемая *реляционной*

верификацией, широко применима в итеративном процессе разработки программного обеспечения, когда текущая и предыдущая версии сравниваются и проверяются на предмет отсутствия нововведённых ошибок. Другой пример применения — доказательство свойств безопасности потока информации, таких как невмешательство и балансировка времени [13], в которых исполнения одной и той же программы сравниваются для различных входов.

Многие автоматические подходы к реляционной верификации основываются на построении *произведения* сравниваемых программ [9, 14–18]. В таком случае реляционная спецификация над несколькими программами (или несколькими исполнениями одной программы) становится функциональной спецификацией над произведением этих программ. В теории такой подход позволяет верифицировать реляционные спецификации существующими подходами, однако на практике большинство существующих техник не могут справиться с потенциально сложной структурой программы-произведения. Проблему можно смягчить *слиянием* определенных циклов в произведении программ (т.е. применением т.н. *стратегий синхронизации*), но зачастую они обнаруживаются либо вручную, либо использованием неточных синтаксических эвристик. Другой недостаток состоит в том, что количество возможных стратегий экспоненциально зависит от количества сливаемых программ. Статья описывает новый полностью автоматический подход к определению стратегий синхронизации, который ведёт к эффективному нахождению *реляционных инвариантов* для произведений программ.

Предложенный подход обобщает одну из самых успешных реализаций PDR А. Гурфинкелем и др. [3] под названием Spacer. PDR инкрементально усиливает функциональную спецификацию (т.е. свойство безопасности) до тех пор, пока она либо не станет индуктивной, либо будет обнаружен контрпример. Он моделирует программы набором логических импликаций над множеством неинтерпретированных предикатных символов, называемых дизъюнктами Хорна с ограничениями. Неформально говоря, дизъюнкты задают семантику неинтерпретированных символов, и при нахождении интерпретации этих символов, выполняющие множество дизъюнктов, можно определить индуктивные инварианты верифицируемой программы. Spacer итеративно аппроксимирует семантику неинтерпретированных символов сверху и снизу; аппроксимации сверху используются для блокировки ложных контрпримеров, апроксимации снизу — для анализа веток программы без раскрутки отношения перехода.

Дизъюнкты, построенные для произведения программ, нелинейны, где неинтерпретированные символы соответствуют сравниваемым программам. В статье предлагается новый направляемый свойством подход, поддерживающий аппроксимации сверху и снизу семантик *групп* предикатов. Он работает сходным образом с обычным исследованием произведения программ, но *без явного вычисления произведения программ*. Более важно, что алгоритм автоматически определяет релевантные группы предикатов, анализируя контрпримеры к индуктивности, полученные на различных стадиях процесса верификации. Это позволяет эффективно отсекать пространство поиска различных стратегий синхронизации, что приводит к увеличению производительности. Без применения предложенного подхода, PDR пытается выводить инварианты для каждого символа в отдельности, что зачастую не полу-

чается, т.к., например, желаемые инварианты не выразимы в языке моделирования поведений программы.

Подход был реализован на базе Spacer и апробирован на тестах, полученных из различных задач реляционной верификации. Эксперименты подтверждают, что для многих систем дизъюнктов Хорна, на которых SPACER не завершается, наш подход способен быстро выводить реляционные инварианты.

Статья устроена следующим образом. В секции 1. собраны предварительные сведения, в секции 2. дано определение реляционных инвариантов систем дизъюнктов Хорна с ограничениями, ранее не представленное в литературе. Алгоритм определения стратегий синхронизации, обобщающий PDR, представлен в секции 3.. В секции 4. представлены экспериментальные данные, секции 5. и 6. завершают статью.

1. Предварительные сведения

1.1. Язык ограничений

Пусть Σ — сигнатура языка первого порядка с равенством, и пусть $\mathcal{M}$ — Σ-структура с носителем $|\mathcal{M}|$. $\mathcal{M}$ — *модель* Σ-предложения φ (т.е. Σ-формулы без свободных переменных), если φ выполняется в $\mathcal{M}$, что обозначается $\mathcal{M} \models \varphi$. Будем называть язык первого порядка, определяемый Σ, *языком ограничений*. Для Σ-формулы с n свободными переменными $\varphi(x_1, \ldots, x_n)$, обозначим $\mathcal{M}(\varphi)$ множество оценок свободных переменных, выполняющих φ в $\mathcal{M}$, т.е. $\mathcal{M}(\varphi) \stackrel{\text{def}}{=} \{\langle a_1, \ldots, a_n\rangle \mid \mathcal{M} \models \varphi(a_1, \ldots, a_n)\} \subseteq |\mathcal{M}|^n$.

1.2. Дизъюнкты Хорна с ограничениями

Пусть $\mathcal{R} = \{P_0, P_1, \ldots, P_n\}$ — конечное множество предикатных символов, которые будем называть *реляционными* (или *неинтерпретированными*) символами. *Дизъюнкт Хорна с ограничением* — это $\Sigma \cup \mathcal{R}$-формула вида

$$\varphi \wedge R_1(\overline{x}_1) \wedge \ldots \wedge R_m(\overline{x}_m) \Rightarrow R(\overline{v}_R),$$

где φ — бесквантворная Σ-формула, $R, R_i \in \mathcal{R}$, $\overline{v}_R$ и $\overline{x}_i$ — векторы переменных. Посылка импликации называется *телом* дизъюнкта C и обозначается $body(C)$, заключение $R(\overline{v}_R)$ называется *головой* дизъюнкта. *Система дизъюнктов* — это конечное множество дизъюнктов. Символ P_0 — специальный символ "запросов", корень всех деревьев вывода системы дизъюнктов. Если в тело дизъюнкта входит не более одного применения неинтерпретированного символа, дизъюнкт называется *линейным*, в противном случае — *нелинейным*. Система дизъюнктов линейна, если каждый дизъюнкт в ней линеен.

1.3. Проблема безопасности

Пара $\langle \mathcal{P}, \varphi_{safe}\rangle$ называется *проблемой безопасности* системы дизъюнктов $\mathcal{P} = \{C_1, \ldots, C_n\}$ относительно *свойства безопасности* φ_{safe} — Σ-формулы над пере-

менными $\overline{v}_{P_0}$. Множество *правил* реляционного символа R, обозначаемое $rules(R)$, — множество дизъюнктов $\mathcal{P}$ с применением R в голове. Без потери общности будем считать, что в голове каждого правила одного и символа R находятся применения R к одному и тому же набору переменных $\overline{v}_R$, т.е. если дизъюнкты $C_i, C_j \in \mathcal{P}$ имеют головы $R(\overline{v}_R)$ и $R(\overline{v}'_R)$, то $\overline{v}_R = \overline{v}'_R$.

Расширим нотацию $body$, обозначив за $body(R)$ дизъюнкцию тел правил для R:

$$body(R) \stackrel{\text{def}}{=} \bigvee_{C \in rules(R)} body(C).$$

Тело слияния реляционных символов $R_1, \ldots, R_m \in \mathcal{R}$ — это конъюнкция их тел

$$body(R_1, \ldots, R_m) \stackrel{\text{def}}{=} body(R_1) \stackrel{+}{\wedge} \ldots \stackrel{+}{\wedge} body(R_m),$$

где $\varphi \stackrel{+}{\wedge} \psi$ — конъюнкция φ и ψ, гарантирующая, что свободные переменные операндов различны $\varphi \wedge \psi$:

$$\varphi(\overline{x}) \stackrel{+}{\wedge} \psi(\overline{y}) \stackrel{\text{def}}{=} (\varphi \wedge \psi)\,(\overline{x} \uplus \overline{y})\,.$$

$\overline{\ell}_R$ обозначает вектор *экзистенциальных* (или *локальных*) переменных R, т.е. свободных переменных $body(R)$, не входящих в переменные головы $\overline{v}_R$.

Пример 1. Пусть $\mathcal{M}$ — комбинация стандартной модели линейной целочисленной арифметики и начальной модели теории алгебраических типов данных с сортом $tree$, заданным конструкторами $leaf : tree$ и $node : \mathbb{N} \times tree \times tree \to tree$. Рассмотрим следующую проблему безопасности, которая 1) считает количество узлов дерева (реляционный символ $size$), 2) суммирует значения в узлах дерева (реляционный символ sum) и 3) строит новое дерево, полученное из старого увеличением всех значений в узлах на 2 (реляционный символ inc):

$$\begin{aligned}
T = leaf \wedge n = 0 &\Rightarrow size(T,n)\\
T = node(v,L,R) \wedge n = 1 + n^L + n^R \wedge size(L,n^L) \wedge size(R,n^R) &\Rightarrow size(T,n)\\
T = leaf \wedge s = 0 &\Rightarrow sum(T,s)\\
T = node(v,L,R) \wedge s = v + s^L + s^R \wedge\\
sum(L,s^L) \wedge sum(R,s^R) &\Rightarrow sum(T,s)\\
T = leaf \wedge U = leaf &\Rightarrow inc(T,U)\\
T = node(v,L,R) \wedge U = node(v+2,L',R') \wedge\\
inc(L,L') \wedge inc(R,R') &\Rightarrow inc(T,U)\\
size(T,n) \wedge sum(T,s) \wedge inc(T,T') \wedge sum(T',s') &\Rightarrow P_0(T,n,s,s')\\
\varphi_{safe} \stackrel{\text{def}}{=} s' = s + 2n
\end{aligned}$$

Требуется доказать, что сумма значений в узлах второго дерева равняется сумме в узлах оригинального плюс удвоенное количество узлов в дереве. Здесь $\mathcal{R} = \{P_0, size, sum, inc\}$, $\overline{v}_{P_0} = \{T, n, s, s'\}$, $\overline{\ell}_{P_0} = \{T'\}$, $\overline{v}_{size} = \{T, n\}$, $\overline{\ell}_{size} = \{v, L, R, n^L, n^R\}$, $body(size) = (T = leaf \wedge n = 0) \vee \big(T = node(v,L,R) \wedge n = 1 + n^L + n^R \wedge size(L,n^L) \wedge size(R,n^R)\big)$.

1.4. Семантика неподвижной точки

Пусть $k_0, k_1, \ldots, k_n$ — арности символов $P_0, P_1, \ldots, P_n$ соответственно. Пусть $\overline{X} = \langle X_0, X_1, \ldots, X_n \rangle$ — кортеж отношений на носителе $|\mathcal{M}|$, $X_i \subseteq |\mathcal{M}|^{k_i}$. За $\left(\mathcal{M}, \overline{X}\right)$ обозначим обогащение $\mathcal{M}\{P_0 \mapsto X_0, P_1 \mapsto X_1, \ldots, P_n \mapsto X_n\}$.

Семантика системы дизъюнктов $\mathcal{P}$ в структуре $\mathcal{M}$ — (поточечно) наименьшая $(n+1)$-ка отношений $\overline{X}$, такая, что для всех $P \in \mathcal{R}$,

$$\left(\mathcal{M}, \overline{X}\right) \models \forall \overline{v}_P \cup \overline{\ell}_P. \left(body(P) \Rightarrow P(\overline{v}_P)\right).$$

Семантика $\mathcal{P}$ — наименьшая неподвижная точка оператора немедленных последствий $\mathcal{P}$ [20]; она всегда существует по теореме Кнастера-Тарского [19, 20]. Будем называть элементы кортежа семантики всей системы семантиками соответствующего символа и обозначать их $[\![P_i]\!]_{\mathcal{M}}$; таким образом, семантикой системы является кортеж $\langle [\![P_0]\!]_{\mathcal{M}}, [\![P_1]\!]_{\mathcal{M}}, \ldots, [\![P_n]\!]_{\mathcal{M}} \rangle$.

Система дизъюнктов *безопасна* по отношению к φ_{safe}, если $[\![P_0]\!]_{\mathcal{M}} \subseteq \mathcal{M}(\varphi_{safe})$. Например, система дизъюнктов в примере 1 безопасна по отношению к свойству безопасности $s' = s + 2n$.

1.5. Доказательства безопасности

Символьная интерпретация (далее просто интерпретация) Π отображает $P \in \mathcal{R}$ в Σ-формулу над $\Sigma \cup \overline{v}_P$. Для $\Sigma \cup \mathcal{R}$-формулы ψ, $[\![\psi]\!]_\Pi$ — это формула, полученная одновременной подстановкой всех применений реляционных символов в ψ формулами их Π-образов с соответствующим переименованием переменных.

Интерпретация Π — *доказательство безопасности* проблемы $\langle \mathcal{P}, \varphi_{safe} \rangle$, если Π безопасна и индуктивна:

$$\mathcal{M} \models \forall \overline{v}_{P_0}. \Pi(P_0) \Rightarrow \varphi_{safe} \qquad \text{(безопасность)}$$

$$\text{для всех } P \in \mathcal{R},\ \mathcal{M} \models \forall \overline{v}_P \cup \overline{\ell}_P. ([\![body(P)]\!]_\Pi \Rightarrow \Pi(P)) \qquad \text{(индуктивность)}$$

Предложение 1. Если существует доказательство безопасности проблемы $\langle \mathcal{P}, \varphi_{safe} \rangle$, то $\mathcal{P}$ безопасна по отношению к φ_{safe}.

Доказательство. Пусть Π — доказательство безопасности. По индуктивности, для кортежа $\overline{Y} = \langle \mathcal{M}(\Pi(P_0)), \mathcal{M}(\Pi(P_1)), \ldots, \mathcal{M}(\Pi(P_n)) \rangle$ имеем

$$\left(\mathcal{M}, \overline{Y}\right) \models \forall \overline{v}_P \cup \overline{\ell}_P. \left(body(P) \Rightarrow P(\overline{v}_P)\right)$$

для всех $P \in \mathcal{R}$. Т.к. $\langle [\![P_0]\!]_{\mathcal{M}}, [\![P_1]\!]_{\mathcal{M}}, \ldots, [\![P_n]\!]_{\mathcal{M}} \rangle$ — поточечно наименьший такой кортеж, имеем $[\![P_0]\!]_{\mathcal{M}} \subseteq \mathcal{M}(\Pi(P_0))$. По безопасности, $\mathcal{M}(\Pi(M)) \subseteq \mathcal{M}(\varphi_{safe})$. Таким образом, имеем $[\![P_0]\!]_{\mathcal{M}} \subseteq \mathcal{M}(\varphi_{safe})$. □

2. Реляционные инварианты

Системы дизъюнктов Хорна с ограничениями широко используются в автоматической верификации для доказательства корректности программ по отношению к спецификациям корректности. Однако, когда речь заходит о верификации реляционных свойств нескольких программ, моделируемых нелинейными дизъюнктами Хорна, рассуждать о таких системах становится сложнее. Зачастую доказательства безопасности не выразимы в языке ограничений. Например, несмотря на то, что система в примере 1 безопасна, *не существует доказательства безопасности, представимого в языке ограничений примера* (т.е. формулой, построенной из арифметических символов, равенства и символов *leaf* и *node*).

Пример 2. Рассмотрим более простой пример:

$$x=0 \wedge z=0 \Rightarrow mul(x,y,z)$$
$$x>0 \wedge x'=x-1 \wedge z=z'+y \wedge mul(x',y,z') \Rightarrow mul(x,y,z)$$
$$x=x' \wedge y=y' \wedge mul(x,y,z) \wedge mul(x',y',z') \Rightarrow P_0(x,y,z,x',y',z')$$
$$\varphi_{safe} \stackrel{\text{def}}{=} z=z'$$

Инвариант $mul(x,y,z) = (z = x \cdot y)$ не представим в линейной целочисленной арифметике, как и любая теоретико-множественная модель этой системы.

В этой секции обобщается классическое понятие доказательства безопасности таким образом, что в обоих этих случаях существует представимое доказательство. Ключевая идея состоит в том, чтобы сопоставлять формулы *группам* реляционных символов, а не каждому символу в отдельности. Это позволяет определять в языке ограничений *отношения* между переменными различных вычислений вместо резюмирования каждого вычисления в отдельности.

2.1. Определение реляционных интерпретаций

Обозначим за $\mathbb{N}^X$ множество мультимножеств на X, т.е. множество всех отображений X на множество натуральных чисел. Если $\overline{x} = x_1, \ldots, x_n$ — вектор элементов X (возможно повторяющихся), мы будем отождествлять его с мультимножеством $\{x_k \mapsto \#x_k\}$, где $\#x_i$ — число вхождений x_i в $\overline{x}$. Мультимножества частично упорядочены включением: для $m_1, m_2 \in \mathbb{N}^X$, $m_1 \subseteq m_2$, если $\forall x \in X, m_1(x) \leq m_2(x)$.

Определение 1. Пусть $\mathcal{P}$ — система дизъюнктов над множеством реляционных символов $\mathcal{R}$. *Реляционная интерпретация* — частичное отображение $\mathbb{N}^{\mathcal{R}}$ на множество формул языка ограничений, отображающее $\overline{R} = \{R_1 \mapsto n_1, \ldots, R_k \mapsto n_k\}$ в формулу над переменными $\overline{v}_{\overline{R}} \stackrel{\text{def}}{=} \underbrace{\overline{v}_{R_1} \uplus \ldots \uplus \overline{v}_{R_1}}_{n_1 \text{ times}} \uplus \ldots \uplus \underbrace{\overline{v}_{R_k} \uplus \ldots \uplus \overline{v}_{R_k}}_{n_k \text{ times}}$.

Пусть E — реляционная интерпретация. Обозначим за $dom(E)$ её область определения. Без потери общности будем считать, что $\mathcal{R} \subseteq dom(E)$: если $R \in \mathcal{R} \backslash dom(E)$, то отобразим R в $\top$. Пусть $R_1, \ldots, R_m \in \mathcal{R}$, φ — формула. Определим реляционную

подстановку индуктивно:

$$[\![\varphi \wedge R_1(\overline{x}_1) \wedge \ldots \wedge R_m(\overline{x}_m)]\!]_E \stackrel{\text{def}}{=} \varphi \wedge \bigwedge_{\substack{R_{i_1},\ldots,R_{i_k} \in \{R_1,\ldots,R_m\} \\ \langle R_{i_1},\ldots,R_{i_k}\rangle \in dom(E)}} E(R_{i_1},\ldots,R_{i_k})(\overline{x}_{i_1},\ldots,\overline{x}_{i_k}) \tag{1}$$

и

$$\left[\!\!\left[\bigwedge_{i=1}^{k}\bigvee_{j=1}^{m_i} F_{i,j}\right]\!\!\right]_E \stackrel{\text{def}}{=} \bigvee_{\substack{1\le j_1\le m_1 \\ \ldots \\ 1\le j_k\le m_k}} [\![F_{1,j_1} \wedge \ldots \wedge F_{k,j_k}]\!]_E \tag{2}$$

Неформально, (1) перебирает все возможные варианты "групповых" подстановок в тело одного дизъюнкта $R_1(\overline{x}_1)\wedge\ldots\wedge R_m(\overline{x}_m)$. В (2) определяется реляционная подстановка в тело слияния нескольких символов, которое, по определению, есть конъюнкция дизъюнкций тел дизъюнктов. В (2) объединяются одним из $m_1\cdot\ldots\cdot m_k$ возможных способов, с дальнейшей групповой подстановкой в каждое из объединённых тел дизъюнктов.

Пример 3. Рассмотрим следующую систему дизъюнктов:

$$\begin{aligned} \varphi_1 &\Rightarrow f(x_1,x_2) \\ \varphi_2 \wedge f(x_1',x_2') \wedge f(x_1'',x_2'') &\Rightarrow f(x_1,x_2) \\ \psi_1 &\Rightarrow g(y) \\ \psi_2 \wedge g(y') &\Rightarrow g(y) \end{aligned}$$

и следующую реляционную интерпретацию:

$$E = \{f \mapsto \top, g \mapsto \eta_1(y), \langle f,g\rangle \mapsto \eta_2(x_1,x_2,y)\}.$$

Результат подстановки E в $body(f,g)$ таков:

$$\begin{aligned} [\![body(f,g)]\!]_E = \Big[\!\Big[\Big(\varphi_1 \vee \big(\varphi_2 \wedge f(x_1',x_2') \wedge f(x_1'',x_2'')\big)\Big) \wedge \\ \Big(\psi_1 \vee \big(\psi_2 \wedge g(y')\big)\Big)\Big]\!\Big]_E = \\ [\![\varphi_1 \wedge \psi_1]\!]_E \vee [\![\varphi_1 \wedge \psi_2 \wedge g(y')]\!]_E \vee \\ \quad [\![\varphi_2 \wedge \psi_1 \wedge f(x_1',x_2') \wedge f(x_1'',x_2'')]\!]_E \vee \\ \quad [\![\varphi_2 \wedge \psi_2 \wedge f(x_1',x_2') \wedge f(x_1'',x_2'') \wedge g(y')]\!]_E = \\ (\varphi_1 \wedge \psi_1) \vee (\varphi_1 \wedge \psi_2 \wedge \eta_1(y')) \vee (\varphi_2 \wedge \psi_1) \vee \\ \quad (\varphi_2 \wedge \psi_2 \wedge \eta_1(y') \wedge \eta_2(x_1',x_2',y') \wedge \eta_2(x_1'',x_2'',y')) \end{aligned}$$

Реляционные интерпретации обобщают "классические" интерпретации: если E — реляционная интерпретация, определённая только на *единичных* мультимножествах (т.е. мультимножествах, состоящих из одного элемента, входящего один раз), а Π — "классическая" интерпретация, отображающая те же символы в те же формулы, то для всех $\Sigma\cup\mathcal{R}$-формул φ, $[\![\varphi]\!]_E$ логически равносильно $[\![\varphi]\!]_\Pi$.

2.2. Реляционные доказательства безопасности

Реляционная интерпретация E — *доказательство безопасности* проблемы $\langle \mathcal{P}, \varphi_{safe} \rangle$, если она безопасна и индуктивна:

$$\mathcal{M} \models \forall \overline{v}_{P_0}.E(P_0) \Rightarrow \varphi_{safe} \qquad \text{(безопасность)}$$
$$\text{для всех } \overline{P} \in dom(E),\ \mathcal{M} \models \forall \overline{v}_{\overline{P}} \cup \overline{\ell}_{\overline{P}}.\big([\![body(\overline{P})]\!]_E \Rightarrow E(\overline{P}) \big). \qquad \text{(индуктивность)}$$

Кроме реляционных подстановок, главная разница между "классическими" и реляционными доказательствами безопасности состоит в том, что последние должны быть индуктивны относительно тел *слияний* символов. Т.е. если $\overline{P} \in dom(E)$ для неединичного мультимножества $\overline{P}$, то E должно быть индуктивно относительно $body(\overline{P})$.

Пример 4. Хотя для системы в примере 2 не существует "классического" доказательства безопасности, представимое в языке линейной целочисленной арифметики, существует реляционное доказательство безопасности:

$$E = \{P_0 \mapsto z = z',\ mul \mapsto \top,$$
$$\langle mul, mul \rangle \mapsto (x^{mul_1} = x^{mul_2} \wedge y^{mul_1} = y^{mul_2} \Rightarrow z^{mul_1} = z^{mul_2})\}$$

Пример 1 также имеет бескванторное реляционное доказательство безопасности:

$$E = \{P_0 \mapsto s' = s + 2n, \quad sum \mapsto \top, \quad inc \mapsto \top,$$
$$\langle size, sum, sum, inc \rangle \mapsto T^{size} = T^{sum_1} = T^{inc} \wedge$$
$$T^{sum_2} = U^{inc_2} \Rightarrow s^{sum_2} = s^{sum_1} + 2n^{size}\}$$

2.3. Корректность

Теорема 1. *Если существует реляционное доказательство безопасности проблемы* $\langle \mathcal{P}, \varphi_{safe} \rangle$*, то* $\mathcal{P}$ *безопасна по отношению к* φ_{safe}*.*

Доказательство. По безопасности E, достаточно показать, что $[\![P_0]\!]_{\mathcal{M}} \subseteq \mathcal{M}(E(P_0))$. Докажем это, построив другую систему дизъюнктов $\mathcal{P}'$ и "классическое" доказательство её безопасности П, используя E.

Для каждого $\overline{P} \in dom(E)$ введём новый реляционный символ $R_{\overline{P}}$. Определим реляционную интерпретацию E', которая отображает $\overline{P}$ на $R_{\overline{P}}(\overline{v}_{\overline{P}})$. Теперь определим $\mathcal{P}'$ над реляционными символами $\{R_{\overline{P}} \mid \overline{P} \in dom(E)\}$. Для каждого $R_{\overline{P}}$, положим $body(R_{\overline{P}}) \stackrel{\text{def}}{=} [\![body(\overline{P})]\!]_{E'}$; голова каждого правила для $R_{\overline{P}}$ — это $R_{\overline{P}}(\overline{v}_{\overline{P}})$.

Например, для системы $\mathcal{P}$ и реляционной интерпретации E в примере 3, $\mathcal{P}'$ следующая:

$$\begin{aligned}
\varphi_1 &\Rightarrow R_f(x_1, x_2)\\
\varphi_2 \wedge R_f(x_1', x_2') \wedge R_f(x_1'', x_2'') &\Rightarrow R_f(x_1, x_2)\\
\psi_1 &\Rightarrow R_g(y)\\
\psi_2 \wedge R_g(y') &\Rightarrow R_g(y)\\
\varphi_1 \wedge \psi_1 &\Rightarrow R_{f,g}(x_1, x_2, y)\\
\varphi_1 \wedge \psi_2 \wedge R_g(y') &\Rightarrow R_{f,g}(x_1, x_2, y)\\
\varphi_2 \wedge \psi_1 \wedge R_f(x_1', x_2') \wedge R_f(x_1'', x_2'') &\Rightarrow R_{f,g}(x_1, x_2, y)\\
\varphi_2 \wedge \psi_2 \wedge R_f({x_1}', {x_2}') \wedge R_f({x_1}'', {x_2}'') \wedge R_g(y') \wedge&\\
R_{f,g}({x_1}', {x_2}', y) \wedge R_{f,g}({x_1}'', {x_2}'', y) &\Rightarrow R_{f,g}(x_1, x_2, y)
\end{aligned}$$

Семантика $R_{\overline{P}}$ в $\mathcal{P}'$ есть декартово произведение $\bigtimes_{p\in\overline{P}} [\![p]\!]_{\mathcal{M}}$, т.к. $R_{\overline{P}}$ применяется к переменным различных отношений и "достигает" каждого состояния, достигаемого каждым $p \in \overline{P}$. В частности,

$$[\![P_0]\!]_{\mathcal{M}} = [\![R_{P_0}]\!]_{\mathcal{M}}\,. \tag{3}$$

Наконец, определим "классическую" интерпретацию Π, отображающую $R_{\overline{P}}$ на $E(\overline{P})$. Π — доказательство безопасности $\mathcal{P}'$: она безопасна и индуктивна по построению. Обратим внимание, что

$$\Pi(R_{P_0}) = E(P_0). \tag{4}$$

Т.к. семантика $\mathcal{P}'$ — поточечно наименьший индуктивный кортеж отношений, имеем

$$[\![R_{P_0}]\!]_{\mathcal{M}} \subseteq \mathcal{M}(\Pi(R_{P_0})). \tag{5}$$

По (3), (4) и (5) получаем:

$$[\![P_0]\!]_{\mathcal{M}} = [\![R_{P_0}]\!]_{\mathcal{M}} \subseteq \mathcal{M}(\Pi(R_{P_0})) = \mathcal{M}(E(P_0)) \subseteq \mathcal{M}(\varphi_{safe}).$$

□

2.4. Быстрая подстановка

По (2), наивная реализация $[\![\varphi]\!]_E$ требует вычисления $m_1 \cdot \ldots \cdot m_k$ правил, что имеет экспоненциальную сложность с ростом k. В этом разделе демонстрируется альтернативный метод, позволяющий избежать явного перечисления всех комбинаций сливаемых правил, что позволяет преодолеть наиболее слабую сторону подходов к синтаксическому преобразованию системы [16, 17].

Ключевая идея состоит в построении *эквивыполнимой* формулы вместо точного вычисления, используя правила (1) и (2).

Пусть $R_1(\overline{x}_1), \ldots, R_m(\overline{x}_m)$ — все применения реляционных символом в φ. Для каждого применения $R_i(\overline{x}_i)$ заведём пропозициональный атом (т.е. нульместный предикатный символ) a_i. Пусть φ' — формула, полученная заменой всех вхождений $R_i(\overline{x}_i)$ на a_i для всех i в φ. Заметим, что $[\![\varphi]\!]_E$ эквивыполнимо с

$$\varphi' \wedge \bigwedge_{\substack{R_{i_1},\ldots,R_{i_k}\in\{R_1,\ldots,R_m\}\\ \langle R_{i_1},\ldots,R_{i_k}\rangle\in dom(E)}} \left(a_{i_1}\wedge\ldots\wedge a_{i_k} \Rightarrow E(R_{i_1},\ldots,R_{i_k})(\overline{x}_{i_1},\ldots,\overline{x}_{i_k})\right).$$

Пример 5. Для примера 3, следующая формула эквивыполнима с $[\![body(f,g)]\!]_E$:

$$\big(\varphi_1 \vee (\varphi_2 \wedge a_1 \wedge a_2)\big) \wedge \big(\psi_1 \vee (\psi_2 \wedge a_3)\big) \wedge \big(a_3 \Rightarrow \eta_1(y')\big) \wedge \\ \wedge \big(a_1 \wedge a_3 \Rightarrow \eta_2(x_1', x_2', y')\big) \wedge \big(a_2 \wedge a_3 \Rightarrow \eta_2(x_1'', x_2'', y')\big)$$

Этот метод используется в реализации RelRecMc (см. секцию 3.); он позволяет сохранить композициональность в выводе реляционных лемм и значительно повышает скорость вычисления реляционной подстановки по сравнению с наивной реализацией.

3. Алгоритм

В данной секции описывается направляемый свойством алгоритм автоматического вывода реляционных инвариантов. Алгоритм расширяет RecMC из [3]; мы заимствуем оттуда нотацию и общую структуру алгоритма и отсылаем читателя для более подробных объяснений и доказательств корректности.

3.1. Леммы и ветви

Алгоритм поддерживает две структуры данных ρ и σ, в которых хранит *достижимые ветви* системы и *леммы* о системе соответственно. Первая отображает $P \in \mathcal{R}$ и натуральное число b на множество формул над $\overline{v}_P$, вторая отображает мультимножество $\overline{P} \in \mathbb{N}^{\mathcal{R}}$ и натуральное число b на множество формул над $\overline{v}_{\overline{P}}$.

Алгоритм использует ρ для нахождения контрпримеров к безопасности и σ для построения реляционных доказательств. В частности, в ρ размещаются *факты о достижимости*, т.е. достижимые ветви системы; $\rho(P,b)$ — множество формул, аппроксимирующих снизу семантику P глубины b, т.е. объединение всех выводов системы сверху вниз высоты b. Двойственно, в σ размещаются факты, *резюмирующие* систему, которые называются *леммами*. Формулы в $\sigma(P,b)$ аппроксимируют сверху семантику P глубины b, они используются для построения реляционного доказательства безопасности.

ρ и σ неявно определяют "классическую" интерпретацию U^b_ρ и реляционную интерпретацию O^b_σ соответственно, аппроксимирующие семантику системы глубины b сверху и снизу[1]:

$$U^b_\rho(P) \stackrel{\text{def}}{=} \bigvee \{\delta \in \rho(P,c) \mid c \le b\}\,,$$
$$O^b_\sigma(\overline{P}) \stackrel{\text{def}}{=} \bigwedge \{\delta \in \sigma(\overline{R},c) \mid \overline{R} \in dom(\sigma), \overline{R} \subseteq \overline{P}, c \ge b\}\,.$$

Леммы для мультимножества $\overline{P}$ включают в себя леммы подмножеств $\overline{P}$.

Мы сокращаем $[\![\pi]\!]_{U^b_\rho}$ и $[\![\pi]\!]_{O^b_\sigma}$ до $[\![\pi]\!]^b_\rho$ и $[\![\pi]\!]^b_\sigma$ соответственно. Для простоты положим значения интерпретаций на уровне -1 $U^{-1}_\rho(P) \stackrel{\text{def}}{=} \bot$ и $O^{-1}_\sigma(\overline{P}) \stackrel{\text{def}}{=} \bot$ для всех $P \in \mathcal{R}$ и $\overline{P} \in \mathbb{N}^{\mathcal{R}}$.

[1] Конъюнкция пустого множества — $\top$, дизъюнкция — $\bot$.

Алгоритм 1: Псевдокод RelRecMc

Вход : Проблема безопасности $\langle \mathcal{P}, \varphi_{safe} \rangle$
Выход: $out \in \langle$`БЕЗОПАСНО/НЕБЕЗОПАСНО`, реляционное доказательство/ контрпример$\rangle$

```
1  b ← 0; ρ ← ∅; σ ← ∅;
2  пока true
3      ⟨res, ρ, σ⟩ ← RelBndSafety(⟨P, φ_safe⟩, b, ρ, σ);
4      если res = ДОСТИЖИМО вернуть ⟨НЕБЕЗОПАСНО, _, ρ⟩;
5      иначе
6          inductive ← true;
7          для всех P̄ such that ⟨P̄, b⟩ ∈ dom(σ)
8              для всех δ ∈ σ(P̄, b)
9                  если ⊨ [[body(P̄)]]^b_σ ⇒ δ
10                     σ ← σ ∪ {⟨P̄, b + 1⟩ ↦ δ};
11                 иначе inductive ← false;
12         если inductive вернуть ⟨БЕЗОПАСНО, O^b_σ, _⟩;
13         b ← b + 1;
```

3.2. Внешний цикл

В алгоритме 1 представлен псевдокод процедуры RelRecMc, которая итеративно добавляет новые ветви в ρ и усиливает леммы σ до тех пор, пока либо ρ не засвидетельствует контрпример (строка 4), или σ не станет индуктивным (строка 12). Итерация номер b RelRecMc проверяет достижимость нарушения свойства за b шагов. Если нарушения не достижимы (строки 6 - 11), σ содержит доказательство безопасности всех выводов системы сверху-вниз высоты b. Затем RelRecMc распространяет все индуктивные леммы из σ на уровень $b+1$ и итерируется, если их не достаточно для доказательства безопасности системы в целом.

3.3. Внутренний цикл

Процедура RelBndSafety, представленная в алгоритме 2, проверяет безопасность всех выводов системы сверху-вниз высоты, ограниченной сверху B. Она формулирует и решает *запросы* $\langle \overline{P}, \pi, b \rangle$, где $\overline{P} \in \mathbb{N}^{\mathcal{R}}$, π — отрицание свойства безопасности для $\overline{P}$, and $b \in \mathbb{N}$. Неформально, ответ на $\langle \overline{P}, \pi, b \rangle$ состоит в определении того, достигает ли $\overline{P}$ π за b шагов.

Запросы помещаются в очередь Q, которая в начале содержит только $\langle P_0, \neg\varphi_{safe}, B \rangle$. Каждая итерация начинается с выбора запроса с наименьшим b (строка 3), на который может быть дан положительный (строка 12) или отрицательный (строка 8) ответ, либо который может породить дочерние запросы, на которые нужно ответить для ответа на данный запрос (строка 31). Когда на все запросы дан ответ, алгоритм возвращает результат `(НЕ-)ДОСТИЖИМО` (строка 34 или 32 соответственно).

Вывод ветвей Если π достижимо за один шаг от предшественников, ограниченных высотой $b-1$ (строка 4), алгоритм выводит по новой ветви для всех $\overline{P}[i]$ (строка 7). Неформально, вместо исследования всех ветвей $\overline{P}[i]$, алгоритм исследу-

Алгоритм 2: Псевдокод RelBndSafety

Вход : Проблема безопасности $\langle \Phi, \varphi_{safe} \rangle$, уровень B, хранилища ρ, σ
Выход : $out \in \langle$ДОСТИЖИМО/НЕДОСТИЖИМО$, \rho, \sigma \rangle$
Данные: Очередь запросов Q

$Q \leftarrow \{\langle P_0, \neg\varphi_{safe}, B\rangle\}$;
пока $(Q \neq \emptyset)$
  выбрать $\langle \overline{P}, \pi, b\rangle$ из Q;
  если $\exists m \,.\, m \models [\![body(\overline{P})]\!]_{\rho}^{b-1} \wedge \pi$
    для $i \leftarrow 1$ **до** $|\overline{P}|$
      $\psi_{\overline{P}[i]} \leftarrow \text{MBP}\left([\![body(\overline{P}[i])]\!]_{\rho}^{b-1}, \overline{\ell}_{\overline{P}[i]}, m \right)$;
      $\rho \leftarrow \rho \cup \{\langle \overline{P}[i], b\rangle \mapsto \psi_{\overline{P}[i]} \mid 1 \le i \le n\}$;
    $Q \leftarrow Q \setminus \{\langle \overline{R}, \eta, c\rangle \mid \overline{R} \subseteq \overline{P}, c \geq b, \bigwedge_{i=1}^{+|\overline{R}|} \psi_{\overline{R}[i]} \wedge \eta \not\Rightarrow \bot\}$;
  иначе если $[\![body(\overline{P})]\!]_{\sigma}^{b-1} \wedge \pi \Rightarrow \bot$
    пусть ψ такая, что $[\![body(\overline{P})]\!]_{\sigma}^{b-1} \Rightarrow \psi$ и $\psi \wedge \pi \Rightarrow \bot$;
    $\sigma \leftarrow \sigma \cup \{\langle \overline{P}, b\rangle \mapsto \psi\}$;
    $Q \leftarrow Q \setminus \left\{ \langle \overline{R}, \eta, c\rangle \mid \overline{P} \subseteq \overline{R}, c \leq b, \left[\!\!\left[\bigwedge_i \overline{R}[i](\overline{v}_{\overline{R}[i]}) \right]\!\!\right]_{\sigma}^{c} \wedge \eta \Rightarrow \bot \right\}$;
  иначе
    пусть $cti \models [\![body(\overline{P})]\!]_{\sigma}^{b-1} \wedge \pi$;
    $\psi \leftarrow \pi$;
    $apps \leftarrow \emptyset$;
    для $i \leftarrow 1$ **до** $|\overline{P}|$
      let $C \in rules(\overline{P}[i])$ be s.t. $cti \models [\![body(C)]\!]_{\sigma}^{b-1}$;
      $\eta \wedge R_1(\overline{x}_1) \wedge \ldots \wedge R_m(\overline{x}_m) \leftarrow body(C)$;
      $\psi \leftarrow \psi \wedge \eta$;
      для $j \leftarrow 1$ **до** m
        если $cti \models [\![R_j(\overline{x}_j)]\!]_{\rho}^{b-1}$
          $\psi \leftarrow \psi \wedge [\![R_j(\overline{x}_j)]\!]_{\rho}^{b-1}$;
        иначе $apps \leftarrow apps \cup \{R_j(\overline{x}_j)\}$;
    $Groups \leftarrow \text{Partition}(apps, \pi, \psi)$;
    для всех $\{i_1, \ldots, i_k\} \in Groups$
      $rels \leftarrow \langle R_{i_1}, \ldots, R_{i_k}\rangle$;
      $vars \leftarrow \overline{x}_{i_1} \ldots \overline{x}_{i_k}$;
      $\psi' \leftarrow \psi \wedge \left[\!\!\left[\bigwedge_{\substack{R_j(\overline{x}_j) \in apps \\ j \notin \{i_1, \ldots, i_k\}}} R_j(\overline{x}_j) \right]\!\!\right]_{\sigma}^{b-1}$;
      $\psi' \leftarrow \text{MBP}\left(\psi', \overline{v}_{\overline{P}} \cup \overline{\ell}_{\overline{P}} \setminus vars, cti\right)$;
      $Q \leftarrow Q \cup \{\langle rels, \psi'[vars \leftarrow \overline{v}_{rels}], b-1\rangle\}$;
если $[\![P_0]\!]_{\rho}^{n} \wedge \neg\varphi_{safe} \not\Rightarrow \bot$ **вернуть** $\langle$*ДОСТИЖИМО*$, \rho, \sigma\rangle$;
утвердить $[\![P_0]\!]_{\sigma}^{n} \wedge \neg\varphi_{safe} \Rightarrow \bot$;
вернуть $\langle$*НЕДОСТИЖИМО*$, \rho, \sigma\rangle$;

ет за раз только одну ветвь $\psi_{\overline{P}[i]}$, выбираемую в направлении к свойству. Каждый запрос $\langle \overline{R}, \eta, c\rangle \in Q$, где η достижимо в обновлённой интерпретации U_{ρ}^{c}, считается отвеченным и удаляется из Q (в частности, $\langle \overline{P}, \pi, b\rangle$).

Для получения символьного представления для ветви $\psi_{\overline{P}[i]}$, алгоритм исполь-

зует *проекцию на основе модели* (model-based projection, MBP) [3, 21]. По данной формуле $\exists \overline{x}.\tau$, где τ — бескванторная формула и модели m, $\text{MBP}(\tau, \overline{x}, m)$ строит бескванторную конъюнкцию литералов τ' такую, что $m \models \tau'$, $\tau' \Rightarrow \exists \overline{x}.\tau$, и если $\mathcal{M}$ допускает элиминацию кванторов, то для каждой формулы τ существует конечное множество моделей $m_1, \ldots, m_k$, что $\exists \overline{x}.\tau \Leftrightarrow \bigvee_{i=1}^{k} \text{MBP}(\tau, \overline{x}, m_i)$. Неформально, последовательность проекций на основе моделей *лениво* устраняют квантор из $\exists \overline{x}.\tau$. При данной m, $\text{MBP}\left(\left[\!\left[body(\overline{P}[i])\right]\!\right]_{\rho}^{b-1}, \overline{\ell}_{\overline{P}[i]}, m\right)$ может рассматриваться как выбор ветви $\overline{P}[i]$, выполняющей m, и устранение из неё всех локальных переменных $\overline{\ell}_{\overline{P}[i]}$. Ленивость в элиминации кванторов позволяет сохранять размер символьных представлений маленьким и позволяет рассматривать только релевантные поведения системы. В частности, хотя $\exists \overline{x}.\tau$ эквивалентно дизъюнкции ветвей, алгоритм рассматривает лишь одну ветвь за запрос; другие ветви будут рассмотрены по требованию в следующих итерациях.

Вывод лемм Если σ достаточно сильна для доказательства недостижимости π (строка 9), RelBndSafety выводит новую лемму вычислением Крейговского интерполянта [2] (далее обозначается ITP) формул $\left[\!\left[body(\overline{P})\right]\!\right]_{\sigma}^{b-1}$ и $\neg\pi$. В результате новая лемма аппроксимирует сверху семантику P глубины b, всё ещё доказывая её безопасность относительно π. Важно, что новая лемма сформулирована только над переменными $\overline{P}$, выражая *отношения* между элементами $\overline{P}$. Каждый запрос $\langle \overline{R}, \eta, c \rangle \in Q$ такой, что обновлённая σ доказывает невыполнимость η, считается отвеченным и удаляется из очереди (в частности, $\langle \overline{P}, \pi, b \rangle$).

Генерация запросов Если ни $\left[\!\left[body(\overline{P})\right]\!\right]_{\rho}^{b-1} \wedge \pi$ выполнимо, ни $\left[\!\left[body(\overline{P})\right]\!\right]_{\sigma}^{b-1} \wedge \pi$ невыполнимо, формулы в ρ на уровне $b-1$ слишком сильны для того, чтобы засвидетельствовать контрпример, а леммы на уровне $b-1$ слишком слабы для доказательства безопасности. В этом случае существует потенциальный контрпример (*контрпример к индуктивности* в терминах IC3 [22]) *cti*, получаемый алгоритмом на строке 14, который должен быть засвидетельствован ρ или заблокирован σ. Алгоритм порождает дочерние запросы, ответы на которые помогут ответить на $\langle \overline{P}, \pi, b \rangle$ либо доказательством недостижимости *cti*, либо свидетельством его достижимости.

Для каждого отношения в $\overline{P}$ алгоритм выбирает дизъюнкт C, выполняющийся в *cti* (*свидетельствующий cti*) (строка 18). Такой дизъюнкт гарантированно существует, т.к. $cti \models \left[\!\left[body(\overline{P})\right]\!\right]_{\sigma}^{b-1}$. Пусть $R_1(\overline{x}_1), \ldots, R_m(\overline{x}_m)$ — все применения реляционных символов в C (строка 19). На следующих шагах алгоритм пытается усилить леммы, чтобы заблокировать *cti*. Для этого алгоритм определяет, какие леммы уровня $b-1$ слишком грубы, разделяя $R_1(\overline{x}_1), \ldots, R_m(\overline{x}_m)$ на две группы: применения, свидетельствующие *cti* (строка 22) и другие применения *apps* (строка 24). Т.к. ветви применений в первой группе уже свидетельствуют контрпример,

[2] Крейговский интерполянт двух формул A и B таких, что $A \wedge B$ невыполнимо, — это формула I, удовлетворяющая трём условиям: 1) $A \Rightarrow I$, 2) $I \wedge B$ невыполнимо, 3) I состоит только из общих для A и B символов.

не имеет смысла усиливать их леммы, поэтому алгоритм пытается усилить только леммы для *apps*.

На этом шаге алгоритм ведёт себя отличным от [3] образом. Вместо усиления лемм для каждого отношения в отдельности, он пытается вывести лемму для *группы* предикатов в *apps*.

Алгоритм параметризован оракулом Partition (строка 25), который разделяет входное множество атомов на список мультимножеств применений. Например, множество атомов $\{f(\overline{x_1}), f(\overline{x_2}), g(\overline{y})\}$ может быть поделено на группы $[\{f \mapsto 2\}, \{g \mapsto 1\}]$ отношений и список векторов $[\langle \overline{x}_1, \overline{x}_2 \rangle, \overline{y}]$ переменных. В результате список *rels* содержит все мультимножества символов, которые будут исследованы на в дочерних запросах.

В нашей реализации Partition разделяет применения на рекурсивную и нерекурсивную группы[3]. Если в рекурсивной группе более $\left|\overline{P}\right|$ элементов, она в свою очередь разбивается на подгруппы размера не более $\left|\overline{P}\right|$. Это останавливает неограниченный рост размеров мультимножеств запросов. Partition угадывает подходящее разбиение рекурсивных применений определений синхронизаций, сохраняющих индуктивность свойства в духе [17] (секция 3.4. демонстрирует его работу на примере 1).

Для каждого мультимножества в *rels* генерирует дочерний запрос, аппроксимирующий снизу множество ошибочных состояний группы. Свойство безопасности j-ой группы — конъюнкция родительского свойства безопасности π (строка 15) ограничений всех дизъюнктов, свидетельствующих *cti* (строка 20), достижимых дочерних состояний, свидетельствующих *cti* (строка 23) и лемм оставшихся мультимножеств в разбиении (строка 29). Неформально, RelBndSafety усиливает свойство безопасности π информацией о достижимых состояниях и дочерних леммах, связанных с *cti*. Далее, алгоритм проектирует свойство безопасности, устраняя все переменные, кроме $vars[j]$, используя проекцию на основе модели (строка 30), переименовывает переменные и помещает новый запрос в очередь (строка 31). Дочерние свойства формулируются над переменными дочерних отношений, для этого переменные переименовываются (строка 31). Аналогично выводу ветвей, использование проекции на основе моделей не портит корректность, поддерживая размер формул в запросах небольшим.

3.4. Пример

В этом разделе показаны несколько итераций алгоритма для задачи в примере 1. После синтаксической предобработки алгоритм работает со следующей системой, в которой переменные различных отношений различны:

[3] Два символа (взаимно) рекурсивны, если они принадлежат одной сильно связной компоненте в орграфе (V, E) с $V = \mathcal{R}$ и $(P, R) \in E$, если и только если R входит в тело некоторого правила для P.

$$
\begin{aligned}
T_1 = \mathit{leaf} \wedge n = 0 &\Rightarrow \mathit{size}(T_1, n)\\
T_1 = \mathit{node}(v_1, L_1, R_1) \wedge n = 1 + n^L + n^R \wedge & \\
\mathit{size}(L_1, n^L) \wedge \mathit{size}(R_1, n^R) &\Rightarrow \mathit{size}(T_1, n)\\
T_2 = \mathit{leaf} \wedge s_2 = 0 &\Rightarrow \mathit{sum}(T_2, s_2)\\
T_2 = \mathit{node}(v_2, L_2, R_2) \wedge s_2 = v_2 + s_2^L + s_2^R \wedge & \\
\mathit{sum}(L_2, s_2^L) \wedge \mathit{sum}(R_2, s_2^R) &\Rightarrow \mathit{sum}(T_2, s_2)\\
T_4 = \mathit{leaf} \wedge U = \mathit{leaf} &\Rightarrow \mathit{inc}(T_4, U)\\
T_4 = \mathit{node}(v_4, L_4, R_4) \wedge U = \mathit{node}(v_4 + 2, L', R') \wedge & \\
\mathit{inc}(L_4, L') \wedge \mathit{inc}(R_4, R') &\Rightarrow \mathit{inc}(T_4, U)\\
A = T_0 \wedge B = T_0 \wedge C = T_0 \wedge D = E \wedge \mathit{size}(A, n_0) \wedge & \\
\mathit{sum}(B, s_0) \wedge \mathit{inc}(C, D) \wedge \mathit{sum}(E, s_0') &\Rightarrow P_0(T_0, n_0, s_0, s_0')\\
\varphi_{\mathit{safe}} &\stackrel{\text{def}}{=} s_0' = s_0 + 2n_0
\end{aligned}
$$

Уровень *0* Алгоритм начинает с вызова RelBndSafety для уровня 0, который кладёт запрос $\langle P_0, s_0' \neq s_0 + 2n_0, 0\rangle$ в Q. И $[\![\mathit{body}(P_0)]\!]_\rho^{-1}$, и $[\![\mathit{body}(P_0)]\!]_\sigma^{-1}$ ложны, поэтому алгоритм попадает на строку 11, где добавляется $\mathrm{ITP}(\bot, \neg\varphi_{\mathit{safe}}) = \bot$ в $\sigma(P_0, 0)$. RelBndSafety завершается с результатом *UNREACHABLE*, но т.к. добавленная лемма не индуктивна, RelRecMc продолжает свою работу на уровне 1.

Уровень *1* RelBndSafety начинает с $Q = \{\langle P_0, s_0' \neq s_0 + 2n_0, 1\rangle\}$. Здесь $[\![\mathit{body}(P_0)]\!]_\rho^0 \equiv \bot$, $[\![\mathit{body}(P_0)]\!]_\sigma^0 \equiv \varphi_0 \stackrel{\text{def}}{=} A = T_0 \wedge B = T_0 \wedge C = T_0 \wedge D = E$. Т.к. $\varphi_0 \wedge \neg\varphi_{\mathit{safe}}$ выполнима, алгоритм получает $\mathit{cti} = \{A, B, C, D, E, T_0 \mapsto \mathit{leaf}; n_0 \mapsto 1; s_0 \mapsto 0; s_0' \mapsto 1\}$ и переходит на строку 14. Затем выбирается единственное возможное правило для P_0 с телом $\varphi_0 \wedge \mathit{size}(A, n_0) \wedge \mathit{sum}(B, s_0) \wedge \mathit{inc}(C, D) \wedge \mathit{sum}(E, s_0')$. Теперь $\mathit{cti} \not\models [\![\mathit{size}(A, n_0)]\!]_\rho^0 \equiv A = \mathit{leaf} \wedge n_0 = 0$, $\mathit{cti} \models [\![\mathit{sum}(B, s_0)]\!]_\rho^0$, $\mathit{cti} \models [\![\mathit{inc}(C, D)]\!]_\rho^0$, $\mathit{cti} \not\models [\![\mathit{sum}(E, s_0')]\!]_\rho^0$, откуда $\mathit{apps} = \{\mathit{size}(A, n_0), \mathit{sum}(E, s_0')\}$ и $\psi \equiv \neg\varphi_{\mathit{safe}} \wedge \varphi_0 \wedge B = \mathit{leaf} \wedge s_0 = 0 \wedge C = \mathit{leaf} \wedge D = \mathit{leaf}$. Т.к. и *inc*, и *sum* не рекурсивны с P_0, Partition ничего не делает: $\textsc{Partition}(\mathit{apps}) = \{\{1, 4\}\}$.

Обозначим $\alpha_1 = \{\mathit{size} \mapsto 1, \mathit{sum} \mapsto 1\}$. Для получения запроса для α_1, алгоритм строит проекцию ψ, устраняя все переменные, кроме A, n_0, E, и s_0', получая $\psi' \equiv \mathrm{MBP}(\psi, \{T_0, B, C, D, s_0\}, \mathit{cti}) \equiv A = \mathit{leaf} \wedge E = \mathit{leaf} \wedge s_0' \neq 2n_0$, которая после переименования переменных становится $\psi_1 \stackrel{\text{def}}{=} T_1 = \mathit{leaf} \wedge T_2 = \mathit{leaf} \wedge s_2 \neq 2n$. Таким образом, Q становится $\{\langle P_0, \neg\varphi_{\mathit{safe}}, 1\rangle, \langle \alpha_1, \psi_1, 0\rangle\}$, и алгоритм уходит на следующую итерацию.

На второй итерации алгоритм выбирает из очереди $\langle \alpha_1, \psi_1, 0\rangle$ как запрос с наименьшим уровнем. Пусть $\beta_1 \stackrel{\text{def}}{=} [\![\mathit{body}(\alpha_1)]\!]_\rho^{-1} \equiv [\![\mathit{body}(\alpha_1)]\!]_\sigma^{-1} \equiv T_1 = \mathit{leaf} \wedge n = 0 \wedge T_2 = \mathit{leaf} \wedge s_2 = 0$. Т.к. $\beta_1 \wedge \psi_1$ не выполнима, алгоритм выводит новую лемму $\delta_1 \stackrel{\text{def}}{=} \mathrm{ITP}(\beta_1, \psi_1) \equiv T_1 = T_2 = \mathit{leaf} \wedge n = s_2 = 0$. Таким образом, $\sigma(\alpha_1, 0) = \{\delta_1\}$, запрос на уровне 0 отвечен и убирается из Q.

На третьей итерации алгоритм снова выбирает $\langle P_0, \neg\varphi_{\mathit{safe}}, 1\rangle$. На этот раз $[\![\mathit{body}(P_0)]\!]_\sigma^0 \equiv \varphi_0 \wedge \delta_1(A, n, B, s_0) \wedge \delta_1(A, n, E, s_0')$. Т.к. теперь $[\![\mathit{body}(P_0)]\!]_\sigma^0 \wedge \neg\varphi_{\mathit{safe}}$ не выполнима, в $\sigma(P_0, 1)$ добавляется $\mathrm{ITP}([\![\mathit{body}(P_0)]\!]_\sigma^0, \neg\varphi_{\mathit{safe}}) \equiv \varphi_{\mathit{safe}}$. Новое окружение не индуктивно, поэтому RelRecMc переходит на уровень 2.

Уровень 2 Запрос $\langle P_0, \neg\varphi_{safe}, 2\rangle$ выбирается из Q. На этот раз $cti = \{A, B, C, T_0 \mapsto node(0,leaf,leaf); D, E \mapsto leaf; n_0, s_0, s'_0 \mapsto 1\}$. Т.к. $cti \not\models [\![size(A,n_0)]\!]^1_\rho$, $cti \not\models [\![sum(B,s_0)]\!]^1_\rho$, $cti \not\models [\![inc(C,D)]\!]^1_\rho$, $cti \not\models [\![sum(E,s'_0)]\!]^1_\rho$, получаем $apps = \{size(A,n_0), sum(B,s_0), inc(C,D), sum(E,s'_0)\}$. Т.к. ни один из символов не рекурсивен с P_0, получаем $Groups = \{\{1,2,3,4\}\}$. Для получения дочернего свойства безопасности алгоритм устраняет T_0 проекцией и уходит на следующую итерацию с $\{\langle P_0, \neg\varphi_{safe}, 2\rangle, \langle\alpha_2, \psi_2, 1\rangle\}$, где $\alpha_2 \stackrel{\text{def}}{=} \{size \mapsto 1, sum \mapsto 2, inc \mapsto 1\}$ и $\psi_2 \stackrel{\text{def}}{=} T_1 = T_2 = T_4 \wedge U = T_3 \wedge s_3 \neq s_2 + 2n$.

На следующей итерации $\langle\alpha_2, \psi_2, 1\rangle$ выбирается из Q. RelRecMc применяет подход, описанный в Секции 2.4. для эффективного вычисления подстановки σ. Положим

$$\begin{aligned}
\beta_{size} &\stackrel{\text{def}}{=} (T_1 = leaf \wedge n = 0) \vee \\
&\quad (T_1 = node(v_1,L_1,R_1) \wedge n = 1 + n^L + n^R \wedge a_L \wedge a_R) \\
\beta_{sum_1} &\stackrel{\text{def}}{=} (T_2 = leaf \wedge s_2 = 0) \vee \\
&\quad (T_2 = node(v_2,L_2,R_2) \wedge s_2 = v_2 + s_2^L + s_2^R \wedge b_L \wedge b_R) \\
\beta_{sum_2} &\stackrel{\text{def}}{=} (T_3 = leaf \wedge s_3 = 0) \vee \\
&\quad (T_3 = node(v_3,L_3,R_3) \wedge s_3 = v_3 + s_3^L + s_2^R \wedge c_L \wedge c_R) \\
\beta_{inc} &\stackrel{\text{def}}{=} (T_4 = leaf \wedge U = leaf) \vee \\
&\quad (T_4 = node(v_4,L_4,R_4) \wedge U = node(v_4 + 2, L', R') \wedge d_L \wedge d_R)
\end{aligned}$$

Здесь a_L, a_R, b_L, b_R, c_L, c_R, d_L и d_R — пропозициональные абстракции применений реляционных символов. Тогда:

$$\begin{aligned}
[\![body(\alpha_2)]\!]^0_\sigma \equiv\ & \beta_{size} \wedge \beta_{sum_1} \wedge \beta_{sum_2} \wedge \beta_{inc} \wedge \\
& (a_L \wedge b_L \Rightarrow \delta_1(L_1, n_L, L_2, s_2^L)) \wedge \\
& (a_L \wedge b_R \Rightarrow \delta_1(L_1, n_L, R_2, s_2^R)) \wedge \\
& (a_L \wedge c_L \Rightarrow \delta_1(L_1, n_L, L_3, s_3^L)) \ldots
\end{aligned}$$

Если вместо этого прямолинейно перевести формулу $[\![body(\alpha_2)]\!]^0_\sigma$ в ДНФ и заменить всевозможные комбинации применений реляционных символов формулой δ_1, мы получим формулу в 2^4 раз большего размера.

$[\![body(\alpha_2)]\!]^0_\sigma \wedge \psi_2$ выполнима с

$$cti = \{T_k \mapsto node(1,leaf,leaf); T_3, U \mapsto node(3,leaf,leaf; \ldots\}$$

для $k \in \{1,2,4\}$. Таким образом, на строке 18 алгоритм выбирает по второму (рекурсивному) правилу для каждого отношения в $\mathcal{R}$.

Предположим теперь, что cti не выполняет ни одну из ветвей в ρ ни одного применения в телах правил. Тогда получаем $apps = \{size(L_1,n^L), size(R_1,n^R), sum(L_2,s_2^L), sum(R_2,s_2^R), sum(L_3,s_3^L), sum(R_3,s_3^R), inc(L_4,L'), inc(R_4,R')\}$ где все применения рекурсивны, и $\psi \equiv \psi_2 \wedge T_1 = node(v_1,L_1,R_1) \wedge n = 1 + n^L + n^R \wedge \ldots$.

Если Partition вернет единственную группу из всех применений, мы получим запрос для мультимножества размера 8, что в результате может породить запрос

для 16 отношений и т.д.; в результате, RelBndSafety расходится. Чтобы контролировать размер мультимножеств, Partition разбивает *apps* на группы размера, не превосходящего $|\alpha_2| = 4$. Однако существует $C_8^4 = 70$ вариантов уже для разбиения 8 отношений на две группы размера 4. Для выбора наилучшего разбиения, Partition применяет следующую эвристику.

Так как формула свойства безопасности в каждом запросе в Q (кроме корневого) есть результат проекции, она представляет собой конъюнкцию литералов. Для каждого подмножества *apps*, PARTITION определяет *максимальное индуктивное подмножество* литералов.

В данном случае множество литералов в ψ_2 является $\{T_1 = T_2, T_1 = T_4, U = T_3, s_3 \neq s_2 + 2n\}$. Например, литерал $T_1 \wedge T_2$ индуктивен относительно $size(R_1,n^R)$ и $sum(R_2,s_2^R)$. Чтобы убедиться в этом, переименуем $T_1 = T_2$ в $R_1 = R_2$ (т.к. T_1, R_1 и T_2, R_2 — первые аргументы соответствующих применений *size* и *sum*) и заметим, что имеет место $\psi \Rightarrow R_1 = R_2$. В нашем случае, вся формула ψ_2 индуктивна относительно групп $\{size(L_1,n^L), sum(L_2,s_2^L), sum(L_3,s_3^L), inc(L_4,L')\}$ и $\{size(R_1,n^R), sum(R_2,s_2^R), sum(R_3,s_3^R), inc(R_4,R')\}$, поэтому Partition возвращает $Groups = \{\{1,3,5,7\},\{2,4,6,8\}\}$. Для обеих групп в Q добавляются запросы α_2 уровня 0.

На следующих итерациях, алгоритм выводит лемму $T_1 = T_2 = T_4 \wedge U = T_4 \Rightarrow s_3 = s_2 + 2n$ и завершает построение реляционного доказательства безопасности.

3.5. Общие свойства

В этой секции утверждаются важные свойства RelRecMc и RelBndSafety.

Теорема 2. *Алгоритм* RelRecMc *корректен, т.е. если он останавливается для проблемы* $\langle \mathcal{P}, \varphi_{safe} \rangle$*, то она безопасна, тогда и только тогда, когда алгоритм вернул* `БЕЗОПАСНО`*.*

Теорема 3. *Алгоритм* RelBndSafety *корректен, т.е. если он возвращает для проблемы* $\langle \mathcal{P}, \varphi_{safe} \rangle$ *на уровне* B `НЕДОСТИЖИМО`*, тогда и только тогда, когда все выводы системы сверху-вниз высоты, ограниченной сверху* B*, безопасны.*

Оба алгоритма проверяют на выполнимость формулы языка ограничений (например, в строке 9 RelRecMc или в строке 4 RelBndSafety). Назовём оракулом выполнимости в $\mathcal{M}$ процедуру, которая для любой формулы языка ограничений φ за один шаг гарантированно проверяет $\mathcal{M} \models \varphi$. Также подразумевается, что такой оракул способен выдавать модели формул, Крейговские интерполянты и проектировать формулы на основе моделей; на практике таким оракулом является SMT-решатель.

Теорема 4. RelBndSafety *полна относительно оракула выполнимости в* $\mathcal{M}$*, т.е. при наличии оракула выполнимости в* $\mathcal{M}$ *для любой проблемы безопасности* $\langle \mathcal{P}, \varphi_{safe} \rangle$*, уровня* $B \in \mathbb{N}$ *и пустых* σ *и* ρ*,* RelBndSafety *останавливается и возвращает* `ДОСТИЖИМО`*, либо* `НЕДОСТИЖИМО`*.*

Теоремы, аналогичные 2, 3 и 4 доказаны в Теоремах 1 и 2 работы [3]. Для того, чтобы превратить доказательство, представленное в [3], достаточно заменить

все интерпретации и семантики символов на их реляционные аналоги. Например, инвариант традиционного алгоритма

$$\text{для всех } P \in \mathcal{R}, [\![P]\!]^b_{\mathcal{M}} \subseteq \mathcal{M}(O^b_\sigma(P))$$

становится инвариантом RelRecMc

$$\text{для всех } \langle P_1, \ldots, P_n\rangle \in dom(\sigma), [\![P_1]\!]^b_{\mathcal{M}} \times \cdots \times [\![P_n]\!]^b_{\mathcal{M}} \subseteq \mathcal{M}(O^b_\sigma(\langle P_1, \ldots, P_n\rangle)).$$

Теорема 5. *При наличии оракула выполнимости в $\mathcal{M}$* RelRecMc *является коразрешающей процедурой проблем безопасности дизъюнктов, т.е. если $\mathcal{P}$ небезопасно, процедура гарантированно остановится и вернёт* НЕБЕЗОПАСНО.

Доказательство. Любая небезопасная система дизъюнктов имеет резолютивное опровержение с деревом вывода некоторой конечной высоты. Пусть H — наименьшая высота среди всех таких деревьев. Т.к. алгоритм RelRecMc итеративно увеличивает глубину b, то по корректности и относительной полноте RelBndSafety все вызовы RelBndSafety в строке 3 остановятся и вернут `НЕДОСТИЖИМО` (в противном случае, существует резолютивное опровержение меньшей глубины). На шаге $b = H$ остановится RelBndSafety и вернёт ответ НЕБЕЗОПАСНО. □

Для систем с конечным пространством состояний RelRecMc — полная разрешающая процедура; для них алгоритм полиномиален от количества состояний.

Если при выполнении строки 25 алгоритма 2 *Groups* содержит только единичные мультимножества, поведение RelBndSafety совпадает с поведением BndSafety из [3]. Другими словами, представленный алгоритм ведёт себя аналогично BndSafety на *линейных* системах дизъюнктов и *обобщает* его поведение на нелинейных системах: если Partition разбивает входные применения на группы размера 1, то алгоритм выводит "классические" доказательства безопасности аналогично оригинальному алгоритму.

В некоторых случаях, когда RecMC [3] не может вывести доказательство безопасности нелинейных систем, т.к. каждая теоретико-множественная модель не представима в языке ограничений, наш алгоритм справляется с поиском реляционного доказательства безопасности. Напротив, если RecMC успешно доказывает или опровергает безопасность системы, наш алгоритм также справляется.

4. Эксперименты

Алгоритм был реализован над Spacer, современным решателем дизъюнктов Хорна в SMT-решателе Z3 [23][4]. Мы сравнили реализацию со Spacer и HoIce [8] на двух наборах тестов[5]. Эксперименты были проведены на компьютере под управлением ОС Arch Linux с процессором Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.

Первый набор тестов содержит 840 проблем из [8], не являющихся проблемами реляционной верификации. Мы сравнили наш алгоритм со Spacer и HoIce и показали его жизнеспособность. Таймаут для этих тестов был 30 секунд. Spacer решил

[4]Реализация доступна по ссылке https://github.com/dvvrd/z3.
[5]Тесты доступны по ссылке https://github.com/dvvrd/spacer-benchmarks.

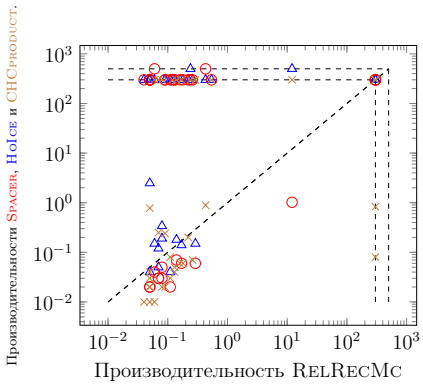

Рис. 1. Сравнение RelRecMc с другими решателями. Каждая точка графика представляет пару времён исполнения теста (сек. × сек.) алгоритмом RelRecMc (ось x) и другим алгоритмом (ось y). Таймауты указаны внутренней пунктирной линией; на внешних пунктирных линиях лежат тесты, на которых решатели завершились с ошибкой времени исполнения.
Fig. 1. RelRecMc vs competitors. Each point in a plot represents a pair of the run times (sec × sec) of RelRecMc (x-axis) and a competitor (y-axis). Timeouts are placed on the inner dashed lines; and crashes are on the outer dashed lines.

788 из 840 проблем с 50 таймаутами и 2 ошибками времени исполнения. Наша реализация решила 807 проблем с 34 таймаутами. Накладные расходы по времени в сравнении со Spacer незначительны (менее 0.1 секунды на 87% проблем). Наша реализация решила большинство проблем, решённых Spacer. Тем не менее, 10 были решены Spacer, но не нашей реализацией; мы объясняем это несовершенством текущей реализации. HoIce решил 808 проблем с 26 таймаутами и 6 ошибками времени исполнения, но и Spacer, и наша реализация RelRecMc затратила меньше времени, чем HoIce на решённых проблемах.

Второй набор тестов содержит 37 проблем реляционной верификации, взятых из [24]. Мы сравнили нашу реализацию со Spacer, HoIce и алгоритмом CHCproduct из [17], реализованным как синтаксическое преобразование входной системы дизъюнктов с последующим решением преобразованной системы решателем Spacer. Схематичное сравнение показано на рис. 1.

Spacer и HoIce решили только 11 из 37 задач с 5-минутным таймаутом. CHCproduct решил 24 задачи, а RelRecMc решил 32 задачи из 37. Важно, что RelRecMc решил задачи, с которыми не справился Spacer после явного синтаксического слияния дизъюнктов. Для больших небезопасных проблем, например, `point-location*`, CHCproduct порождает экспоненциальное количество правил, расходуя всё отведённое время, в то время как другие решатели находят контрпример за секунды.

5. Обзор литературы

Большинство подходов к реляционной верификации основаны на автоматическом или интерактивном анализе программ-произведений [9–12,14–18,25]. Все такие под-

ходы трактуют верификатор функциональных спецификаций как чёрный ящик. Поэтому они вынуждены предопределять стратегию синхронизации. Напротив, наш подход не строит произведение программ явно, а использует SMT-решатель для построения и стратегий синхронизации, и реляционных инвариантов одновременно.

Декартова логика Хоара [26, 27] для доказательств k-свойств безопасности состоит из множества правил и эвристик для выравнивания циклов сравниваемых программ. Они анализируют условия циклов, ветвления, реляционные пред- и постусловия. Для выбора стратегии синхронизации в работе [27] определяется максимальное множество циклов, завершение каждого из которых влечёт завершение других (иначе реляционные инварианты не обнаруживаются, даже если они существуют). Напротив, наш подход не учитывает завершения циклов и может строить реляционные инварианты для циклов с различными количествами итераций.

Существуют подходы к трансформации нелинейных систем дизъюнктов, которые используют существующие решатели для автоматического построения реляционных инвариантов [16, 17]. Преобразование CHCprodcut [17] строит декартово произведение множества правил системы дизъюнктов. Когда правила перемножаемых реляционных символов имеют более одного рекурсивного применения в теле, преобразование CHCprodcut определено не единственным образом. Для этого применяется расширенная техника *синхронного произведения*, которая пытается выбрать произведение, соединяющее структурно схожие рекурсивные применения. Альтернативно, [16] предлагает трансформацию, основанную на известном подходе fold/unfold. Хотя результирующая система дизъюнктов решается проще, сложность трансформации растёт экспоненциально от числа перемноженных предикатных символов. Для сравнения, наш подход обрабатывает рекурсивные группы по требованию, используя модели, полученные от SMT-запросов, и поэтому не ведёт к экспоненциальному взрыву в сложности.

Недавний подход [18] наиболее близок к данной работе. Он анализирует контрпримеры для определения нетривиальных стратегий синхронизации, но использует заданное пользователем множество предикатов для построения реляционных инвариантов. Наш подход не требует предикатов и строит инварианты интерполяцией, эффективно используя преимущества, унаследованные от [3]. В будущем мы планируем поддержать определение других стратегий синхронизации.

6. Заключение и дальнейшие планы

В статье был представлен новый подход к решению нелинейных систем дизъюнктов Хорна, основанный на направляемой свойством достижимости. Его ключевая особенность состоит в автоматическом выводе реляционных инвариантов, аппроксимирующих сверху семантику групп неинтерпретированных символов. Подход способен автоматически определять, какие предикаты должны быть сгруппированы, а для каких будет достаточно вывести индивидуальные инварианты. Подход был реализован поверх решателя Spacer, была показана его практическая полезность на тестовых наборах различных задач реляционной верификации.

В будущем планируется разработка подхода к автоматическому определению

нетривиальных стратегий синхронизации, ищущего различные раскрутки частей программ-произведений; это даст возможность более гибкого поиска реляционных инвариантов.

Список литературы / References

[1] Krystof H., Bjørner N, "Generalized Property Directed Reachability", *LNCS, Springer*, **7317** (2012), 157–171.

[2] Cimatti A., Griggio A., Mover S., Tonetta S., "IC3 Modulo Theories via Implicit Predicate Abstraction", *LNCS, Springer*, **8413** (2014), 46–61.

[3] Komuravelli A., Gurfinkel A., Chaki S., "SMT-Based Model Checking for Recursive Programs", *Formal Methods in System Design*, **48**:3 (2016), 175–205.

[4] Jovanović D., Dutertre B., "Property-Directed k-induction", *FMCAD,IEEE*, 2016, 85–92.

[5] Fedyukovich G., Bodík R., "Accelerating Syntax-Guided Invariant Synthesis", *LNCS, Springer*, **10805** (2018), 251–269.

[6] Непомнящий В.А. и др., "Ориентированный на верификацию язык C-light", *Системная информатика: Сб. науч. тр. РАН. Сиб. отд-ние. Ин-т систем информатики*, **9** (2004), 51–134; [Nepomniaschy V.A. et al., "Verification oriented language C-light", *Sistemnaya Informatika: Sb. Nauch. Tr. RAN. Sib. Otd-nie. In-t Sistem Informatiki*, **9** (2004), 51–134, (in Russian).]

[7] Мандрыкин М. У., Мутилин В. С., Хорошилов А. В., "Введение в метод CEGAR — уточнение абстракции по контрпримерам", *Труды Института системного программирования РАН*, **24** (2013); [Mandrykin M. U., Mutilin V. S., Khoroshilov A. V., "Introduction to CEGAR — Counter-Example Guided Abstraction Refinement", *Proceedings of ISP RAS*, **24** 2013.]

[8] Champion A., Kobayashi N., Sato R., "HoIce: An ICE-Based Non-linear Horn Clause Solver", *Asian Symposium on Programming Languages and Systems, Springer*, 2018, 146–156.

[9] Lahiri S. K., McMillan K. L., Sharma R., Hawblitzel C., "Differential Assertion Checking", *FSE, ACM*, 2013, 345–355.

[10] Almeida J. B., Barbosa M., Barthe G., Dupressoir F., Emmi M., "Verifying Constant-Time Implementations", *USENIX*, 2016, 53–70.

[11] Kiefer M., Klebanov V., Ulbrich M., "Relational Program Reasoning Using Compiler IR", *LNCS, Springer*, **9971** (2016), 149–165.

[12] Beckert B., Bingmann T., Kiefer M., Sanders P., Ulbrich M., Weigl A., "Relational Equivalence Proofs Between Imperative and MapReduce Algorithms", *LNCS, Springer*, **11294** (2018), 248–266.

[13] Athanasiou K. et al., "SideTrail: Verifying Time-Balancing of Cryptosystems", *LNCS, Springer*, **11294** (2018), 215–228.

[14] Barthe G., Crespo J. M., Kunz C., "Relational Verification Using Product Programs", *LNCS, Springer*, **6664** (2011), 200–214.

[15] Felsing D. et al., "Automating Regression Verification", *ASE, ACM*, 2014, 349–360.

[16] De Angelis E. et al., "Relational Verification Through Horn Clause Transformation", *LNCS, Springer*, **9837** (2016), 147–169.

[17] Mordvinov D., Fedyukovich G., "Synchronizing Constrained Horn Clauses", *EPiC Series in Computing, EasyChair*, **46** (2017), 338–355.

[18] Shemer R., Gurfinkel A., Shoham S., Vizel Y., "Property Directed Self Composition", *LNCS, Springer*, **11561** (2019), 161–179.

[19] Clarke E. M., "Program Invariants as Fixedpoints", **21**:4 (1979), 273–294.

[20] Apt K. R., *From Logic Programming to Prolog*, Prentice Hall London, 1997.

[21] Bjørner N., Janota M., "Playing with Quantified Satisfaction", *LPAR (short papers)*, **35** (2015), 15–27.

[22] Bradley A. R., "SAT-Based Model Checking without Unrolling", *LNCS, Springer*, **6538** (2011), 70–87.

[23] De Moura L., Bjørner N., "Z3: An Efficient SMT Solver", *LNCS, Springer*, **4963** (2008), 337–340.

[24] Mordvinov D., Fedyukovich G., "Verifying Safety of Functional Programs with Rosette/Unbound", *CoRR.*, **abs/1704.04558.** (2017).

[25] Strichman O., Veitsman M., "Regression Verification for Unbalanced Recursive Functions", *LNCS*, **9995** (2016), 645–658.

[26] Sousa M., Dillig I., "Cartesian Hoare Logic for Verifying k-safety Properties", *PLDI, ACM*, 2016, 57–69.

[27] Pick L., Fedyukovich G., Gupta A., "Exploiting Synchrony and Symmetry in Relational Verification", *LNCS, Springer*, **10981** (2018), 164–182.

Mordvinov D. A., "Property-Directed Inference of Relational Invariants", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 550–571.

DOI: 10.18255/1818-1015-2019-4-550-571

Abstract. Property Directed Reachability (PDR) is an efficient and scalable approach to solving systems of symbolic constraints also known as Constrained Horn Clauses (CHC). In the case of non-linear CHCs, which may arise, e.g., from relational verification tasks, PDR aims to infer an inductive invariant for each uninterpreted predicate. However, in many practical cases this reasoning is not successful, as invariants should be derived for groups of predicates instead of individual predicates. The article describes a novel algorithm that identifies these groups automatically and complements the existing PDR technique. The key feature of the algorithm is that it does not require a possibly expensive synchronization transformation over the system of CHCs. We have implemented the algorithm on top of a up-to-date CHC solver Spacer. Our experimental evaluation shows that for some CHC systems, on which existing solvers diverge, our tool is able to discover relational invariants.

Keywords: relational verification, constrained horn clauses, property-directed reachability, relational invariants

On the authors:
Dmitry A. Mordvinov, orcid.org/0000-0002-6437-3020, senior reseacher,
St Petersburg University and JetBrains Research,
28 Universitetskiy pr., Petrodvorets 198504, Russia, e-mail: dmitry.mordvinov@jetbrains.com

Acknowledgments:
This work was supported by JetBrains Research.

Моделирование и анализ информационных систем. Т. 26, №4 (2019), с. 572–582
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 572–582

Computer system organization

DOI: 10.18255/1818-1015-2019-4-572-582

УДК 004.021, 517.929

Особенности вычислительной реализации алгоритма оценки ляпуновских показателей систем с запаздыванием

Горюнов В. Е.

Поступила в редакцию 23 октября 2019
После доработки 6 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. Рассматривается вычислительная реализация алгоритма оценки спектра показателей Ляпунова для систем дифференциальных уравнений с запаздывающими аргументами. Учитывая, что для таких систем, а также для краевых задач не удается доказать известную теорему Оселедеца, которая позволяет эффективно вычислять искомые величины, приходится говорить лишь об оценках характеристических показателей, в каком-то смысле близких к ляпуновским. В данной работе предложены две методики обработки решений линеаризованных на аттракторе систем, одна из которых основана на базисе импульсных функций, а другая — на базисе тригонометрических функций. Продемонстрирована гибкость применения указанных алгоритмов в случае квазиустойчивых структур, когда несколько показателей Ляпунова близки к нулю. Разработанные методы тестируются на логистическом уравнении с запаздыванием. Полученные результаты иллюстрируют “близость” оцениваемых характеристик и показателей Ляпунова.

Ключевые слова: спектр показателей Ляпунова, динамическая система с запаздыванием, численный алгоритм, уравнение Хатчинсона

Для цитирования: Горюнов В. Е., "Особенности вычислительной реализации алгоритма оценки ляпуновских показателей систем с запаздыванием", *Моделирование и анализ информационных систем*, **26**:4 (2019), 572–582.

Об авторах:
Горюнов Владимир Евгеньевич, orcid.org/0000-0002-0512-6986, аспирант,
Ярославский государственный университет им. П. Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: salkar@ya.ru

Благодарности:
Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта №18-29-10055.

Введение

Рассматривается расширение стандартного алгоритма [1] вычисления нескольких первых показателей Ляпунова для систем дифференциальных уравнений с запаздывающими аргументами.

Отметим, что ляпуновские показатели для систем с запаздыванием могут не вполне корректно оцениваться численно. Дело в том, что для конечномерных систем имеет место известная теорема Оселедеца [2], в которой утверждается, что линеаризованная на устойчивом решении система всегда является правильной по Ляпунову. Это позволяет в определении ляпуновских показателей [3] заменить верхний предел на обычный и численно оценивать эти величины. В случае уравнений с запаздывающими аргументами и краевых задач такую теорему доказать не удается. Поэтому при разработке алгоритмов вычисления ляпуновских показателей важно иметь модельное уравнение с запаздыванием, для которого спектр может быть вычислен каким-либо другим способом. Наличие такой задачи позволяет протестировать разработанный алгоритм и убедиться в его работоспособности. В статьях [4–6] вычисляются спектры ляпуновских показателей, однако обоснования предложенного алгоритма, как, впрочем, и тестирующего примера авторы не приводят, в отличие от работы [7], которая послужила основой для представленных в данной статье методик. Отдельно упомянем [8] — одну из первых работ по данной тематике. Отметим, что все результаты, описанные в этой статье, носят экспериментальный характер.

1. Описание алгоритма

Опишем процесс получения оценок первых K показателей Ляпунова в случае систем дифференциальных уравнений с запаздывающими аргументами следующего вида:

$$\dot{x} = F\big(x, x(t-h_1), x(t-h_2), \ldots, x(t-h_s)\big), \tag{1}$$

где для $\forall t\ x(t) \in \mathbb{R}^N$, N — размерность системы, $h_i \in \mathbb{R}$ $(i = 1, \ldots, s)$, $h_1 > h_2 > \ldots > h_s > 0$.

В качестве фазового пространства примем пространство непрерывных на отрезке $[-h_1, 0]$ функций в $\mathbb{R}^N$, а именно $C([-h_1, 0]; \mathbb{R}^N)$.

Для численного решения системы (1) с начальными условиями

$$x_0(t) = f(t), \quad t \in [-h_1, 0], \quad f(t) \in C([-h_1, 0]; \mathbb{R}^N) \tag{2}$$

будем использовать метод Дормана–Принса пятого порядка (DOPRI54) с переменной длиной шага [9].

Таким образом, будем решать систему (1) с соответствующим начальным условием (2) выбранным методом до момента времени Θ, достаточного для приближения траектории решения к изучаемому аттрактору. При этом на промежутке $t \in [\Theta - h_1, \Theta]$ получим функцию $x^{(0)}(t) \in C([\Theta - h_1, \Theta]; \mathbb{R}^N)$, которая станет новым начальным условием системы (1).

Дополним систему уравнений (1) следующими K идентичными системами:

$$\begin{aligned} \dot{u}_j = \frac{\partial F}{\partial x}\Big|_{x=x_*(t)} \cdot u_j + \frac{\partial F}{\partial x(t-h_1)}\Big|_{x(t-h_1)=x_*(t-h_1)} \cdot u_j(t-h_1) + \ldots + \\ + \frac{\partial F}{\partial x(t-h_s)}\Big|_{x(t-h_s)=x_*(t-h_s)} \cdot u_j(t-h_s), \end{aligned} \tag{3}$$

где $j = 1, \ldots, K$, $x_*(t)$ — решение системы уравнений (1) с начальным условием $x(t) = x^{(0)}(t)$ при $t > \Theta$. Они представляют собой линеаризованные на решении $x_*(t)$ системы уравнений (1). Далее будем обозначать $u_j = u_j(t) \stackrel{\text{def}}{=} \big(u_{1j}(t), \ldots, u_{Nj}(t)\big)^{\mathrm{T}}$, для которого введем норму

$$\|u_j(t)\|^2 \stackrel{\text{def}}{=} \int\limits_{t-h_1}^{t} |u_{1j}(\tau)|^2 d\tau + \ldots + \int\limits_{t-h_1}^{t} |u_{Nj}(\tau)|^2 d\tau. \tag{4}$$

Для каждой системы уравнений (3) используем начальные условия в виде ортонормированных импульсных функций, например:

$$u_j(t) = \begin{cases} \sqrt{\dfrac{K}{Nh_1}} & \text{при } t \in \Big[(\Theta - h_1) + (j-1)h_1/K, \\ & \qquad (\Theta - h_1) + jh_1/K\Big], \quad j = 1, \ldots, K, \\ 0 & \text{иначе,} \end{cases} \tag{5}$$

либо в виде ортонормированных тригонометрических функций:

$$u_j(t) = \begin{cases} 1, & j = 1, \\ \sin\big(j\pi(t-\Theta)/h_1\big)/\sqrt{2}, & j = 2, 4, 6, \ldots \\ \cos\big((j-1)\pi(t-\Theta)/h_1\big)/\sqrt{2}, & j = 3, 5, 7, \ldots \end{cases} \tag{6}$$

Решая совместно систему (1) с начальным условием $x(t) = x^{(0)}(t)$ и системы уравнений (3) с начальными условиями (5) или (6) на промежутке $t \in [\Theta, \Theta + T]$, где $T \geq h_1$, получаем для каждой из линеаризованных систем решение $u_j^{(1)}(t) = \big(u_{1j}^{(1)}(t), \ldots, u_{Nj}^{(1)}(t)\big)^{\mathrm{T}}$, $j = 1, \ldots, K$.

Учитывая, что решения $u_j^{(1)}(t)$ ведут себя экспоненциально, необходимо их перенормировывать через определенные промежутки времени. Заметим, что проблему представляют как неограниченный рост решений, так и их стремление к нулю. Таким образом, на промежутке $t \in [\Theta + T - h_1, \Theta + T]$ усредняем внутри каждого из M равных временных интервалов длины h_1/M вычисленные решения линеаризованных систем $u_{1j}^{(1)}(t), \ldots, u_{Nj}^{(1)}(t)$, в результате чего получаем кусочно-непрерывные функции $\tilde{u}_{1j}^{(1)}(t), \ldots, \tilde{u}_{Nj}^{(1)}(t)$ соответственно, которые используем в одном из описанных ниже методов.

Метод импульсных функций.

- Применяем метод Грама–Шмидта [10] к $\tilde{u}_j^{(1)}(t)$.
- При этом после процедуры ортогонализации каждой функции, но перед ее перенормировкой вычисляем величины $\xi_j^{(1)} = \|\tilde{u}_{\text{ort } j}^{(1)}\|$, где $\|\cdot\|$ — норма, определенная в (4), $\tilde{u}_{\text{ort } j}^{(1)}(t)$ — ортогонализированная система функций $\tilde{u}_j^{(1)}(t)$.
- Продолжаем решать системы (1), (3), при этом в качестве начальных условий для линеаризованных систем используем полученную ортонормированную систему функций.

Для применения данного метода требуется в качестве начальных условий для линеаризованных систем выбрать систему ортонормированных импульсных функций (5).

Метод тригонометрических функций.

- Предварительно переводим функции $\tilde{u}_j^{(1)}(t)$ в систему векторов $v_j^{(1)} \in \mathbb{R}^{MN}$ по следующему правилу: $v_{mnj}^{(1)} = \tilde{u}_{nj}^{(1)}(\Theta+T-h_1+(m-1/2)h_1/M)$, $m = 1, \ldots, M$, $n = 1, \ldots, N$, $j = 1, \ldots, K$.
- К получившейся системе векторов применяем дискретное преобразование Фурье [11], в результате чего получаем комплекснозначные векторы $c_j^{(1)} \in \mathbb{C}^{MN/2+1}$.
- Векторы $c_j^{(1)}$ разделяем на пары действительнозначных векторов, состоящие из действительных и мнимых частей $d_j^{(1)} \in \mathbb{R}^{MN+2}$.
- К системе векторов $d_j^{(1)}$ применяем метод Грама–Шмидта.
- При этом после процедуры ортогонализации каждого вектора, но перед его перенормировкой вычисляем величины
$$\xi_j^{(1)} = \|d_{\mathrm{ort}\ j}^{(1)}\| = \Big(\sum_{j=1}^{K}\sum_{i=1}^{NM+2} |d_{\mathrm{ort}\ ij}^{(1)}|^2\Big)^{\frac{1}{2}},$$
где $d_{\mathrm{ort}\ j}^{(1)}$ — ортогонализированная система векторов $d_j^{(1)}$, $j = 1, \ldots, K$.
- Полученные ортонормированные действительнозначные векторы переводим обратно в комплекснозначные, к которым применяем обратное дискретное преобразование Фурье, таким образом получая векторы $w_j^{(1)} \in \mathbb{R}^{MN}$.
- Строим систему функций $\hat{u}_j^{(1)}(t)$ по правилу: $\hat{u}_{nj}^{(1)}(t) = w_{mnj}^{(1)}$ при $t \in [\Theta + T - h_1+(m-1)h_1/M; \Theta+T-h_1+mh_1/M]$, $m = 1, \ldots, M$, $n = 1, \ldots, N$, $j = 1, \ldots, K$.
- Продолжаем решать системы (1), (3), при этом в качестве начальных условий для линеаризованных систем используем систему функций $\hat{u}_j^{(1)}(t)$.

Для применения данного метода требуется в качестве начальных условий для линеаризованных систем выбрать систему ортонормированных тригонометрических функций (6).

Повторяем описанный процесс на временных интервалах $[\Theta+kT-h_1, \Theta+kT]$, $k > 1$, в результате чего обработке алгоритмом подвергаются соответствующие решения $u_j^{(k)}(t)$. Оценка показателей Ляпунова в таком случае вычисляется по формуле

$$\lambda_j = \lim_{L\to\infty} \frac{\sum_{k=1}^{L} \ln \xi_j^{(k)}}{LT}, \quad j = 1, \ldots, K. \tag{7}$$

Отметим, что выбор времени перенормировки T можно осуществлять двумя различными способами: через равные промежутки времени или динамически [12]. В последнем случае на каждом шаге алгоритма придется хранить не только $\xi_j^{(k)}$, но и T_k. Тогда формула оценки показателей Ляпунова будет иметь вид

$$\lambda_j = \lim_{L\to\infty} \frac{\sum_{k=1}^{L} \ln \xi_j^{(k)}/T_k}{L}, \quad j = 1, \ldots, K.$$

Описанная структура алгоритма позволяет начинать вычисления и до выхода решения на изучаемый аттрактор, особенно когда сам процесс приближения решения к аттрактору может быть сопряжен с большими вычислительными трудностями, как это показано в [13] для случая нескольких близких к нулю показателей Ляпунова у квазиустойчивых структур. В таком случае первые шаги, соответствующие временным интервалам, в которых решение еще не приблизилось к аттрактору на достаточное расстояние, должны отбрасываться и никак не учитываться в сумме из формулы (7). Также рекомендуется отбрасывать несколько первых шагов и в общем случае, поскольку процесс формирования нового ортонормированного базиса линеаризованных систем может заметно отразиться на спектре показателей при не слишком большом количестве учтенных шагов L (см. подобный прием в [14]).

Теперь перейдем к результатам тестирования описанных методов.

2. Тестирование на примере уравнения Хатчинсона

Вычислительные эксперименты проводились для уравнения Хатчинсона [15], которое имеет следующий вид:

$$\dot{x} = rx(t)\big(1 - x(t-1)\big). \tag{8}$$

В работе [16] показано, что ненулевые решения уравнения (8) асимптотически устойчивы при $r \in (0, \pi/2)$, причем при $r \in (0, e^{-1})$ монотонно, а при $r \in (e^{-1}, \pi/2)$ колебательным образом решение стремится к единице. Кроме того, единичное решение обладает глобальной устойчивостью при $r \le 37/24$ [16–18], а в [19, 20] приведен алгоритм, который допускает улучшение этой оценки. В случае единичного состояния равновесия при $r < \pi/2$ показатели Ляпунова для уравнения Хатчинсона совпадают с вещественными частями корней характеристического квазиполинома $P(\lambda) \equiv \lambda + r\exp(-\lambda)$, $\lambda = \tau + i\omega$. Для их вычисления используется система алгебраических уравнений:

$$\tau + re^{-\tau}\cos\omega = 0, \quad \omega - re^{-\tau}\sin\omega = 0.$$

Округленные компоненты решения τ_i данной системы при разных значениях параметра r представлены во втором столбце таблиц 1–3. Будем называть их эталонными значениями.

Уравнение (8) дополняется системой линеаризованных уравнений:

$$\dot{u}_j = r\big(1 - x(t-1)\big)u_j(t) - rx(t)u_j(t-1), \quad j = 1, \ldots, K.$$

Для всех опытов применялись следующие параметры:

- количество вычисляемых показателей Ляпунова $K = 10$;
- время выхода на аттрактор $\Theta = 150$;
- время до перенормировки решений линеаризованных систем $T = 4$;
- количество пересчетов показателей Ляпунова $L = 5000$.

Таблица 1. Первые 10 оценок показателей Ляпунова ($\check{\lambda}_i$ – методом импульсных функций, $\tilde{\lambda}_i$ – методом тригонометрических функций) для уравнения Хатчинсона при $r = 1.5$, а также абсолютная разность σ_i и относительная разность ρ_i между ними и эталонными значениями.
Table 1. The first 10 estimates of Lyapunov exponents ($\check{\lambda}_i$ – by the method of impulse functions, $\tilde{\lambda}_i$ – by the method of trigonometric functions) for the Hutchinson equation at $r = 1.5$ as well as the absolute difference σ_i and the relative difference ρ_i between the obtained and the reference values.

i	τ_i	$\check{\lambda}_i$	$\check{\sigma}_i$	$\check{\rho}_i$	$\tilde{\lambda}_i$	$\tilde{\sigma}_i$	$\tilde{\rho}_i$
$M = 100$							
1	−0.0328	−0.0338	0.0010	0.0305	−0.0335	0.0007	0.0213
2	−0.0328	−0.0338	0.0010	0.0305	−0.0336	0.0008	0.0244
3	−1.6509	−1.6544	0.0035	0.0021	−1.6540	0.0031	0.0019
4	−1.6509	−1.6544	0.0035	0.0021	−1.6541	0.0032	0.0019
5	−2.2447	−2.2491	0.0044	0.0020	−2.2488	0.0041	0.0018
6	−2.2447	−2.2491	0.0044	0.0020	−2.2489	0.0042	0.0019
7	−2.6130	−2.6179	0.0049	0.0019	−2.6178	0.0048	0.0018
8	−2.6130	−2.6179	0.0049	0.0019	−2.6179	0.0049	0.0019
9	−2.8811	−2.8866	0.0055	0.0019	−2.8868	0.0057	0.0020
10	−2.8811	−2.8866	0.0055	0.0019	−2.8869	0.0058	0.0020
$M = 1000$							
1	−0.0328	−0.0329	0.0001	0.0030	−0.0326	0.0002	0.0061
2	−0.0328	−0.0330	0.0002	0.0061	−0.0327	0.0001	0.0030
3	−1.6509	−1.6513	0.0004	0.0002	−1.6509	0.0000	0.0000
4	−1.6509	−1.6513	0.0004	0.0002	−1.6510	0.0001	0.0001
5	−2.2447	−2.2452	0.0005	0.0002	−2.2448	0.0001	0.0000
6	−2.2447	−2.2452	0.0005	0.0002	−2.2449	0.0002	0.0001
7	−2.6130	−2.6134	0.0004	0.0002	−2.6132	0.0002	0.0001
8	−2.6130	−2.6134	0.0004	0.0002	−2.6132	0.0002	0.0001
9	−2.8811	−2.8815	0.0004	0.0001	−2.8813	0.0002	0.0001
10	−2.8811	−2.8815	0.0004	0.0001	−2.8813	0.0002	0.0001
$M = 2000$							
1	−0.0328	−0.0329	0.0001	0.0030	−0.0325	0.0003	0.0091
2	−0.0328	−0.0330	0.0002	0.0061	−0.0326	0.0002	0.0061
3	−1.6509	−1.6512	0.0003	0.0002	−1.6507	0.0002	0.0001
4	−1.6509	−1.6512	0.0003	0.0002	−1.6508	0.0001	0.0001
5	−2.2447	−2.2450	0.0003	0.0001	−2.2446	0.0001	0.0000
6	−2.2447	−2.2450	0.0003	0.0001	−2.2447	0.0000	0.0000
7	−2.6130	−2.6132	0.0002	0.0001	−2.6130	0.0000	0.0000
8	−2.6130	−2.6132	0.0002	0.0001	−2.6130	0.0000	0.0000
9	−2.8811	−2.8812	0.0001	0.0000	−2.8811	0.0000	0.0000
10	−2.8811	−2.8812	0.0001	0.0000	−2.8811	0.0000	0.0000

Таблица 2. Первые 10 оценок показателей Ляпунова ($\check{\lambda}_i$ – методом импульсных функций, $\tilde{\lambda}_i$ – методом тригонометрических функций) для уравнения Хатчинсона при $r = 1.0$, а также абсолютная разность σ_i и относительная разность ρ_i между ними и эталонными значениями.
Table 2. The first 10 estimates of Lyapunov exponents ($\check{\lambda}_i$ – by the method of impulse functions, $\tilde{\lambda}_i$ – by the method of trigonometric functions) for the Hutchinson equation at $r = 1.0$ as well as the absolute difference σ_i and the relative difference ρ_i between the obtained and the reference values.

i	τ_i	$\check{\lambda}_i$	$\check{\sigma}_i$	$\check{\rho}_i$	$\tilde{\lambda}_i$	$\tilde{\sigma}_i$	$\tilde{\rho}_i$
$M = 100$							
1	−0.3181	−0.3195	0.0014	0.0044	−0.3192	0.0011	0.0035
2	−0.3181	−0.3196	0.0015	0.0047	−0.3194	0.0013	0.0041
3	−2.0623	−2.0663	0.0040	0.0019	−2.0659	0.0036	0.0017
4	−2.0623	−2.0663	0.0040	0.0019	−2.0660	0.0037	0.0018
5	−2.6532	−2.6580	0.0048	0.0018	−2.6578	0.0046	0.0017
6	−2.6532	−2.6580	0.0048	0.0018	−2.6578	0.0046	0.0017
7	−3.0202	−3.0257	0.0055	0.0018	−3.0256	0.0054	0.0018
8	−3.0202	−3.0257	0.0055	0.0018	−3.0257	0.0055	0.0018
9	−3.2878	−3.2938	0.0060	0.0018	−3.2943	0.0065	0.0020
10	−3.2878	−3.2938	0.0060	0.0018	−3.2943	0.0065	0.0020
$M = 1000$							
1	−0.3181	−0.3183	0.0002	0.0006	−0.3180	0.0001	0.0003
2	−0.3181	−0.3184	0.0003	0.0009	−0.3181	0.0000	0.0000
3	−2.0623	−2.0628	0.0005	0.0002	−2.0622	0.0001	0.0000
4	−2.0623	−2.0628	0.0005	0.0002	−2.0624	0.0001	0.0000
5	−2.6532	−2.6537	0.0005	0.0002	−2.6533	0.0001	0.0000
6	−2.6532	−2.6537	0.0005	0.0002	−2.6534	0.0002	0.0001
7	−3.0202	−3.0207	0.0005	0.0002	−3.0204	0.0002	0.0001
8	−3.0202	−3.0207	0.0005	0.0002	−3.0205	0.0003	0.0001
9	−3.2878	−3.2882	0.0004	0.0001	−3.2880	0.0002	0.0001
10	−3.2878	−3.2882	0.0004	0.0001	−3.2880	0.0002	0.0001
$M = 2000$							
1	−0.3181	−0.3182	0.0001	0.0003	−0.3178	0.0003	0.0009
2	−0.3181	−0.3184	0.0003	0.0009	−0.3180	0.0001	0.0003
3	−2.0623	−2.0626	0.0003	0.0001	−2.0620	0.0003	0.0001
4	−2.0623	−2.0626	0.0003	0.0001	−2.0622	0.0001	0.0000
5	−2.6532	−2.6534	0.0002	0.0001	−2.6531	0.0001	0.0000
6	−2.6532	−2.6534	0.0002	0.0001	−2.6531	0.0001	0.0000
7	−3.0202	−3.0204	0.0002	0.0001	−3.0202	0.0000	0.0000
8	−3.0202	−3.0204	0.0002	0.0001	−3.0202	0.0000	0.0000
9	−3.2878	−3.2879	0.0001	0.0000	−3.2877	0.0001	0.0000
10	−3.2878	−3.2879	0.0001	0.0000	−3.2877	0.0001	0.0000

Таблица 3. Первые 10 оценок показателей Ляпунова ($\check{\lambda}_i$ – методом импульсных функций, $\tilde{\lambda}_i$ – методом тригонометрических функций) для уравнения Хатчинсона при $r = 0.5$, а также абсолютная разность σ_i и относительная разность ρ_i между ними и эталонными значениями.
Table 3. The first 10 estimates of Lyapunov exponents ($\check{\lambda}_i$ – by the method of impulse functions, $\tilde{\lambda}_i$ – by the method of trigonometric functions) for the Hutchinson equation at $r = 0.5$ as well as the absolute difference σ_i and the relative difference ρ_i between the obtained and the reference values.

i	τ_i	$\check{\lambda}_i$	$\check{\sigma}_i$	$\check{\rho}_i$	$\tilde{\lambda}_i$	$\tilde{\sigma}_i$	$\tilde{\rho}_i$
$M = 100$							
1	−0.7941	−0.7959	0.0018	0.0023	−0.7957	0.0016	0.0020
2	−0.7941	−0.7961	0.0020	0.0025	−0.7957	0.0016	0.0020
3	−2.7721	−2.7770	0.0049	0.0018	−2.7766	0.0045	0.0016
4	−2.7721	−2.7770	0.0049	0.0018	−2.7767	0.0046	0.0017
5	−3.3533	−3.3591	0.0058	0.0017	−3.3588	0.0055	0.0016
6	−3.3533	−3.3591	0.0058	0.0017	−3.3589	0.0056	0.0017
7	−3.7173	−3.7237	0.0064	0.0017	−3.7237	0.0064	0.0017
8	−3.7173	−3.7237	0.0064	0.0017	−3.7237	0.0064	0.0017
9	−3.9835	−3.9906	0.0071	0.0018	−3.9914	0.0079	0.0020
10	−3.9835	−3.9907	0.0072	0.0018	−3.9914	0.0079	0.0020
$M = 1000$							
1	−0.7941	−0.7942	0.0001	0.0001	−0.7939	0.0002	0.0003
2	−0.7941	−0.7944	0.0003	0.0004	−0.7940	0.0001	0.0001
3	−2.7721	−2.7726	0.0005	0.0002	−2.7722	0.0001	0.0000
4	−2.7721	−2.7726	0.0005	0.0002	−2.7723	0.0002	0.0001
5	−3.3533	−3.3539	0.0006	0.0002	−3.3536	0.0003	0.0001
6	−3.3533	−3.3539	0.0006	0.0002	−3.3537	0.0004	0.0001
7	−3.7173	−3.7178	0.0005	0.0001	−3.7177	0.0004	0.0001
8	−3.7173	−3.7178	0.0005	0.0001	−3.7177	0.0004	0.0001
9	−3.9835	−3.9840	0.0005	0.0001	−3.9839	0.0004	0.0001
10	−3.9835	−3.9840	0.0005	0.0001	−3.9839	0.0004	0.0001
$M = 2000$							
1	−0.7941	−0.7941	0.0000	0.0000	−0.7937	0.0004	0.0005
2	−0.7941	−0.7943	0.0002	0.0003	−0.7939	0.0002	0.0003
3	−2.7721	−2.7724	0.0003	0.0001	−2.7719	0.0002	0.0001
4	−2.7721	−2.7724	0.0003	0.0001	−2.7720	0.0001	0.0000
5	−3.3533	−3.3536	0.0003	0.0001	−3.3533	0.0000	0.0000
6	−3.3533	−3.3536	0.0003	0.0001	−3.3533	0.0000	0.0000
7	−3.7173	−3.7175	0.0002	0.0001	−3.7173	0.0000	0.0000
8	−3.7173	−3.7175	0.0002	0.0001	−3.7173	0.0000	0.0000
9	−3.9835	−3.9836	0.0001	0.0000	−3.9835	0.0000	0.0000
10	−3.9835	−3.9837	0.0002	0.0001	−3.9835	0.0000	0.0000

Вычисленные оценки ляпуновских показателей, абсолютная и относительная разности для удобства округлены до четвертого знака после запятой.

Как видно из таблиц 1–3, точность вычисления показателей зависит от величины выбранного разбиения. При увеличении M в 10 раз со 100 до 1000 относительная погрешность уменьшилась на порядок. В крайнем случае, когда количество точек разбиения равно количеству вычисляемых показателей Ляпунова, достигаемая точность мала. Отметим, что в некоторых случаях, например, для режимов модели из [13], метод тригонометрических функций может оказаться неприменим ввиду потери информации на этапе применения дискретного преобразования Фурье.

Заключение

Таким образом, проведенные численные эксперименты показывают, что при выборе достаточного количества точек разбиения M оцениваемые характеристики могут оказаться качественно близкими к показателям Ляпунова. В случае небольшой размерности системы (1) выбор методики влияет на скорость расчетов в зависимости от характера исследуемого решения. В частности, если решение является относительно сглаженным, то метод тригонометрических функций работает быстрее метода импульсных функций. Если же решение содержит большое количество участков с достаточно острыми пиками, то метод импульсных функций является предпочтительным по скорости выполнения. Кроме того, при увеличении количества вычисляемых показателей Ляпунова K и числа точек разбиения M, а также размерности исследуемой системы N становится эффективным применение многопроцессорных параллельных систем.

Список литературы / References

[1] Купцов П. В., “Вычисление показателей Ляпунова для распределённых систем: преимущества и недостатки различных численных методов”, *Изв. вузов “ПНД”*, **18**:5 (2010), 93–112; [Kuptsov P. V., “Computation of Lyapunov Exponents for Spatially Extended Systems: Advantages and Limitations of Various Numerical Methods”, *Izv. VUZ. Applied Nonlinear Dynamics*, **18**:5 (2010), 93–112, (in Russian).]

[2] Оселедец В. И., “Мультипликативная эргодическая теорема. Характеристические показатели Ляпунова динамических систем”, Тр. ММО, **19**, 1968, 179–210; [Oseledets V. I., “A multiplicative ergodic theorem. Lyapunov characteristic numbers for dynamical systems”, *Trans. Moscow Math. Soc.*, **19** (1968), 197–231.]

[3] Былов Б. Ф., Виноград Р. Э., Гробман Д. М., Немыцкий В. В., *Теория показателей Ляпунова и ее приложения к вопросам устойчивости*, Наука, 1966; [Bylov B. F., Vinograd R. E., Grobman D. M., Nemytskiy V. V., *Teoriya pokazateley Lyapunova i ee prilozheniya k voprosam ustoychivosti*, Nauka, 1966, (in Russian).]

[4] Балякин А. А., Рыскин Н. М., “Особенности расчета спектров показателей Ляпунова в распределенных системах с запаздывающей обратной связью”, *Изв. вузов “ПНД”*, **15**:6 (2007), 3–21; [Balyakin A. A., Ryskin N. M., “Peculiarities of Calculation of the Lyapunov Exponents Set in Distributed Self-Oscillated Systems with Delayed Feedback”, *Izv. VUZ. Applied nonlinear dynamics*, **15**:6 (2007), 3–21, (in Russian).]

[5] Балякин А. А., Блохина Е. В., “Вычисление спектра показателей Ляпунова для распределенных систем радиофизической природы”, *Изв. вузов “ПНД”*, **16**:2 (2008), 87–110; [Balyakin A. A., Blokhina E. V., “Peculiarities of Calculation of the Lyapunov

Exponents Set in Distributed Self-Oscillated Systems with Delayed Feedback", *Izv. VUZ. Applied Nonlinear Dynamics*, **16**:2 (2008), 87–110, (in Russian).]

[6] Колоскова А. Д., Москаленко О. И., Короновский А. А., "Метод расчета спектра показателей Ляпунова для систем с запаздыванием", *Письма в ЖТФ*, **44**:9 (2018), 19–25; [Koloskova A. D., Moskalenko O. I., Koronovskii A. A., "A Method for Calculating the Spectrum of Lyapunov Exponents for Delay Systems", *Technical Physics Letters*, **44**:5 (2018), 374–377.]

[7] Алешин С. В., "Оценка инвариантных числовых показателей аттракторов систем дифференциальных уравнений с запаздыванием", *Вычислит. технологии в естеств. науках: методы суперкомп. моделир.*, 2014, 10–17; [Aleshin S. V., "The Numerical Evaluation of Attractors Exponents of Delay Differential Equations System", *Comp. Technologies in Sciences. Methods of Simul. on Supercomputers*, 2014, 10–17, (in Russian).]

[8] Farmer J. D., "Chaotic Attractors of an Infinite-Dimensional Dynamical System", *Physica D: Nonlinear Phenomena*, **4**:3 (1982), 366–393.

[9] Hairer E., Nørsett S. P., Wanner G., *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer-Verlag Berlin Heidelberg, 2008.

[10] Cheney W., Kincaid D., *Linear Algebra: Theory and Applications*, Sudbury, Mass: Jones and Bartlett Publishers, 2009.

[11] Нуссбаумер Г., *Быстрое преобразование Фурье и алгоритмы вычисления сверток*, Радио и связь, 1985; [Nussbaumer H. J., *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag Berlin Heidelberg, 1981.]

[12] Глызин Д. С., Глызин С. Д., Колесов А. Ю., Розов Н. Х., "Метод динамической перенормировки для нахождения максимального ляпуновского показателя хаотического аттрактора", *Дифференц. уравнения*, **41**:2 (2005), 268–273; Glyzin D. S., Glyzin S. D., Kolesov A. Yu., Rozov N. Kh., "The Dynamic Renormalization Method for Finding the Maximum Lyapunov Exponent of a Chaotic Attractor", *Differ. Equ.*, **41**:2 (2005), 284–289.

[13] Aleshin S. V., Glyzin D. S., Glyzin S. D., Goryunov V. E., "Estimation of Lyapunov Exponents for Quasi-Stable Attractors of Dynamical Systems with Time Delay.", *Journal of Physics: Conference Series*, **1163** (2019), 012045.

[14] Kuptsov P. V., Kuznetsov S. P., "Violation of Hyperbolicity in a Diffusive Medium with Local Hyperbolic Attractor", *Phys. Rev. E.*, **80** (2009), 01620513.

[15] Hutchinson G. E., "Circular Causal Systems in Ecology", *Ann. N.Y. Acad. Sci.*, **50** (1948), 221–246.

[16] Hale J., *Theory of Functional Differential Equations*, Springer-Verlag New York, 1977.

[17] Wright E. M., "A Non-Linear Difference-Differential Equation", *J. Reine Angew. Math.*, **194** (1955), 66–87.

[18] Kakutani S., Markus L., "On the Nonlinear Difference-Differential Equation $y'(t) = (A - By(t-\tau))y(t)$", *Contributions to the Theory of Nonlinear Oscillations*, **4** (1958), 1–18.

[19] Кащенко С. А., "К вопросу об оценке в пространстве параметров области глобальной устойчивости уравнения Хатчинсона", Нелинейные колебания в задачах экологии. Ярославль: ЯрГУ, 1985, 55–62; [Kaschenko S. A., "K voprosu ob otsenke v prostranstve parametrov oblasti global'noy ustoychivosti uravneniya Khatchinsona", *Nelineynye kolebaniya v zadachakh ekologii. Yaroslavl: YarGU*, 1985, 55–62, (in Russian).]

[20] Кащенко С. А., "Асимптотика решений обобщённого уравнения Хатчинсона", *Модел. и анализ информ. систем*, **19**:3 (2012), 32–62; [Kaschenko S. A., "Asymptotics of Solutions of the Generalized Hutchinson's Equation", *Modeling and Analysis of Information Systems*, **19**:3 (2012), 32–62, (in Russian).]

Goryunov V. E., "Features of the Computational Implementation of the Algorithm for Estimating the Lyapunov Exponents of Systems with Delay", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 572–582.

DOI: 10.18255/1818-1015-2019-4-572-582

Abstract. We consider the computational implementation of the algorithm for Lyapunov exponents spectrum numerical estimation for delay differential equations. It is known that for such systems, as well as for boundary value problems, it is not possible to prove the well-known Oseledets theorem which allows us to calculate the required parameters very efficiently. Therefore, we can only talk about the estimates of the characteristics in some sense close to the Lyapunov exponents. In this paper, we propose two methods of linearized systems solutions processing. One of them is based on a set of impulse functions, and the other is based on a set of trigonometric functions. We show the usage flexibility of these algorithms in the case of quasi-stable structures when several Lyapunov exponents are close to zero. The developed methods are tested on a logistic equation with a delay, and these tests illustrate the "proximity" of the obtained numerical characteristics and Lyapunov exponents.

Keywords: Lyapunov exponents spectrum, dynamical system with delay, numerical algorithm, Hutchinson equation

On the authors:
Vladimir E. Goryunov, orcid.org/0000-0002-0512-6986, postgraduate student,
P. G. Demidov Yaroslavl State University,
Sovetskaya str., 14, Yaroslavl, 150003, Russia, e-mail: salkar@ya.ru

Acknowledgments:
The reported study was funded by RFBR according to the research project № 18-29-10055.