УДК 004.43

Парадигмальный подход к факторизации определений языков и систем программирования

Городняя Л.В. (Институт систем информатики СО РАН,

Новосибирский государственный университет)

Статья посвящена проблеме факторизации определений языков и систем программирования. В качестве основного параметра факторизации выбрана семантическая декомпозиция в рамках анализа парадигм программирования. Такой выбор позволяет выделять автономно развиваемые типовые компоненты систем программирования. Типовые компоненты должны быть приспособлены к конструированию различных информационных систем. Кроме того, их существование позволяет формировать методику обучения разработке компонентов информационных систем. Попутно показана дистанция в понятийной сложности между программированием и разработкой систем программирования.

Ключевые слова: *определение языков программирования, факторизации определений, декомпозиция программ, критерии декомпозиции, парадигмы программирования, семантические системы, разработка программ, методы обучения программированию.*

1. Введение

Многие работы по методам разработки программных систем зависят от практичности подходов к декомпозиции программ, что можно рассматривать как проблему факторизации программ и средств их представления на базе языков программирования (ЯП), отлаживаемых с помощью систем программирования (СП). Общее понятие факторизации основано на декомпозиции сложных объектов в произведение более простых объектов, из которых перемножение даёт исходный объект при условии, что выбор более простых объектов обусловлен определённым фактором, позволяющим предельно простые объекты отличать от более сложных. Даже при факторизации чисел эта задача, имеющая строгое определение, становится сложной при переходе к большим числам. Такое понятие достаточно естественно может быть перенесено на представления программ, определений ЯП и реализаций СП при условии определения факторов для выделения простых объектов и определения техники произведения для восстановления исходного сложного объекта из результата факторизации.

Если техника произведения может быть сведена к общепринятым методам сборки программ из типовых компонентов, то выбор факторов для выделения простых составляющих в случае программ обладает значительным разнооборазием. Не удивительно, что на весьма представительных конференциях, посвященных обсуждению всех проблем программирования, рассматриваются лишь отдельные штрихи проблемы факторизации программ на материале конкретных наиболее популярных ЯП, таких как Си, Java, Python [7].

В данной статье рассматривается парадигмальный подход к выбору факторов и аналогов произведения для факторизуемых представлений программ, определений ЯП и реализаций СП, использующий семантическую декомпозицию формализованных определений. Проблема факторизации в программировании осложнена разнообразием используемых средств и широким спектром противоречащих друг другу критериев качества программ. Основная цель декомпозиции программ — обеспечение многократности использования отлаженных фрагментов. Именно многократность использования является аргументом доверия программным системам. Конструирование систем из отлаженных компонентов не всегда обладает удобной комбинаторикой. Возможность улучшения конструктива без чрезмерного роста трудозатрат на повторное программирование и отладку обусловлено выбором критериев декомпозиции программ. Нередко критерии декомпозиции отражают структуру декомпозируемого текста программы. Чаще имеет место учёт особенностей квалификации разработчиков программы. При решении сложных задач критерии декомпозиции программ отражают актуальность подзадач по шагам разработки. Долгоживущие программы эксплуатируются дольше времени её авторского сопровождения.

Системы программирования обычно проектируются именно как долгоживущие программы. Это достаточная причина высокой вероятности неавторского улучшения СП. Всё это объясняет, почему нужны объективные критерии декомпозиции ЯП, отражающие возможность автономного развития выделенных компонентов СП. К этой проблеме примыкает слабость методики обучения разработке компонентов программ и программных комплексов. Для разработки СП такая методика усложнена необходимостью полноценного ознакомления с основными и фундаментальными парадигмами программирования (ПП), включая параллельные вычисление [2]. Основные парадигмы имеют производственное значение. Они поддерживают жизненный цикл разработки программ. Фундаментальные парадигмы имеют образовательное значение. Они обеспечивают формирование расширяемой системы понятий, необходимых для освоения экстенсивно развивающегося пространства задач, решаемых с помощью ИТ.

Описание парадигмального подхода к факторизации программ начинается с общего представления о семантических системах, семантике языков разного уровня, особенностях СП и учебного программирования, дающего основания для относительного определения языков и систем программирования (ЯСП) с учётом прагматики и методики парадигмального анализа ЯП. В заключении приведены основные идеи инструментальной поддержки парадигмального подхода к факторизации ЯСП.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 18-07-01048-а.

2. Общее представление

Достаточно объективной основой факторизации СП является семантическая декомпозиция определения ЯП. Результаты такой декомпозиции можно формулировать на базе предложенного С.С. Лавровым понятия "семантическая система". Это понятие расширяет понятие «алгебраическая система» заданием явного правила вычислений [4]. Для точности приведены пояснения к используемым общеизвестным терминам, необходимым для чёткого описания рассматриваемых методов декомпозиции программ. Эти методы в перспективе приводят к парадигмальному анализу определений ЯСП.

Семантическая система — это <**V, F, R**>, где:

V - основное множество значений,

F - конечный набор функций, возможно принадлежащих основному множеству,

R - правило применения функций к значениям, возможно входящее в набор функций,

Семантически подобные системы в разных ЯП могут обладать разными особенностями реализационной прагматики (РП). Прежде всего это относится к различию систем по схемам пре- или пост-вычислений. Далее реализация ЯП может тяготеть к интерпретации или компиляции программ. При переходе к СП могут различаться методы реализации функций техникой макроподстановки или вызова подпрограмм. Список других штрихов реализационных различий можно продолжать. Это означает существование поливариантности компонентов РП, поддерживающих разные парадигмы программирования. Такие компоненты пригодны для автономного развития.

Практичные критерии декомпозиции, кроме того, требуют учёта и представления специфики определений типовых компонентов в ясной форме, допускающей речевую практику. Помимо формальной определённости класса семантических систем нужна лингвистическая ясность, дающая достаточные основания для лаконичного объяснения

особенностей каждой семантической системы, что можно соотнести с отмеченной в 1970-е А.П. Ершовым проблемой Лексикона программирования, осложнённой свойством естественных языков изменять словарь при смене поколений. Кроме того, выбор основных семантических систем следует обосновывать прецедентом аппаратной реализации, показывающим достижимость эффективной реализации.

Обычно для любого множества значений **V** реализационно различимы следующие виды функций семантических систем:

- методы вычислений (**V*** → **V+**),
- средства доступа к памяти (**T : A** → **V**),
- особенности управления вычислениями {**F** → **{0,1}**},
- обратимая комплексация данных (**E** ↔ **S**).

Таким видам соответствуют разные правила **R**, определяющие классы семантических систем, особенности реализации которых описаны ниже, в таблицах 1-5. Пояснения к словам, размещённым в клетках таблиц, не являются определениями терминов, слова выполняют лишь роль индексов для размещения в соответствующих клетках неформализованных понятий и средств ЯП, выделяемых при парадигмальном анализе.

3. Базовая семантика

Рассмотрим более детально основные виды эффективно реализуемых функций.

(**V*** → **V+**) — методы вычислений. Функции таких семантических систем отображают некоторое число однотипных значений, возможно ни одного, в одно или несколько значений этого же типа. В некоторых ЯП такому классу может принадлежать до десятка семантических систем над разными типами значений. Соответствующее правило **R** обычно заключается в предварительном вычислении операндов **V*** с последующей выдачей **V+**.

(**T : A** → **V**) — средства доступа к памяти. Этот класс семантических систем характеризуется введением множества адресов **A** и специальной таблицы **T.** Соответствующее правило **R** позволяет по определённой (возможно неявно) таблице **T** поддерживать доступ к значениям. Существуют ЯП, в которых такая таблица допускает расширение и полиморфность РП. Обычно правило **R** предполагает предварительное вычисление операнда **V** при заданном **A**, хотя бывает, что и **A** можно вычислять.

{**F** → **{0,1}**} — особенности управления вычислениями. Этот класс семантических систем требует выделения одного или двух значений-переключателей **{0|1}.** Переключатели нужны для выбора ветви вычислений или обхода отдельных действий. В некоторых ЯП такие

значения реализованы как отдельный тип данных. Соответствующее правило **R** может использовать переключатели разного происхождения. Чаще всего - из ранее выполненных вычислений. В таком случае ход вычислений может зависеть от промежуточных результатов. Бывает и учёт независимых регистров. При выполнении функций **F** могут меняться значения переключателей.

(**E** ↔ **S**) — обратимая комплексация данных. Этот класс семантических систем разделяет множество значений **V** на элементарные (**E**) и составные (**S**). Во многих ЯП используется несколько таких семантических систем - разные структуры данных. Различие структур данных поддерживает разные дисциплины доступа к хранимым в памяти значениям. В некоторых ЯП допускается программирование структур данных. Чаще встречается конструирование из встроенных наиболее удобно реализуемых шаблонов. Соответствующее правило **R** должно быть подчинено определённой аксиоматике, гарантирующей наличие обратных функций в процессе комплексации. Встречаются иногда отсутствие обратных функций, обычно мотивируемое соображениями эффективности.

На уровне базовой семантики ЯП существенные различия видов функций семантических систем можно представить Таблицей 1.

Таблица 1

Семантическая декомпозиция минимального ядра ЯП

Уровень\Класс	***V***	***V* → V+***	***T : A → V***	***{F → {0,1}}***	***E ↔ S***
Ядро	Значение	Операции	Память	Управление	Вектор

Ядро — семантический базис. Позволяет полное определение ЯП получать как консервативное расширение ядра. Обычно ядро приспособлено и к неконсервативному расширению пополнением набора библиотечных функций, реализуемых на уровне аппаратуры. Это позволяет реализации СП для любого ЯП поддерживать разные парадигмы программирования, необходимые для поддержки полного жизненного цикла программ чтобы достигать эффективности независимо от исходных возможностей ЯП.

Значение — минимальное представление объектов из области приложения языка.

Операции — минимальный комплект для обработки значений.

Память — введение адресов для лаконичного и уникального представления значений (указатели, идентификаторы, переменные, метки).

Управление — разметка выполнимости последовательности элементов программы из (операций, функций, действий и т.п.) специально выбранными значениями, например, **{0|1}** или **{True | False}** или **{Nil | T}**.

Вектор — обратимое конструирование одноуровневых комплектов, рассматриваемых как целостность, из которых можно восстанавливать исходные элементы. На уровне ядра достаточно одной структуры — вектора, списка, очереди или т. п.

Различия классов семантических систем на уровне правил применения функций к значениям выражены таблицами 2-5.

Таблица 2

Семантическая декомпозиция макрорасширения ядра ЯП

Уровень\ Класс	***V***	***V* → V+***	***T : A → V***	***{F → {0,1}}***	***E ↔ S***
Ядро	Значение	Операции	Память	Управление	Вектор
Макро	Данное	Функции	Задание	Блоки	Стек

Макро — пополнение ядра средствами обработки представлений выражений, используемых с целью укрупнения любых конструкций. Укрупнение поддерживает консервативное расширение ЯП. Кроме того, оно способствует лаконизму текстов программ. Макротехника позволяет наследовать отлаженность компонентов программ. Бывает важным исключать дубли фрагментов текста, кода и структур данных. Простейший механизм макрогенерации обычно присутствует в СП как препроцессор. Так может быть устроена техника кодогенерации и обработки шаблонов при компиляции программ. Реализация укрупнений может быть функционально эквивалентна вызову подпрограмм. Взаимозаменяемость макроподстановки и вызова подпрограмм нередко используется при оптимизации программ.

Данное — хранимое значение или выражение, допускающее уникальность экземпляра, доступного по адресу, возможно, на внешнем устройстве.

Функции — укрупнение операций с возможной параметризацией операндов. Реализационная прагматика может отличаться техникой передачи параметров через стек или специальное поле аргументов или неявно. Последнее позволяет обработку памяти формально рассматривать как функцию с неявным аргументом, выполняющим роль таблицы соответствия адресов и значений.

Задание — хранимое именованное выражение с возможностью многократного выполнения.

Блоки — хранимое выражение или код программы, представляющий составные действия, ветвления, циклы, вызовы функций, обычно с локализацией переменных.

Стек — схема организации данных, с определённой дисциплиной доступа для поддержки иерархии, возможно с защитой независимых блоков.

4. Семантика языков высокого уровня

Повышение уровня ЯП обеспечивается средствами укрупнения данных. Это приводит к понятию «иерархия» и оперированию блоками программы. Дальнейшее наращивание объёмов разрабатываемых программ отчасти достигается автоматизацией контроля некоторых условий корректности применения операций и функций к их операндам. Становятся важными понятия «предикат» и «тип переменных», удобно проверяемые при компиляции.

Таблица 3

Семантическая декомпозиция диагностического дополнения ядра ЯП

Уровень\ Класс	***V***	***V* → V+***	***T : A → V***	***{F → {0,1}}***	***E ↔ S***
Ядро	Значение	Операции	Память	Управление	Вектор
Макро	Данное	Функции	Задание	Блоки	Стек
Границы	Исключения	Предикаты	Типы переменных	Проверка логики	Вариант

Границы — методы проверки вычислимости выражений и выполнимости программ. Цель задания границ — снижение трудоёмкости отладки программ упрощением обнаружения ошибок. При отсутствии ошибок проверка воспринимается как накладные расходы. Встречаются механизмы установки ловушек на непредусмотренные ситуации и программирования обработки прерываний с возможностью продолжения вычислений.

Исключения — выбор специальных значений для разметки ненужных ситуаций. В некоторых ЯП вводят значения типа “Error”. При переходе к СП происходит добавление текстовых шаблонов для формирования диагностических сообщений об исключительных ситуациях.

Предикаты — специальные функции, позволяющие определять типы значений или сравнивать значения независимо от расположения данных в памяти. Роль предиката может выполнять любая функция при подходящих договорённостях и схеме реагирования на её результаты.

Типы переменных — связывание типа значения с переменной, хранящей его в памяти.

Проверка логики — соответствие типа данных или значений операциям обработки значений или доступа к памяти, возможно с учётом условий вычислимости.

Вариант — схема организации данных без определённой дисциплины доступа для организации перебора равноправных элементов или блоков. Необходимо как механизм удостоверения принципиальной выполнимости вычислений при частичной постановке задачи.

Таблица 4

Семантическая декомпозиция практичного обобщения ядра ЯП

Уровень\Класс	***V***	***V* → V+***	***T : A → V***	***{F → {0,1}}***	***E ↔ S***
Ядро	Значение	Операции	Память	Управление	Вектор
Макро	Данное	Функции	Задание	Блоки	Стек
Границы	Исключения	Предикаты	Типы переменных	Проверка логики	Вариант
Общность	Неопределённость	Мульти	Устройства	Отображение	Ввод-вывод

Общность — дополнительные средства обеспечения отладки и применении программ, поддерживающие возможность разумного продолжения вычислений при любых исходных данных и аварийных ситуациях.

Неопределённость — вводятся специальные дополнительные значения и ловушки (_|_, Error, Future). Такое расширение множества значений позволяет учитывать в текстах программ некоторые отдельные особенности процесса разработки, отладки и схемы жизненного цикла программ.

Мульти — допускается произвольное число операндов операций и аргументов функций. Возможно просачивание определений на однородные структуры данных, позволяющее определения над элементарными данными автоматически распространять на более сложные данные.

Устройства — механизм неявного расширения области действия операций и функций на устройства ввода-вывода, рассматриваемые как обобщение памяти.

Отображение — возможность регулярного применения функции к серии данных благодаря использованию представлений функций в качестве аргументов функций более высокого порядка.

Ввод-вывод — средства приёма данных с внешних устройств и размещения данных на внешних носителях данных, включая средства доступа к устройствам с уровня программы. Обычно подразумевается аксиоматика, требующая совместимости форматов ввода-вывода: для всякого вводимого данного существует эквивалентное ему выводимое данное и обратно — если данное может быть выведено, то его можно ввести без потерь.

5. Языки сверх высокого уровня

Существуют задачи с повышенными требованиями к качеству программ их решения. В частности, могут быть многочисленные важные ограничения на время и ресурсы, используемые при эксплуатации программ. Всё это обуславливает ряд трудно формализуемых уровней, здесь условно названных термином «надёжность». Часть решений на таких уровнях могут быть представлены средствами ЯП, другие поддерживаются

системой программирования или средой разработки программ, возможно допускающей использование разных ЯП, включая совместное применение библиотек.

Таблица 5

Семантическая декомпозиция высокопроизводительного расширения ЯП.

Уровень\Класс	***V***	***V* → V+***	***T : A → V***	***{F → {0,1}}***	***E ↔ S***
Ядро	Значение	Операции	Память	Управление	Вектор
Макро	Данное	Функции	Задание	Блоки	Стек
Границы	Исключения	Предикаты	Типы переменных	Проверка логики	Вариант
Общность	Неопределённость	Мульти	Устройства	Отображение	Ввод-вывод
Надёжность	Параметры обстановки и сети устройств	Отношения	Передача данных	Дисциплина обслуживания	Комплексы

Надёжность — дополнительные средства обеспечения качества программ при отладке и применении программ. Часть требований к качеству программ связано с применением программ на сетевом и многопроцессорном оборудовании. Чаще всего это поддержка параллельных вычислений, что весьма по разному влияет на особенности правил применения функций к значениям.

Параметры обстановки и сети устройств — вводятся специальные дополнительные значения и формы для представления конфигураций устройств. Такое расширение множества значений позволяет учитывать некоторые отдельные особенности процесса разработки отлаживаемых программ и практики их применения. Кроме того, появляется возможность учёта в программах динамики распределённых информационных систем.

Отношения — декларативное объявление контролируемых зависимостей. Существуют ЯП, защищающие сохранение таких объявленных условий.

Передача данных — механизм неявного расширения области действия операций над памятью на периферийные устройства. Такой механизм необходим для ЯП, нацеленных на поддержку параллельных вычислений и создание распределённых информационных систем. Обычно возникают средства работы с копиями данных, их репликации и клонирования.

Дисциплина обслуживания — возникают механизмы формирования пространств итераций и программирования методов доступа к разнородной памяти. Возможно запараллеливание тела цикла или витков рекурсивной функции на разные устройства - процессоры. Такие пространства поддерживают многопоточность или многопроцессорность в случае слабых зависимостей между соседними итерациями. При оптимизации программ возможно сведение рекурсии к циклу, а для верификации часто требуется обратное.

Комплексы — структура данных, допускающая разметку и кратность вхождения составляющих. Такая структура используется как средство повышения эксплуатационных

характеристик программы. Комплексы позволяют представлять тиражирование готового или подобного конструктива без избыточного копирования.

6. Переход к системам программирования

Реализация каждой из семантических систем, выделенных из определения ЯП, может быть выполнена разными методами. От методов требуется сохранение достаточной автономии систем и их соответствие спецификации. Спецификация в свою очередь должна быть согласована с правилом R, определяющим класс семантической системы. Кроме того, значения при разработке СП обретают поливариантность используемых представлений. По меньшей мере используются одновременно тексты, структуры и коды с полным спектром переходов от одного представления к другому. В результате удаётся смягчать избыточную сложность отладки программ ценой повышения трудоёмкости разработки СП. Возможна верифицируемая комбинаторика семантических систем по мере улучшении СП. При создании СП число реализуемых промежуточных семантических систем резко возрастает. Это зависит от сложности задач системной поддержки внутренних форматов данных. Дополнительные сложности вносит оптимизация программ. Ещё сложнее решение проблем производительности вычислений. Для долгоживущих программ свой вклад даёт и факторизации компонентов, приспособленных к многократному применению в разных СП. Возникает ряд дополнительных уровней, характеризуемых своими наборами понятий:

Окружение — операционная система, поддерживающая реальные процессы, оперируя понятиями Объекты, Процессор и Устройства, Действия, Файлы, Процесс, Конфигурация.

Реализация — процесс или результат создания исполнимого кода программы, оперируя понятиями Шаблоны кода, Генерация программы, Атрибуты, Линейные участки (SSA-формы[1]), Размеченное множество.

Оптимизация — приведение программы к форме, обладающей заданными свойствами, оперируя понятиями Критерии, Преобразования, Эквиваленты, Маршрут, Орграф.

Факторизация — приведение программы к декомпозированной форме, части которой обладают определённой автономностью при улучшении, исполнении или применении программ. Оперирует понятиями Комплекты, Процессоры, Библиотеки, Потоки, Сеть.

Производительность — приведение программы к форме, дающей более высокую скорость исполнения. Оперирует понятиями Время, Синхронизация, Сложное действие, Барьеры, Синхросеть.

---

1 SSA-формы - линейные участки с однократным присваиванием. Играют важную роль в методах оптимизации программ.

Каждый такой уровень включает в себя полный образ реализуемого ЯП. Это пропорционально увеличивает понятийную сложность разработки СП. Сравнение систем понятий уровня ЯП и СП показывает значительное наращивание дистанции между программированием и системным программированием, что более подробно рассмотрено в [2].

7. Классификатор парадигм и стилей программирования

Как правило под парадигмами понимают особенности мышления, характерные для выбора способа решения некоторого класса задач. О парадигмах программирования начали говорить с середины 1970-ых годов при осознании кризиса технологии программирования. Вскоре внимание от этой темы отвлеклось на задачи освоения новой элементной базы. Понятие «парадигма программирования» пока не имеет строгого определения, хотя термин стал модным и активно используется при объявлении новых способов работы с программами.

Возникает вопрос: в какой мере анонсы различных ПП реально отражают разные особенности мышления? Далее: как оценить, насколько новая парадигма отличается от уже имеющихся? Ответы на эти вопросы выведены из результатов анализа и сравнения наиболее популярных основных и фундаментальных парадигм программирования, поддержанных в большом числе ЯП. Рассмотрены особенности и многих производных парадигм [10,11], сводимых к композиции основных и фундаментальных. Кроме того, показана зависимость успешного применения ПП от фазы интервальной схемы процесса разработки программ. Схема успешной разработки отражает уровень изученности решаемых задач. Практичность результата разработки обуславливается прагматикой традиционных решений реализации базовой семантики. Эффективность таких решений зависит от подразумеваемых аппаратных средств.

Программирование решения любой непростой задачи происходит в определённой схеме поддержки жизненного цикла программ. Эксплуатация программы нередко ориентирована на развитие широкого спектра аппаратуры и расширение категорий пользователей. По этой причине многие долгоживущие и новые ЯП поддерживают сразу несколько различных парадигм. Мультипарадигмальность может быть представлена разными пропорциями в поддержке отдельных парадигм. Для определения таких пропорций нужна парадигмальная декомпозиция определений ЯСП. Прежде всего пропорции связаны с оценкой уровня изученности решаемых задач. К оценке примыкают требования ограничения новизны в программистских проектах для прогнозирования времени жизни проекта. Эти два параметра позволяют корректно обосновывать выбор средств и методов разработки программ, включая

выбор ЯСП. Материал для такого обоснования дают известные прагматические аспекты истории ЯСП. Они представлены фактами развития парадигм программирования.

Особое место среди парадигм занимает функциональное программирование (ФП). Прежде всего с этой парадигмой связан лаконичный стиль представления программ. Процесс ФП нацелен на предельно общие методы решения задач. Общность даёт представление универсальных функций, допускающих полный контроль правильности вычислений.

В результате формируется потенциал развиваемых средств информационной обработки. При пошаговой технологии разработки программ ФП может использоваться как инструмент прототипирования других парадигм. Это образует основу функционального подхода к описанию парадигм программирования и ЯСП. Реально ФП способно выполнять роль парадигмального моделирования новых информационных систем. Это образует полноценный полигон для языкотворчества. Первое применение такого полигона находится в области проблемно-ориентированных ЯСП. Основы ФП исторически привлекают внимание специалистов при каждой смене элементной базы. Другая линия - неожиданное расширение сферы применения ИТ.

Очередной этап надежд на ФП связан с актуальностью проблем параллельных вычислений (ПВ). Многие попытки классификации парадигм программирования не рискуют признать ПВ самостоятельной парадигмой. Чаще встречается включение ПВ в общеизвестные парадигмы в виде расширений, дополнительных библиотек, указаний компилятору или специальных механизмов. Обычно встраиваются в ЯП отдельные средства без претензий на общее решение проблем ПВ. Многообразие моделей взаимодействующих процессов ждёт своего решения уже более 70-ти лет. Ряд проблем проявилось ещё в докомпьютерную эпоху. Всё это говорит о сложности проблемы. Не исключено, что решение проблем ПВ слишком чувствительно к методам обучения, зависит от образовательных механизмов. Это повышает требования к определению парадигмы параллельного программирования и поддерживающих её ЯСП. Предстоит дальнейшее исследование парадигм параллельного программирования. Важно достичь возможности привлечения разных моделей взаимодействия программируемых процессов. Нужна опора на языки лаконичного представления программ. Растёт актуальность создания СП, обеспечивающих не только тестирование, но и расширяемый спектр средств отладки и верификации программ. Это может дать техническую основу для подготовки производительных программных систем.

В этом плане поучителен опыт разработки и развития долгоживущих ЯСП. Характерно внимание проблемам отладки и расширения программ. Долгоживущи СП необходимо обладают мультипарадигмальностью даже для изночально монопарадигмальных ЯП (Haskell,

F#). Такое расширение позволяет успешно применять СП на протяжении полного жизненный цикла программ. В этом плане жизненный цикл программ для ПВ обладает большей длительностью, выходящей за пределы основных ПП.

Обычно парадигмальные различия проявляются на всех уровнях определения ЯСП (лексика, синтаксис, семантика, прагматика). Особенно чётко видна парадигмальная характеристика на уровне прагматики. Именно эта разница служит основанием для классификации парадигм программирования и отделения их от подходов, стилей, методов и т. п.

8. Представление относительной семантики ЯСП

Подход к сравнению и классификации ЯСП осложнён возрастанием объёмов их описаний и сложности реализаций. Первое формальное описание языка Pure Lisp занимало примерно полторы страницы. На них было представлено около десяти понятий. Для их освоения было достаточно около 250-ти упражнений. В сравнении с этим, описание нового стандарта C++ занимает примерно 1300 страниц. Формальное представление грамматики – 32 страницы. Используется определение 280-ти понятий. Для отладки компилятора C++ предлагается более 6000 тестов.

Реализация Lisp-интерпретатора поддерживала встроенные средства компиляции программируемых функций. Это обеспечивало гладкий переход от отладки программ к их эффективному применению. Некоторые диалекты Lisp-a и сферы их приложения выполнили роль прототипов для создания новых ИТ не только в области искусственного интеллекта. Язык использовался как средство проектирования, прототипирования, символьной обработки, исследования методов оптимизации и преобразования программ, гипертекстов и многого другого. Первые реализации языка Lisp в СССР показали высокую моделирующую силу ФП. Она оказалась достаточной для решения сложных задач в области верификации программ. На языке Lisp были проведены исследования доказательных построений (и проводятся в настоящее время [12]) и решения задач химии, биологии и языкознания. До сих пор многие из таких задач рассматриваются как весьма сложные. Дальнейшее усложнение многих классов задач связано с переходом к большеобъёмным массивам. Доминирование парадигмы императивного программирования успешно поддержало эффективные решения хорошо изученных задач по ранее отлаженным алгоритмам. При переходе к большеобъёмным массивам эта парадигма слабо приспособлена к разработке новых методов, требующих заметного объёма отладки на фоне распределённых систем. Это провоцирует массовое языкотворчество в сфере новых и экспериментальных постановок задач. Такие

задачи неустранимо обладают исследовательским компонентом, требующим более мощных и более общих парадигм.

Создание новых проблемно-ориентированных ЯП выполняет работу по компенсации недостаточности основных ПП для новых задач. Такую работу пытались и до сих пор пытаются выполнить на базе парадигмы объектно-ориентированного программирования. Большие надежды возлагались на разработку и стандартизацию проблемно-ориентированных интерфейсов. Приходится констатировать, что формально полезные абстракции на практике редко получают удачное конкретное наполнение и применение. Наследование отлаженных программистских решений обычно происходит на уровне библиотечных модулей или инструментов в СП. Строгой формализации чаще предпочитают успешное применение редактируемых образцов программ и многократно используемых шаблонов языковых конструкций. Это позволяет определения новых ЯП формулировать относительно известных ЯСП. Для этого нужны средства определений в стиле описания границ сходства и различия. Такие границы можно описывать в форме сопоставления парадигмальных моделей. Парадигмальные модели позволяют таким образом выделять автономно развиваемые модули. Взаимонезависимость парадигм удобна для синтеза СП из типовых программных компонентов, поддерживающих отдельную парадигму. Модули, соответствующие отдельным семантическим системам, могут иметь разные конкретизации для различных парадигм. Сгруппированные для поддержки разных ПП модули при такой декомпозиции могут обладать взаимозаменяемостью при смене парадигмы. Накопление таких модулей образует технологическую основу для экспериментальной разработки новых ЯП, определения которых должны быть декомпозированы для пошагового определения СП.

Программные инструменты для создания новых ЯСП могут использовать относительное определение семантики со ссылками на известные ЯП. Достаточно констатировать, что вычисления организованы как в языке Pascal, работа с памятью как в языке Lisp, управление вычислениями как в языке C#, структуры данных как в языке APL. Формально это напоминает технику теоретических работ на уровне свободно интерпретируемых схем программ. Плодотворность такого подхода хорошо видна на разнообразии схем, подобных сетям Петри. Чёткая формулировка относительной семантики позволит использовать банк улучшаемых верифицируемых компонентов информационных систем. Учитывая сложность решения вопросов верификации для обеспечения надёжности и безопасности программ, именно такой подход может обеспечить многократность использования результатов, оправдывающую трудозатраты. Ряд методов конструирования учебных систем программирования также может повысить эффективность экспериментального

программирования на базе банка типовых компонентов ЯСП. Возможность апеллировать к известным ЯСП может гарантировать быстрое ознакомление с новыми средствами создаваемых ЯП.

Проблема автоматизации ПВ — одно из важнейших направлений современных исследований в области ИТ. Стремительный прогресс современного оборудования опережает развитие методов организации высокопроизводительной информационной обработки данных. Развитие моделей параллелизма в языках высокого уровня характеризуется значительным разнообразием. Такое разнообразие трудно поддержать в рамках одного ЯП. Поэтому от современной системы ПВ требуется не только лаконизм понятного представления программ. Нужна ещё и конструктивность высокоуровневых определений, допускающих совмещение разноязыковых средств. Кроме того, заметные проблемы связаны с расширяемостью спектра используемых архитектур. В целом, на пути к автоматизации ПВ возникают существенные затруднения. Часть затруднений мало зависит от результатов фундаментальных исследований. Многое обусловлено формированием технологий и человеческим фактором.

Часть трудностей может быть смягчена активизацией применения программируемых методов преобразования программ. Условно это можно назвать суперпрограммированием. Имеются прецеденты работ такого направления. Многое может дать подход на основе трансформационно-операционной семантики языков параллельного программирования. В целом всё это приводит к актуальности задач изменения подходов к организации СП. Нужны информационно-инструментальные средства не только встроенного в компиляторы, но и программируемого анализа и преобразования программ. Такие средства требуют сопровождения и конструирования распределенной измерительной среды для экспериментальных исследований производительности информационных систем. Измерительная среда даёт аргументы для пополнения корпуса отлаженных программ и тестовой базы. К этому примыкает задача конструирования уточняемых функций при разработке программ. Технически это похоже на перегрузку операций в ООП. Полезна возможность уточнение начинать макетированием программ с помощью тестов, спецификаций и описаний, что несколько перекликается с идеями обработки недоопределённой информации.

Анализ парадигмальных особенностей и тенденций развития ЯСП приводит к ряду важных выводов. Прежде всего, возрастание объёма определений ЯП вызвано в значительной мере попытками поддержать в рамках одного языка все средства, необходимые в полном жизненном цикле программ. Рост объёмов определений можно нормализовать

методикой относительных описаний, используя предварительные знания основных ПП. Пропорции между разными парадигмами допускают отладку средствами ФП. Достаточно развиты средства синтаксически управляемого синтеза определений, позволяющие от средств ФП автоматически переходить к любому конкретному синтаксису. Гибкость таких средств может быть повышена техникой сопоставления с образцами и другими методами. Достаточно важно, что решение проблем ПВ требует поддержки своих парадигм, выходящих за пределы удобного применения основных. В любом случае, независимо от ПП, обеспечение надёжности и безопасности ПО требует пересмотра традиционных решений о возможностях СП. Решение таких, трудно формализуемых, проблем связано с образовательными аспектами гуманитарной составляющей информатики. Особого внимания требует зависимость практики программирования от имплицитного научения. Здесь имеет место проявление интуиции, инсайта и неявного знания. Это также даёт предпосылки для пересмотра структуры СП на современном оборудовании и мощных возможностях ИТ.

9. Эксплуатационная прагматика базовых ПП

Рассматривая полный жизненный цикл программ как ряд схем решения задач с помощью определения, разработки и отладки программируемых решений, можно обратить внимание на методы пошаговой или непрерывной разработки экспериментальных и развивающихся программных систем, позволяющие ряд начальных версий создавать техникой макетирования. Само понятие макета допускает постепенное уточнение прототипов программ. Определение функционирования таких уточняемых прототипов можно выполнять методом вертикального слоения программ А.Л. Фуксмана [5,6]. Концептуально этот метод является предшественником аспектно-ориентированного программирования. Отдельные слои могут быть подвергнуты семантической и парадигмальной декомпозиции. Это обеспечивает использование резервов синтаксически управляемого конструирования обработчиков символьной информации [2]. Такие резервы позволяют поддержать непрерывную улучшаемость системы по мере её отладки и внедрения. В процессе улучшения появляется возможность выделять типовые компоненты, пригодные для включения в разные системы. Компоненты систем обладают общими семантическими подсистемами, зависящими от реализационной и эксплуатационной прагматик.

Основные ПП обладают на уровне эксплуатационной прагматики достаточно чёткими различиями. Такие различия как схема программы, формат элементов программы, особенности процесса выполнения программы, граница между правильностью программы и отказом в продолжении выполнения. Для решения задач, обладающих стабильной

постановкой, характерны парадигмы функционального и императивно-процедурного программирования. Решение подверженных развитию задач естественнее выражается в парадигме логического и объектно-ориентированно программирования. При переходе к особо сложным постановкам задач, решение которых требует коллективных усилий и многопроцессорных комплексов, возникает необходимость в парадигмах параллельного программирования. Постановка задачи обуславливает выбор решений в таком трёхмерном парадигмальном пространстве.

Функциональное программирование - программа представляется деревом, вершины которого функции. При обходе дерева функции поочерёдно взаимодействуют с аргументами. Число и типы аргументов должны соответствовать определению функции. Аргументами могут быть фрагменты дерева. Это позволяет организовывать локальное изменение дерева. Результатом программы считается результат последней вычисленной функции. Функция может быть применена к аргументам или дать отказ. В последнем случае обход дерева прерывается и программа признаётся невыполнимой. Программисту при отладке следует изменить программу так, чтобы обход дерева мог быть завершён.

Императивно-процедурное программирование - программа представляется размеченной последовательностью необходимых команд над общей памятью. Порядок выполнения команд подчинён императивной логике управления, задающей последовательность, начинающуюся с известной стартовой метки. Исполнение команды заключается в изменении значения некоторых регистров памяти. Очередная команда исполняется немедленно. Отсутствие учёта и проверки каких-либо соответствий или условий готовности даёт экономию времени исполнения. Результатом программы считаются значения ряда регистров, получившиеся по завершении программы. Неуспешное исполнение команды приводит к передаче управления обработчику прерываний. Обработчик может продолжить выполнение программы. Как правило до исполнения программы, на этапе компиляции, выполняется анализ исполнимости команд. Возможно некоторое изменение их последовательности из соображений оптимизации. Это приводит к определённому расширению пространства допустимых решений, возможно не известных пользователю. Выполнение программы завершает исполнение специальной команды или отсутствие очередной команды. Программист при отладке обычно доводит программу к форме, не требующей обработки прерываний и завершаемой для определённости специальной командой.

Логическое программирование - программа представляется рядом "равноправных" вариантов. Любой вариант может выполниться успешно или дать отказ, влекущий выполнение другого варианта. Программа признаётся невыполнимой только если нет ни

одного успешно выполняющегося варианта. В случае неуспеха программисту при отладке следует добавить варианты, чтобы хотя бы один был успешным. Результатом программы считается результат любого из успешных вариантов или их объединение. Пространство допустимых процессов расширяется выполнением неуспешных вариантов, результаты которых не сказываются на окончательном результате.

Объектно-ориентированное программирование - программа представляется иерархией классов объектов. Класс содержит описания методов обработки объектов и спецификацию прав доступа методов к элементам объектов. Выполнение программы начинается с головной функции. Вызов функции сводится к применению методов к объектам при соответствии прав доступа. Пространство допустимых процессов включает кодирование специальных сигнатур для проверки соответствия метода и объекта. Это позволяет исключить перебор при выборе одного из определений метода, функции или операции, обладающих внешне одинаковыми представлениями. При несоответствии программист может при отладке изменить спецификацию прав доступа, перегрузить некоторые определения, дополнить или перестроить иерархию классов, чтобы нужный метод мог быть применён к объекту.

Переход к параллельным вычислениям заставляет программы представлять сетью, узлы которой - действия или данные. Действия выполнимы при определённых условиях готовности. При обходе сети выполняются готовые действия, возможно без ожидания завершения смежных действий. По существу на практике различаются два подхода к параллелизму. Асинхронный подход ориентирован на независимые потоки действий, возможно обменивающиеся сообщениями. Другой подход ориентирован на организацию взаимодействия процессов над общей памятью, допускающих императивную синхронизацию на уровне аппаратуры.

При асинхронном подходе отказ при выполнении некоторого действия не препятствует выполнению других действий, кроме него самого, что расширяет пространство допустимых процессов в сравнении с ФП. При выполнении одних действий могут меняться условия готовности других. Результатом программы считаются результаты ряда необходимых действий. Обход сети прерывается при отсутствии готовых к исполнению действий. Программа признаётся невыполнимой, если не выполнены отдельные необходимые действия. Программисту при отладке следует изменить программу так, чтобы обход сети обеспечивал выполнение необходимых действий при любом порядке обхода сети.

Взаимодействие процессов над общей памятью допускает императивную синхронизацию на уровне аппаратуры. Это может требовать более жёстких границ выполнимости очередного действия. Для императивной синхронизации процессов программа представляется сетью,

узлы которой — вилкообразные разветвления линейных участков, выполнимых при достижении разветвления. При обходе сети действия линейного участка выполняются неимперативно, т. е., возможно без непрерывности и ожидания завершения смежных действий. Отказ при выполнении некоторого действия не препятствует выполнению других действий и других участков, но текущий участок может быть помечен как дефектный, что расширяет пространство допустимых процессов. Программа завершается по мере выполнения всех бездефектных участков или признаётся невыполнимой, если нет успешно продолжаемых участков. Результатом программы считаются равноправные результаты всех участков. Программисту при отладке следует изменять программу так, чтобы обход сети обеспечивал выполнение необходимых участков при любых дефектах обхода сети.

10. Методика учебного программирования

Последние годы характеризуются всплеском интереса к созданию новых проблемно-ориентированных языков программирования. Это порождают заметно расширяющийся спектр проблем образовательного и реализационного характера. Парадигмальные модели языков и систем программирования могут быть полезны при создании классификатора для парадигмальной характеристики языков программирования. Такая классификация нацелена на стратификацию представления особенностей семантики языков и систем программирования на автономно развиваемые компоненты. Выделение компонентов в процессе пошаговой разработки экспериментальных систем программирования можно связать с формированием схем изучения и преподавания системного программирования. Это поспособствует снижению трудоёмкости разработки новых языков и систем программирования.

В разных источниках упоминается от 20-ти до 70-ти парадигм программирования. Список парадигм видоизменяется и расширяется в зависимости от актуальности тех или иных проблем программирования. Влияет и мода на особенности популярных ИТ. Бум языкотворчества в области проблемно-ориентированных ЯП показывает важность парадигмальных различий в определениях языков при выборе инструментов для практических работ. Успех выбора зависит от выработки рекомендаций по предпочтению инструментальных средств при создании новых программных систем и развитии ИТ. Разработка методов анализа определений ЯСП с целью установления их парадигмальной характеристики может дать подходящую основу для таких рекомендаций.

Практичность целей требует уточнить определение термина «парадигмы программирования». Необходимо отличать парадигмы от подходов, стилей, дисциплин,

техник, технологий, методов, методик, методологий программирования. Затем следует выразить отношения между базовыми и производными парадигмами. При выборе типовых компонентов для конструирования новых ЯП можно учесть историю появления наиболее популярных парадигм. Это даёт возможность уяснить причины и особенности их успешного применения на разных категориях задач разработки программных систем. Работа со сложными конструкциями требует методики лаконичного представления определений ЯСП. Лаконизм позволяет видеть парадигмальные характеристики с общих позиций. Всё это даёт возможность вывода системы обучения программированию. Такая система должна включать методы преподавания программирования на базе парадигмальных моделей. Ключевое значение имеют парадигмальные модели, представимые средствами функционального программирования. Это парадигма обладает высокой моделирующей силой. Лаконизм и понимаемость парадигмальных моделей может быть достаточной для описания учебных языков при обучении параллельным вычислениям. Такие модели пригодны для выработки ясных рекомендаций по технологической схеме разработки учебных материалов по программированию [8,9].

Общепринята практика экономить время подготовки специалистов погружением их в современный производственный инструментарий. Такая практика, минуя основы ради быстрого выхода в производство, закономерно приводит к провалам при быстрой смене технологий. Учебные языки программирования позволяют в лаконичной форме понять базовые ПП. В перспективе можно показать пути их развития в ближайшем будущем. Именно учебные языки и системы программирования призваны смягчать образовательные проблемы развития ИТ. В конце 1950-ых годов прецедент такого учебного языка был создан как введение в программирование на языке Lisp через изучение его подмножества Pure Lisp. Так была решена задача привлечения энтузиастов ради развёртывания исследований в области искусственного интеллекта. И сейчас основные идеи этого уровня используются в развитии теории автоматизации доказательств [12]. Есть основания решать трудные проблемы организации параллельных вычислений начиная с проблемы начального ознакомления с миром параллелизма. Это можно бы сделать на материале робототехники в стиле Lego-Logo, смягчив зависимость от физического материала использованием современных средств компьютерной графики.

Разработка учебных СП и игровых тренажёров – прекрасный полигон для экспериментов по совершенствованию и изучению техники компиляции программ. Материал для постановки учебных задач даёт создание новых проблемно-ориентированных ЯП. В этом плане перспективна разработка факультативного курса «Учебная информатика». Такой курс

непременно должен включать в себя ознакомление с основными явлениями параллелизма. Введение в параллелизм для школьников может дать курс «Начала параллелизма» с практикумом на базе языка начального обучения. Важно как можно раньше проявить и помочь осознать интуитивные способности понимать и прогнозировать взаимодействия процессов. Не трудно видеть, что с такими задачами в обычной жизни реально справляется большинство.

На базе опыта Новосибирских Школ юных программистов специально разработан язык СИНХРО для раннего ознакомления с миром параллелизма [3]. Этот язык даёт основные представления о технике программирования взаимодействующих процессов и методах конструирования программ. Представление данных обеспечивает возможность чёткого разделения процессов хранения данных и вычисления хранимых значений. Язык содержит средства макрогенерации, обогащённой управляемым контролем синтаксической и семантической правильности текстов программ. Ядро языка содержит механизм синхронизации процессов, включая согласование итерирования и рекурсии в терминах барьеров. Предполагается, что полноценное овладение современным миром ИТ более надёжно в форме игр на базе учебных языков и СП и проблемно-ориентированных игровых тренажёров. Созрела необходимость пересмотра методов организации и производственных СП с целью рационального использования возможностей ИТ, таких как видео, цвет, геометрия, шрифты и игры.

Эти вопросы проявляются на фоне быстрой эволюции технических средств и расширения сферы их применения. Решение ряда проблем осложнено обилием и разнообразием доступных весьма противоречивых учебных материалов. Это обостряет проблему достоверности представленного знания. К счастью теперь массово доступны используемые для самообучения и взаимообучения механизмы Интернет-поисковиков. Это даёт возможность выявления наиболее значимых позиций в своде информационных материалов.

11. Анализ схем изучения и преподавания ЯСП

Инвариант обучения программированию подразумевает некоторую математическую преамбулу. Обычно это даётся как обзор основных разделов классической дискретной математики. Многие понятия математики стали общепринятым аппаратом описания компьютерного мира, формальным языком постановки задач и методов их алгоритмического решения. За редким исключением не придано значение моделям непрерывной и неклассической математики. Разнообразие аксиоматических теорий и многозначных логик слабо представлены в программах обучения программистов. Тем не менее, абстрактные

механизмы имеющие место в реальной практике программирования и реализации ЯСП, находят своё воплощение в форме ПП. Есть достаточные основания для формирования более широких учебных программ по прикладной математике информационных систем, особенно для специализации по системному программированию. Классификация фундаментальных парадигм программирования отражает именно их образовательное значение. При анализе ПП с этой точки зрения отмечается важность разделённости ПП в определениях разных ЯП.

Не менее важна гуманитарная составляющая процессов разработки программ. Роль человеческого фактора имеет яркие проявления в истории информатики. Начиная с первых идей автоматизации вычислений и до наших дней можно привести многочисленные примеры индивидуального вклада отдельных личностей и коллективов в становление информатики и программирования. Движущие силы информатики тесно связаны с яркими лидерами, их способностями, индивидуальностью, характерами и образованием. Это всё в свою очередь базируется на природных механизмах имплицитного научения. Таким образом, результаты исследований в области этологии человека могут рассматриваться как исходный ориентир образовательной информатики. Именно такой ориентир, обуславливающий гуманитарные аспекты программирования и программистского образования, является важнейшим фактором успешной практики в программировании. Возможно, так устроена любая ноаая отрасль науки и производства. Молодость программирования как науки и технологии ещё не позволила переварить значительный объём имплицитного знания, не перевела его на удобный вербальный уровень. Потому программирование сохраняет зависимость от латентного обучения и организации специальных учебных практикумов. Общедоступность мощных информационных ресурсов резко повышает актуальность внимания к гуманитарным аспектам программистского образования. Раннее введение в учебный мир программирования на материале робототехники способно быстро довести до решения проблем достоверности знаний, представленных в сетях даже в форме большеобъёмных массивов.

Упорядочение учебно-методических материалов наиболее перспективно в рамках функционального подхода к системному представлению прикладных программ учебного назначения. Такой подход даёт методику лаконичного представления результатов парадигмального анализа схем изучения ЯСП. Особенно важно отследить схему практических занятий по подготовке и отладке программ. Существенно наладить работу в разных контекстах или на разных комплексах. Практика на базе производственных СП обычно отвлекает от понимания базовых возможностей ядра ЯСП. Именно специальный компьютерный практикум призван сформировать и проявить интуитивную базу программирования на примере работы с учебными макетами и учебными ЯП. Требуется

комплект учебных ЯП полноценного освоения разных парадигм программирования. Комплект ЯП должен быть достаточным для понимания типовых проблем и методов разработки программ.

Практика обучения обычно дополняется системой предвузовской подготовки талантливой молодежи для сознательного выбора профессии. Специализация в области программирования и информационных технологий в этом не исключение. Проблемой является акцент на активизацию творческих способностей, противоречащий массовому убеждению в доступности решений почти всех задач. Включение осознания инсайтов в процессе получения новых знаний затруднено чрезмерным богатством доступных знаний, уровень достоверности которых требует проверки. В принципе, такое положение дел может давать пищу для микроотркытий.

Результативность такого подхода была объективно показана историей Новосибирских Школ юных программистов. Выпускники этой Школы создали первый комплект прикладных программ автоматизации школьного учебного процесса «Школьница». Примерно такая дополнительная образовательная линия теперь фактически существует в форме Интернет-доступа к информационным материалам, дистанционных школ, консультаций, конференций и конкурсов. Причём, не только по программированию, но и по другим видам творчества.

Преподавание парадигм параллельного программирования в университетских образовательных программах и специализации по ИТ уже получило признание в мировой практике. Осознано её образовательное значение, но методические и образовательные проблемы ПВ ещё не преодолены. Так же не вполне осознан образовательный аспект фундаментальных и основных ПП. Слабо представлен полный спектр ПП в профессиональной подготовке информатиков. Для специализации информатиков изучение ПП должно быть чем-то вроде таблицы Менделеева в химии или алгебры в математике. На уровне высшего образования уже накоплен опыт преподавания ПП бакалаврам и магистрантам. Пора учесть, что результативность обучения практическому программированию может быть повышена активизацией парадигмального подхода к формированию программ обучения. ФП даёт лаконичные формы для достаточно глубокого понимания парадигмальных различий изучаемых ЯСП. Оно полезно и для формирования интуиции, при условии, что инвариант обучения программированию содержит гуманитарный аспект. Функциональные модели дают систематизированное представление основных и фундаментальных ПП, включая имплицитное знание. Важный материал даёт история программирования, информатики и вычислительной техники, особенно рассказы участников и очевидцев, дополненные результатами исследований по истории науки.

12. Заключение

Парадигмы программирования по существу различаются на уровне прагматики и интуитивных моделей понимания решаемых задач. Они определяют компромисс между трудоёмкостью разработки программ и производительностью их применения. Выбор парадигмы зависит от ряда характеристик постановки решаемой задачи. Такие параметры как уровень изученности задач, ранг абстрагирования понятий, степень организованности структур данных, полнота отлаженности алгоритмов и программ позволяют систематизировать свод исторически сложившихся парадигм программирования. Парадигмальный анализ постановки задачи позволяет представить проект программы в виде композиции из парадигмальных моделей. Такие модели допускают реализацию в виде автономно развиваемых программных модулей. Именно это способствует их многократному использованию в разных программных комплексах.

Изучение парадигм программирования начинается с проявления интуитивных представлений о взаимодействии процессов. Отладка такого взаимодействия может сопровождаться образцами методов разработки программ и стилей программирования. Разнообразие стилей и методов доступно как корпус свободно распространяемых программ и свод учебно-методических материалов. Основное препятствие - трудоёмкость систематизации этого обширного пространства, что делает актуальной задачу автоматизации парадигмального анализа программ.

Представление результатов парадигмального анализа возможно в рамках экспертной системы при условии выбора подходящих средств визуализации схем понятий, некоторые из возможных приёмов визуализации уже намечены и рассмотрены [2]. Инструментарий автоматизации парадигмального анализа можно создать как взаимодействие экспертной системы с гибким комплексом средств сопоставления программ с разными моделями, подобными системам верификации программ или проверки доказательств теорем. Такие системы уже пригодны для анализа нормализованных форм программ в стиле функционального программирования на языке Haskell.

Список литературы

1. Городняя, Л.В. Парадигмы программирования: анализ и сравнение / Из-во СО РАН, РФФИ 17-11-00042. 216 с.
2. Городняя Л.В. Пространство решений в языках и системах программирования. //Научный сервис в сети Интернет: труды XX Всероссийской научной конференции (17-22 сентября 2018

г., г. Новороссийск). -- М.: ИПМ им. М.В.Келдыша, 2018. - [Электронный ресурс]. URL: http://keldysh.ru/abrau/2018/theses/46.pdf (дата обращения 06.09.2018).

3. Городняя Л.В. Язык параллельного программирования СИНХРО, предназначенный для обучения. Новосибирск, ИСИ СО РАН, 30 с. (Препринт / ИСИ СО РАН; № 180) [Электронный ресурс]. URL: https://www.iis.nsk.su/files/preprints/gorodnyaya_180.pdf (дата обращения 06.09.2018).
4. Лавров С.С. Методы задания семантики языков программирования / Программирование, 1978. N 6. С. 3-10.
5. Михалкович С.С., Налбандян Ю.С. Адольф Львович Фуксман – математик и программист. [Электронный ресурс]. URL: http://www.sorucom.org/pdf/SORUCOM_2017.pdf (дата обращения 06.09.2018), [Электронный ресурс]. URL: http://plc.sfedu.ru/files/slides/day-1/Mikhalkovich.pdf (дата обращения 06.09.2018).
6. Фуксман А.Л. Технологические аспекты создания программных систем - М. : Статистика, 1979. – 183 с.
7. ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) [Электронный ресурс]. URL: *https://2017.splashcon.org/home* (дата обращения 06.09.2018).
8. Gorodnyaya L.V., Andreyeva T.A. Programming paradigms in higher education. // Bulletin of the Novosibirsk Computing center. Series: Computer science. – № 38. – 2015. – pp. 67-90.
9. Gorodniaia L., Andreyeva T.. Study of Programming Paradigms. // 10th International Technology, Education and Development Conference. 7-9 March, 2016, Valencia, Spain. //INTED2016 Proceedings. – 2016. – pp. 7482-7491. ISBN: 978-84-608-5617-7 ISSN: 2340-1079 doi: 10.21125/inted.2016.0768
10. Peter Van Roy (2009-05-12). "Programming Paradigms for Dummies: What Every Programmer Should Know" (PDF). info.ucl.ac.be. Retrieved 2018-01-22.
11. Peter Van Roy. Сайт с материалами по системе Mozart, поддерживающей учебный мульти-парадигмальный язык программирования Oz. [Электронный ресурс]. URL: http://sourceforge.net/projects/mozart-oz/ (дата обращения 06.09.2018)
12. Voevodsky, V. “A1-homotopy theory”, *Doc. Math.*, Proceedings of the International Congress of Mathematicians, Vol. I (Berlin, 1998), no. Extra Vol. I, 1998, 579–604 p.

Приложение 1

Аббревиатуры и обозначения

ПВ — параллельные вычисления.

ПП — парадигмы программирования.

РП — реализационная прагматика.

СП — система программирования.

ФП — функциональное программирование.

ЯП — язык программирования.

ЯСП — языки и системы программирования.

A — адреса в памяти.

E — элементарные значения.

F — функции или операции.

R — правило применения операций к значениям.

S — составные или конструируемые значения.

T — таблица соответствия адресов хранимым значениям.

V — значения.

V* — любое число значений, возможно ни одного.

V+ — любое число значений, хотя бы одно должно быть.

УДК 004

УДК 004.822:004.89

Подход к реализации паттернов содержания при разработке онтологий научных предметных областей

Загорулько Ю.А. (Институт систем информатики СО РАН, Новосибирский государственный университет),

Боровикова О.И. (Институт систем информатики СО РАН)

Рассматривается подход к разработке и реализации такого вида паттернов онтологического проектирования, как паттерны содержания. Использование таких паттернов при построении онтологий научных предметных областей позволяет, с одной стороны, обеспечить единообразное и согласованное представление всех сущностей разрабатываемой онтологии, с другой – сэкономить человеческие ресурсы и избежать типичных ошибок онтологического моделирования.

Ключевые слова: *онтология, паттерны онтологического проектирования, паттерны содержания, научная предметная область.*

1. Введение

Онтологии широко используются для формализации знаний в научных предметных областях. С помощью онтологии можно не только удобно представить все необходимые понятия моделируемой области, но и обеспечить их единообразное и согласованное описание. (Заметим, что под научной предметной областью (НПО) здесь понимается предметная область, охватывающая некоторую научную дисциплину или область научных знаний во всех ее аспектах, включая характерные для нее объекты и предметы исследования, применяемые в ней методы исследования, выполняемую в ней научную деятельность и полученные в рамках ее исследований научные результаты.)

Разработка онтологии любой предметной области является довольно сложным и трудоемким процессом, поэтому для его облегчения предлагаются различные методологии и подходы. В последнее время интенсивно развивается подход, базирующийся на применении паттернов онтологического проектирования (Ontology Design Patterns или ODP) [3, 5, 12]. Согласно этому подходу ODP представляют собой документально зафиксированные описания проверенных на практике решений типовых проблем онтологического моделирования. Они создаются для того, чтобы упорядочить и облегчить процесс

построения онтологий и помочь разработчикам избежать типичных ошибок онтологического моделирования.

Несмотря на то, что использование паттернов онтологического проектирования позволяет сэкономить человеческие ресурсы и повысить качество разрабатываемых онтологий, в настоящее время только одна методология построения онтологий, а именно методология экстремального проектирования онтологий XD (eXtreme Design methodology) [9], предложенная в рамках проекта NeOn [14] , открыто заявляет об использовании ODP.

Заметим также, что существует не так много инструментов разработки онтологий, поддерживающих использование паттернов онтологического проектирования. К ним, например, относится плагин для инструмента разработки онтологий проекта NeOn, а также плагин для редактора онтологий WebProtégé [13]. Однако эти средства покрывают только часть возможных задач, связанных с паттернами. Так, совсем нет инструментов, поддерживающих построение, поиск и извлечение паттернов из онтологий, и очень мало средств, поддерживающих сбор, обсуждение и распространение паттернов. К последним в какой-то мере можно отнести каталоги паттернов онтологического проектирования [8, 11, 15], также активно развиваемые в последнее время.

В статье рассматривается подход к реализации такого вида паттернов онтологического проектирования, как паттерны содержания [16], которые играют важную роль в предложенной авторами методологии разработки онтологий научных предметных областей [2, 3].

Работа выполнена при финансовой поддержке РФФИ (проект № 16-07-00569) и Президиума СО РАН (Блок 43.2. Комплексной программы ФНИ СО РАН II.1).

2. Введение в паттерны онтологического проектирования

Как было сказано выше, паттерны онтологического проектирования предназначены для описания решений типовых проблем, возникающих при разработке онтологий.

В зависимости от вида проблем, для решения которых создаются паттерны, различают структурные паттерны (Structural ODPs), паттерны соответствия (Correspondence ODPs), паттерны содержания (Content ODPs), паттерны логического вывода (Reasoning ODPs), паттерны представления (Presentation ODPs) и лексико-синтаксические паттерны (Lexico-Syntactic ODPs) [12].

Упрощенная версия типизации паттернов, предложенной в проекте NeOn [14], представлена на Рис.1.

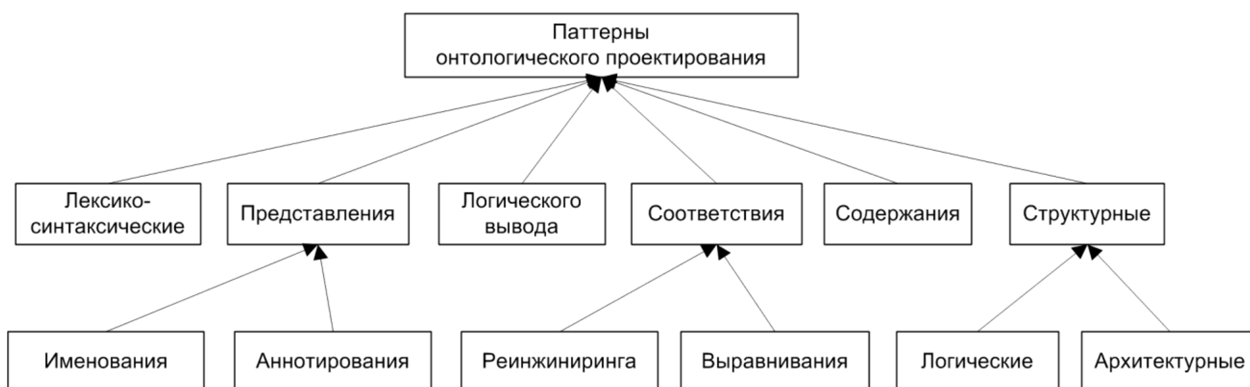

Рис.1. Типы паттернов онтологического проектирования.

Наиболее востребованными паттернами при разработке онтологий непосредственно инженерами знаний являются структурные паттерны, паттерны содержания и паттерны представления. Рассмотрим их подробнее.

Структурные паттерны либо фиксируют способы решения проблем, вызванных ограничениями выразительных возможностей языков описания онтологий, либо задают общую структуру и вид онтологии. Паттерны, предназначенные для преодоления проблемы выразительности языков, называются логическими структурными паттернами (Logical ODP). Паттерны, содержащие предложения по организации онтологии в целом, относятся к архитектурным паттернам (Architectural ODP).

Паттерны содержания задают способы представления типовых фрагментов онтологий, на основе которых могут строиться онтологии целого класса предметных областей. На основе этих паттернов можно описать более сложные (составные) паттерны содержания, дополняя их недостающими компонентами и вводя дополнительные отношения. Однако следует учитывать, что при использовании составных паттернов могут возникнуть проблемы их сборки и интеграции [6] .

Паттерны представления определяют рекомендации по именованию, аннотированию и графическому представлению элементов онтологии. Применение этих паттернов должно повысить «читаемость» онтологии, а также удобство и простоту ее использования.

На данный момент не существует единого стандарта для описания паттернов, но чаще всего они описываются в формате, предложенном на портале ассоциации ODPA (Association for Ontology Design & Patterns) [8], созданном в рамках проекта NeOn [14]. В соответствии с этим форматом описание паттерна включает его графическое представление, текстовое описание, набор сценариев и примеров использования, ссылки на другие паттерны, в которых он используется, а также сведения о названии паттерна, его авторе и области применения.

Методология экстремального проектирования онтологий XD [9] предлагает также снабжать каждый паттерн содержания набором квалификационных вопросов (Competency questions). Эти вопросы могут использоваться как на этапе разработки паттернов, так и для поиска нужных паттернов при разработке конкретной онтологии.

Разработка и использование паттернов онтологического проектирования опирается на 5 основных операций над ними [12]: импорт (import), специализация (specialization), обобщение (generalization), композиция (composition) и расширение (expansion).

3. Разработка и применение паттернов содержания при построении онтологий НПО

При создании онтологии НПО важно обеспечить возможность единообразного и согласованного представления используемых в ней научных понятий и их свойств. Одним из способов решения этой проблемы является применение паттернов содержания, описывающих понятия, характерные для большинства научных предметных областей и выполняемой в них научной деятельности. Набор таких паттернов предоставляет методология построения онтологий, разработанная в рамках технологии создания тематических интеллектуальных научных интернет-ресурсов [2].

Следует заметить, что эта методология кроме паттернов содержания включает также паттерны представления и структурные логические паттерны [3, 17], которые используются в паттернах содержания. С помощью структурных логических паттернов, например, представляются атрибутированные отношения (бинарные отношения с атрибутами), многоместные отношения и области допустимых значений, заданные множествами элементарных значений. Последние, следуя традиции, принятой в реляционной модели данных, будем называть доменами.

На Рис.2 приведен паттерн представления домена, в котором сам домен задается перечислимым классом *Класс1*, включающим конечный набор различных индивидов (объектов или экземпляров класса) *объект1*,…, *объектn*, определяющих все возможные значения некоторого свойства объектов класса *Класс2*. При этом класс *Класс1* является наследником служебного класса *Домен*.

На Рис. 3 представлен пример использования данного паттерна для описания домена “Географический тип”.

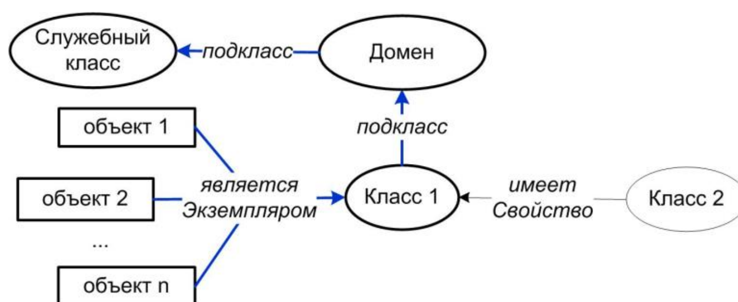

Рис.2. Структурный логический паттерн для представления домена.

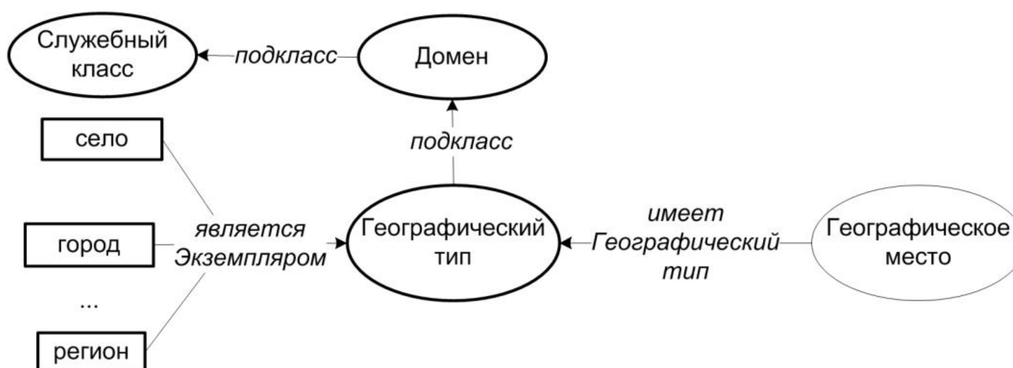

Рис.3. Пример использования паттерна для представления домена ”Географический тип”.

Заметим, что на рисунках, содержащих графическое представление паттернов, классы обозначаются в виде эллипсов, а индивиды и атрибуты – в виде прямоугольников. Связь типа «класс-класс» показывается сплошной прямой линией, а связь типа «класс-атрибут» – прерывистой. При этом обязательные связи показываются на рисунках линией синего цвета, а связываемые ими классы, атрибуты и индивиды представляются фигурами, обведенными жирной линией.

4.1. Разработка паттернов содержания

В рамках методологии построения онтологий НПО разработаны паттерны содержания для представления следующих понятий: *Объект исследования*, *Предмет исследования*, *Метод исследования*, *Раздел науки*, *Научный результат*, *Персона*, *Организация*, *Деятельность*, *Проект*, *Публикация* и др.

При разработке паттернов содержания для каждого из них был определен набор квалификационных вопросов, представляющих его содержание. С помощью этих вопросов был выявлен обязательный и факультативный состав элементов паттерна и описаны требования к ним, которые были представлены в виде аксиом и ограничений.

Рассмотрим некоторые паттерны содержания подробнее.

Как правило, научная деятельность реализуется через проекты, программы, экспедиции и экспертизы. Графическое представление паттерна, предназначенного для описания научной

деятельности, приведено на Рис.4. В этом паттерне отражено требование, состоящее в том, что любая деятельность должна иметь название и при ее описании необходимо давать ссылку на объект исследования, изучению которого посвящена данная деятельность, на раздел науки, по теме которого она выполняется, и на персону или организацию, входящую в состав ее участников.

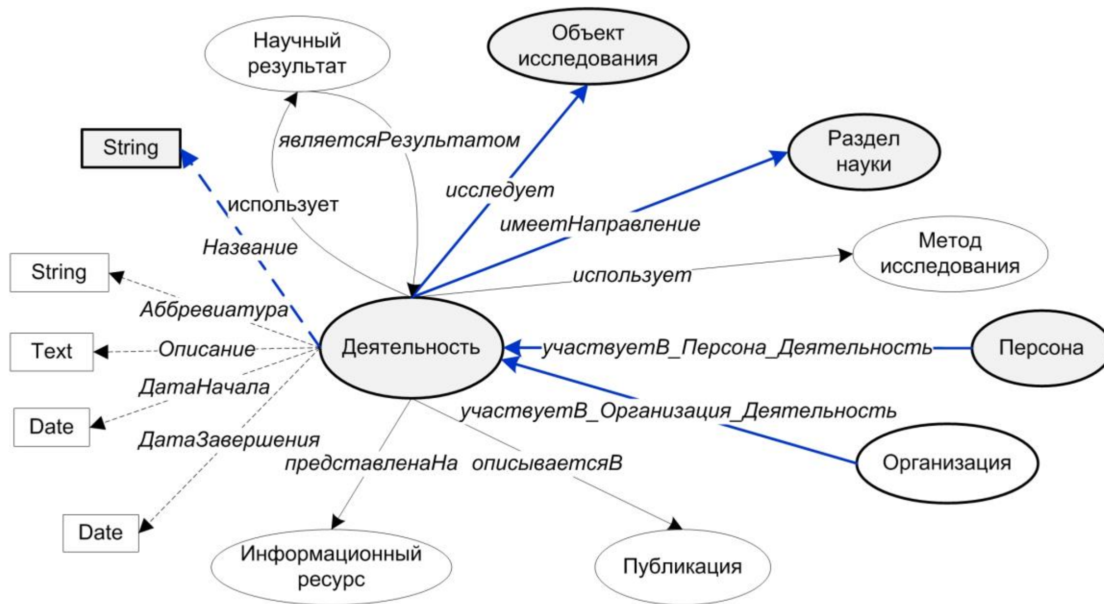

Рис.4. Паттерн для описания научной деятельности.

На Рис.5 приведен паттерн, предназначенный для описания методов исследования, используемых в научной деятельности. В соответствии с ним описание метода должно включать название и описание метода, а также ссылки на решаемые с помощью него задачи, раздел науки, в котором он используется, и автора метода (персону или организацию).

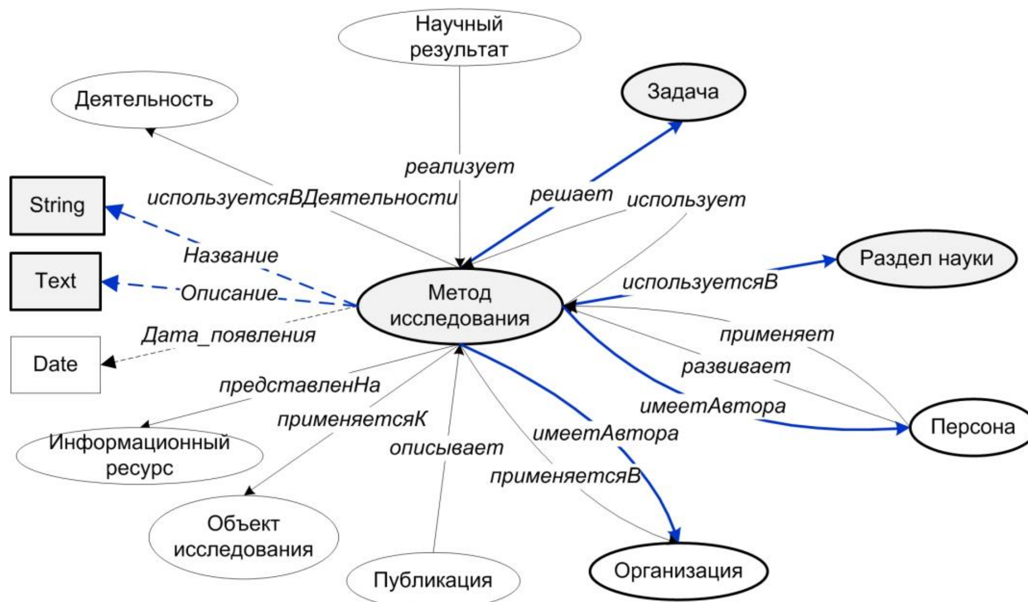

Рис.5. Паттерн для описания метода исследования.

На Рис.6 приведен паттерн, предназначенный для описания научного результата. В этом паттерне отражено требование: при описании научного результата необходимо указывать его название и давать ссылку на деятельность, при выполнении которой он был получен, и на публикацию, в которой он был зафиксирован.

Рис.6. Паттерн для описания научного результата.

Нетрудно заметить, что невозможно описать ни один паттерн содержания изолированно, т.е. не используя понятия из других паттернов. Например, в паттерне для описания научного результата (см. Рис. 6), кроме «центрального» понятия *Научный результат*, также используются понятия *Деятельность*, *Предмет исследования*, *Раздел науки* и др. Это объединяет все паттерны содержания в единую систему паттернов, а представляемые ими понятия в единую систему понятий, задающую связанное описание моделируемой предметной области.

Фактически, паттерны содержания являются фрагментами онтологии, которые после специализации содержащихся в них понятий и дополнения необходимыми понятиями и свойствами, становятся составными частями разрабатываемой онтологии НПО.

4.2. Реализация паттернов содержания

Все паттерны содержания рассматриваемой методологии реализованы средствами языка OWL [7]. Ниже приведен фрагмент OWL-реализации паттерна содержания, предназначенного для описания научной деятельности в формате Turtle. (Заметим, что в этом фрагменте используются названия онтологических элементов из пространства имен онтологии *inir: http://www.semanticweb.org/IIS/ontologies/INIR_Ontology#*.)

Опишем содержательные компоненты, которые включает этот паттерн.

Прежде всего, это набор используемых в нем классов и аксиома непересекаемости этих классов (*AllDisjointClasses*):

```
:ИнформационныйРесурс rdf:type owl:Class;
:МетодИсследования rdf:type owl:Class;
:ОбъектИсследования rdf:type owl:Class;
:Организация rdf:type owl:Class;
:Персона rdf:type owl:Class;
:Публикация rdf:type owl:Class;
:РазделНауки rdf:type owl:Class;
:НаучныйРезультат
[ rdf:type owl:AllDisjointClasses; owl:members
( :Деятельность
:ИнформационныйРесурс
:МетодИсследования
:ОбъектИсследования
:Организация
:Персона
:Публикация
:РазделНауки
:НаучныйРезультат)] .
```

Набор свойств класса *Деятельность* (в нотации языка OWL – *ObjectProperty* и *DatatypeProperty*) выглядит следующим образом:

```
:имеетНаправление_Деятельность_Раздел rdf:type owl:ObjectProperty;
 rdfs:domain :Деятельность;
 rdfs:range :РазделНауки;
:исследует_Деятельность_Объект rdf:type owl:ObjectProperty;
 rdfs:domain :Деятельность;
 rdfs:range :ОбъектИсследования;
:использует_Деятельность_Метод rdf:type owl:ObjectProperty;
 rdfs:domain :Деятельность;
 rdfs:range :МетодИсследования;
```

```
:описываетсяВ_Деятельность_Публикация rdf:type owl:ObjectProperty;
 rdfs:domain :Деятельность;
 rdfs:range :Публикация;
:участвуетВ_Персона_Деятельность rdf:type owl:ObjectProperty;
 rdfs:domain :Персона;
 rdfs:range :Деятельность;
:являетсяРезультатом_Результат_Деятельность rdf:type owl:ObjectProperty;
 rdfs:domain :НаучныйРезультат;
 rdfs:range :Деятельность;
:являетсяРесурсом_Ресурс_Деятельность rdf:type owl:ObjectProperty;
 rdfs:domain :ИнформационныйРесурс;
 rdfs:range :Деятельность;
:участвуетВ_Организация_Деятельность rdf:type owl:ObjectProperty;
 rdfs:domain :Организация;
 rdfs:range :Деятельность;
:Деятельность_Название rdf:type owl:DatatypeProperty;
rdfs:domain :Деятельность;
rdfs:range rdfs:Literal;
:Деятельность_Аббревиатура rdf:type owl:DatatypeProperty;
rdfs:domain :Деятельность;
rdfs:range rdfs:Literal;
:Деятельность_Описание rdf:type owl:DatatypeProperty;
rdfs:domain :Деятельность;
rdfs:range rdfs:Literal;
```

Далее для каждого экземпляра (представителя) класса *Деятельность* с помощью свойства *FunctionalProperty* накладывается требование к указанию только одной даты начала и одной даты завершения:

```
:Деятельность_ДатаЗавершения rdf:type owl:DatatypeProperty,
owl:FunctionalProperty;
rdfs:domain :Деятельность ;
rdfs:range rdfs:Literal;
:Деятельность_ДатаНачала rdf:type owl:DatatypeProperty, owl:FunctionalProperty;
```

```
rdfs:domain :Деятельность ;
rdfs:range rdfs:Literal;
```

Наконец для каждого экземпляра класса *Деятельность* путем накладывания ограничений на свойства описываются следующие требования на количество и тип, связанных с этим классом понятий:

- должно быть задано хотя бы одно направление деятельности;
- должен быть задан хотя бы один объект исследования;
- в деятельности должна участвовать, по крайней мере, одна персона;
- у деятельности должно быть, по крайней мере, одно название.

```
:Деятельность rdf:type owl:Class;
owl:equivalentClass
[ rdf:type owl:Restriction;
owl:onProperty :имеетНаправление_Деятельность_Раздел ;
owl:someValuesFrom : РазделНауки] ,
[ rdf:type owl:Restriction;
owl:onProperty :исследует_Деятельность_Объект ;
owl:someValuesFrom :ОбъектИсследования] ,
 [rdf:type owl:Restriction;
owl:onProperty [owl:inverseOf :участвуетВДеятельности] ;
owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger;
 owl:onClass [ rdf:type owl:Class ;owl:unionOf (:Организация :Персона )]
[ rdf:type owl:Restriction;
owl:onProperty :Деятельность_Название ;
owl:minCardinality "1"^^xsd:nonNegativeInteger] ;.
```

4.3. Поддержка использования паттернов содержания

Методология построения онтологий НПО с использованием паттернов содержания поддерживается нашим редактором данных, который позволяет пополнять онтологию индивидами классов, для которых реализованы паттерны содержании. При этом редактор по имени класса, выбранного пользователем, находит соответствующий ему паттерн и на его основе строит форму, содержащую поля для заполнения свойств объекта этого класса (см. Рис.7).

При создании объекта учитываются ограничения на значения и кардинальность его свойств, заданные в паттерне. Заметим, что такие ограничения используются не только для контроля вводимой информации, но и в качестве потенциального источника значений свойств создаваемого (редактируемого) объекта. Так, в зависимости от типа ограничений пользователь может выбрать значения для каждого свойства либо из описания домена, заданного соответствующим структурным паттерном в качестве области значений этого свойства, либо из динамически сформированного по текущей версии онтологии списка индивидов класса, указанного в качестве области значений этого свойства.

Рис.7. Окно редактора данных

На Рис.7 показано окно редактора данных в момент создания объекта с названием *Метод недоопределенных вычислений* для онтологии НПО «Поддержка принятия решений в слабоформализованных областях» [1, 3, 4] на основе паттерна метода исследования (см. Рис. 5). При создании этого объекта контролируется наличие значений у его атрибутов *название* и *описание*, а также существование ссылок на объекты классов *Персона, Задача* и *Раздел науки* (учитываются ограничения на кардинальность связей *имеетАвтора-Персону*, *решаетЗадачу* и *используетсяВРазделеНауки*). При этом, когда пользователь переходит к заданию автора метода, автоматически формируется список из индивидов класса *Персона* (см. окно с

фамилиями и инициалами в правом нижнем углу Рис.7), из которого пользователь может выбрать нужную персону.

5. Заключение

В статье обсуждены вопросы применения паттернов онтологического проектирования для разработки онтологий научных предметных областей. Показано, что при использовании таких паттернов обеспечивается единообразное и согласованное представление всех сущностей разрабатываемой онтологии, экономятся человеческие ресурсы и сокращается число ошибок при разработке онтологий.

Представлен подход к реализации паттернов содержания, играющих ведущую роль в предлагаемой методологии разработки онтологий НПО. Данная методология и описанные в статье паттерны содержания показали свою практическую полезность при разработке онтологий различных научных предметных областей («Поддержка принятия решений в слабоформализованных областях» [1, 4], «Активная сейсмология» [10] и др.).

Список литературы

1. Загорулько Г.Б. Разработка онтологии для интернет-ресурса поддержки принятия решений в слабоформализованных областях // Онтология проектирования. 2016. Т. 6, №4(22). С. 485-500.
2. Загорулько Ю.А., Загорулько Г.Б., Боровикова О.И. Технология создания тематических интеллектуальных научных интернет-ресурсов, базирующаяся на онтологии // Программная инженерия. 2016. Т. 7. № 2. С. 51-60.
3. Загорулько Ю.А., Боровикова О.И., Загорулько Г.Б. Применение паттернов онтологического проектирования при разработке онтологий научных предметных областей // Аналитика и управление данными в областях с интенсивным использованием данных: сборник научных трудов XIX Международной конференции DAMDID/RCDL’2017. Москва: ФИЦ ИУ РАН, 2017. С. 332-340.
4. Загорулько Ю.А., Боровикова О.И., Загорулько Г.Б., Шестаков В.К. Использование паттернов для разработки онтологии информационно-аналитического интернет-ресурса «Поддержка принятия решений» // Информационные и математические технологии в науке и управлении. 2017. № 3. С. 144-153.
5. Ломов П.А. Применение паттернов онтологического проектирования для создания и использования онтологий в рамках интегрированного пространства знаний // Онтология проектирования, 2015. Т. 5, № 2(16). С.233-245.

6. Ломов П.А. Автоматизация синтеза составных онтологических паттернов содержания // Онтология проектирования. 2016. Т.6, №2(20). С.162-172. DOI: 10.18287/2223-9537-2016-6-2-162-172.
7. Antoniou G., Harmelen F. Web Ontology Language: OWL // Handbook on Ontologies. Berlin: Springer Verlag, 2004. P. 67-92.
8. Association for Ontology Design & Patterns, 2018. URL: http://ontologydesignpatterns.org (дата обращения: 22.10.2018).
9. Blomqvist E., Hammar K., Presutti V. Engineering Ontologies with Patterns: The eXtreme Design Methodology. P. Hitzler, A. Gangemi, K. Janowicz, A. Krisnadhi, V. Presutti (eds.). Ontology Engineering with Ontology Design Patterns. Studies on the Semantic Web, IOS Press, 2016. P. 23-50.
10. Braginskaya L., Kovalevsky V., Grigoryuk A., Zagorulko G. Ontological approach to information support of investigations in active seismology // Proceedings of the 2nd Russian-Pacific Conference on Computer Technology and Applications (RPC),Vladivostok, Russky Island, Russia, 25-29 September, 2017. P. 27-29. IEEE Xplore digital library. URL: http://ieeexplore.ieee.org/document/8168060 (дата обращения: 22.10.2018) ISBN: 978-1-5386-1206-4. DOI: 10.1109/RPC.2017.8168060.
11. Dodds L., Davis I. Linked Data Patterns. 2012. URL: http://patterns.dataincubator.org/book (дата обращения 22.10.2018).
12. Gangemi A., Presutti V. Ontology Design Patterns // Handbook on Ontologies. Springer, 2009. P. 221-243.
13. Hammar K. Ontology Design Patterns in WebProtégé. Proceedings of 14th International Semantic Web Conference (ISWC-2015). Posters & Demonstrations Track. CEUR Workshop Proceedings, 2015. Vol. 1486. URL: http://ceur-ws.org/Vol-1486/paper_50.pdf (дата обращения: 22.10.2018).
14. NeOn Project, 2018. URL: http://neon-project.org/nw/Welcome_to_the_NeOn_Project.html (дата обращения: 22.10.2018).
15. Ontology Design Patterns (ODPs) Public Catalog, 2009. URL: http://odps.sourceforge.net (дата обращения: 22.10.2018).
16. Presutti V., Daga E., Gangemi A., Blomqvist E. eXtreme Design with Content Ontology Design Patterns. Proceedings of the Workshop on Ontology Patterns (WOP 2009), Washington D.C., USA, 2009. Vol. 516. P. 83-97.
17. Zagorulko Y., Borovikova O., Zagorulko G. Pattern-Based Methodology for Building the Ontologies of Scientific Subject Domains. In: New Trends in Intelligent Software Methodologies, Tools and Techniques. Proceedings of the 17th International Conference SoMeT_18. H. Fujita and E. Herrera-Viedma (Eds.). Series: Frontiers in Artificial Intelligence and Applications, Vol. 303. Amsterdam: IOS Press, 2018. P. 529–542.

УДК 004.43

Разработка параллельной предикатной программы решения системы линейных уравнений методом Гаусса-Жордано

Каблуков И.В. (Институт систем информатики СО РАН),

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается технология предикатного программирования для реализации параллельной программы решения системы линейных уравнений методом Гаусса-Жордано. Предикатная программа изначально параллельна и не требует распараллеливания. Описывается построение предикатной программы и ее оптимизирующая трансформация с получением эффективной параллельной императивной программы.

Ключевые слова: *параллельное программирование, трансформации программ, функциональное программирование.*

1. Введение

В настоящей работе представлена технология предикатного программирования для реализации параллельной программы решения системы линейных уравнений методом Гаусса-Жордано. В течение десяти лет на данной программе демонстрируется техника работы с массивами в магистерском курсе НГУ «Предикатное программирование». Стимулом публикации послужила работа [4] с описанием реализации асинхронно-потоковой программы алгоритма разложения Холецкого для вычисления квадратного корня матрицы. Схема этого алгоритма во многом похожа на схему Гаусса-Жордано.

Эффективность предикатных программ достигается применением *оптимизирующих трансформаций*. Они определяют отличную от классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

В предикатном программировании нет распараллеливания. Предикатная программа изначально параллельна. В языке предикатного программирования P имеются параллельный оператор и конструкция векторного параллелизма «определение массива». Задача лишь в том, чтобы донести исходный параллелизм до конечной императивной программы.

Во втором разделе дается краткое описание языка P [3]. В третьем разделе подробно описывается построение предикатной программы решения системы линейных уравнений методом Гаусса-Жордано. Далее представлена последовательность оптимизирующих трансформаций с получением эффективной параллельной программы. Природа параллелизма и особенности реализации параллельных программ исследуются в разделе 5. В разделе 6 проводится сравнение с подходами потокового параллелизма. Заключение содержит замечания по реализации параллельных программ для разных платформ.

2. Предикатное программирование

Предикатная программа является предикатом (логической формулой) в форме вычислимого оператора. Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [3] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
{ <оператор> }
```

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов. Они используются для улучшения понимания программ и для дедуктивной верификации [6, 10, 19].

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Вызов вида H(x: y), где x – набор выражений, а y – набор переменных, эквивалентен оператору присваивания y = H(x). Здесь вызов программы H представлен в форме вызова функции H(x).

В языке P имеется развитая система типов: подтипы, структуры, множества, алгебраические типы, предикатные типы, массивы. Значением типа **array**(E, J) является *массив* с *элементами массива* типа E и *индексами* конечного типа J. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами. Массив всегда вычисляется полностью для всех его элементов. Для элемента

массива используется привычная форма записи A[j] вместо функциональной формы A(j); соответствующая предикатная форма есть A(j**:** z).

Операции с массивами определяются через *оператор каррирования* A(j**:** z) {B(x, j**:** z)} [12], где аргументы j и результаты z являются формальными параметрами, а оператор B(x, j**:** z) – телом определяемой предикатной программы A. Результатом исполнения оператора каррирования является новая предикатная программа A(j**:** z), получаемая фиксацией текущих значений переменных набора x в операторе B(x, j**:** z). В случае, когда A – массив, дополнительно происходит вычисление всех элементов массива A.

В языке P оператор каррирования для массива A записывается в виде оператора присваивания A = **for** (j) {B(x, j)}. Конструкция в правой части оператора – *определение массива*. Значением определения массива является новый массив, в котором для всякого допустимого индекса j соответствующий элемент массива равен значению выражения B(x, j). Вычисления элементов массива для разных индексов j могут быть реализованы параллельно. Это векторный параллелизм, который становится матричным в случае, когда j – пара индексов. Рассмотрим пример:

```
type Vec = array (int, 1..5);
Vec  A;
A =  for (j) { j*j };
```

Здесь описывается массив A с элементами типа **int** и индексами от 1 до 5. Результатом исполнения данного фрагмента является массив A со значениями элементов: 1, 4, 9, 16, 25.

Определение части массива представляется следующим оператором каррирования:

A(j**:** z) { **if** (m(x)≤j & j≤n(x)) B(x, j**:** z) **else** C(j**:** z) } .

В языке P данная конструкция записывается в виде оператора:

A = C **for** (j) {**case** m(x)..n(x)**:** B(x, j) } .

При исполнении реализуется замена части массива C в диапазоне индексов от m(x) до n(x). Модифицированное значение массива присваивается массиву A. При этом исходный массив C остается неизменным. В случае, когда m(x) и n(x) совпадают, вместо диапазона m(x)..n(x) указывается просто m(x). Если j – пара индексов, то диапазоны могут быть по обоим индексам; в этом случае они разделяются запятой.

В определении части массива может быть несколько **case**-частей. Например, оператор A = C **for** (j) { **case** m..n**:** B(x, j) **case** p..k**:** D(x, j) } является эквивалентом оператора A = (C **for** (j) { **case** m..n**:** B(x, j) }) { **case** p..k**:** D(x, j) } .

Введем понятие *гиперфункции* – программы с несколькими *ветвями* результатов [3, разд. 3.1, 3.3]. Гиперфункция A(x: y: z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются. *Вызов гиперфункции* записывается в виде A(x: y #M1: z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций [2], переводящих программу на императивное расширение языка **P**:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 6, 10, 11, 19].

3. Программа решения системы линейных уравнений методом Гаусса-Жордано

Рассматривается классический алгоритм решения системы линейных уравнений метода Гаусса-Жордано. Это метод исключения переменных с приведением к эквивалентной системе треугольного вида. Метод работает, если не появляется нулей на главной диагонали. Здесь представлен более общий алгоритм, который успешно «борется» с нулями и, в частности, распознает вырожденность системы уравнений.

Пусть имеется система линейных уравнений:

$$\text{LinEq(n, a, b, x)} \cong \sum_{j=1}^{n} a_{ij}x_j = b_i, i = 1..n$$

Здесь a – квадратная матрица размерности n×n, b – вектор свободных членов, x – вектор неизвестных переменных, значения которых требуется вычислить.

Для построения спецификации задачи необходимо формализовать предикат LinEq. Определим типы вектора и матрицы:

```
nat n;
type I1n = 1..n;
type VEC = array (real, I1n);
type MATR = array (real, I1n, I1n);
```

Переменная n, являющаяся параметром задачи, определена здесь как глобальная, что позволяет опускать ее в программах, формулах и параметрических типах I1n(n), VEC(n) и MATR(n). Введем вспомогательную формулу подсчета суммы значений некоторой функции f, заданной параметром, для аргументов от 1 до k:

```
formula SUM(nat k, predicate(nat: real) f: real) = k=0? 0: SUM(k - 1, f) + f(k);
```

Наконец, формула LinEq определяет набор равенств, соответствующий системе линейных уравнений:

```
formula LinEq(MATR a, VEC b, x) =
        forall i=1..n. SUM( n, predicate(nat j: real) {a[i, j] * x[j]} ) = b[i];
```

Здесь, конструкция **predicate**(**nat** j**: real**) {a[i, j] * x[j]} литерально определяет функцию f(j) = a[i, j] * x[j].

Представленная ниже программа Lin является гиперфункцией. Определяется, вырождена ли матрица a. В случае невырожденной матрицы решается система уравнений – вычисляется вектор неизвестных x. При этом реализуется первая ветвь гиперфункции. Если обнаруживается, что матрица a вырождена, то завершение гиперфункции Lin реализуется по второй ветви.

```
Lin(MATR a, VEC b : VEC x : )
pre  n > 0
pre 1: det(a) != 0
post 1: LinEq(a, b, x)
{       TriangleMatr(a, b:  MATR c, VEC d :  #2);
        Solution(c, d:  VEC x ) #1
}
```

В гиперфункции Lin две ветви результатов, причем у второй ветви (после второго двоеточия в первой строке) результатов нет. Условие n > 0 есть общее предусловие программы. Предусловие первой ветви det(a) != 0 определяется как ненулевой детерминант

матрицы a. Постусловие по первой ветви – предикат LinEq(n, a, b, x). Вторая ветвь не имеет постусловия, так как у нее нет результатов.

Гиперфункция TriangleMatr реализует приведение исходной системы уравнений a*x = b к эквивалентной системе уравнений c*x = d с треугольной матрицей c. Если в процессе преобразования матрицы a обнаруживается ее вырожденность, то исполнение гиперфункции завершается ветвью 2. Оператор перехода #2 во второй ветви вызова TriangleMatr определяет завершение гиперфункции Lin второй ветвью.

Программа Solution реализует решение системы c*x = d для треугольной матрицы c. Оператор перехода #1 после вызова Solution определяет завершение программы Lin первой ветвью.

Определим спецификацию программы TriangleMatr. Условие того, что новая система уравнений c*x = d эквивалентна исходной невырожденной системе уравнений a*x = b, представлено формулой postLin.

```
formula postLin(MATR a, VEC b, MATR c, VEC d) =
      forall VEC x, y. LinEq(a, b, x) & LinEq(c, d, y) ⇒ x = y
```

Формула triangle определяет второе условие: матрица c является верхней треугольной, т.е. все элементы главной диагонали равны 1, а элементы ниже главной диагонали – нулевые.

```
formula triangle(MATR c) =
      (forall i=1..n. c[i, i] = 1) & forall i,j=1..n. (i > j ⇒ c[i, j] = 0)
```

Программа TriangleMatr представлена ниже.

```
TriangleMatr(MATR a, VEC b: MATR a′, VEC b′ :  );
pre  n > 0
pre 1: det(a) != 0
post 1: postLin(a, b, a′, b′) & triangle(a′)
{       Jord(a, b, 1:  a′, b′ #1 :  #2) };
```

Здесь применяется метод обобщения исходной задачи TriangleMatr. Рассмотрим более общую задачу Jord, в которой имеется дополнительный аргумент k – номер текущей строки матрицы a. Формула triaCol вводит дополнительное предусловие в программе Jord: матрица a триангулирована для столбцов от 1 до k-1, т.е. элементы главной диагонали до k-1 равны единице, а элементы ниже этих диагональных элементов равны нулю, см. Рис.1.

```
formula triaCol(nat k, MATR a) =
      forall i=1..k-1. (a[i, i] = 1) & forall i = 1..n, j=1..k-1. (i > j ⇒ a[i, j] = 0);
```

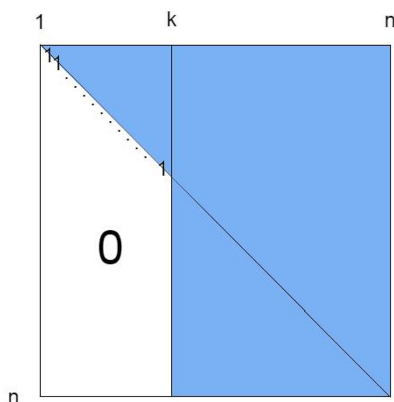

Рис.1

Задача TriangleMatr сводится к задаче Jord при k = 1.

Программа гиперфункции Jord приведена ниже.

```
Jord(MATR a, VEC b, nat k: MATR a′, VEC b′ : )
pre 1 ≤ k ≤ n & triaCol(k, a)
pre 1: det(a) != 0
post 1: postLin(a, b, a′, b′) & triangle(a′)
{       diagEl(a, b, k: MATR c, VEC d : #2);
        norm(k, c, d: MATR e, VEC f);
        subtrLines(k, e, f: MATR g, VEC h);
        if (k = n) { a′ = g || b′ = h #1 }
        else   Jord(g, h, k+1:  a′, b′ #1 : #2)
};
```

Программа Jord реализует следующую последовательность действий. Вызов программы diagEl обеспечивает истинность условия a[k, k] ≠ 0, обменивая k-ю строку матрицы a с последующими строками. Если удается обеспечить a[k, k] ≠ 0, то вызов программы norm преобразует систему уравнений к эквивалентной системе, в которой a[k, k] = 1. Вызов subtrLines реализует вычитание k-й строки из последующих строк матрицы так, чтобы занулить k-й столбец. Если k<n, реализуется рекурсивный вызов программы Jord с параметром k+1.

Цель гиперфункции diagEl – обеспечить истинность условия a[k, k] ≠ 0. Если a[k, k]=0, делается попытка обменять k-ю строку матрицы a с одной из последующих строк. Если во всех строках нулевые элементы, то матрица a вырождена, и тогда программа завершается второй ветвью гиперфункции.

```
diagEl(MATR a, VEC b, nat k: MATR a′, VEC b′: )
pre  1 ≤ k ≤ n & triaCol(k, a)
pre 1: det(a) != 0
post 1: a′[k, k] != 0 & postLin(a, b, a′, b′)
{       if (a[k, k] != 0) { a′ = a || b′ = b #1}
        else perm(a, b, k, k+1: a′, b′ #1 : #2)
}
```

Гиперфункция perm является обобщением diagEl. Имеется параметр m, причем m>k. Дополнительное предусловие: элементы k–го столбца в строках от k до m-1 – нулевые. Программа perm обменивает k–ю строку с одной из строк от m до n, если такое возможно.

```
perm(MATR a, VEC b, nat k, m: MATR a′, VEC b′ : )
pre  1 ≤ k ≤ n & k<m ≤ n+1 & triaCol(k, a) & forall i = k..m-1. a[i, k] = 0
pre 1: det(a) != 0
post 1: a′[k, k] != 0 & postLin(a, b, a′, b′)
{       if (m>n) #2
        else if (a[m, k] != 0) {
                b′ = b for (I1n j) {case k: b[m] case m: b[k]} ||
                a′ = a for (I1n i, j) {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
                #1
        } else  perm(a, b, k, m + 1: a′, b′ #1 : #2)
}
```

Программа norm приводит диагональный элемент a[k, k] к единице путем деления k–ой строки на этот диагональный элемент:

```
norm(nat k, MATR a, VEC b: MATR a′, VEC b′)
pre  1 ≤ k ≤ n & a[k, k] != 0 &  triaCol(k, a)
post a′[k, k] = 1 & postLin(a, b, a′, b′)
{       b′ = b for (I1n j) { case k: b[k] / a[k, k] } ||
        a' = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
}
```

Программа subtrLines приводит к нулю все элементы в k–ом столбце ниже главной диагонали путем вычета k–ой строки из всех нижележащих строк с подходящими множителями:

```
subtrLines(nat k, MATR a, VEC b: MATR a′, VEC b′)
pre  1 ≤ k ≤ n & a[k, k] = 1 &  triaCol(k, a)
post triaCol(k+1, a) & postLin(a, b, a′, b′)
{ b′ = b for (I1n i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
  a' = a for (I1n i, j) {case (k+1..n, k): 0 case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
}
```

Фрагмент **case** (k+1..n, k)**:** 0 специально выделен отдельно из общего случая.

Программа Solution находит решение системы уравнений a*x = b для верхнетреугольной матрицы a.

```
Solution(MATR a, VEC b:  VEC x )
pre  n > 0 & triangle(a)
post LinEq(a, b, x)
{        uniMat(a, b, n:  x ) }
```

Решение реализуется последовательным приведением к нулю всех элементов матрицы a выше главной диагонали. Применяется метод обобщения исходной задачи Solution, которая сводится к более общей задаче uniMat. Имеется параметр k – номер текущего столбца матрицы a. Формула nulColl вводит дополнительное предусловие в программе uniMat: в матрице a все элементы выше главной диагонали с индексами от k+1 до n равны нулю.

```
formula nulColl(nat k, MATR a) = forall j=k+1..n, i=1..j-1. a[i, j] = 0;
```

Программа uniMat последовательно обнуляет столбцы матрицы выше главной диагонали, начиная с последнего столбца.

```
uniMat(MATR a, VEC b, nat k:  VEC x )
pre  n > 0 & triangle(a) & nulCol(k, a)
post LinEq(a, b, x)
{        if (k = 1) x = b
         else { subtrCol(a, b, k : MATR c, VEC d);
                  uniMat(c, d, k-1:  x )
         }
}
```

Для очередного k–го столбца программа subtrCol обнуляет все элементы выше главной диагонали.

```
subtrCol(MATR a, VEC b, nat k: MATR a’, VEC b’)
pre  n > 0 & triangle(a) & nulCol(k, a)
post nulCol(k-1, a) & postLin(a, b, a’, b’)
{        b’ = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
         a' = a for (I1n i, j) {case (1..k-1, k): 0 }
}
```

4. Оптимизирующие трансформации

Эффективная императивная программа получается из предикатной программы применением системы оптимизирующих трансформаций. Определим набор трансформаций для программы решения системы линейных уравнений Lin.

На первом этапе применяются трансформации склеивания переменных, реализуемых применением команд вида x ← y, z, задающих замену всех вхождений переменных y и z на

переменную x. Корректность склеивания обеспечивается на основе потокового анализа программы [2].

На втором этапе проводится замена хвостовой рекурсии циклом **for**. Также реализуются упрощения. Результатом проведения двух этапов трансформации является следующая программа:

Склеивания: a ← c; b ← d, x;

```
Lin(MATR a, VEC b:  b : )
{       TriangleMatr(a, b:  a, b :  #2);
        Solution(a, b:  b ) #1
}
```

Склеивания: a ← a′; b ← b′;

```
TriangleMatr(MATR a, VEC b:  a, b :  );
{       Jord(a, b, 1:  a, b #1:  #2) }
```

Склеивания: a ← c, e, g, a′; b ← d, f, h, b′;

```
Jord(MATR a, VEC b, nat k:  a, b : )
{       for(;;) {
                diagEl(a, b, k: a, b : #2);
                norm(k, a, b: a, b);
                subtrLines(k, a, b: a, b);
                if (k = n) #1
                else  k = k + 1
        }
}
```

Склеивания: a ← a′; b ← b′;

```
diagEl(MATR a, VEC b, nat k:  a, b : )
{       if (a[k, k] != 0) #1
        else perm(a, b, k, k+1: a, b #1 | #2)
}
```

Склеивания: a ← a′; b ← b′;

```
perm(MATR a, VEC b, nat k, m:  a,  b : )
{       for(;;) {
            if (m>n) #2
            else if (a[m, k] != 0) {
               b = b for (I1n j) {case k: b[m] case m: b[k]} ||
               a = a for (I1n i, j) {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
               #1
            } else  m = m + 1
        }
}
```

Склеивания: a ← a′; b ← b′;

```
norm(nat k, MATR a, VEC b:  a,  b)
{       b = b for (I1n j) {case k: b[k] / a[k, k] } ||
        a = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
}
```

Склеивания: a ← a′; b ← b′;

```
subtrLines(nat k, MATR a, VEC b:  a,  b)
{ b = b for (I1n i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
  a = a for (I1n i, j) { case (k+1..n, k): 0 case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
}
```

Склеивание: b ← x;

```
Solution(MATR a, VEC b:   b )
{       uniMat(a, b, n:  b ) }
```

Склеивания: a ← c; b ← d, x;

```
uniMat(MATR a, VEC b, nat k:   b )
{       for(;;) {
                if (k = 1) return;
                subtrCol(a, b, k : a, b);
                k = k - 1
        }
}
```

Склеивания: a ← a′; b ← b′;

```
subtrCol(MATR a, VEC b, nat k:  a,  b)
{       b = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
        a = a for (I1n i, j) {case (1..k-1, k): 0 }
}
```

На третьем этапе, после устранения рекурсии, проводится открытая подстановка тел программ на места вызовов программ. Подстановка программ реализуется в следующей последовательности:

```
perm → diagEl;
subtrLines, norm, diagEl → Jord → TriangleMatr → Lin;
subtrCol → uniMat → Solution → Lin;
```

Результатом указанной серии подстановок является следующая программа.

```
Lin(MATR a, VEC b:  b : )
{       for(nat k = 1; ; k=k+1) {
                if (a[k, k] != 0) #M1;
                nat m;
                for(m = k +1;;m = m + 1) {
                        if (m>n) #2;
                        if (a[m, k] != 0) break
                };
                { b = b for (I1n j) {case k: b[m] case m: b[k]} ||
                  a = a for (I1n i, j) {case (k, k..n): a[m, j]  case (m, k..n): a[k, j]}
                };
M1:             { b = b for (I1n j) {case  k: b[k] / a[k, k] }||
                  a = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
                };
                { b = b for (I1n i) {case  k+1..n: b[i] - b[k] × a[i, k]} ||
                  a = a for (I1n i, j) {case (k+1..n, k): 0
                                        case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
                };
                if (k = n) break
        }
        for(nat k = n; k != 1; k = k - 1) {
                b = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
                a = a for (I1n i, j) {case (1..k-1, k): 0 }
        }
        #1
}
```

На четвертом этапе раскрываются операции с массивами. Для формы вида a = a **for** каждая **case**–часть определяет независимый параллельный цикл. За исключением первых двух форм, реализующих обмен элементов. Там возникает групповой оператор присваивания, который раскрывается с помощью дополнительной переменной. Получаем итоговую программу на императивном расширении языка **P**:

```
Lin( MATR a, VEC b:  b : )
{       for(nat k = 1; ; k=k+1) {
                if (a[k, k] != 0) #M1;
                nat m;
                for(m = k +1;;m = m + 1) {
                        if (m>n) #2;
                        if (a[m, k] != 0) break
                }
                {  | b[k], b[m] | = | b[m], b[k] | ||
                   for (nat j= k..n) | a[k, j], a[m, j] | = | a[m, j], a[k, j] |
                }
M1:             {  b[k] = b[k] / c[k, k] ||
                   a[k, k] = 1 ||
                   for (nat j= k+1..n) a[k, j] = a[k, j] / a[k, k]
                }
                {  for (nat i= k+1..n) b[i] = b[i] - b[k] × a[i, k] ||
                   for (nat j= k+1..n, k) a[k, j] =0 ||
                   for (nat i= k+1..n, nat j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]
                };
                if (k = n) break
        }
        for(nat k = n step -1 to 2) {
                for (nat i=1..k-1) b[i] = b[i] - b[k] × a[i, k] ||
                for (nat i=1..k-1) a[i, k] = 0
        }
        #1
}
```

Такие операторы (отмеченные красным цветом), как обнуление k-го столбца, ранее в программах subtrLines и subtrCol, и присваивание a[k, k] = 1, являются избыточными, поскольку соответствующие элементы матрицы a не используются в дальнейших вычислениях. Удаление указанных фрагментов программы – нетривиальная трансформация, требующая специального потокового анализа на элементах матрицы.

Все циклы, кроме полученных устранением рекурсии, допускают параллельное исполнение. Наиболее значимым является оператор цикла:

```
for (nat i= k+1..n, nat j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]
```

Здесь проводится наибольший объем вычислений. Данный цикл допускает реализацию в форме матричного параллелизма.

5. Параллелизм предикатных программ

Распараллеливание программ не является самоцелью – оно применяется исключительно в целях повышения быстродействия программ.

Классом предикатных программ является класс программ-функций [8], не взаимодействующих с внешним окружением программы. В предикатных программах нет физического параллелизма, который возможен в автоматных программах [5, 9, 13], являющихся продолжением параллельных физических процессов, протекающих в разных точках пространства и осуществляющих обмен информацией. Но есть параллелизм другого рода, логической природы.

5.1. Источники параллелизма

Параллелизм предикатных программ имеется в языке P_0 [12], из которого построен язык предикатного программирования P [3]. Источником параллелизма в языке P_0 являются два оператора: параллельный оператор и оператор каррирования.

Параллельный оператор B(x: y) || C(x: z) допускает параллельное независимое исполнение подоператоров B(x: y) и C(x: z). При этом исполнение B(x: y) и исполнение C(x: z) никак между собой не взаимодействуют, поскольку наборы y и z не содержат общих переменных. Ввиду тождества:

$$\{B(x: y) \mid\mid C(x: z)\};\ F(y, z: u) \equiv F(B(x), C(x): u)$$

параллелизм присутствует при вычислении значений разных аргументов в вызове предиката или функции. В частности, параллелизм возможен при вычислении аргументов любой бинарной операции, например, плюс «+».

Если предикатная программа транслируется на язык Си или C++, параллельный оператор кодируется в виде последовательного оператора. В подавляющем большинстве случаев издержки на организацию параллельных процессов превышают выигрыш от распараллеливания.

Параллелизм простых операторов может быть частично учтен в архитектуре широких команд вычислительных систем «Эльбрус» [14]. Адекватная реализация параллельного оператора возможна лишь на уровне интегральных схем или его аналога ПЛИС (FPGA).

Вторым источником параллелизма (векторного и матричного) является оператор каррирования [12] – аналог лямбда-функции в языках функционального программирования. Оператор каррирования для массива записывается в виде конструкций: определение массива и определение частей массива. Оптимизирующая трансформация этих конструкций дает циклы с векторным или матричным параллелизмом.

Параллелизм, реализуемый параллельным оператором и определением массива, будем называть *логическим параллелизмом* в противовес физическому параллелизму в автоматных программах.

5.2. В классе программ-функций только логический параллелизм. Распараллеливания нет в предикатном программировании

Оператор суперпозиции, условный оператор, параллельный оператор и конструкция «определения массива» определяют базис языка P и являются естественными формами построения алгоритма [7]. Параллельный оператор и определение массива являются двумя формами логического параллелизма предикатных программ. Предикатная программа не нуждается в распараллеливании. Она изначально параллельна. Задача реализации в том, чтобы донести этот параллелизм до конченой программы, получаемой в результате применения набора оптимизирующих трансформаций.

Универсальность предикатного программирования можно сформулировать следующим тезисом. Если для некоторой математической задачи, постановку которой можно формализовать в виде предусловия и постусловия, существует алгоритм решения задачи, то этот алгоритм можно записать в виде предикатной программы.

Апробация предикатного программирования на большом числе задач показала, что построение предикатных программ не только потенциально возможно. Технология предикатного программирования имеет существенные преимущества перед традиционной технологией императивного программирования.

Справедлив другой тезис. Для всякой математической задачи, постановку которой можно формализовать в виде предусловия и постусловия, и для всякой императивной программы, которая решает данную задачу, существует предикатная программа, решающая данную задачу, причем императивная программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций.

Отсюда следует чисто математическая природа всех программ из класса программ-функций, т.е. класса невзаимодействующих программ. Всякой программе соответствует некоторый предикат, т.е. вычислимая логическая формула. Эта формула композируется из базисных операторов языка P_0. Среди композиций – параллельный оператор и определение массива. Другого параллелизма нет. При этом фрагменты программ, которые могут исполняться параллельно, между собой не взаимодействуют.

Итак, логический параллелизм полностью характеризует параллелизм не только предикатных, но и императивных программ для всего класса задач дискретной и вычислительной математики.

5.3. Параллельное и конкурентное программирование

Мы противопоставляем логический и физический параллелизмы программ. Логический параллелизм реализуется в классе программ-функций. Это самый большой класс программ, содержащий, в частности, программы для задач дискретной и вычислительной математики.

Физический параллелизм может присутствовать в программах класса программ-процессов [8] (или реактивных систем), реализующих постоянное взаимодействие с окружением программы. Этот класс также называется классом *автоматных программ*. Важнейшим подклассом является класс контроллеров систем управления. Исполнение автоматной программы и ее параллельных подпрограмм инициируется различными физическими процессами через сообщения и датчики в окружении программы. Разные физические процессы обычно протекают параллельно. Соответственно, параллельно исполняются различные инициируемые ими подпрограммы. Эти подпрограммы могут обращаться к общим данным, что может стать причиной конфликта по доступу.

Если предикатной программе, а также императивной программе для класса задач вычислительной математики соответствует предикат, то автоматной программе соответствует конечный автомат [5, 9, 13]. Различная природа предикатных и автоматных программ предопределяет разные технологии программирования для этих классов программ. Язык автоматного программирования универсален и позволяет закодировать любой алгоритм. Но не следует использовать его для программ-функций – программа станет существенно сложнее. В частности, не рекомендуется программировать в автоматном стиле чисто вычислительные блоки, появляющиеся в составе контроллеров систем управления. Различные особенности технологии автоматного программирования описаны в работе [5, разд.4].

Параллельное программирование для автоматных программ – неудачный и неадекватный перевод для concurrent programming. Это вызывает путаницу, в частности, является причиной того, что параллельное программирование для вычислительной математики и concurrent programming рассматриваются вместе под общим заголовком «в области параллельного и распределенного программирования» (стр.24) в Программе фундаментальных исследований:

http://www.ras.ru/FStorage/Download.aspx?id=62d335ba-2aea-4803-85ee-fd0cd37aba4b

В последнее время, наряду с термином «распределенное программирование» часто используется «конкурентное программирование».

6. Сравнение с подходами потокового параллелизма

Кризис программирования 1960-ых гг. не был успешно преодолен. Появившиеся новые технологии и языки программирования лишь смягчили остроту этого кризиса. В этом причина затянувшейся стагнации в области программной инженерии. Особая острота кризиса программирования в параллельном программировании. Большое число архитектур вычислительных машин, языков программирования и технологий. Высокая сложность и трудоемкость программирования, проблемы надежности. Дороговизна разработки новых архитектур. Все это отмечалось в многочисленных обзорах по параллельному программированию.

В обзоре [4] рассматриваются языки и системы потокового параллелизма с ориентацией на несколько разных платформ. Как средство решения проблем эффективности и удобства программирования поставлена задача построения единого универсального максимально абстрактного языка для математического описания алгоритмов, с которого можно автоматически (с возможными подсказками) транслировать в разные языки существующих платформ.

Дополнительный интерес к работе [4] вызван рассмотрением там алгоритма разложения Холецкого для вычисления квадратного корня матрицы. Схема этого алгоритма во многом похожа на схему Гаусса-Жордано, используемую в нашей работе. А это значит, что алгоритм Гаусса-Жордано можно было бы записать на языке UPL(G) [4] в виде асинхронно-потоковой реализации аналогично алгоритму разложения Холецкого в работе [4]. В асинхронно-потоковой модели результаты промежуточных вычислений не записываются в память, а доставляются непосредственно аргументам для всех операторов программы, где эти результаты далее используются. Памяти нет – данные прикрепляются непосредственно к аргументам операторов. Оператор срабатывает лишь при поступлении всех его аргументов.

Интригующим было бы сравнение алгоритма Гаусса-Жордано с применением матричного параллелизма и асинхронно-потоковой реализации этого алгоритма на языке UPL(G). Однако, несмотря на многочисленные попытки в мире, до сих пор нет успешной асинхронно-потоковой архитектуры.

Эффективность алгоритма Гаусса-Жордано зависит главным образом от эффективности реализации цикла (см. конец разд. 4):

```
for (nat i= k+1..n, nat j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]
```

Предельная эффективность здесь достигается применением специализированного матричного процессора. Единственный способ достичь подобного результата в асинхронно-потоковой архитектуре, например для языков Пифагор [16, 17] или Sisal [15], это суметь положить параллельные списки Пифагора или потоки Sisal'а на матричный процессор. Использовать декларированные преимущества неявного параллелизма здесь не удастся. Очередной вычисленный элемент a[i, j] весьма проблематично упреждающе использовать на следующих итерациях цикла по k. В частности, необходимо гарантировать, что строка матрицы, содержащая a[i, j], не будет обмениваться. Кроме того, надо не попасть в цикл нормирования по диагональному элементу. Необходимо также обеспечить доступность элементов a[k, j] и a[i, k]. В принципе, подобное реализуемо применением изощренного потокового анализа, однако вряд ли обеспечит приемлемую эффективность.

7. Заключение

В настоящей работе в технологии предикатного программирования представлена реализация параллельной программы решения системы линейных уравнений по схеме Гаусса-Жордано. Без распараллеливания. Поскольку предикатная программа изначально параллельна благодаря наличию в языке P параллельного оператора и определения массива. Задача лишь в том, чтобы донести исходный параллелизм до конечной императивной программы.

В языке P не предусмотрено других специальных средств реализации распараллеливания. Разумеется, следует выбрать адекватный алгоритм, соответствующий эффективной реализации на целевой платформе. Например, для параллельного суммирования массива чисел необходимо выбрать подходящую модификацию каскадной схемы суммирования. Но при этом программа остается чисто предикатной, не отягощенной специальными конструкциями подобно языкам в работе [4], в частности, позволяет провести дедуктивную верификацию.

Реализация параллелизма проводится полностью на стадии оптимизирующей трансформации предикатной программы. Реализация с выходом на пакеты MPI и OpenMP не представляет особой сложности. Реализация для ПЛИС [18] и интегральных схем потребует применения более мощного и глубокого потокового анализа трансформируемой программы. В частности, необходимо будет реализовать протяжку диапазонов значений скалярных переменных [18]. Для реализации на ПЛИС выбирается соответствующий алгоритм. Предикатная программа определенным образом структурируется с ориентацией на ПЛИС.

При реализации на базе языка Си структуризация обычно реализуется автоматически. Однако в предикатном программировании оптимизации подобного рода возлагаются на программиста. В дальнейшем следует исследовать возможность реализации специальных структурирующих трансформаций.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. Т. 4 (33). С. 79-94.
2. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. — Новосибирск, 2017. — С. 21-48. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения: 22.10.2018).
3. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с., [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения: 22.10.2018).
4. Климов А.В. Обзор подходов к созданию мульти-платформенной среды параллельного программирования. Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции. М.: ИПМ им. М.В.Келдыша, 2017. С. 260-276. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://keldysh.ru/abrau/2017/19.pdf
5. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/lift1.pdf (дата обращения: 22.10.2018).
6. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
7. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. — Институт Математики СО РАН, Новосибирск, 2018. — С. 156-165. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf (дата обращения: 22.10.2018).
8. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения: 22.10.2018).
9. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. [Электронный ресурс]. Систем. требования:

Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/req_k.pdf (дата обращения: 22.10.2018).

10. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
11. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
12. Шелехов В.И. Семантика языка предикатного программирования // Знания-Онтологии-Теории (ЗОНТ-2015). — Новосибирск, 2015. — С 206-215. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения: 22.10.2018).
13. Шелехов В.И. Язык и технология автоматного программирования // «Программная инженерия», №4, 2014. – С. 3-15. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf (дата обращения: 22.10.2018).
14. Волконский В.Ю., Брегер А.В., Бучнев А.Ю., Грабежной А.В., Ермолицкий А.В., Муханов Л.Е., Нейман-заде М.И., Степанов П.А., Четверина О.А. Методы распараллеливания программ в оптимизирующем компиляторе / ЗАО «МЦСТ», 2013. 28с. URL: http://www.mcst.ru/metody-rasparallelivaniya-programm-v-optimiziruyushhem-kompilyatore (дата обращения: 23.11.2018).
15. Касьянов В. Н., Стасенко А. П. Язык программирования Sisal 3.2 // Методы и инструменты конструирования программ. Новосибирск: ИСИ СО РАН, 2007. С. 56– 134.
16. Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. – 2005. – № 1 (10). – С. 71–89.
17. Легалов А.И., Васильев В.С., Матковский И.В., Ушакова М.С. Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ // Труды Института системного программирования РАН, том 29, вып. 5, 2017, С. 165-184.
18. Cardoso J.M.P., Diniz P.C. Compilation Techniques for Reconfigurable Architectures. Springer. 2009. 230p.
19. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.

УДК 004.05

Дедуктивная верификация и оптимизация предикатной программы конкатенации строк

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Для любой программы на языке Си можно построить эквивалентную предикатную программу, провести её дедуктивную верификацию, применить к ней набор оптимизирующих трансформаций и в результате получить исходную программу на языке Си.

Данный метод иллюстрируется для известной библиотечной программы конкатенации строк strcat. Описывается построение, дедуктивная верификация и оптимизирующая трансформация предикатной программы конкатенации строк как объектов алгебраического типа «список» в языке предикатного программирования. Разработан аппарат сканирования списков и новый метод кодирования списков через массивы.

Анализируется возможность применения технологии предикатного программирования для дедуктивной верификации программ на языке Си.

***Ключевые слова**: дедуктивная верификация, трансформации программ, алгебраические типы данных, строки в языке Си.*

1. Введение

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Преимущество по времени верификации оценивалось примерно в 5 раз. Для программы быстрой сортировки с двумя опорными элементами зафиксировано преимущество в 10 раз [20]. Отметим чрезвычайно высокую сложность проведения формальной спецификации и дедуктивной верификации императивных программ.

В языке предикатного программирования P [8] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением *оптимизирующих трансформаций*. Они определяют отличную от

классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной [6, 7];
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 11, 15, 16, 22].

Программа принадлежит *классу программ-функций* [14], если она не взаимодействует с внешним окружением программы; точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы. Программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов. Предикатная программа относится к классу *программ-функций*.

Предикатное программирование универсально. Для всякой императивной программы, принадлежащей классу программ-функций и решающей некоторую математическую задачу, можно построить эквивалентную предикатную программу. При этом императивная программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций.

Необходимость дедуктивной верификации библиотечных программ обычно требуется в случае, когда они вызываются в составе программ, для которых проводится дедуктивная верификация. В нашем случае, программа конкатенации строк `strcat` вызывается в модуле безопасности Linux Security Modules (LSM) ядра ОС Linux. Для программы LSM реализуется дедуктивная верификация на соответствие собственной спецификации, а также специально разработанной модели политики безопасности [3]. Дедуктивная верификация программы LSM необходима для сертификации ОС Astra Linux Special Edition [9] высоким уровнем доверия в соответствии со стандартом ГОСТ Р ИСО/МЭК 15408-3 [2].

Для программы strcat на языке Си построена эквивалентная предикатная программа. Процесс ее дедуктивной верификации описан в Приложении 1. Доступно доказательство формул корректности [5] в системе PVS. Применение набора оптимизирующих оптимизаций к предикатной программе дает императивную программу, эквивалентную исходной на языке Си.

Строки определены как частный случай алгебраического типа «список» в языке P [8]. Разработан новый метод кодирования списков через массивы с использованием двух указателей, а также аппарат сканирования списков.

Во втором разделе дается краткое описание языка предикатного программирования. Метод дедуктивной верификации описывается в третьем разделе. В четвертом разделе определено построение предикатной программы strcat. В следующем разделе описывается процесс оптимизирующей трансформации программы. Аппарат сканирования списков представлен в шестом разделе. В седьмом разделе анализируется причина хороших показателей дедуктивной верификации в предикатном программировании. В восьмом разделе рассматриваются особенности сертификации предикатных программ. В девятом разделе описывается опыт проектов по дедуктивной верификации библиотек на языке Си. В заключении суммируются результаты работы.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [8] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }
```

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [11, 15, 22]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> **else** <оператор2>
<имя программы>(<список аргументов>**:** <список результатов>)
<тип> <пробел> <список имен переменных>

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **struct**(T_1 f_1, ..., T_n f_n) является *структура* из n значений, именуемых *полями* f_1, f_2,..., f_n и имеющих типы T_1, T_2,..., T_n, соответственно.

Значением *алгебраического типа* **union**(K_1, …, K_n) является один из конструкторов K_1, K_2, …, K_n. *Конструктор* K(T_1 f_1, ..., T_m f_m) определяется *именем конструктора* K и набором полей как у структуры; набор полей может быть пустым. Для объекта s алгебраического типа *распознаватель* K?(s) является истинным, если объект s определяется как конструктор вида K. Алгебраический тип может быть рекурсивно определяемым.

Типичными объектами алгебраических типов являются списки и деревья. *Список* определяет последовательность элементов некоторого произвольного типа T, являющегося параметром. Описание типа списка следующее:

```
type list (type T) = union( nil,  cons(T car, list(T) cdr) );
```

Алгебраический тип list имеет два конструктора: nil, обозначающий пустой список, и cons, определяющий список, первый элемент которого представлен полем car, а остальная часть списка («хвост» – все элементы, кроме первого) определяется полем cdr. Тип list считается определенным в языке P и не требует описания в программе.

Пусть s – переменная типа список. Тогда s.car определяет первый элемент – «голову» списка s, s.cdr – список без первого элемента – «хвост» списка s. Распознаватель nil?(s) определяет принадлежность списка s конструктору nil, т.е. пустоту списка. Вместо nil?(s) обычно используется s = nil.

Для списков s и u определены операции: len(s) – длина списка, s + u – конкатенация списков s и u. В качестве операндов конкатенации допускаются также элементы списка.

При трансляции предикатной программы в императивный язык основным способом представления списка является массив. Другими возможными альтернативами кодирования списка являются: односвязный список, двунаправленный список, кольцевой список. В языке

P введены дополнительные конструкции, обеспечивающие эффективную реализацию списков через массивы [19].

3. Дедуктивная верификация

Предикатная программа относится к классу *программ-функций* [14]. Программа-функция должна всегда **нормально завершаться** с получением результата, поскольку бесконечно работающая и невзаимодействующая программа бесполезна.

Спецификацией предикатной программы H(x**:** y) являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Спецификация записывается в виде: [P(x), Q(x, y)].

Для языка **P₀** построена формальная операционная семантика $\mathcal{R}$(H)(x, y) и доказано тождество $\mathcal{R}$(H) = H [17]. На базе языка **P₀** последовательным расширением и сохранением тождества $\mathcal{R}$(H) = H построен язык предикатного программирования **P** [8].

Тотальная корректность программы относительно спецификации определяется формулой:

$$H(x: y)\ \mathbf{corr}\ [P(x), Q(x, y)] \cong \forall x.\ P(x) \Rightarrow [\forall y.\ H(x: y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ H(x: y)$$

Формулу тотальной корректности будем представлять в виде правила **COR**:

$$\mathbf{COR}:\ \frac{\forall x,y.\ P(x)\ \&\ H(x: y) \Rightarrow Q(x, y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x: y)}{H(x: y)\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [10], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности. Корректность правил доказана [4] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Часть формул доказывается автоматически SMT-решателем CVC4. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [24]. Данный метод опробован для дедуктивной верификации более сотни программ [11, 15, 18, 22].

Предположим, что наборы переменных x, y и z не пересекаются, а x может быть пустым. Ниже приведены некоторые правила доказательства корректности операторов.

$$\mathbf{QP}: \frac{B(x\!: y)\ \mathbf{corr}\ [P(x), Q(x, y)];\ C(x\!: z)\ \mathbf{corr}\ [P(x), R(x, z)];}{\{B(x\!: y)\ ||\ C(x\!: z)\}\ \mathbf{corr}\ [P(x), Q(x, y)\ \&\ R(x, z)]}$$

$$\mathbf{QC}: \frac{\begin{array}{l} B(x\!: y)\ \mathbf{corr}\ [P(x)\ \&\ E(x), Q(x, y)]; \\ C(x\!: z)\ \mathbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x, y)] \end{array}}{\{\mathbf{if}\ (E(x))\ B(x\!: y)\ \mathbf{else}\ C(x\!: y)\}\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

Далее следует правило для частного случая оператора суперпозиции, соответствующего *сведению к более общей задаче* C(x, z**:** y).

$$\mathbf{RB}: \frac{\begin{array}{l} \forall z\ C(x, z\!: y)\ \mathbf{corr^*}\ [P_C(x, z), Q_C(x, y)]; \\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x)); \\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ ); \end{array}}{C(x, B(x)\!: y)\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

Запись вида z = B(x) является эквивалентом B(x**:** z). Истинность трех посылок правила **RB** гарантирует корректность следующей программы:

H(x**:** y) **pre** P(x) **post** Q(x, y) { C(x, B(x)**:** y) }

В случае рекурсивного вызова C(x, B(x)**:** y) обозначение **corr*** означает, что первая посылка опускается, а $P^*_C(x, B(x))$ заменяется на $P_C(x, B(x))\ \&\ m(x) < m(v)$. Здесь m – натуральная функция *меры*, строго убывающая на аргументах рекурсивных вызовов, а v обозначает аргументы рекурсивной программы C.

4. Построение программы конкатенации строк

4.1. Постановка задачи

Имеется следующая программа конкатенации строк на языке Си:

```
char *strcat(char *dest, const char *src)
{
	char *tmp = dest;
	while (*dest)
		dest++;
	while ((*dest++ = *src++) != '\0');
	return tmp;
}
```

Строка по указателю src добавляется к строке по указателю dest. Предполагается, что указатель dest ссылается на участок памяти достаточного размера для результата strcat.

Требуется построить соответствующую предикатную программу, провести ее дедуктивную верификацию и трансформировать ее в эффективную императивную программу, эквивалентную приведенной выше.

4.2. Предикатная программа конкатенации строк

Построим предикатную программу конкатенации строк dest и src. Результат конкатенации – итоговое значение переменной dest. В языке P конкатенация строк определена операцией «+» [8, разд. 7.3]. Реализация строк в трансляторе должна быть поддержана специальной библиотекой. Поэтому программа проста:

```
strcat(string dest, src: string dest′) post dest′ = dest + src
{ dest′ = dest + src };
```

Здесь штрих в имени dest′ означает, что в итоговой императивной программе переменные dest и dest′ должны быть склеены: dest′ заменяется на dest.

Исходная программа на языке Си (разд. 4.1) соответствует библиотечной программе для реализации строковой операции «+». Нам предстоит построить соответствующую предикатную программу с реализацией строкового типа через тип списка list.

Строковый тип **string** является предопределенным в языке P. Его определение имеет вид:

```
type string = list(char);
```

Для произвольного объекта типа **string** используется традиционное представление в виде массива литер, завершающегося нулем, причем нуль не входит в значение строки. Дадим точное определение типа **string** для данного стандартного представления. Для строки s через sval обозначим *собственно значение* строки без завершающего нуля. Предикат nozero(sval) постулирует, что в значении строки недопустимо использование нуля. Далее операция «+» везде является операцией конкатенации для списков, реализуемая в библиотечной поддержке языка P.

```
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or sval.car≠zero & nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
```

В приведенной последовательности описаний строковый тип определяется как множество последовательностей литер, не содержащих нуля и дополненных нулем в конце.

Проверка строки s на пустоту реализуется операцией s.car = \0, а не s = nil. В итоге, типы list и **string** несовместимы: со строковым объектом нельзя работать как со списком, в

частности, нельзя подставлять строковый объект параметром типа list. Библиотеки для списков неприменимы для строковых объектов.

В представленной ниже предикатной программе strcat, реализующей конкатенацию строк dest и src, сначала для первой строки dest выделяется ее собственно значение s, после чего реализуется конкатенация списков s и src.

```
strcat(string dest, src: string dest’)
post ∃ listchar sval. isVal(s, sval) & dest’ = sval + src
{   strval(dest: listchar s);
    listcat(s, src: dest’)
};
```

Программа strval, представленная здесь как библиотечная, определяет собственно значение s для строки dest.

```
strval(string dest: listchar s) post dest = s + zero;
```

Реализация программы strval в виде предикатной программы описана ниже в разд. 6.

Программа listcat реализует оператор dest = s + src, где «+» – операция конкатенации списков. Учитывая особенности представления строк, здесь нужна предикатная программа, приведенная ниже.

```
listcat(listchar s, string src: string dest) post dest = s + src measure len(src)
{   listchar s1 = s + src.car;
    if (src.car = zero) dest = s1 else listcar(s1, src.cdr: dest)
};
```

Итак, полная предикатная программа состоит из программ strcat и listcat. Программа strval представлена в разд. 6.

5. Трансформации

Определим набор трансформаций, превращающих программу конкатенации строк, состоящую из программ strcat и listcat, в эффективную императивную программу. На первом этапе реализуется трансформация склеивания переменных. Например, операция склеивания dest ← dest’, s реализует замену всех вхождений переменных dest’ и s на переменную dest.

Склеивание: dest ← dest’, s.

```
strcat(string dest, src: dest)
{   strval(dest: dest);
   listcat(dest, src: dest)
};
```

Склеивание: dest ← s, s1.

```
listcat(listchar dest, string src: dest)
{   dest = dest + src.car;
    if (src.car = zero) dest = dest else listcar(dest, src.cdr: dest)
};
```

На втором этапе проводится замена хвостовой рекурсии циклом в программе listcat:

```
listcat(listchar dest, string src: dest) post dest = s + src
{  loop {
     dest = dest + src.car;
     if (src.car = zero) {dest = dest; break} else |dest, src| = |dest, src.cdr|
   }
};
```

Раскрывается групповой оператор присваивания: в результате остается оператор src = src.cdr. Удаляется оператор dest = dest.

```
listcat(listchar dest, string src: dest) post dest = s + src
{  loop {
     dest = dest + src.car;
     if (src.car = zero) break;
     src = src.cdr
   }
};
```

На третьем этапе программа listcat подставляется на место вызова в strcat.

```
strcat(string dest, src: dest)
{   strval(dest: dest);
    loop {
        dest = dest + src.car;
        if (src.car = zero) break;
        src = src.cdr
    }
};
```

На четвертом этапе трансформаций списки кодируются через массивы. Стандартный метод кодирования списков через массивы описан в работе [13, разд.5.4]. Здесь применяется новый метод: кодирование списков через указатели, отличающийся от кодирования односвязными списками [12].

Список s кодируется массивом литер. Работа с массивом реализуется с помощью двух указателей sF и sL:

```
type STR = *char;
STR sF, sL;
```

Указатель sF «смотрит» на начало массива и используется для чтения. Указатель sL ссылается на следующий элемент за последним элементом списка s и предназначен для записи.

Операции со списками языка **P** необходимо представить эквивалентными операциями для массивов через пару указателей. Ниже приведено представление операций и операторов программы strcat.

```
src.car    →     *srcF;
dest = dest + src.car   →     *destL = *srcF; destL++
src = src.cdr     →     srcF++;
s = strval(dest)    →   sF = destF; for(sL = destF;*sL != zero; sL++);
```

Представленный способ кодирования вызова библиотечной программы strval определяется в разд. 6. Итоговая программа представлена ниже. В ней destF заменено на dest, srcF – на src.

```
strcat(char *dest, const char *src)
{   char *destL;
    for(destL = dest;*destL != zero; destL++); //strval(dest: dest);
    loop {
       *destL = *src; destL++;                    //dest = dest + src.car;
       if (*src = zero) break;
       src++                                      //src = src.cdr
    }
};
```

Сопоставляем с исходной программой на языке Си:

```
char *strcat(char *dest, const char *src)
{
        char *tmp = dest;
        while (*dest)
                dest++;
        while ((*dest++ = *src++) != '\0');
        return tmp;
}
```

Полученная в результате трансформаций программа почти эквивалентна исходной. Отличие в том, что в исходной программе указатель src продвинут дальше на одну позицию. Другая особенность: при завершении программы destL «смотрит» не на нуль, а на следующий за нулем. Это значит, что данный алгоритм не лучший для реализации серии конкатенаций.

6. Сканирование списков

Библиотечная программа strval, используемая при построении программы strcat для вычисления собственно значения строки, реализуется в виде предикатной программы применением *механизма сканирования списков*. *Сканирование* непустого списка a реализуется продвижением по нему другого списка b таким образом, что a = b + c для некоторого остатка – списка c. Сначала применяется операция **init**, в результате которой b = nil и c = a. Далее применяется операция **step**, которая начальный элемент списка c перемещает в конец списка b. Определение операций **init** и **step** представлено ниже в виде предикатных программ.

```
init(list(T) a: list(T) b, c) pre  a ≠ nil post b = nil & c = a
{ b = nil || c = a };

step(list(T) a, b, c: list(T) b′, c′)
pre a = b + c & c ≠ nil post b′ = b + c.car & c′ = c.cdr
{ b′ = b + c.car || c′ = c.cdr };
```

Программа strval реализуется сканированием вида dest = s + d, завершающимся при d = zero. Чтобы упростить программу, определим переменную dest глобальной. Результатом программы strval является переменная s – собственно значение строки dest.

```
string dest;
strval( : listchar s) post dest = s + zero
{ scan(init(dest): s) };
```

Программа scan является обобщением программы strval. Она реализует сканирование вида dest = s0 + d и завершается при d = zero.

```
scan(listchar s0, string d: listchar s)
pre dest = s0 + d post dest = s + zero measure len(d)
{ if (d.car = zero) s = s0 else scan(step(s0, d): s) };
```

Определим трансформации для программы strval. На первом этапе для программы scan проводится склеивание: s ← s0.

```
scan(listchar s, string d: s)
{ if (d.car = zero) s = s else scan(step(s, d): s) };
```

Далее хвостовая рекурсия в scan заменяется циклом. Удаляется оператор s = s. На месте рекурсивного вызова появляется групповой оператор присваивания |s, d| = step(s, d).

```
scan(listchar s, string d: s)
{ loop { if (d.car = zero) break; |s, d| = step(s, d) } };
```

На следующем этапе тело программы scan подставляется на место вызова в strval.

```
strval( : listchar s)
{  string d; |s, d| = init(dest);
   loop { if (d.car = zero) break;  |s, d| = step(s, d) }
};
```

Каждый список кодируется массивом литер и двумя указателями по схеме, описанной в разд. 5. Первый указатель «смотрит» на первый элемент списка, второй указатель указывает на элемент, непосредственно следующий за последним элементом списка. Определим описание переменных для указателей программы strval:

```
type STR = *char;
STR destF, sF, sL, dF;
```

Распределение указателей после очередного цикла работы программы strval показано на Рис.1. Список d размещается в памяти непосредственно после списка s. Для указателей реализуются следующие равенства: sF = destF и sL = dF. Учитывая это, далее вместо указателя dF будем использовать sL.

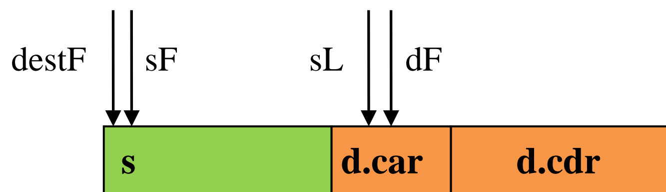

Рис. 1. Схема сканирования

Ниже приведено представление операций и операторов программы strval.

```
|s, d| = init(dest)   →   sF = destF; sL = destF;
|s, d| = step(s, d)  →  sL++;
d.car        → *sL;
```

Результат кодирования строковых операций представлен ниже.

```
strval(string dest: listchar s)
{   sF = destF; sL = destF;
   loop { if (*sL = zero) break;  sL++ }
};
```

Преобразование к циклу **for** дает итоговую программу:

```
strval(string dest: listchar s)
{ sF = destF; for (sL = destF; *sL != zero;  sL++); }
```

В дальнейшем, при трансформации предикатных программах вызов strval будет кодироваться данным фрагментом кода; см. кодирование s = strval(dest) в разд. 5.

7. Кривая логика императивных программ

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Преимущество по времени верификации более чем в 5 раз. Какова причина этого? Кривая логика императивных программ.

Программа из класса программ-функций [14] имеет две основы: логику и вычислимость, причем логика является первичной, главной. Предикатная программа – предикат, вычислимая логическая формула. Набор операторов языка **P0** [17], на базе которого строится язык **P**, определяется Таблицей 1 [17, разд. 3.2]:

H(x**:** y) ≅ ∃z. B(x**:** z) & C(z**:** y)	H(x**:** y) { B(x**:** z); C(z**:** y) }
H(x**:** y, z) ≅ B(x**:** y) & C(x**:** z)	H(x**:** y, z) { B(x**:** y) || C(x**:** z) }
H(x**:** y) ≅ (e ⇒ B(x**:** y)) & (¬e ⇒ C(x**:** y))	H(x**:** y) { **if** (e) B(x**:** y) **else** C(x**:** y) }
H(x**:** y) ≅ B(x~**:** y)	H(x**:** y) { B(x~**:** y) }
H(A, x**:** y) ≅ A(x**:** y)	H(A, x**:** y) { A(x**:** y) }
H(x**:** D) ≅ ∀y,z. D(y**:** z) ≡ B(x, y**:** z)	H(x**:** D) { D(y**:** z) {B(x, y**:** z)} }
H(A, x**:** D) ≅ ∀y,z. D(y**:** z) ≡ A(x, y**:** z)	H(A, x**:** D) { D(y**:** z) {A(x, y**:** z)} }

Таблица 1. Вычислимые предикаты и их программная форма.

Для предикатной программы используется программная форма в правой колонке. При этом каждому оператору предикатной программы непосредственным очевидным образом сопоставляется соответствующая логическая формула. Соответствие между предикатной программой и ее логикой простое и естественное.

Если к программе применяется оптимизирующее преобразование, то ее логика не исчезает. Она усложняется, искривляется. Усложняется также соответствие между логикой и программой. Поскольку императивная программа получается из эквивалентной предикатной программы применением серии оптимизирующих трансформаций, то искривление логики императивной программы довольно существенное. Искривляется также и формальная спецификация императивной программы по сравнению со спецификацией соответствующей предикатной программы. В частности, инварианты циклов принципиально сложнее предусловий соответствующих рекурсивных программ, преобразуемых в циклы.

Сложность формальных спецификаций все чаще становится предметом обсуждения в последних работах. В качестве одного из способов решения проблемы предлагается приблизить язык спецификаций к языку программирования [23].

8. Особенности сертификации

Сертификация программ, разработанных с применением дедуктивной верификации, налагает дополнительные требования на технологию программирования и сопутствующие инструменты. Определим набор действий *оценщика* – эксперта, которому в соответствии с правилами стандарта ГОСТ Р ИСО/МЭК 15408-3 [2] надлежит сертифицировать программы, разработанные по технологии предикатного программирования.

Например, для сертификации программы strcat, представленной в данной статье, объектом оценки стал бы материал разделов 4–6 и Приложения 1. То есть описание построения предикатной программы, ее оптимизирующей трансформации и дедуктивной верификации. Для удобства анализа в Приложении 1 показывается каждое применяемое правило доказательства корректности вместе с его конкретизацией. Дополнительно требуется свидетельство того, что применяемое правило корректно, т.е. присутствует в списке правил, доказанных [4] в системе PVS.

После разработки производственной системы предикатного программирования, появятся требования к ее корректности и предоставления набора свидетельств, оценка которых позволит определить уровень доверия к ней. Во-первых, необходимо свидетельство того, что набор правил, применяемых генератором формул корректности в подсистеме дедуктивной верификации, покрывается списком правил, доказанных [4] в системе PVS, вместе со свидетельством корректности работы самого генератора. Необходимы также свидетельства корректности трансляции с языка **P**, а также корректности реализации оптимизирующих трансформаций. Обеспечить такие два свидетельства чрезвычайно трудно. Разработка моделей трансляции и оптимизирующей трансформации, их анализ и верификация должны повысить уровень доверия к системе. Дополнительным артефактом оценки может стать набор листингов программ после каждой автоматически применяемой трансформации.

9. Другие работы по верификации библиотечных программ

В работе [21] описывается дедуктивная верификация 26 библиотечных функций языка Си, в основном для работы со строковыми объектами, в рамках международного проекта по верификации ядра ОС Linux. Используется стек инструментов дедуктивной верификации AstraVer [27] с выходом на большое число SMT-решателей. Доказательство генерируемых условий корректности функций реализуется исключительно применением SMT-решателей с использованием простых лемм, которые возможно доказываются применением интерактивных систем доказательства типа PVS [24]. Формальные спецификации были

написаны для исходного кода библиотечных функций, и лишь коды двух функций подвергнуты предварительной модификации. Результаты работы [21] существенно превосходят достижения других известных проектов, в частности, представленных работами [25, 26], по дедуктивной верификации библиотечных функций языка Си.

10. Заключение

В настоящей работе представлена технология предикатного программирования для построения, оптимизирующей трансформации и дедуктивной верификации библиотечной программы конкатенации строк strcat. Разработан новый способ кодирования списков через массивы с использованием двух указателей. Данный способ соответствует традиционному стилю работы со строками через указатели. Разработан аппарат сканирования списков, используемый для нахождения конца строки. Возможно, существуют другие решения по кодированию строковых объектов.

Дедуктивная верификация предикатных программ существенно проще и быстрее, чем дедуктивная верификация аналогичных императивных программ. В предикатном программировании затраты на формальную спецификацию и дедуктивную верификацию в 3-6 раз превышают затраты на разработку программы в традиционном стиле с применением тестирования. По сравнению с императивным программированием это принципиально расширяет возможности применения дедуктивной верификации в производственном программировании.

Чтобы обеспечить высокий уровень доверия к инструментам верификации, необходимо гарантировать корректность применяемой технологии и сопутствующих инструментов. В рамках третьего релиза транслятора с языка P разработана модель внутреннего представления транслируемой программы. В дальнейшем планируется формализовать эту модель и провести ее верификацию. На базе этой модели предполагается также построить модель оптимизирующих трансформаций.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. Т. 4 (33). С. 79-94.
2. ГОСТ Р ИСО/МЭК 15408-3-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности.

3. Девянин П.Н., Ефремов Д.В., Кулямин В.В., Петренко А.К., Хорошилов А.В., Щепетков И.В. Моделирование и верификация политик безопасности управления доступом в операционных системах // Коллективная монография, версия 1.3. ИСП РАН, 2018. 181с. [Электронный ресурс]. URL: http://www.ispras.ru/publications/security_policy_modeling_and_verification.pdf (дата обращения 12.11.2018)
4. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 12.11.2018)
5. Доказательство формул корректности программы strcat в системе PVS. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/strcat.zip (дата обращения 12.11.2018)
6. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. Новосибирск, 2017. С. 21-48. Электрон. журн. 2018. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения 12.11.2018)
7. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. Новосибирск, 2012. 6с. (Препр. / ИСИ СО РАН; N 167).
8. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с., [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения 12.11.2018).
9. Операционные системы Astra Linux. [Электронный ресурс]. URL: http://www.astralinux.ru (дата обращения 12.11.2018).
10. Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия». 2016. № 5. С. 202-210. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf (дата обращения 12.11.2018).
11. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
12. Шелехов В.И. Дедуктивная верификация и реализация предикатной программы инвертирования односвязных списков. ИСИ СО РАН, Новосибирск, 2018. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/listinvert.pdf (дата обращения 12.11.2018).
13. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. Институт Математики СО РАН, Новосибирск, 2017. С. 156-165. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf (дата обращения 12.11.2018).
14. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. С. 531–538. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения 12.11.2018).
15. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).

16. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
17. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения 12.11.2018).
18. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы '2017», Ростов-на-Дону. 2017. С.258-262. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf (дата обращения 12.11.2018).
19. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, ИСИ СО РАН, Новосибирск. Электрон. журн. 2014, №3. С. 25-43. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/Listcharing.pdf (дата обращения 12.11.2018).
20. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами // Научный сервис в сети Интернет. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf (дата обращения 12.11.2018).
21. Ефремов Д.В, Мандрыкин М.У. Формальная верификация библиотечных функций ядра Linux. Труды ИСП РАН, том 29, вып. 6, 2017. С. 49-76. DOI: 10.15514/ISPRAS-2017-29(6)-3
22. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
23. Utting M., Pearce D.J., Groves L. Making Whiley Boogie! // Integrated Formal Methods. LNCS 10510, 2017. p. 69-84.
24. PVS Specification and Verification System. SRI International. [Электронный ресурс]. URL: http://pvs.csl.sri.com/ (дата обращения 12.11.2018).
25. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in // NFM 2014. LNCS 8430, 2014. P. 343–358.
26. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.
27. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. URL: http://linuxtesting.org/astraver, 15.10.2017 (дата обращения 30.11.2018)

Приложение 1.
Дедуктивная верификация программы strcat

Приведем полный набор определений, необходимых для верификации программы strcat. Сначала определим формулы для предусловий и постусловий.

```
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or sval.car≠zero & nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
formula Pstrcat(string dest, src) = Str(src) & Str(dest);
formula Qstrcat(string dest, src, dest') = ∃ listchar sval. isVal(dest, sval) & dest' = sval + src;
```

Программа strcat:

```
strcat(string dest, src: string dest') post Qstrcat(dest, src, dest')
{   strval(dest: listchar s);
    listcat(s, src: dest')
};

strval(string s: listchar s') post isVal(s, s');
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);
listcat(listchar s, string src: string dest) post Qlistcat(s, src, dest);
```

Следует учитывать, что для аргументов типа **string** в предусловие необходимо добавить предикат Str от аргумента. Аналогично, для результата типа **string** в постусловие добавляется предикат Str от результата. Это учтено в приведенных формулах для предусловий и постусловий.

Сначала применяется правило **RS** для оператора суперпозиции strval(dest: listchar s); listcat(s, src: dest'), составляющего тело программы strcat:

$$\mathbf{RS}:\ \frac{\begin{array}{l} B(x\!:\ z)\ \mathbf{corr}^*\ [P_B(x),\ Q_B(x, z)];\\ \forall z.\ C(x, z\!:\ y)\ \mathbf{corr}^*\ [P_C(x, z),\ Q_C(x, z, y)];\\ \forall z, y.\ (\ P(x)\ \&\ Q_B(x, z)\ \&\ Q_C(x, z, y) \rightarrow\ Q(x, y)\ )\\ \forall z.\ (\ P(x)\ \&\ Q_B(x, z) \rightarrow\ P^*_C(x, z)\ );\\ P(x) \rightarrow\ P^*_B(x); \end{array}}{B(x\!:\ z);\ C(x, z\!:\ y)\ \mathbf{corr}\ [P(x),\ Q(x, y)]}$$

Здесь B(x: z) соответствует strval(dest: listchar s), а C(x, z: y) – listcat(s, src: dest'). Ниже конкретизация правила **RS**.

strval(dest**:** s)B(x: z) **corr*** [Str(dest), isVal(dest, s)];
listcat(s, src**:** dest′) **corr*** [Plistcat(s, src), Qlistcat(s, src**,** dest′)];
Pstrcat(dest, src) & isVal(dest, s) & Qlistcat(s, src**,** dest′) → Qstrcat(dest, src**,** dest′)
Pstrcat(dest, src) & isVal(dest, s) → Plistcat(s, src);
RS: Pstrcat(dest, src) → Str(dest);
strval(dest**:** s); listcat(s, src**:** dest′) **corr** [Pstrcat(dest, src), Qstrcat(dest, src**,** dest′)]

Первая посылка определяет корректность программы strval, считающейся здесь предопределенной. Корректность программы listcat, определяемая второй посылкой, будет рассмотрена ниже. Посылки с третьей по пятую определяют следующие формулы корректности:

Pstrcat(dest, src) & isVal(dest, s) & Qlistcat(s, src**,** dest′) ⇒ Qstrcat(dest, src**,** dest′);
Pstrcat(dest, src) & isVal(dest, s) ⇒ Str(src);
Pstrcat(dest, src) ⇒ Str(dest);

Истинность двух последних формул очевидна. Первая формула будет доказана в PVS.

Рассмотрим программу listcat и построим для нее формулы корректности. Задание меры len(src) обязательно для рекурсивных программ.

```
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);
listcat(listchar s, string src: string dest) post Qlistcat(s, src, dest) measure len(src)
{   listchar s1 = s + src.car;
    if (src.car = zero) dest = s1 else listcar(s1, src.cdr: dest)
};
```

Тело программы listcat является оператором суперпозиции. В данном случае удобнее применить следующую модификацию правила **QS**:

P(x) → ∃z. B(x**:** z);
∀z. C(x, z: y) **corr** [P(x) & B(x**:** z), Q(x, y)];
QS: B(x: z); C(x, z: y) **corr** [P(x), Q(x,y)]

Приведем конкретизацию данного правила **QS**.

Plistcat(s, src) → ∃ s1. s1 = s + src.car;
if ... **corr** [Plistcat(s, src) & s1 = s + src.car, Qlistcat(s, src**,** dest)];
QS: s1 = s + src.car; **if** ... **corr** [Plistcat(s, src), Qlistcat(s, src**,** dest)]

Посылки правила **QS** определяют следующие формулы для listcat.

Str(src) ⇒ ∃s1. s1 = s + src.car;
if (src.car = zero) dest = s1 **else** listcar(s1, src.cdr**:** dest) **corr**
[Str(src) & s1 = s + src.car, Qlistcat(s, src**,** dest)];

Истинность первой формулы очевидна. Вторая формула определяет новую цель. Здесь применяется правило **QC** для условного оператора.

QC:

$$\frac{\begin{array}{l}B(x\!: y)\ \mathbf{corr}\ [P(x)\ \&\ E(x),\ Q(x, y)];\\ C(x\!: z)\ \mathbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x, y)]\end{array}}{\{\mathbf{if}\ (E(x))\ B(x\!: y)\ \mathbf{else}\ C(x\!: y)\}\ \mathbf{corr}\ [P(x),\ Q(x, y)]}$$

Конкретизация правила **QC**:

QC:

dest = s1 **corr** [Str(src) & s1 = s + src.car & src.car = zero, Qlistcat(s, src**,** dest)];
listcar(s1, src.cdr**:** dest) **corr**
[Str(src) & s1 = s + src.car & ¬ src.car = zero, Qlistcat(s, src**,** dest)]
———————————————————————————————
if (src.car = zero) dest = s1 **else** listcar(s1, src.cdr**:** dest) **corr**
[Str(src) & s1 = s + src.car, Qlistcat(s, src**,** dest)]

Посылки правила **QC** определяют следующие две цели:

dest = s1 **corr** [Str(src) & s1 = s + src.car & src.car = zero, Qlistcat(s, src**,** dest)]
listcar(s1, src.cdr**:** dest) **corr**
[Str(src) & s1 = s + src.car & src.car ≠ zero, Qlistcat(s, src**,** dest)]

Для первой цели применяется правило **COR**, соответствующее определению «**corr**».

COR:

$$\frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\!: y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\!: y)\end{array}}{H(x\!: y)\ \mathbf{corr}\ [P(x),\ Q(x, y)]}$$

Конкретизация правила **COR**:

COR:

Str(src) & s1 = s + src.car & src.car = zero & dest = s1 ⇒ Qlistcat(s, src**,** dest);
Str(src) & s1 = s + src.car & src.car = zero ⇒ ∃s1. dest = s1
———————————————————————————————
dest = s1 **corr** [Str(src) & s1 = s + src.car & src.car = zero P(x), Qlistcat(s, src**,** dest)]

Посылки правила определяет следующие формулы корректности:

Str(src) & s1 = s + src.car & src.car = zero & dest = s1 ⇒ Qlistcat(s, src**,** dest);
Str(src) & s1 = s + src.car & src.car = zero ⇒ ∃dest. dest = s1;

Истинность второй формулы очевидна. Первая формула будет доказана в PVS.

Для второй цели правила **QC** для рекурсивного вызова listcar(s1, src.cdr**:** dest) применим правило **RB**.

RB:

$$\frac{\begin{array}{l}\forall z\ C(x, z\!: y)\ \mathbf{corr^*}\ [P_C(x, z),\ Q_C(x, y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ );\end{array}}{C(x, B(x)\!: y)\ \mathbf{corr}\ [P(x),\ Q(x, y)]}$$

Здесь B(s, src) = (s1, src.cdr), предикат P_B = src ≠ nil. Первая посылка правила **RB** для программы listcar отсутствует ввиду рекурсивности listcar. Далее, предикат P^*_C(x, B(x)) определяется следующим образом: P^*_C(B(s, src)) = P_C(B(s, src)) & len(src.cdr)<len(src).

Конкретизация правила **RB**:

```
     listcat(s, src: dest) corr* [Str(src), Qlistcat(s, src, dest)];
     Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr) & len(src.cdr)<len(src);
     Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr, dest) ⇒ Qlistcat(s, src, dest);
RB:  ----------------------------------------------------------------------------------------------------
     listcar(s1, src.cdr: dest) corr
     [Str(src) & s1 = s + src.car & src.car ≠ zero, Qlistcat(s, src, dest)]
```

Ввиду рекурсивности listcar первая посылка опускается. Вторая и третья посылки правила **RB** определяют следующие формулы корректности:

```
Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr) & len(src.cdr)<len(src);
Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr, dest) ⇒
                                                                    Qlistcat(s, src, dest);
```

Рассмотрим верификацию программы strval. Предварительно проведем открытую подстановку для вызовов стандартных операций **init** и **step**.

```
formula Pscan(listchar s, string d, dest) = Str(d) & Str(dest) & dest = s + d;
formula Qstrcat(string dest, src, dest′) = ∃ listchar sval. isVal(dest, sval) & dest′ = sval + src;
strval(string dest: listchar s) post isVal(dest, s)
{ scan(nil, dest, dest: s) };
```

```
                         ∀z C(x, z: y) corr* [Pc(x, z), Qc(x, y)];
                         P(x) ⇒ PB(x) & P*C(x, B(x));
                         ∀y ( P(x) & Qc(B(x), y) ⇒ Q(x, y) );
                   RB:  ----------------------------------------
                            C(x, B(x): y) corr [P(x), Q(x, y)]
B(s0, d, dest) = |nil, dest, dest|
                 scan(s0, dest: s) corr* [Pscan(s0, d, dest), isVal(dest, s)];
                 Str(dest) ⇒ Pscan(nil, dest, dest);
                 ∀y (Str(dest) & isVal(dest, s) ⇒ isVal(dest, s) );
          RB:   ------------------------------------------------------
                     scan(nil, d, dest: s) corr [Str(dest), isVal(dest, s)]
```

Цель: scan(s0, dest: s) **corr*** [Pscan(s0, d, dest), isVal(dest, s)]

Применяется обобщение задачи

```
scan(listchar s0, string d, dest: listchar s)
pre Pscan(s0, d, dest) post isVal(dest, s) measure len(d)
{ if (d.car = zero) s = s0 else scan(s0 + d.car, d.cdr: s) };
```

QC:
B(x: y) **corr** [P(x) & E(x), Q(x, y)];
C(x: z) **corr** [P(x) & ¬E(x), Q(x, y)]
————————————————————
{**if (**E(x)) B(x**:** y) **else** C(x**:** y)} **corr** [P(x), Q(x, y)]

Конкретизация **QC**.

QC:
s = s0 **corr** [Pscan(s0, d, dest) & d.car = zero, isVal(dest, s)];
scan(s0 + d.car, d.cdr**:** s) **corr** [Pscan(s0, d, dest) & ¬ d.car = zero, isVal(dest, s)]
————————————————————
{ **if** (d.car = zero) s = s0 **else** scan(s0 + d.car, d.cdr**:** s)} **corr**
[Pscan(s0, d, dest), isVal(dest, s)]

Для первой посылки применяется определение **corr**.

COR:
∀x,y. P(x) & H(x**:** y) ⇒ Q(x, y);
∀x. P(x) ⇒ ∃y. H(x**:** y)
————————————————————
H(x**:** y) **corr** [P(x), Q(x, y)]

lemma COR1: Pscan(s0, d, dest) & d.car = zero & s = s0 ⇒ isVal(dest, s)
scan(s0 + d.car, d.cdr**:** s) **corr** [Pscan(s0, d, dest) & ¬ d.car = zero, isVal(dest, s)]
formula PSC(listchar s0, **string** d, dest) = Pscan(s0, d, dest) & ¬ d.car = zero;

RB:
∀z C(x, z**:** y) **corr*** [P_C(x, z), Q_C(x, y)];
P(x) ⇒ P_B(x) & P^*_C(x, B(x));
∀y (P(x) & Q_C(B(x), y) ⇒ Q(x, y));
————————————————————
C(x, B(x)**:** y) **corr** [P(x), Q(x, y)]

Здесь B(s0,d,dest) = | s0 + d.car, d.cdr, dest |, P_B(s0,d,dest) = cons?(d).

Конкретизация **RB**.

RB:
∀z C(x, z**:** y) **corr*** [P_C(x, z), Q_C(x, y)];
PSC(s0, d, dest) ⇒ cons?(d) & Pscan(s0 + d.car, d.cdr, dest) & len(d.cdr)<len(d)
∀y (PSC(s0, d, dest) & isVal(dest, s) ⇒ isVal(dest, s));
————————————————————
scan(s0 + d.car, d.cdr**:** s) **corr** [PSC(s0, d, dest), isVal(dest, s)]

lemma RB22: PSC(s0, d, dest) ⇒
cons?(d) & Pscan(s0 + d.car, d.cdr, dest) & len(d.cdr)<len(d)

В итоге получаем следующую теорию для доказательства в системе интерактивного доказательства PVS.

```
theory strcat {
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
formula Pstrcat(string dest, src) = Str(src) & Str(dest);
formula Qstrcat(string dest, src, dest′) = ∃ listchar sval. isVal(dest, sval) & dest′ = sval + src;
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);

lemma QS3: Pstrcat(dest, src) & isVal(dest, s) & Qlistcat(s, src, dest′) ⇒ Qstrcat(dest, src, dest′);
lemma COR: Str(src) & s1 = s + src.car & src.car = zero & dest = s1 ⇒ Qlistcat(s, src, dest);
lemma RB2: Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr);
lemma RB3: Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr, dest) ⇒
                                                                 Qlistcat(s, src, dest);

formula Pscan(listchar s, string d, dest) = Str(d) & Str(dest) & dest = s + d;
formula PSC(listchar s0, string d, dest) = Pscan(s0, d, dest) & ¬ d.car = zero;
lemma COR1: Pscan(s0, d, dest) & d.car = zero & s = s0 ⇒ isVal(dest, s)
lemma RB22: PSC(s0, d, dest) ⇒ cons?(d) & Pscan(s0 + d.car, d.cdr, dest) & len(d.cdr)<len(d)
}
```

Последние четыре строки получены при верификации программы strval.

В этой теории необходимо доказать четыре леммы, соответствующие полученным выше формулам корректности. Спецификации этой теории пришлось существенно модифицировать, чтобы они смогли пройти семантический контроль при трансляции в системе PVS. Ниже приведена итоговая теория на языке PVS.

```
strcat: THEORY BEGIN
  char: TYPE = below[256];
  zero: char = 0;
  listchar: TYPE = list[char];
  s0, s, s1, sval: VAR listchar

  val(s, sval):bool = s = append(sval, cons(zero,null));
  nozero(sval): RECURSIVE bool = CASES sval OF
        null: TRUE, cons(x,y): NOT (x = zero) & nozero(y) ENDCASES MEASURE length(sval);
  Str(s):bool = EXISTS sval: val(s, sval) & nozero(sval);
  string: TYPE = {s:listchar | Str(s)};

  d, dest, dest9, src: VAR string

  Pstrcat(dest, src):bool = Str(src) & Str(dest);
  Qstrcat(dest, src, dest9):bool = EXISTS sval: val(dest, sval) & dest9 = append(sval, src);
  Plistcat(s, src):bool = Str(src);
  Qlistcat(s, src, dest):bool = dest = append(s,src) & Str(dest);
  Pscan(s0, d, dest):bool = Str(d) & Str(dest) & dest = append(s0, d);
  PSC(s0, d, dest):bool = Pscan(s0, d, dest) & NOT (car(d) = zero);

  Notnil:    LEMMA Str(src) IMPLIES cons?(src);
  Empstr:    LEMMA car(src) = zero IMPLIES src = cons(zero, null)
  Notemp:  LEMMA src = append(s, cons(zero, null)) & NOT car(src) = zero IMPLIES cons?(s)
  appCons: LEMMA cons?(src) IMPLIES append(cons(car(src), null), cdr(src)) = src

  QS3: LEMMA  Pstrcat(dest, src) & val(dest, s) & Qlistcat(s, src, dest9)
                                                 IMPLIES  Qstrcat(dest, src, dest9);
  COR: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & car(src) = zero & dest = s1
                                                         IMPLIES Qlistcat(s, src, dest);
  RB2: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & NOT car(src) = zero
                           IMPLIES cons?(src) & Str(cdr(src)) & length(cdr(src)) < length(src);
  RB3: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & NOT car(src) = zero &
                                    Qlistcat(s1, cdr(src), dest)     IMPLIES Qlistcat(s, src, dest);
  COR1: LEMMA  Pscan(s0, d, dest) & car(d) = zero & s = s0 IMPLIES val(dest, s)
  RB22: LEMMA PSC(s0, d, dest) IMPLIES cons?(d) &
                     Pscan(append(s0, cons(car(d), null)), cdr(d), dest) & length(cdr(d))<length(d)
END strcat
```

В этой теории введены дополнительные простые леммы Notnil, Empstr, Notemp и appCons для упрощения доказательства основных формул корректности. Доступно доказательство [5] всех лемм данной теории, включая TCC, в системе PVS:

http://persons.iis.nsk.su/files/persons/pages/strcat.zip

УДК 004.432:004.054

Reflex Language: a Practical Notation for Cyber-Physical Systems

Liakh T.V., Rozov A.S., Zyubin V.E.

(Institute of Automation and Electrometry SB RAS,

Novosibirsk State University)

This paper introduces a conceptual framework for complex control algorithms in form of hyperprocess model. To demonstrate the practical value of the model we describe grammar and translational semantics of a process-oriented language, known as Reflex or "C with processes". Expressive properties of the presented notation are shown on an example control algorithm design for a hand dryer device. Finally, we give a short report about practical application of the language and results, which have been obtained during its usage.

Keywords: Reflex language, control software, IEC 61131-3, process-oriented programming, POP, programmable logic controller, PLC, cyber-physical system, hybrid system, finite state machine, FSM, finite state automaton, FSA, controller, plant, syntax, semantics, industrial automation, reactive system

1. Introduction

Industrial control systems are nowadays most commonly implemented using programmable logic controllers (PLCs). A typical PLC consists of a central control board hosting the processor, in bundle with multiple extension modules that provide digital I/O, analog-to-digital and digital-to analog conversions, etc. All of control logic, including timing, computation, continuous control and peripheral communication, is specified in the PLC software. Thus in control systems, software development expenses surpass those of hardware development and constitute the larger part of the total project cost.

The specificity of control algorithms calls for specialized design patterns, languages and models to be utilized in development of PLC software. Numerous theoretical studies have been published, with design patterns, models and methods (e. g. TCSP, CCS, I/O-Automata, TLA, FSM and their timed extensions [1–12], to name a few) that address the characteristic issues arising in control software development. Despite this diversity of approaches available, the industry still favours IEC 61131-3 [13] standard languages as their primary and ultimate solution

in the field. It incorporates languages for relay logic (LD), functional block diagrams (FBDs), sequential function charts (SFCs) and assembler-like specifications (IL). However, as the complexity of control software increases and quality is of higher priority, the 35 years old technology underlying the IEC 61131-3 approach is not able to address the present-day requirements [14]. The IEC standard is ambiguous, universally critisized for its lack of expressiveness and is even cited as an example of standardization misuse [15]. The shortcomings of the IEC 61131-3 languages have become particularly evident after new types of cyber-physical systems started to emerge and rapidly spread, such as embedded systems, smart-devices, IoT-devices, etc. This motivates researchers to enrich the IEC 61131-3 development model with OO concepts [16], or rather develop alternative approaches, e. g. [17, 18].

The idea of designing special mathematical models for specification of control software is not new. A large amount of related work has been published over the years. Unfortunately, the effort was distributed among different fields: reactive systems, programming, logic, parallel system, homeostatic systems, so called real time systems, robotics, etc., see e. g. [1, 4, 5, 9, 19] Though meany of the presented ideas look promising, these theoretical results find little to no use in the industry. Attaining a practical solution for control system specification requires a holistic approach.

We strongly believe that a useful notation for control software can be designed over a thorough analysis of the target task. The models and concepts underlying such a notation would have to reflect the key features of control systems. Further, the notation itself should be comprehensible and easy-to-use for the programmers already engaged in the industry. It therefore needs to conform with the current trends in programming languages. In this work we introduce a programming language based on this idea.

The paper is organized as follows: in Section 2 we analyze the domain of control systems and extract the key features of control software. Section 3 introduces the hyperprocess mathematical model that reflects the aforementioned features. Section 4 presents the syntax of Reflex programming language that is based on the hyperprocess. Section 5 shows semantics of the language. Section 7 concludes the paper and lines out directions for our future work.

This work has been supported by the Federal Agency for Scientific Organizations (project AAAA-A17-117060610006-6) and the Russian Foundation for Basic Research (grant 17-07-01600).

2. Specific Features of Control Software

When considering a modern control system, we generally picture a digital controller connected to a controlled plant. The plant represents external environment of control system and consists of hardware and equipment within which physical processes take place. The plant and controller are connected via sensors and actuators. Sensors read data from the plant and pass it to the control system. The controller then acts, or rather reacts on this input by producing control values for the actuators, which in turn alter the flow of the physical processes in the plant.

Input from the sensors is supplied continuously. The controller detects events within the data-flow and consequently reacts on them with accordance to its algorithm. Therefore, unlike transformational software, control algorithms operate indefinitely, which for digital controllers automatically implies cyclic execution. A general control algorithm is thus presented with the following pattern: input of data about current state of controlled plant from sensors – data processing and determination of required reaction – and data output, which changes current state of the actuators and consequently the state of the plant.

Technological processes tend to involve multiple stages and the way control software reacts to events, needs to change over time. Control algorithms have polymorphic behavior which cannot be defined by the knowledge of inputs alone, but depends on a history of events.

As the plants are dynamical systems, control software has to function time-dependently, i. e. accordingly to the plant dynamics. This means that control algorithms accommodate delays, latencies, idles, pauses, watchdogs, timeouts and other notions connected to time intervals.

Another important feature of control algorithms is that they are almost always highly concurrent. The system needs to control multiple physical processes and communicate with a variety of hardware peripherals – all at the same time. As physical processes in the plant evolve independently, the sequence of events is arbitrary. Therefore any attempts to describe the control system within a single monolithic block leads to a combinatorial explosion of complexity [20]. Avoiding this requires the system to be split into a multitude of independent or loosely coupled control flows.

Lastly, we ought to mark the hierarchical structure of any complex control algorithm that reflects artificial nature of the external environment, the designer plan that is implemented in architecture of the facilities [21]. Taking into account the previous remarks we can say that the hierarchical structure consists of chains that are independently executed in parallel. This

means that divergence and convergence of control flow are the base for a significant part of control algorithm. The algorithm structure can be arbitrary, irregular and ever closed on itself.

To summarize our analysis, we list the key features of control systems that we deem most significant for control software development:

- openness – i. e. communication with an external environment,
- cyclic and indefinite execution,
- event-driven polymorphic behavior,
- operations with time intervals,
- concurrency,
- hierarchical structure.

3. Hyperprocess Model

From practice it is apparent that the Finite State Automaton (FSA) concept is particularly promising for use in logical control. FSAs implicitly assume presence of an external environment, can execute cyclically and exhibit event-driven polymorphic behavior. However, our analysis of control system features shows that the model also needs to support concurrency, hierarchy and timed operations.

Furthermore, the FSA model is tailored for hardware implementation. This is due to historical circumstances of when the model was created [22]. Negative effects of this become apparent even on the conceptual level, in the terms “input alphabet” and “output alphabet” which, while simple and convenient for theoretical studies, appear awkward and obscure from a modern programmer’s standpoint. This leads to general misunderstanding and discourages usage of a potentially very powerful concept [23]. Efficiency in practice requires that more conventional programming concepts, like variables and statements be supported. With this in mind, we have constructed an FSA-based model of control software, that is suitable for the domain of control software development. Here we will outline its basic structure and key properties that are necessary to lay a foundation for Reflex language. A more detailed description of the hyperprocess model can be found in [24].

The control software is represented as a hyperprocess – an ordered set of processes, which are cyclically activated with the period T_H. Mathematically, the hyperprocess is defined by a triplet:

$$H = < T_H, P, p_1 >, \text{ where} \tag{1}$$

- T_H is the period of activation,
- P is a finite nonempty ordered set of processes ($P = p_1, p_2, \ldots, p_M$, where M is the number of processes),
- p_1 is the first marked process, $p_1 \in P$.

At this point we can assume that a process is just a function, or a set of instructions (in programming sense). Note that the word "ordered" refers to textual description of a program. It should also be noted that a kind of perfect synchrony hypothesis [25] is assumed in this model. In contrast to the original hypothesis, which states that neither computation nor communication takes any physical time, we assume a relaxed statement to be true: the latency period for calculation overhead is less or equal than the period of activation T_H. This seems to be a less idealistic and quite reasonable condition for software implementation.

A process denotes a polymorphic subroutine – it is a set of mutually exclusive subroutines (i.e. blocks of sequential program code) which is handled as a unified entity. We will further refer to these subroutines as process state functions. For any cycle of hyperprocess activation, each process is reduced to one of its state functions as determined by the current state of that process. That state function is called the current function of the process and provides instructions to be executed during that hyperprocess activation. In particular, it can contain instructions that change the state of the process for the next cycle. State functions containing no instructions are referred to as passive and correspond to inactive states of the process. Each process also keeps track of how many hyperprocess cycles it has spent in its current state.

Formally, i-th process is a quintuple

$$p_i =< F_i, F_i^p, f_i^1, f_i^{cur}, T_i >, \text{ where} \qquad (2)$$

- $p_i \in H$, $i = 1, 2, \ldots, M$,
- F_i is a set of mutually exclusive functions,
- F_i^p is a set of mutually exclusive passive functions, $F_i^p \subset F_i$,
- f_i^1 is the first function (or marked active function), $(f_i^1 \in F_i) \wedge (f_i^1 \notin F_i^p)$,
- f_i^{cur} is the current function, $f_i^{cur} \in F_i$,
- T_i is the current time.

In programming, particularly in C, the term function is equivalent to a subroutine – a set of instructions or statements, that specify mathematical calculations, conditional actions etc. In the hyperprocess model we rather prefer to accentuate the event-driven and reactive features of the model. A state function is therefore defined as a set of events and reactions to the events.

Formally, j-th functions of i-th process is a twain

$$f_{ji} = < X_{ji}, Y_{ji} >, \text{ where} \tag{3}$$

- X_{ji} is a set of events,
- Y_{ji} is a set of reactions.

As events we consider any changes or superpositions of changes inside or outside the hyperprocess that are of importance to the control algorithm. The event is connected to a reaction it stimulates. Reactions are superpositions of actions, including calculations, change of output values, state transitions, communication with other processes, etc. A state with no events has has no reactions:

$$(X_{ji} = \varnothing) \Rightarrow (Y_{ji} = \varnothing). \tag{4}$$

Passive functions can then be defined as follows in terms of events and reactions:

$$(f_{ji} \in F_i^p) \Leftrightarrow (X_{ji} = \varnothing). \tag{5}$$

In essence, the process concept is a modification of the FSA model. The input and output alphabets have been removed and states of automaton along with its transition relation have been replaced with state functions. Transitions between states are part of the instructions within state functions. The input and output alphabets have been replaced with events and reactions. The transition relation of automaton is spread across state functions and expressed with a special reaction:

$$set_state(p_i, f_i^j) \equiv (f_i^{cur} := f_i^j, T_i := 0).$$

Thus, the original FSA model was preserved within the process model. Describing software with multiple automata provides logically parallel execution with granularity of the functions. The hyperprocess execution can be described in a following way: the hyperprocess is cyclically activated with a period T_H. upon each activation the current state function f_i^{cur} for each process $p_i \in P$ is executed. With each activation, the process time T_i for the process is incremented. Execution of a state function consists of sequentially testing for each of its monitored events and executing corresponding reactions for events that are detected.

To provide means for time-tracking and communication between processes, special events and reactions are defined. A process can check whether another process is in a passive state:

$$passive(p_i) \Leftrightarrow (f_i^{cur} \in F_i^p).$$

For time tracking purposes, a timeout event can be monitored:

$$timeout(p_i, T_{timeout}) \Leftrightarrow (T_i > T_{timeout}).$$

This event is triggered once the process p_i has been executing with the same state function

for a number of hyperprocess cycles given by $T_{timeout}$. To allow divergence and convergence of control flow, processes can start and stop other processes:

$$start(p_i) \equiv (f_i^{cur} := f_i^1, T_i := 0),$$

$$stop(p_i) \equiv (f_i^{cur} := f_i^{stop}), \text{ where } f_i^{stop} \in F_i^p.$$

These two reactions in conjunction with the *passive* event facilitate arrangement of the processes into a hierarchic structure, with higher level processes starting lower-level processes being a rough equivalence of calling subroutines in procedural programming. With this model we therefore have preserved the original cyclic, event-driven and polymorphic nature of the FSA model, and extended it to support concurrency, hierarchy and time tracking.

4. Introduction to Reflex Syntax

The hyperprocess model presented above, underlies the Reflex programming language, which is a dialect of the C programming language. The C language has been chosen for better learnability as many mainstream programming languages nowadays have C-like syntax. Additional constructs have been introduced for controlling processes and tracking time intervals. Two kinds of passive functions are used: the STOP-function is for a normal finishing of the process, and the ERROR-function is to indicate an abnormal finish. Reflex also provides constructs for linking internal software variables to physical I/O signals. The direct and inverse transformations between registers of I/O modules and internal variables are automated.

Reflex syntax is demonstrated here using a simple example of a program controlling a hand dryer like those often found in public restrooms (Listing 1). A formal Reflex syntax definition in EBNF can be found in Appendix A.

Here, the program uses input from an IR sensor, indicating presence of hands under the dryer and controls the fan and heater with a joint output signal. The basic requirement is that the dryer is on while hands are present and turns off automatically otherwise. As the person using the dryer rubs and turns their hands, the signal from the binary IR sensor will pulse between "on" and "off". To avoid erratic toggling of the dryer heater and fan, the program should not react to this switching and the actuators should only be turned off once the sensor signal is a steady "off". The algorithm can meet such requirement only by measuring the duration of the off state of the sensor. In this case, a duration longer than a certain given value (for example, 1 s) would be regarded as the "hands removed" event.

```
PROGR HandDryerController {
  TACT 100;
  CONST ON 1;
  CONST OFF 0;
  /* direction, name, address, offset, size of the port */
  INPUT  SENSOR_INPUT_PORT  0 0 8; /*IR sensor input port*/
  OUTPUT ACTUATOR_OUTPUT_PORT 1 0 8; /*output to heater and fan*/
  PROC Init {
    BOOL S_HANDS_UNDER_DRYER = {SENSOR_INPUT_PORT[1]} FOR ALL;
    BOOL C_TURN_ON_DRYER = {ACTUATOR_OUTPUT_PORT[1]} FOR ALL;
    STATE Waiting {
      IF (S_HANDS_UNDER_DRYER == ON) {
        C_TURN_ON_DRYER = ON;
        SET NEXT;
      }
      ELSE C_TURN_ON_DRYER = OFF;
    }
    STATE Drying {
      IF (S_HANDS_UNDER_DRYER == ON) RESET TIMEOUT;
      TIMEOUT 10 { SET STATE Waiting; }
    }
  } /* \PROC */
}  /* \PROGRAM */
```

Listing 1: Hand dryer example in Reflex

General program structure. A Reflex program is an aggregate of concurrently running communicating processes defined in textual form with the `PROC` keyword:

```
PROC <process name> { <process body> }
```

The process that is defined first in program text is the process p_1 that is initially in active state upon start of the program.

The `TACT` directive at the top sets the hyperprocess activation period T_H specified in milliseconds.

Main part of the process body is a list of state function definitions:

```
STATE <state name> { <state body> }
```

The first defined state in process body corresponds to it starting state f_i^1 into which the process is transitioned by a `START PROC` instruction (or initially for p_1). Passive states `STOP` and `ERROR` are not explicitly defined in the program.

Statements. A state body is defined with a sequential block of code, that includes statements like expression calculations (this include assignments), C-like selection statements `IF/ELSE` and `SWITCH` that define events and their corresponding reactions, process control statements and one optional timeout statement.

The syntax for the selection statements is very similar to that in C:

```
IF (<expression>) <statement> ELSE <statement>
```

Each of the `<statement>` blocks here can either be a single, semicolon terminated statement, or a compound statement enclosed in braces.The `ELSE` part is optional. Nested `IF`s are supported, i.e. each of the `<statements>` blocks above can in turn contain selection statements, much like in C language. The C-like `SWITCH` statement has also been included in Reflex:

```
SWITCH(<expression>) {
    CASE <value>:
        <statement>
    ...
    DEFAULT:
        <statement>
}
```

Reflex-specific process control constructs include state transitions, control statements and activity predicates that can be used in expressions. State transitions set the process state for the next activation cycle:

```
SET STATE <state name>;
```

A reserved keyword `NEXT` can be used here in lieu of explicit state name to denote a transition to the state that is defined next to the current along the text.

The `START/STOP/ERROR` statements allow processes to start/stop other processes and to stop themselves - either normally or in error state. These statements are responsible for divergence and convergence of control flow:

```
START PROC <process name>;
STOP PROC <process name>;
STOP;
ERROR;
```

To provide means for tracking time, timeout statements have been introduced in Reflex:

```
TIMEOUT <value in hyperprocess clocks> <statement>
```

This statement can only be used once in a state function and should then be the last statement in the state body. It allows to specify a reaction to the event of the process spending more than the specified amount of time in its current state. It is worth noting that though the timeout value here is specified in hyperprocess clocks and so the accuracy depends on the hyperprocess activation period T_H defined with the `TACT` directive at the top of the program text.

One use of timeouts is to implement a non-blocking delay, but a more typical application is for when a process waits for some external event, and that event not arriving withing a reasonable time interval would mean a malfunction of external components. Specifying a timeout serves as

a safety mechanism to prevent the process from locking in the waiting state forever, not unlike hardware watchdogs found in microcontrollers. However, it is often the case that some other event can signify that the system is still functioning normally, and the process can stay in its current state for more time. In this situation the `RESET TIMEOUT;` statement can be used.

Expressions. Expressions in Reflex can be used in conditions for `IF` or `SWITCH` selection statements or in expression statements:

```
<expression>;
```

A typical example of this kind of expression statement is assignment of value to a variable. Reflex supports most of expressions found in C, including assignments, comparisons, arithmetic, bitwise and logical operations.

One type of expressions that is new in Reflex is the process activity check predicates. Processes are able to check whether other processes are in their active or passive states using the selection statement in conjunction with `ACTIVE/PASSIVE` predicates, e.g.:

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

In the above example, the `IF` condition will be satisfied if the process at question is currently in any state other than `STOP` and `ERROR`.

Ports and variables. The process body can contain variable definitions with port bindings and scope directives:

```
<type> <variable name> = <port binding> <scope directive>;
```

Supported types are `BOOL` for Boolean values as well as `INT`, `SHORT`, `LONG`, `FLOAT` and `DOUBLE` that behave the same way as in C. The `FOR ALL` scope directive is to indicate that this variable can be used by any processes in the program. Port binding makes the variable being read into from an input port or written into the port if that port is defined as output. Ports used in the program are defined before the process definitions in the following format:

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

Reflex also supports constant definitions:

```
CONST <constant name> <constant value>;
```

Supported types for constant values here are the same as for variables. Unnamed constants can also be used in expressions, though it is generally deemed a bad practice in programming.

5. Translational Semantics

Semantics of a programming language can be preserved when that language is translated into another (target) programming language [26]. This is especially valuable if the target language is widely used and its semantics are well-known and affixed in standards. A translator has been implemented, that parses programs written in Reflex and produces C code, that can consequently be compiled into executable code for the target platform. C has been chosen as the target language for two reasons. Firstly, C served as the basis for Reflex, and many lower-level Reflex syntax elements have the same meaning as their identical C counterparts. Secondly, the C language is widely spread, used across many application domains and can be easily compiled for almost any known computational platform, thus providing cross-platform compatibility. The semantics of Reflex is therefore defined by this translation and here we use it to demonstrate the meaning of Reflex programs, by providing equivalent C code for key Reflex constructs.

Structure and Execution of an Equivalent C Program

The states of all processes are stored in an array of the following structure:

```
struct StateWord{
  unsigned long T; /*Ticks spent without a state change*/
  unsigned char cur_state; /*Index of current state*/
}ProcStates[PROC_NUM]; /*Array of process states*/
#define STATE_OF_ERROR       254
#define STATE_OF_STOP        255
```

Listing 2: State word structure

Here T corresponds to time T_i in the process model and `cur_state` is the index of current state function: $f_i^{cur} = f_i^j \Rightarrow j - 1 =$ `cur_state`. The first state function f_i^1 has an index of 0 and indices of special states stop and error are defined as the last two indices in the range. Note that `cur_state` is 8-bit here, as typical process in practice has 5 states only and few processes need more than 10 states.

Upon start, the program performs platform-dependent initialization, including that of the time service, which tracks the activation period T_H. Process states are also initialized: all times are set to zero, and all state indices are set to STOP, except for the first process, which is set to be in its first state. After that the program runs in an infinite loop, activating the hyperprocess with period T_H, specified by the `TACT` directive (Listing3). In the presented implementation, the time service is external to Reflex program and its interface is comprised of the `TH_Elapsed` flag.

Alternative implementations could have the infinite loop be empty and hyperprocess activation placed inside a timer interrupt service routine.

```
void main (void){
  SysInit(); /*Platform specific code*/
  InitProcesses(); /*Set p1 to start, the rest to STOP*/
  for(;;){ /*Indefinite cycic execution*/
    if(TH_Elapsed){/*Activation period TH*/
      TH_Elapsed = 0;
      HyperProcess(); /*Activate hyperprocess*/
    }
  }
}
```

Listing 3: Program execution

When activated, the hyperprocess reads and stores all used input port values, consequently executes all processes, writes all output port values that have been changed and, finally, increments time counters of all processes. Together with the previous listing, this makes up a simple round-robin cooperative concurrency algorithm. The reasons behind Input and Output functions are discussed later in the text. `IncrementProcTicks` here increments the `T` values in the `ProcStates` array for all processes. These values are further used in timeout statements.

```
void HyperProcess (void){
  Input(); /*Read and store input port values*/
  Process0(); /*Execute first process*/
  Process1(); /*Execute second process*/
  ...
  ProcessN(); /*Execute last process*/
  Output(); /*Output port values that have changed*/
  IncrementProcTicks(); /*Increase all T values by 1*/
}
```

Listing 4: Hyperprocess execution

Below is the equivalent C code for process `Init` from the hand dryer example demonstrated in 4. It uses a classic switch-style implementation of a finite state automaton. Another common FSA implementation in C involves tabular representation of the state machine with function pointers used for state functions. However, we find that the switch-based version is simpler, provides a more readable C-code and causes less overhead, especially with a significantly limited stack size. Resulting C code could be made even more readable by using enumerators for process and state indices.

```
void Process0 (void) { /*Process Init*/
  switch (ProcStates[0].cur_state) {
    case 0: /*State Waiting*/
```

```
      if (POV0 == C_0) { /*IF(S_HANDS_UNDER_DRYER == 0)*/
        POV1[1] = C_0; /*C_TURN_ON_DRYER = ON;*/
        Set_State(0, 1); /*SET  NEXT;*/
      }
      else POV1[1] = C_1; /*C_TURN_ON_DRYER = OFF;*/
      break;
    case 1: /*State Drying*/
      if (POV0 == C_0) /*IF(S_HANDS_UNDER_DRYER == ON)*/
        Set_State(0, 1); /*RESET TIMEOUT;*/
      if (Timeout(0, 12)) /*TIMEOUT 12*/
        Set_State(0, 0); /*SET STATE Waiting*/
      break;
    default:
      break;
  }
}
```

Listing 5: Example of process execution

Semantics of Special Control Statements and Expressions

Many statements and expressions defined in Reflex, are borrowed from C along with their semantics. Hence, out focus is on those constructs that are unique to Reflex and not present in C. Here we define the semantics of specialized constructs that are necessary for process functioning and inter-process communication, namely the state transition, the start/stop/restart operators, the timeout statement, and the active/passive expressions.

Given this code is found inside a state function of process p_i, state S has index j and process P has index k, and statement `<RS>` translates to statement `<CS>` in the resulting code, the following translation rules apply for specialized Reflex statements and logical expressions:

```
SET STATE s;  →  { ProcStates[i].cur_state = j;
                   ProcStates[i].T = 0;

RESTART;  →  { ProcStates[i].cur_state = 0;
               ProcStates[i].T = 0;

START PROC P;  →  { ProcStates[k].cur_state = 0;
                    ProcStates[k].T = 0;

STOP PROC P;  →  ProcStates[k].cur_state = STATE_OF_STOP;

STOP;  →  ProcStates[i].cur_state = STATE_OF_STOP;

ERROR;  →  ProcStates[i].cur_state = STATE_OF_ERROR;

RESET TIMEOUT;  →  ProcStates[i].T = 0;

TIMEOUT N <RS>  →  IF(ProcStates[i].T > N) <CS>

PROC P IN STATE ACTIVE  ⇔  { ProcStates[i].cur_state != STATE_OF_STOP &&
                              ProcStates[i].cur_state != STATE_OF_ERROR
```

```
PROC P IN STATE PASSIVE ⇔ { ProcStates[i].cur_state == STATE_OF_STOP ||
                            ProcStates[i].cur_state == STATE_OF_ERROR
```

Variables and Ports So far we have defined semantics for most syntax constructs that are new in Reflex, i.e. not present in C. One important point that has been left out so far, is how Reflex treats variables that are associated with I/O ports. In our hand dryer example we have two variables associated with I/O ports. The input variable `S_HANDS_UNDER_DRYER` contains the binary value from the IR sensor and the output variable `C_TURN_ON_DRYER` is written by the program to control the joint fan/heater actuator. While all internal variables are treated in Reflex the same way as they would be in C, variables bound to ports have a somewhat different semantics.

For input variables, their value is read from the corresponding bit of a corresponding port in the Input function at the start of hyperprocess execution (see Listing 4) and cannot be modified during the hyperprocess execution.

As for the output variables, their values are read in the Input function as well, but are stored in a double-buffered manner, i.e. each value is stored in two instances, both used when evaluating and executing expressions. One instance is only used in read operations and maintains its initial value throughout the hyperprocess execution, so that each process reads the same value, regardless of their order. The second instance stores the modified value, used in write operations. This can be seen in Listing 5: variable `C_TURN_ON_DRYER` is translated into an array with two values `char POV1[2]`, and only the second value `POV1[1]` is used in write operations. At the end of hyperprocess execution, the Output function is called, that checks what output values have been modified and stores those as corresponding bits in the output ports.

6. Practical Example

The Reflex language has passed strong examination in several complex projects. One such application was in an automated control system for a single crystal growth furnace [27]. The task involved typical industrial environment and equipment with distributed functioning, intelligent sensors, four-coordinate motion subsystem, actuators, gas-vacuum subsystem with doubled pumps, cooling subsystem, heater subsystem, conventional logic and analog I/O modules, complex algorithm with active reaction on faults. A Micro PC with a CPU685E, 300 MHz [28] processor has been used as computational platform.

Рис. 1: Single Crystal Growth Furnace

The control system has been implemented with all control logic written in Reflex. The following results have been obtained:

- the control algorithm has naturally fit the hyperprocess model with a surprisingly large quantity of concurrent processes (more than 760),
- low time consumption for process execution has been logged (2 microseconds per process),
- at peak load, full cycle of hyperprocess execution took 8 milliseconds (about 10 microseconds a process),
- the Reflex language showed to be easily understood by project members who were previously not familiar with programming,
- locality of change was preserved throughout the whole development of the system,

Overall, the language has received favorable comments from programmers, and what is more, very good responses have been received from control system developers with strong background in the IEC 61131-3 standard. The language has been praised for being more flexible and comfortable as compared to the IEC languages. The flexibility of the approach and Reflex language was recently used in the project on dynamic verification of control programs [29].

7. Conclusion

In this work we have presented a novel programming language Reflex that provides a practical notation for digital control systems. The hyperprocess model underlying Reflex takes is compliant with primary features of control software – cyclic execution, event-driven and polymorphic behavior, concurrency, etc. The language syntax is similar to C and provides time-interval operations as well as means for inter-process communication that encourage hierarchical organization of control software. A translator from Reflex to C has been implemented and semantics of key Reflex constructs has been demonstrated based on this translation.

The language has been successfully used in multiple industrial applications and demonstrated promising qualities, producing easily readable, maintainable code and overall robust and dependable software.

Further direction of work would be to research applicability of verification methods to software specified in Reflex.

References

1. Hoare C. A. R. Communicating Sequential Processes. Prentice Hall, 1985. – 256 p.
2. Harel D. Statecharts: a Visual Formalism for Complex Systems. / In: Science of Computer Programming 8. Elsevier Science Publishers B. V., North-Holland. 1987. P. 231–274.
3. Lynch N., Tuttle M. An Introduction to Input/Output Automata // CWI Quarterly. 1989. No 2. P. 219–246.
4. Berry G. The Foundations of Esterel // In: Plotkin, G., Stirling, C., and Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, Foundations of Computing Series. 2000. P. 425–454.
5. Milner R. Communication and Concurrency. Series in Computer Science. Prentice Hall, 1989. – 300 p.
6. Shoshmina I. V. Developing formal temporal requirements to distributed program systems // System Informatics. 2016. No. 8 P. 33–42.
7. Davis J., Schneider S. An Introduction to Timed CSP // PRG-75, PRG Programming Research Group Oxford. 1989.
8. Chen L., Anderson S., and Moller F. A Timed Calculus of Communicating Systems // Technical Report LFCS-90-127, University of Edinburgh. 1990.
9. Kaynar D. K., Lynch N., Segala R., Vaandrager F. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems // 24th IEEE International Real-Time Systems Symposium (RTSS'03), IEEE Computer Society Cancun, Mexico. 2003). P. 166–177.
10. Staroletov S. Design and Implementation a Software for Water Purification with Using Automata

Approach and Specification Based Analysis // System Informatics. 2017. No10. P. 33–44.

11. Shabaldina N., Gromov M. Using BALM-II for deriving parallel composition of timed finite state machines with outputs delays and timeouts: work-in-progress // System Informatics. 2016. No. 8 P. 33–42.
12. Abadi M., Lamport L. An Old-fashioned Recipe for Real Time // ACM Transactions on Programming Languages and Systems, Vol. 16 (5), ACM Press New York, Sept 1995. P. 1543–1571.
13. IEC 61131-3: Programmable controllers Part 3: Programming languages, International Electrotechnical Commission Std., Rev. 2.0, 2003.
14. Basile F., Chiacchio P., and Gerbasio D. On the Implementation of Industrial Automation Systems Based on PLC // IEEE Trans. on Automation Science and Engineering, Oct 2013, vol. 10, no. 4, pp. 990–1003.
15. Crater K. C. When Technology Standards Become Counterproductive. Control Technology Corporation. 1992.
16. Thramboulidis K. and Frey G. An MDD Process for IEC 61131-based Industrial Automation Systems // in 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA11), September 5-9, 2011, Toulouse, France, 2011. P. 1–8
17. IEC 61499: Function Blocks for Industrial Process Measurement andControl Systems, Parts 1 – 4, International Electrotechnical Commission Std., Rev. 1.0, 2004/2005.
18. Wagner F., Schmuki R., Wagner T., and Wolstenholme P. Modeling Software with Finite State Machines.Boston, MA, USA: Auerbach Publications, 2006.
19. Malyshkin V.E. Parallel computing technologies 2016 // The Journal of Supercomputing, Vol. 73, Iss. 2, Springer, 2017. P. 607–608.
20. Zyubin V.E. Multicore Processors and Programming. Open Systems J., N7-8. 2005. P. 12–19 (in Russian).
21. Zyubin V.E. Text and Graphics: What Language Does Programmer Need? // Open Systems J., 2004. No 1. P. 54–58 (in Russian).
22. Glushkov V. M. The Synthesis of Digital Automata. PhisMathGis, Moscow. 1962 (in Russian)
23. Wagner F., Wolstenholme P.: Misunderstandings about State Machines // IEE J. Computing and Control Engineering, Vol. 16. No 4, Aug 2004. P. 45.
24. Zyubin V. E. Hyper-automaton: A Model of Control Algorithms // in IEEE International Siberian Conference on Control and Communications (SIBCON-2007). Proceedings of the IEEE International Siberian Conference on Control and Communications, O. Stukach, Ed. Tomsk, Russia: IEEE, 2007, pp. 51–57. Available: https://doi.org/10.1109/SIBCON.2007.371297.
25. Kof L., Schätz B. Combining Aspects of Reactive Systems // In: Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics. Novosibirsk. 2003. P. 239–243.
26. Slonneger K. and Kurtz B. L. Formal syntax and semantics of programming languages / Addison-Wesley Reading, 1995, – 340 p.
27. Lubkov A. A., Zyubin V. E., Kurochkin A. V., Budnikov K. I. A Control System Architecture for a Single Crystal Growing Furnace.// In: Proc. of the Second IASTED Int. Conf. Automation,

Control, and Applications. 2005. P. 166–169.
28. CPU686E. CPU Card. Technical Specifications. Fastwel Inc. 2005.
29. Liakh T., Zyubin V. Model Checking of Industrial Control Algorithms in Combination with Virtual Objects // System Informatics. 2016. No 8. P. 11–20. (In Russian).

A. Appendix A. Reflex formal syntax in EBNF

```
program =    "PROGR", id, "{", [tact], [{const_or_enum_spec}],
             [{function_decl}], [{register_spec}], {process_spec},
             "}";

tact  =   "TACT", int_num, ";";

const_or_enum_spec  = const_spec | enumerator_spec;
const_spec          = "CONST", const_id, const_exp_body, ";";
enumerator_spec     = "ENUM", "{", enumerator_list, "}";
enumerator_list     = enumerator | (enumerator, ",", enumerator_list);
enumerator          = const_id | (const_id, "=", const_exp_body);
const_exp_body      = const_pref_term |
                     (const_pref_term, {const_infix, const_pref_term});
const_pref_term     = [const_prefix], const_term;
const_prefix        = "~" | "!" | "+" | "-";
const_infix         = "+" | "-" | "*" | "/" | "%" | "<<" | ">>" | "&" |
                      "^" | "|" | "&&" | "||";
const_term          = number | const_id | "(", const_exp_body,")"  ;
const_id            = id;

function_decl = c_type_spec, func_id, "(", c_type_spec_list, ")", ";";
c_type_spec_list  = c_type_spec | (c_type_spec, ",", c_type_spec_list);
func_id = id;

register_spec = ("INPUT" | "OUTPUT"), reg_id, addr_1, addr_2,
                register_size;
addr_1        = int_num;
addr_2        = int_num;
register_size = "8" | "16";
reg_id        = id;

process_spec    = "PROC", proc_id, "{", [var_list], {func_state}, "}";
proc_id         = id;
var_list        = {var_spec | var_decl};
var_spec        = (phys_var_spec | calc_var_spec),
                  visibility_spec, ";";
phys_var_spec   = int_type_spec, var_id, "=",
                  "{", reg_bits_spec_list, "}";
reg_bits_spec_list  = reg_bits_spec |
                      (reg_bits_spec, ",", reg_bits_spec_list);
reg_bits_spec   = reg_id, "[", int_num, "]";
calc_var_spec   = (c_type_spec | "BOOL"), var_id;
visibility_spec = "LOCAL" | ("FOR", "ALL") | ("FOR", proc_id_list);
proc_id_list    = proc_id | (proc_id, "," , proc_id_list);

var_decl        = "FROM", "PROC", proc_id, var_id_list, ";";
var_id_list     = var_id | (var_id, ",", var_id_list);


func_state      = "STATE", func_state_id, "{",
                  ([func_state_body],timeout_react_spec) |
```

```
                    timeout_react_spec, "}";
func_state_body = {react_spec};
react_spec      = ";" | ("{", func_state_body, "}") | switch_spec |
                    event_react_spec | start_spec | stop_spec |
                    error_spec | loop_decl | set_cur_sf_spec |
                    restart_cur_proc_spec | reset_timer_spec |
                    var_equation;

switch_spec     = "SWITCH", "(", event, ")", "{", {case_spec}, "}";
case_spec       = "CASE", int_num, ":", func_state_body,
                    ["BREAK", ";"];

event_react_spec  = event_react_body, [rev_event_react_body];
event_react_body  = "IF", "(", event, ")", react_spec;

rev_event_react_body= "ELSE", react_spec;

start_spec      = "START", proc_id, ";";
stop_spec       = "STOP", [proc_id], ";";
error_spec      = "ERROR", [proc_id], ";";
loop_decl       = "LOOP", ";";
set_cur_sf_spec = "SET", ("STATE", func_state_id) | "NEXT", ";";
restart_cur_proc_spec = "RESTART", ";";
reset_timer_spec      = "RESET", "TIMEOUT", ";";

var_equation    = var_id, "=", event, ";");

event = var_pref_post_term |
        var_pref_post_term, {var_infix, var_pref_post_term};
var_pref_post_term  = var_prefix, term, [var_postfix];
var_prefix    = [("~" | "!" | "++" | "--" | "+" | "-" | "*" | "&")],
                ["(", c_type_spec,")"];
var_postfix   = "++" | "--";
var_infix     = "+" | "-" | "*" | "/" | "%" | "<<" | ">>" | "&" | "^" |
                "|" | "&&" | "||" | "=" | "*=" | "/=" | "%=" | "+=" |
                "-=" | "<<=", | ">>=" | "&=" | "^=" | "|=" | "<" | ">"|
                "<=" | ">=" | "==" | "!=";
term  = number | const_id | var_id | function |
        situation | "(", event, ")";
function  = func_id, "(", func_param_list, ")";
func_param_list = event | (event, {",", event});

situation   = "PROC", proc_id, "IN", "STATE",
              (func_state_id | "ACTIVE" | "PASSIVE");

timeout_react_spec  = "TIMEOUT", (number | const_id | var_id),
                        react_spec;
func_state_id = id;

var_id  = id;

c_type_spec     = "VOID" | "FLOAT" | "DOUBLE" |
                  (["SIGNED" | "UNSIGNED"],
                  ("SHORT" | "INT" | "LONG"));
int_type_spec   = "BOOL" | "SHORT" | "INT" | "LONG";
float_type_spec = "FLOAT" | "DOUBLE";

id  = letter, {letter | decimal_digit};

number      = int_num | float_num;
int_num     = octal_num | decimal_num | hex_num;

letter        = ("A" ... "Z") | ("a" ... "z") | "_";
```

```
octal_num     = "0", [{octal_digit}], ["U" | "u"] | ["L" | "l"];
octal_digit   = "0" ... "7";
decimal_num   = (decimal_digit - "0"), [{decimal_digit}], ["U" | "u"] |
                ["L" | "l"];
decimal_digit = octal_digit | "8" | "9";
hex_num       = hex_prefix, {hex_digit}, ["U" | "u"] | ["L" | "l"];
hex_digit     = decimal_digit | ("A" ... "F") | ("a" ... "f");
float_num     = decimal_float_num | hex_float_num;

decimal_float_num = (fractional_part, [exponent_part], [float_suffix])|
                    (decimal_sequence, exponent_part, [float_suffix]);
hex_float_num     = hex_prefix, (hex_fractional_part | hex_sequence),
                    bin_exponent_part, [float_suffix];

hex_prefix  = "0x" | "0X";

fractional_part = ([decimal_sequence], ".", decimal_sequence) |
                  (decimal_sequence, ".");
exponent_part   = ("e" | "E"), [sign], decimal_sequence;
sign            = "+" | "-";

hex_fractional_part = ([hex_sequence], ".", hex_sequence) |
                      (hex_sequence, ".");

bin_exponent_part = ("p" | "P"), [sign], decimal_sequence;

hex_sequence        = {hex_digit};
decimal_sequence    = {decimal_digit};

float_suffix    = "f" | "F" | "l" | "L";
```

Listing 6: Reflex Grammar Specification