УДК 004.05 DOI: 10.17587/prin.13.155-167

В. И. Шелехов, канд. техн. наук, зав. лаб., vshel@iis.nsk.su,
Институт систем информатики им. А. П. Ершова СО РАН,
Новосибирский государственный университет

Автоматное программирование на базе системы моделирования и верификации Event-B

Представлен новый язык автоматного программирования, построенный расширением языка спецификаций Event-B. При разработке моделей в Event-B появляется возможность использования методов автоматного программирования в дополнение к популярному методу уточнений. Технология автоматного программирования на базе Event-B демонстрируется на примере задачи управления движением на мосту из руководства по системе Event-B. Предложено более простое решение с верификацией в инструменте Rodin. Эффективность методов верификации в Event-B подтверждена нахождением трех нетривиальных ошибок в приведенном решении.

Ключевые слова: *автоматное программирование, Event-B, уточнение, требования, дедуктивная верификация, трансформации программ, функциональное программирование*

Введение

Event-B [1] — это метод формальной спецификации и верификации систем в программной и системной инженерии, успешно используемый при разработке производственных систем управления, особенно в железнодорожном транспорте и метрополитене. Система Event-B реализована на платформе Rodin [2]. Метод верификации в Event-B гарантирует обнаружение многих серьезных ошибок. Несмотря на большое число разнообразных приложений, существует ряд факторов, сдерживающих широкое внедрение подхода Event-B, что отмечается в обзоре [3].

Автоматное программирование [4—8] ориентировано на класс реактивных систем, в частности, систем управления. Автоматная программа определяет автомат в виде гиперграфовой композиции сегментов кода. Технология автоматного программирования предлагает комплекс методов для разработки, оптимизации и верификации автоматных программ.

Построен новый язык автоматного программирования расширением языка спецификаций Event-B. Автоматная программа легко кодируется в языке спецификаций Event-B. Это дает возможность использования технологии автоматного программирования в разработке систем управления для критической инфраструктуры на платформе Rodin [2]. Данный стиль разработки иллюстрируется на примере задачи управления движением автомобилей на мосту из руководства по системе Event-B [1]. Предложено более простое решение с верификацией в инструменте Rodin. Эффективность методов верификации в Event-B подтверждена нахождением трех нетривиальных ошибок в приведенном решении.

В разд. 1 настоящей статьи определен класс программ-процессов в контексте других классов программ, описаны язык и технология автоматного программирования. В разд. 2 представлен метод Event-B. Далее в разд. 3 описано построение языка автоматного программирования на базе языка спецификаций Event-B. В разд. 4 даны два решения задачи управления движением автомобилей на мосту из руководства по Event-B [1]. Проиллюстрированы уточнения из руководства [1]. Описана разработка автоматной программы, исправляющей недостатки реализации в Event-B. В разд. 5 описана верификация автоматной программы в инструменте Rodin [2]. Обзор работ дан в разд. 6. В заключении описаны итоговые положения и замечания по технологии автоматного программирования на базе системы Event-B.

1. Автоматное программирование

1.1. Классы программ

Автоматные программы требуют других методов разработки и верификации, нежели программы вычисления некоторого результата по набору аргументов. Например, метод верификации Флойда—Хоара [9, 10] не применим для автоматных программ. Существуют разные классы программ со своими особенными методами разработки, спецификации, верификации, моделирования и оптимизации.

Классификация программ [11] определяет невзаимодействующие программы (или программы-функции), реактивные системы (или программы-процессы), языковые процессоры и операционные среды. Отметим, что такая классификация неполна и не покрывает существующего разнообразия видов программ.

Класс программ-функций (невзаимодействующих программ). Программа принадлежит этому классу,

если она не взаимодействует с внешним окружением. Точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод был собран в конце программы. Программа обязана всегда завершаться, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Следовательно, программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов.

Класс программ-процессов (реактивных систем). Программа-процесс является реактивной системой, реагирующей на определенный набор событий (сообщений) во внешнем окружении программы. Программа-процесс является либо автоматной программой, либо она определяется в виде композиции нескольких автоматных программ, исполняемых параллельно и взаимодействующих между собой через прием/посылку сообщений и разделяемые переменные.

Автоматная программа состоит из одного или нескольких сегментов. *Сегмент* имеет один вход, помеченный меткой — *управляющим состоянием*. Сегмент имеет один или несколько выходов. Автоматная программа определяет автомат в виде гиперграфа с набором управляющих состояний в качестве вершин и набором сегментов в качестве ориентированных гипердуг. *Состояние* автоматной программы определяется значениями набора переменных, модифицируемых в программе.

Языковые процессоры — это интерпретаторы программ, компиляторы, оптимизаторы, трансформаторы и другие процессоры.

Класс программ "операционная среда". Программа данного класса является автоматом общего вида. Это, например, операционная система в целом или ее фрагмент, например, система файлов. Операционная среда в каждый текущий момент функционирования имеет несколько активных позиций. Каждая позиция функционирует как независимая автоматная программа со своим набором управляющих и внутренних состояний.

Полезность универсальных методов. Метод признается универсальным, если он применим для некоторого класса программ. Метод полезен для конкретной программы, если его применение дает преимущества по сравнению с некоторой типовой технологией, не использующей данный метод.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая при этом будет значительно сложнее аналогичной функциональной или императивной программы. Автоматное программирование не полезно для программ-функций.

Полезность метода объектно-ориентированного программирования определяется возможностью инкапсуляции (упрятывания) разнообразных деталей реализации, предоставляя наружу более простой интерфейс. Если не происходит существенного упрощения внутреннего интерфейса программы, то объектно-ориентированные конструкции просто загромождают программу, удлиняя и усложняя ее. Между тем имеется достаточно стойкая тенденция, иногда подкрепленная местными стандартами в разных средах разработчиков и провоцируемая такими языками, как Java, злоупотребления объектно-ориентированным программированием. Кроме того, в редких случаях объектно-ориентированное программирование требуется в полном объеме. В подавляющем большинстве случаев можно было бы ограничиться более простыми возможностями на уровне абстрактных типов данных [12].

1.2. Язык автоматного программирования

Программа-процесс содержит фрагменты, соответствующие программам-функциям. Поэтому язык автоматного программирования должен быть построен как расширение некоторого *базисного* языка (императивного или функционального) для класса программ-функций. Таким способом определено автоматное расширение [5, 6] языка предикатного программирования P [4, Разд. 10]. Язык автоматного программирования можно построить расширением любого языка для класса программ-функций и даже расширением языка спецификаций.

Программа-процесс состоит из секций и автоматных программ. *Секция* определяется следующей конструкцией:

section <Имя секции> **extends** <Имя секции>
{ <Описания типов, констант, переменных
и инвариантов> }

В секции описываются переменные состояния программы (возможно, части состояния), а также константы и типы. Обычно секция помещается перед автоматной программой. Имя секции может отсутствовать. Секция может быть построена расширением другой секции при наличии **extends** <Имя секции>. В секции также помещаются общие инварианты для всей программы.

Автоматная программа определяется следующей конструкцией:

process <Имя программы>(< Описания аргументов>)
{ <Сегменты кода> }

Аргументы могут отсутствовать. Автоматная программа может быть вызвана внутри другой автоматной программы. Произвольный <Сегмент кода> представляется следующей конструкцией:

<Имя управляющего состояния>:
inv <Формула>; <Оператор>

Исполнение <Оператора> завершается оператором перехода вида #M, реализующим переход на начало сегмента с управляющим состоянием M. <Формула> определяет инвариант, который должен быть истинным в начале сегмента для данного управляющего состояния.

Представленная структура управления автоматной программы аналогична используемой в языке Фортран, применяемом в основном для задач вы-

числительной математики, но не для реактивных систем.

Сегмент, код которого не содержит вызовов других процессов, является *атомарным*: исполнение сегмента должно быть единым, неделимым актом; это означает, что до завершения исполнения сегмента все другие параллельно исполняемые процессы останавливаются.

Сегмент кода следующего вида:

M: if (<условие$_1$>&...& <условие$_n$>)
{ <действие$_1$>; ...; <действие$_m$> **#L** }

может быть записан в виде правила [7]:

M: <условие$_1$>,..., <условие$_n$> →
<действие$_1$>, ..., <действие$_m$> **#L**

Сегмент вида **M: if** (C) A **else** B **#L** может быть представлен парой правил:

M: C → A **#L**
M: B **#L**

Правила исполняются в порядке их следования. Второе правило **M:** B **#L** может сработать только при ложном условии C.

Для операторов A и B оператор A || B определяет параллельное исполнение операторов A и B. Оператор A | B определяет недетерминированный выбор для исполнения одного из операторов, A или B.

В языке автоматного программирования определяются также операторы приема и посылки сообщений, действия со временем: установка таймера, оператор задержки по времени и др.

1.3. Спецификация и верификация автоматных программ

Методы верификации, успешно применяемые для программ-функций, в частности, логика Хоара—Флойда [9, 10], непригодны для верификации автоматных программ. Эти методы могут применяться лишь для фрагментов автоматных программ, соответствующих программам-функциям.

Задача разработки программы-процесса формулируется в виде набора требований. *Требование* — утверждение, определяющее потребность и связанные с ней измеримые условия и ограничения. Требования к реактивной системе включают требования окружения и функциональные требования, определяющие поведение системы. Существенными являются также нефункциональные требования надежности, безопасности, защищенности, отсутствия дедлоков и др. Разработка требований должна проводиться в соответствии со стандартом ISO/IEC/ IEEE 29148 [13].

Спецификация программы-процесса является частью требований или непосредственно вытекает из требований. Спецификация определяется общими инвариантами и инвариантами управляющих состояний. Отметим, что инварианты управляющих состояний и общие инварианты принципиально отличаются от инвариантов циклов императивной программы. Инварианты управляющих состояний часто являются слабыми или отсутствуют, т. е. оказываются тождественно истинными.

Далее будет показано, каким образом инвариант управляющего состояния преобразуется в общий инвариант.

Общий инвариант формулируется для автоматной программы в секции непосредственно перед автоматной программой. Общий инвариант должен быть истинным в начале каждого сегмента. Общий инвариант может быть представлен темпоральной формулой.

Истинность общего инварианта гарантируется доказательством следующей серии формул корректности. Для каждого управляющего состояния необходимо доказать, что из истинности общего инварианта в начале соответствующего сегмента следует истинность общего инварианта для модифицированных значений переменных на каждом выходе из данного сегмента.

Для любого фрагмента программы, соответствующего программе-функции, необходимо доказать истинность предусловия для данного фрагмента. Например, для операции деления необходимо доказать, что выражение, являющееся делителем, не равно нулю.

Другие свойства программ-процессов, которые подлежат верификации, — это отсутствие взаимной блокировки процессов (дедлоков) и завершение конечных процессов.

Доказательство истинности всех перечисленных видов формул корректности позволяет избежать многих серьезных ошибок, но оно не гарантирует полной корректности программы в отличие от верификации программ-функций, где правильность спецификации и доказательство формул корректности гарантируют корректность программы.

1.4. Технология автоматного программирования

Разработка программы-процесса начинается с определения набора требований. Фиксируются объекты и их атрибуты. Формулируются требования окружения, функциональные требования, требования безопасности и др. Достаточно часто автоматная программа строится простым переписыванием функциональных требований. Их удобнее записывать в виде логических правил [7]. Требования безопасности формализуются в виде общих инвариантов.

Основной метод автоматного программирования — это метод декомпозиции автоматной программы в виде нескольких сегментов кода. Каждый сегмент кода автоматной программы соответствует некоторому управляющему состоянию и реализует независимую более простую подзадачу. Сужения для подзадачи фиксируются инвариантом управляющего состояния.

Гиперграфовая композиция автоматной программы является более общей по сравнению с композициями операторов в традиционных языках, таких как Си. Гиперграфовая композиция является предельно гибкой. Любую автоматную программу можно предста-

вить композицией двух сегментов. Гиперграфовая композиция трех независимых процессов позволила упростить программу управления лифтом [6].

Сегмент кода, реализующий программу-функцию и имеющий несколько выходов, является *гиперфункцией* [4, 14, 15]. Гиперфункции обеспечивают дополнительные возможности оптимизации программ.

Для получения более эффективной программы применяются эквивалентные автоматные трансформации, существенно меняющие структуру автоматной программы [8].

Для упрощения интерфейса программы полезно использовать методы объектно-ориентированного и аспектно-ориентированного программирования. Отметим, что редко требуется использовать объектно-ориентированное программирование в полном объеме — достаточно ограничиться типовыми конструкциями на уровне абстрактных типов данных [12].

Стиль языка Фортран для автоматной программы в целом контрастирует со стилем языка Си внутри сегмента кода, особенно при использовании циклов вида **while** и **for**. Желательно ограничить использование циклов **while** и **for** в автоматных программах.

2. Система Event-B и платформа Rodin

Event-B [1] — это метод формальной спецификации и верификации систем в программной и системной инженерии с использованием нотации теории множеств и логики первого порядка. Спецификация на языке Event-B состоит из компонентов двух видов: контекстов и машин. *Контекст* определяет множества и константы — статическую часть спецификации. *Машина* содержит: переменные, инварианты, события. События машины определяют процесс в виде недетерминированного автомата. Значения переменных формируют текущее *состояние* процесса.

Текущее состояние спецификации может быть изменено событием. *Событие* определяет охранные условия и действия. *Охранные условия* ограничивают множество возможных состояний, в которых данное событие может произойти. *Действия* реализуют присваивания переменным. Эффект более сложного действия можно определить before-after-предикатом, связывающим значения исходных и модифицированных переменных. Для всякого события инварианты должны оставаться истинными после модификаций переменных действиями события.

Все события атомарны и могут реализоваться, если истинны их охранные условия. Если одновременно истинны охранные условия нескольких событий, то только одно из них может исполниться в данный момент; событие для исполнения выбирается недетерминированным образом.

Спецификация Event-B состоит из последовательности машин, в которой очередная машина является *уточнением* предыдущей машины. Очередная машина содержит новые переменные и события, расширяющие предыдущую машину. Причем поведение новой машины в проекции на наследуемые объекты и события полностью идентично поведению предыдущей машины.

Платформа Rodin [2] определяет среду для разработки и верификации спецификаций на языке Event-B. Доказательство генерируемых по спецификации формул корректности поддерживается автоматическими и интерактивными средствами доказательства с применением SMT-решателей [2]. Степень и возможности автоматизации доказательства существенно выше, чем в системах доказательства PVS [16], Why3 [17] и Coq [18]. В частности, автоматизировано применение эквивалентных замен. В дереве текущего процесса доказательства явно обозначены позиции, по которым пользователь может продолжить доказательство, выбрав одну из предлагаемых альтернатив.

В Event-B имеются средства обнаружения ситуаций взаимной блокировки (дедлока), а также доказательства завершения конечных процессов.

3. Язык автоматного программирования на базе Event-B

В автоматном программировании можно представить любую технологию для программ-процессов. Далее определим построение нового языка автоматного программирования расширением языка спецификаций Event-B.

Произвольное событие далее будем представлять в виде оператора:

if (<Охранные условия>) { <Действия> }

или в виде правила:

<Охранные условия> → <Действия>

В типичном случае действия реализуют присваивание нескольким переменным. Например, { c := C; d := D }. Выражения C и D вычисляются и их значения одновременно присваиваются переменным c и d.

Допустим, машина M состоит из событий A, B, E и F. Тогда машину M будем представлять следующим процессом с единственным управляющим состоянием cycle:

```
process M
{ cycle: [nA:] A | [nB:] B | [nE:] E | [nF:] F #cycle }
```

Здесь nA, nB, nE и nF — имена событий A, B, E и F. Имена событий, альтернатив недетерминированной композиции, представлены здесь в квадратных скобках в отличие от имен управляющих состояний.

В новом языке автоматного программирования будем использовать примитивные типы BOOL, INT, NAT языка Event-B и оператор присваивания <Переменная> := <Выражение>.

Некоторые конструкции языка Event-B заменяются другими, более привычными. Предикат x ∈ BOOL будем записывать в виде x: BOOL. Не используется событие INITIALISATION. Начальная инициализация переменных реализуется в секции перед автоматной

программой. Например, x: BOOL := false. Вместо конструкции partition со сложной семантикой используется описание типа перечисления **enum** в стиле языка Си.

4. Управление движением автомобилей по мосту

Задача управления движением автомобилей по мосту представлена в качестве первого примера в известной книге J.-R. Abrial, являющейся руководством по системе Event-B [1]. На базе этого примера определяются основные положения технологии Event-B.

4.1. Требования к системе управления

Содержательное описание. Имеется мост, соединяющий материк и остров. Мост узкий и позволяет двигаться автомобилям по нему только в одну сторону. Имеются два светофора, установленных при въезде на мост с материка и с острова. У каждого светофора два цвета: красный и зеленый. Автомобилям запрещено движение на красный светофор при въезде на мост. Имеются четыре сенсора. Каждый сенсор находится на некотором участке автомобильной трассы и способен фиксировать ситуацию, когда автомобиль находится на этом участке трассы. Первый сенсор находится перед въездом на мост с материка. Второй сенсор — на мосту перед съездом на остров. Третий сенсор — перед въездом на мост с острова. Четвертый сенсор на мосту перед съездом на материк.

Число автомобилей, которые могут находиться на острове, ограничено. Необходимо построить контроллер, который переключает светофоры, используя показания сенсоров, и таким образом обеспечивает безопасное движение автомобилей по мосту.

Далее проведем анализ, детализацию и именование требований.

Объекты окружения: материк, остров, мост, 2 светофора, 4 сенсора.

Атрибуты светофора: красный; зеленый.

Атрибуты сенсора: on — сенсор включен; off — сенсор выключен.

Имена светофоров: mtl — светофор на материке при въезде на мост; itl — светофор на острове при въезде на мост.

Имена сенсоров: mlOut — сенсор при въезде на мост с материка; ilIn — сенсор при съезде с моста на остров; ilOut — сенсор при въезде на мост с острова; mlIn — сенсор при съезде с моста на материк.

Определение. Сенсор *отключается*, если он изменяет свое значение с on на off.

Каждый сенсор сопряжен с некоторым участком автомобильной трассы. Сенсор включен, если на данном участке находится автомобиль. Отключение сенсора означает, что автомобиль съехал с данного участка.

Сформулируем функциональные требования.

RF1. Отключение сенсора mlOut — автомобиль въехал на мост с материка.

RF2. Отключение сенсора ilIn — автомобиль съехал с моста на остров.

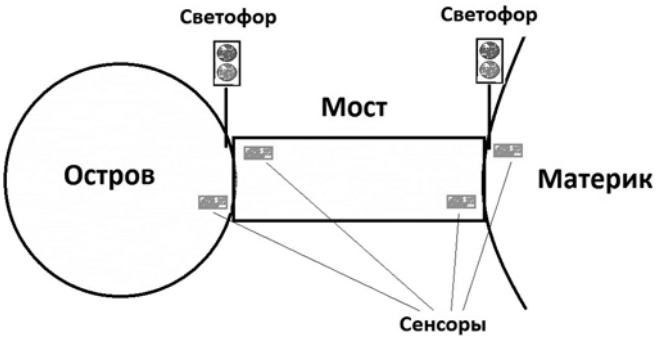

Рис. 1. Схема моста со светофорами и сенсорами

RF3. Отключение сенсора ilOut — автомобиль въехал на мост с острова.

RF4. Отключение сенсора mlIn — автомобиль съехал с моста на материк.

RF5. Число автомобилей на острове ограничено.

RF6. Движение автомобилей по мосту возможно либо с материка на остров, либо с острова на материк.

Определим требования безопасности.

RS1. При красном светофоре mtl запрещено движение с материка на мост.

RS2. При красном светофоре itl запрещено движение с острова на мост.

Представленная система управления движением по мосту в целом и архитектура оборудования (рис. 1) в частности имеют серьезные недостатки и не могут использоваться на практике. Ничто не может помешать автомобилю встать на сенсор, а затем поехать в обратном направлении. Автомобиль может также заехать на мост, а потом съехать с него задним ходом. Поэтому данную задачу построения контроллера управления движением по мосту можно рассматривать только как учебную.

4.2. Разработка от простейшей модели

Построение спецификации управления движением по мосту представлено в книге [1] в виде простейшей модели и последующих трех уточнений. Для иллюстрации технологии Event-B воспроизведем начальную модель и первые два уточнения на языке автоматного программирования.

В начальной модели мост и остров рассматриваются как единое целое. В модели фиксируются автомобили, уходящие с материка, и автомобили, приходящие на материк. Пусть n — число автомобилей на мосту и на острове. Введем константу d, определяющую максимальное число автомобилей на острове. Представим (рис. 2) автоматную программу для начальной машины с двумя событиями: уход автомобиля с материка и приход автомобиля на материк.

Легко обнаружить, что в модели есть ошибки. Первая ошибка: нет гарантии, что после действия n:=n+1 не будет превышен лимит автомобилей d. Эффективность метода Event-B [1] в том, что он позволяет обнаруживать все ошибки такого рода в процессе доказательства формул корректности. В данном

```
section St0 {
    const d: NAT
    n: NAT
    inv  n ≤ d
}
process carBridge0er {
   cycle: [ML_out:] n := n+1 |
          [ML_in:] n := n-1
          #cycle
}
```

Рис. 2. Начальная модель с ошибками

```
section St0 {
    const d : NAT  // максимальное число автомобилей на острове
    axiom d > 0
    n: NAT := 0
    inv  n ≤ d
}
process carBridge0 {
   cycle: [ML_out:] n < d → n := n+1 |
          [ML_in:]  n > 0 → n := n-1
          #cycle
}
```

Рис. 3. Исправленная начальная модель

```
section St1 extends St0{
    a: NAT := 0
    b: NAT := 0
    c: NAT := 0
    inv a+b+c = n
    inv a=0 ∨ c = 0
}
process carBridge1 refines carBridge0 {
   cycle: [ML_out:]  a+b<d,  c=0 → a:=a+1 |
          [ML_in:]   c>0  →  c:=c-1 |
          [IL_out:]  b>0, a=0 →  b:=b-1, c:=c+1 |
          [IL_in:]   a>0  →  a:=a-1, b:=b+1
          #cycle
}
```

Рис. 4. Первое уточнение

случае будет представлена формула корректности: n+1⩽d, которая здесь недоказуема. Для исправления ошибки необходимо добавить условие: n<d. Вторая ошибка: для действия n:=n–1 будет сгенерирована формула корректности n-1 ∈ NAT, которая оказывается недоказуемой. Необходимо вставить условие: n>0. Третья ошибка: в начальный момент работы машины не выполняется формула корректности: n⩽d. Причина в том, что переменная n не инициализирована. В секции St0 необходимо вставить инициализацию: n:= 0. После исправления данных ошибок появляется четвертая ошибка. При поиске блокировок (дедлоков) генерируется формула корректности: n⩽d ⇒ n<d ∨ n>0, которая недоказуема при d=0. В случае d=0 действительно возникает дедлок, поэтому необходимо условие d>0 в виде аксиомы. На рис. 3 дана исправленная начальная модель.

Определим уточнение начальной модели (рис. 4). Будем различать автомобили, находящиеся на мосту и на острове. Заменим старую переменную n тремя новыми переменными: a — число автомобилей, движущихся по мосту с материка на остров; b — число автомобилей на острове; c — число автомобилей, движущихся по мосту с острова на материк. Вводится *связующий инвариант* a+b+c=n, определяющий отношение между старыми и новыми переменными. Появляются два новых события: автомобиль съезжает с моста на остров (IL_in) и автомобиль въезжает на мост с острова (IL_out).

Можно видеть, что события ML_out и ML_in в уточненной модели отличаются от этих же событий в начальной модели. Для этих событий в Event-B генерируются формулы корректности, доказательство которых гарантирует идентичность поведения старых и новых событий. Корректность уточнения для добавленных событий IL_out и IL_in будет обеспечена, если эти события не меняют значений старых переменных, т. е. значения переменной n.

Следующая модель (рис. 5) строится как уточнение модели carBridge1. Это уточнение другого вида. В модели carBridge1 уточнялись структуры данных. Здесь же происходит добавление новых структур данных и новых событий.

Появляются светофоры: mtl — на материке при въезде на мост; itl — на острове при въезде на мост. Инвариант mtl=red ∨ itl=red соответствует требованию RF6. Два других инварианта отражают следующие правило корректности уточнения: охранное условие некоторого события в уточненной модели должно быть не слабее аналогичного условия в предыдущей модели.

На пустом мосту новые события Mtl_green и Itl_green переключают светофоры так, чтобы стало возможным движение в противоположную сторону. Далее, поскольку мост все еще пустой, может сработать другое событие, переключающее светофоры. В результате получим бесконечный процесс быстрого

```
section St2 extends St1{
    type Colour = enum {red, green}
    mtl: Colour := red
    itl: Colour := red
    inv mtl=red ∨ itl=red
    inv mtl=green => a+b<d & c=0
    inv itl=green => b>0 & a=0
}
process carBridge2 refines carBridge1{
   cycle: [ML_out:] mtl=green → a:= a+1 |
          [ML_in:]  c>0  → c:=c-1 |
          [IL_out:] itl=green  →  b:=b-1, c:= c+1 |
          [IL_in:]  a>0 →  a:=a-1, b:=b+1 |
          [Mtl_green:]  mtl=red, c=0, b<d  →  mtl:=green, itl:=red |
          [Itl_green:]  itl=red, a=0, b>0 →  itl:=green, mtl:=red
          #cycle
}
```

Рис. 5. Второе уточнение

переключения светофоров. Чтобы не допустить такую ситуацию, дальнейшее переключение светофоров блокируется в модификации машины carBridge2 до появления автомобиля на мосту с противоположной стороны. Данное решение неудовлетворительно, поскольку ожидание автомобиля с противоположной стороны может оказаться долгим.

Правильное решение следующее. Изменение направления движения возможно, если с противоположной стороны имеется автомобиль, стоящий на сенсоре в ожидании зеленого светофора.

В третьем уточнении добавляются сенсоры. Модель оказалась сложной. Для проведения уточнения потребовалось ввести 11 новых переменных.

Уточнение в Event-B фактически единственный метод. Он позиционируется как универсальный и эффективный для декомпозиции сложных моделей. В данном примере из руководства [1] уточнения были введены в учебных целях. Третье уточнение оказалось неоправданно сложным. Было бы намного проще построить модель, не прибегая к уточнениям.

Метод уточнений является эффективным для иерархических моделей. Построение искусственной иерархии обычно не дает хорошего результата. Метод уточнений не всегда полезен.

4.3. Разработка по технологии автоматного программирования

Изменения числа автомобилей на мосту и на острове определяются через изменения показаний сенсоров. Функционирование сенсоров естественно определить независимыми процессами, вычисляющими число автомобилей на мосту и на острове. Управление светофорами удобнее реализовать другим процессом. Полная программа-процесс управления движением автомобилей на мосту представляется параллельной композицией пяти автоматных программ:

```
process carBridge {
    LightControl || ML_out || IL_in || IL_out || ML_in
}
```

Рассмотрим автоматную программу управления светофорами LightControl. Очевидными являются следующие три управляющих состояния:

- zero — мост пуст, оба светофора красные;
- right — реализуется движение слева направо, т. е. с острова на материк;
- left — реализуется движение справа налево, т. е. с материка на остров.

Каждое управляющее состояние определяет независимую подзадачу, более простую по сравнению с исходной задачей.

В состоянии right реализуется управление движением с острова на материк. Эта задача намного проще исходной задачи. Например, нет необходимости проверять ограничение числа автомобилей на острове. Если мост становится пустым, необходимо перейти в состояние zero.

В состоянии left реализуется управление движением с материка на остров. Если мост становится пустым, происходит переход в состояние zero. При появлении автомобиля, въезжающего на мост с материка, также проверяется ограничение a+b⩽d. В ситуации a+b=d светофор mtl устанавливается красным. Для того чтобы перейти в состояние zero, необходимо сначала освободить мост. Освобождение моста контролируется в дополнительном управляющем состоянии leftscan.

В состоянии zero мост пуст. Оба светофора красные. Переход в состояние right или left становится возможным, когда у моста на сенсоре останавливается автомобиль в ожидании зеленого светофора. Чтобы отслеживать остановку автомобиля на сенсоре перед мостом, вводятся две новые переменные. Переменная ml_out=true, если сенсор mlOut включен, т. е. на нем со стороны материка в данный момент находится автомобиль. Переменная il_out = true, если сенсор ilOut включен — на нем находится автомобиль со стороны острова.

Очевидно, что после перехода с zero на right или left немедленно произойдет возврат на zero, потому что автомобиль не успеет заехать на пустой мост. Получим быстрое мерцание светофора с зеленого на красный. Чтобы избежать мерцания светофоров, переход с zero реализуется в новые состояния right0 и left0, где ожидается появление на мосту первого автомобиля.

Если b=d в состоянии left0, то при появлении первого автомобиля на мосту будет нарушено ограничение a+b⩽d. В связи с этим, при переходе с zero на left0 необходимо дополнительно проверять условие b<d.

Возникает ситуация блокировки (дедлока) в состояниях right0 и left0, если первый автомобиль появился на мосту и успел выехать с моста раньше, чем сработал сегмент кода в состояниях right0 или left0. Подобное возможно, когда процесс LightControl будет приостановлен. Во избежание блокировки введен признак пустого моста empty, который гасится в состояниях right0 или left0. При истинном значении empty автомобиль не может покинуть мост.

На рис. 6 представлена секция для полной программы управления движением по мосту. Константа d,

```
section St0 {
    const d : NAT
    axiom d > 0
    a: NAT := 0
    b: NAT := 0
    type Colour = enum {red, green}
    mtl: Colour := red
    itl:  Colour := red
    ml_out: BOOL := false
    il_out: BOOL := false
    empty: BOOL := false
    inv  a + b ≤ d
    inv mtl = red ∨ itl = red
}
```

Рис. 6. Секция для полной программы

```
process LightControl {
    zero: inv a=0 & mtl = red & itl = red & not empty;
          if (il_out) {itl := green; empty := true #right0}
          else if (ml_out & b< d) {mtl := green; empty := true #left0}
          else #zero
    right0: inv mtl = red & itl = green & empty;
           if (a=0) #right0 else { empty := false #right}
    right:  inv mtl = red & itl = green & not empty;
            a=0 → itl := red #zero
    left0:  inv itl = red & b< d & empty;
           if (a=0) #left0 else { empty := false #left }
    left:   inv itl = red & not empty;
           if (a+b=d) {mtl := red; #leftscan}
           else if (a=0) {mtl := red #zero }
           else #left
    leftscan: inv mtl = red & itl = red & not empty;
             a=0 → #zero
}
```

Рис. 7. Автоматная программа LightControl

переменные b, mtl, и itl определяются также как и в описанной выше реализации. Переменная a определяет число автомобилей на мосту в любом из направлений движения. Переменная c исключена.

На рис. 7 представлена автоматная программа LightControl. Каждое управляющее состояние снабжается инвариантом, который должен быть истинным в начале соответствующего сегмента.

4.4. Программы управления сенсорами

Простейшее функционирование сенсора представляется программой из двух сегментов включения и выключения сенсора:

```
off: #on
on: #off
```

В действительности, автоматные программы управления сенсорами нагружены вычислением значений переменных a, b, ml_out и il_out. Программы ML_out и IL_out реализуют переключения с on на off при зеленом светофоре.

Программы управления сенсорами должны успевать отслеживать изменения показаний сенсоров. Это следует сформулировать в виде следующих требований.

RF7. Включение сенсора при появлении на нем автомобиля должно быть зафиксировано раньше, чем автомобиль съедет с сенсора.

RF8. Выключение сенсора при съезде с него автомобиля должно быть зафиксировано раньше, чем другой автомобиль заедет на этот сенсор.

На пустом мосту, при a=0, невозможна ситуация "автомобиль покидает мост", т. е. переход с on на off. В действительности такого и не произойдет. Если автомобиль находится на сенсоре или заезжает на него, то мост не пуст. Поэтому необходимо вставить дополнительное условие: a>0. Без него оператор a:=a–1 окажется некорректным. Аналогичным образом, необходимо вставить условие b>0 для блокировки ситуации въезда на мост с острова при отсутствии на нем автомобилей.

Программы управления сенсорами представлены далее.

```
process ML_out {
    off: ml_out: = true #on
    on: if (mtl = green){ a:=a+1; ml_out:=false #on1
        } else #on
    on1: if (a + b = d){ mtl:=red #off } else #off
}
```

В программе ML_out при зеленом светофоре mtl в состоянии on автомобиль въезжает на мост, увеличивая число автомобилей на мосту. Если их число на мосту и острове достигает d, то тут же в состоянии on1 светофор mtl переключается на красный. В принципе, такое переключение должно немедленно произойти в состоянии left программы LightControl. Если удалить установку mtl:=red в состоянии on1, то потенциально, хотя и крайне маловероятно, следующий автомобиль успеет съехать с материка на мост ранее, чем программа LightControl выполнит оператор mtl:=red в состоянии left.

```
process IL_in {
    off: a>0, not empty, itl=red → #on
    on: a:=a-1; b:=b + 1 #off
}
```

В программе IL_in в сегменте off используется условие **not** empty для предотвращения дедлока в состоянии left0. Отметим, что в случае истинного значения empty автомобиль все равно съедет с моста, однако исполнение сегмента off будет задержано до погашения признака empty в сегменте left0. Такая задержка при отсутствии других автомобилей не является критичной.

Если удалить условие itl=red в сегменте off программы IL_in, то автомобиль, только что въехавший на мост с острова, сможет вернуться назад по данному сегменту off. Понятно, что в действительности такого не произойдет, поскольку обратное движение запрещено. Здесь же приходится блокировать такую возможность вставкой условия itl=red.

```
process ML_in {
    off: itl=green, a>0, not empty → #on
    on: a:=a-1 #off
}
```

В программе ML_in используется условие itl=green. Это условие не проверяется при съезде с моста на материк в силу отсутствия там светофора. Данное условие блокирует возможность возвращения автомобиля, только что въехавшего на мост с материка.

```
process IL_out {
    off: b>0 → il_out:=true #on
    on: itl=green → b:=b-1, a:=a+1, il_out:=false #off
}
```

Когда один автомобиль заезжает на мост, а другой автомобиль съезжает с моста, две разные программы, исполняемые параллельно, будут пытаться, возможно, одновременно, изменить значение переменной a. Подобное становится критичным, когда программы

управления сенсорами реализуются разными процессорами.

Чтобы исключить возможность одновременного доступа к переменной a разными процессами, в работе [19] используется монитор для пересчета переменных a и b, управляемый семафорами.

5. Опыт верификации в системе Rodin

Программа управления движением автомобилей по мосту, представленная выше, была закодирована в виде спецификации на языке Event-B [1] и верифицирована в инструменте Rodin [2]. Доступна реализация в Rodin: https://persons.iis.nsk.su/files/persons/pages/bridge.zip

Кодирование переходов по управляющим состояниям проводится методом, аналогичным Switch-технологии [20]. Управляющие состояния программы LightControl кодируются как значения переменной state типа перечисления State:

type State = **enum** { zero, right0, right, left0, left, leftscan }

Сегмент кода **M:** A **#L** заменяется оператором **if** (state = **M**) {A; state := **L**}. Гиперграфовая композиция сегментов кода заменяется одним бесконечным циклом, тело которого — недетерминированная композиция модифицированных сегментов. Предварительно сегменты кода будет удобнее представить в виде правил. Программа LightControl преобразуется к виду, показанному на рис. 8.

Каждая альтернатива недетерминированной композиции непосредственно переписывается в виде события на языке Event-B [1]. Например, альтернатива для управляющего состояния leftscan представляется следующим событием:

```
EVENT Leftscan
WHERE
    grd1: state = leftscan
    grd2: a = 0
THEN
    act1: state := zero
END
```

Инварианты управляющих состояний могут быть преобразованы в общие инварианты. Они оказываются полезными при доказательстве формул корректности. Например, инвариант для управляющего состояния zero может быть представлен общим инвариантом:

state = zero ⇒ a=0 & mtl=red & itl=red & **not** empty

Кодирование автоматных программ для сенсоров можно провести без управляющих состояний off и on, используя переменные ml_out и il_out. Для программ IL_in и ML_in применяется автоматная трансформация [8], заменяющая композицию

```
off: A #on
on: B #off
```

одним сегментом off: A; B #of. Далее оператор A; B кодируется событием Event-B.

Пять автоматных программ, исполняемых параллельно в полной программе управления движением на мосту, кодируются в одной машине на языке Event-B. Итоговая смесь всех событий в рамках одной машины фактически реализует полное перемешивание (интерливинг) атомарных сегментов пяти автоматных программ.

Сто сгенерированных формул корректности инвариантов были доказаны автоматически в системе Rodin. Кроме одной формулы, доказанной с помощью интерактивных средств доказательства. Инварианты управляющих состояний были преобразованы в общие инварианты и добавлены к двум основным инвариантам программы.

Дальнейшая верификация проводилась применением *аниматора* — компонента Rodin, реализующего пошаговое исполнение спецификации Event-B с визуализацией текущего состояния, возможностью выбора любого из возможных вариантов исполнения и возможностью вернуться назад в процессе исполнения. Теоретически число вариантов исчисляется астрономическим числом. Однако фактически в каждый момент доступны для исполнения не более трех событий — остальные события заблокированы, поскольку их охранные условия ложные. Поэтому оказалось возможным эффективно перебрать с помощью аниматора ключевые варианты исполнения и обнаружить следующие ошибки в программе управления движением на мосту.

Сначала (первая ошибка) был обнаружен дедлок в состояниях right0 и left0, когда первый автомобиль на мосту выезжает с него раньше срабатывания сегмента в состояниях right0 или left0. Был введен признак empty для пустого моста. Вторая ошибка:

```
process LightControl {
  State state := zero;
  Cycle: [Zero1:] state=zero, il_out → itl := green, empty := true, state:=right0 |
         [Zero2:] state=zero, ml_out, b<d → mtl := green, empty := true, state:=left0 |
         [Right0:] state= right0, a ≠ 0 → empty := false, state := right |
.....................................................................................
         [Leftscan:] state = leftscan, a = 0 → state := zero
       #Cycle
}
```

Рис. 8. Кодирование управляющих состояний программы LightControl

дедлок в состоянии left0 при b=d. Для исправления ошибки введено дополнительное охранное условие b<d в состоянии zero. Третья ошибка обнаружена на одном из вариантов анимации. После въезда на мост первого автомобиля с острова этот автомобиль неожиданно возвращался на остров. Здесь оказалось возможным срабатывание сегмента off процесса IL_in. Для исправления ошибки введено дополнительное охранное условие itl=red.

Таким образом, система Rodin показала свою эффективность при верификации автоматных программ.

Четвертая ошибка обнаружена при написании статьи. В программе ML_out в сегменте on использовалось условие a+b<d для пропуска автомобиля на мост с материка. Отметим, что сенсор реагирует только на появление на нем или съезде с него автомобиля. А автомобиль может съехать только при зеленом светофоре; условие a+b<d не может воздействовать на автомобиль. После пересчета a:=a+1 в сегменте on далее сегмент left программы LightControl при истинности условия a+b=d устанавливает красный светофор. Потенциально, хотя и маловероятно, следующий автомобиль может успеть въехать на мост раньше (при зеленом светофоре), чем сегмент left выполнит оператор mtl:=red. По этой причине проверка a+b=d и установка mtl:=red должны проводиться в программе ML_out сразу же после пересчета a:=a+1 в сегменте on. После исправления данной ошибки для доказательства формул корректности потребовался дополнительный инвариант: @thm_ml_out a+b = d ⇒ mtl=red.

Удаление переменной c оказалось неправильным решением, усложняющим программу. Это решение спровоцировало третью ошибку, усложнило доказательство, а также потребовало вставить дополнительную проверку itl=green в сегменте on программы ML_in.

6. Обзор работ

Автоматное программирование. Автоматные методы программирования заложены во многих известных языках, таких как SDL [21], UML, TLA+ [22], Event-B [1], Дракон [23], а также LD, FBD и SFC, определяемых стандартом IEC 61131-3 для программируемых логических контроллеров. Большинство этих языков являются графическими.

Автоматное программирование [5—8, 20, 24] развивается исключительно в России, начиная с 1990-х годов. Оно доказало свою эффективность на множестве разнообразных приложений. Автоматная программа конструируется в виде гиперграфовой композиции сегментов кода, более общей в сравнении с композициями операторов в традиционном программировании. Каждый сегмент кода метится управляющим состоянием и реализует независимую подзадачу, условие которой частично отражается инвариантом управляющего состояния. Концепция автоматного программирования на базе управляющих состояний ранее предложена в работах [20, 24].

Задача управления движением автомобилей по мосту. Задача используется в работе [25] для иллюстрации метода построения спецификации Event-B снизу, начиная с более простых компонентов (сенсоров и светофоров), которые затем включаются в основную спецификацию. Механизм построения снизу интегрирован с техникой уточнений, реализуемой сверху. Ранее предлагались другие методы конструирования спецификации из независимых модулей. Модульная декомпозиция реализована в классическом методе B [26], однако была утеряна при разработке Event-B.

В работе [25] построены универсальные модели сенсоров и светофоров, каждая применением трех уточнений. Основная модель строится в виде цепочки из восьми машин, полученных уточнениями и включениями подмоделей. Итоговая реализация получилась громоздкой — одна из машин с включенными подмоделями длиною в 379 строк. Единственная машина, полученная из пяти автоматных программ, содержит 165 строк.

Применение уточнений при разработке модели управления движением на мосту в руководстве по Event-B [1] и в работе [25] не упрощает модель, а усложняет ее. Это обнаруживается при сопоставлении с реализацией в автоматном программировании.

Кодирование автоматных программ. Существуют разные способы представления автоматных программ:

- стиль языка Фортран, используемый в представленном подходе;
- графические языки;
- кодирование по Switch-технологии [20];
- использование рекурсивных вызовов вместо операторов перехода;
- предметно-ориентированные языки (DSL).

Язык автоматного программирования сочетает в себе стили языков Си и Фортран. Подобный стиль встречается также в программах ядра ОС Linux[1].

Использование меток в TLA+ [22] внешне похоже на управляющие состояния в автоматном программировании. Декларировано, что метки всего лишь фиксируют участки атомарности в спецификации TLA+.

Алгоритм на графическом языке Дракон [23] эргономично представляет гиперграфовую композицию автоматной программы: ветка алгоритма непосредственно соответствует сегменту кода; все переходы по именам веток реализуются по контуру алгоритма.

В Switch-технологии [20] управляющие состояния становятся значениями некоторой переменной, а сегменты кода — альтернативами оператора switch. При таком преобразовании программа становится длиннее, сложнее и менее эффективной. В этом можно убедиться, сравнив программу LightControl (подразд. 4.3) с ее преобразованной версией в разд. 5. Для языков, не содержащих оператора перехода, это лучший способ кодирования автоматных программ.

Сегмент кода автоматной программы можно представить в виде рекурсивной функции. Аргументами функции являются переменные состояния програм-

[1] Например, https://elixir.bootlin.com/linux/v5.11.8/source/kernel/bpf/hashtab.c#L1467

мы. Функция вычисляет набор значений модифицированного состояния программы при завершении исполнения сегмента. Оператор перехода на другой сегмент заменяется вызовом рекурсивной функции для соответствующего сегмента. Преобразованная программа длиннее и сложнее, чем в предыдущих способах кодирования автоматных программ. Данный способ применяется в алгебрах процессов CCS [27], CSP [28] и др., а также в функциональном языке Erlang [29], ориентированном на разработку распределенных вычислительных систем. Отметим, что гиперграфовая композиция могла бы стать другим базисом для построения алгебр процессов.

Худшим способом является кодирование автомата программы в специальном предметно-ориентированном языке (*domain-specific language*, DSL). Это, например, языки AbC [30] и Microsoft P [31, 32]. Набор элементарных конструкций в таких языках ограничен. Исполнение программ реализуется через интерпретатор. Имеется также возможность трансляции программы на язык Си.

В событийно-ориентированной (*event-driven*) парадигме [33] программа представлена в виде цикла, в котором последовательно просматривается определенный набор событий (сообщений). Для всякого события запускается соответствующий обработчик события. Автоматная программа непредставима в такой схеме, особенно когда обработка одного события реализуется в разных сегментах кода.

Заключение

В статье определен новый язык автоматного программирования на базе языка спецификаций Event-B [1]. Его использование дает возможность разработки программ по технологии автоматного программирования с применением разнообразных эффективных инструментов верификации системы Rodin [2]. Такая возможность особенно актуальна для разработки систем управления в критической инфраструктуре с высокой ценой ошибки. Наличие трудно обнаруживаемых ошибок в распределенных системах — типичное явление, что убедительно продемонстрировано в разд. 5.

Спецификация Event-B представляет собой цепочку уточнений машин. Машина определяется недетерминированной композицией событий. А событие — эквивалент простого условного оператора. В автоматном программировании кроме недетерминированной композиции допускается ряд других. Основной является гиперграфовая композиция по управляющим состояниям. Возможны также параллельная композиция процессов, объектно-ориентированная и аспектно-ориентированная композиции. Процесс может быть вызван из другого процесса. Имеются также механизм сообщений и действий со временем. Преимущества автоматного программирования продемонстрированы на известной задаче управления движением автомобилей на мосту.

Автоматную программу нетрудно переписать в спецификацию Event-B. Техника такого переписывания показана в разд. 5. Однако предварительно следует устранить вызовы процессов, вызовы программ-функций, определяемых независимо от автоматной программы, и вызовы предикатов, используемых в формулах. Вместо вызова программы-функции можно использовать ее спецификацию через before-after-предикат. Иногда трудно найти эквивалент в Event-B для последовательного оператора: A; B. Здесь можно воспользоваться автоматной трансформацией, заменяющей сегмент M: A; B #N комбинацией M: A #K и K: B #N.

Полноценная реализация системы автоматного программирования должна предусматривать генерацию формул корректности для автоматных программ в интеграции с традиционной системой верификации на базе логики Хоара [9] для фрагментов автоматных программ, относящихся к классу программ-функций. Верификация автоматных программ должна быть поддержана инструментами, которые по своим возможностям интегрируют системы Rodin [2] и Why3 [17].

Доступная авторская реализация в Rodin задачи управления движением автомобилей на мосту представлена по ссылке https://persons.iis.nsk.su/files/persons/pages/bridge.zip.

Список литературы

1. **Abrial J.-R.** Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586 p.
2. **Rodin** User's Handbook. Version 2.8 / Jastram M. (editor). 2014. 184 p.
3. **Butler M. J., Körner P., Krings S., Lecomte T., Leuschel M., Mejia L.-F., Voisin L.** The First Twenty-Five Years of Industrial Use of the B-Method // Formal Methods for Industrial Critical Systems (FMICS). 2020. P. 189—209.
4. **Карнаухов Н. С., Першин Д. Ю., Шелехов В. И.** Язык предикатного программирования Р. Новосибирск: ИСИ СО РАН, 2018. 45 с. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf
5. **Шелехов В. И.** Язык и технология автоматного программирования // Программная инженерия. 2014. № 4. С. 3—15. URL: http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf
6. **Тумуров Э. Г., Шелехов В. И.** Технология автоматного программирования на примере программы управления лифтом // Программная инженерия. 2017. Том 8, № 3. С. 99—111. URL: http://novtex.ru/prin/full/pi317_web-99-111.pdf
7. **Шелехов В. И.** Разработка автоматных программ на базе определения требований // Системная Информатика. 2015. № 1. С. 1—29. URL: http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
8. **Шелехов В. И.** Оптимизация автоматных программ методом трансформации требований // Программная инженерия. 2015. № 11. С. 3-13. URL: http://novtex.ru/prin/full/pi1115_web-3-13.pdf
9. **Hoare C. A. R.** An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576—585.
10. **Floyd R. W.** Assigning meanings to programs // Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. AMS, 1967. P. 19—32.
11. **Шелехов В. И.** Классификация программ, ориентированная на технологию программирования // Программная инженерия. 2016. Том 7, № 12. С. 531—538. URL: http://novtex.ru/prin/full/pi1216_web-531-538.pdf
12. **Liskov B., Zilles S.** Programming with abstract data types // Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. 1974. SIGPLAN Notices 9. P. 50—59.
13. **Systems** and software engineering — Life cycle processes — Requirements engineering. ISO/IEC/ IEEE 29148, 2011, 95 p.
14. **Шелехов В. И.** Разработка программы построения дерева суффиксов в технологии предикатного программирования // Препр. ИСИ СО РАН, № 115. Новосибирск, 2004. 52 с.

15. **Шелехов В. И.** Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования // Препр. ИСИ СО РАН, № 164. Новосибирск, 2012. 30 с.
16. **Owre S., Rushby J. M., Shankar N.** PVS: Specification and Verification System // CADE-11: Automated Deduction. LNCS. 1992. Vol. 607. P. 748—752.
17. **Why 3.** Where Programs Meet Provers. URL: http://why3.lri.fr
18. **The Coq Proof Assistant.** URL: https://coq.inria.fr/
19. **Шелехов В. И.** Сравнение технологий автоматного программирования и Event-B // Системная информатика. 2021. № 18. С. 53—84. URL: https://persons.iis.nsk.su/files/persons/pages/bridge.pdf
20. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. URL: http://is.ifmo.ru/books/switch/1
21. **Specification** and description language (SDL). ITU-T Recommendation Z.100 (03/93). URL: http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf
22. **Lamport L.** Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2021. 382 p.
23. **Пароджанов В. Д.** Язык ДРАКОН. Краткое описание. М., 2009. 124 с. URL: http://drakon.su/_media/biblioteka/drakondescription.pdf
24. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. 2-е изд. СПб.: Питер, 2020. 176 с.
25. **Hoang T. S., Dghaym D., Snook C., Butler M.** A Composition Mechanism for Refinement-Based Methods // 22nd International Conference on Engineering of Complex Computer Systems (ICECCS), 2017. P. 100—109.
26. **Abrial J.-R.** The B-book — assigning programs to meanings. Cambridge University Press, 2005. 816 p.
27. **Milner R.** Communication and Concurrency. International Series in Computer Science. Prentice Hall, 1989. 260 p.
28. **Hoare C. A. R.** Communicating Sequential Processes. Prentice Hall International. 1985. 260 p.
29. **Larson J.** Erlang for concurrent programming // ACM Queue. 2008. No. 5. P. 18—23.
30. **Nicola R. D., Duong, T., Inverso O.** Verifying AbC Specifications via Emulation // Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles, ISoLA 2020, Part II. LNCS 12477. P. 261—279.
31. **Liu P., Wahl T., Lal A.** Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers // Computer Aided Verification. 2019. LNCS 11562. P. 386—404.
32. **Desai A., Qadeer S., Seshia S. A.** Programming safe robotics systems: Challenges and ad-vances. // Leveraging Applications of Formal Methods, Verification and Validation. Ver-ification. 2018. LNCS 11245. P. 103—119.
33. **Ferg S.** Event-Driven Programming: Introduction, Tutorial, History. SourceForge, 2006. 59 p.

Automata-based Software Engineering with Event-B

V. I. Shelekhov, vshel@iis.nsk.su, A. P. Ershov Institute of Informatics Systems, Novosibirsk, 630090, Russian Federation

Corresponding author:

Shelekhov Vladimir I., Head of Laboratory, A. P. Ershov Institute of Informatics Systems, Novosibirsk, 630090, Russian Federation
E-mail: vshel@iis.nsk.su

Received on February 13, 2022
Accepted on February 24, 2022

A new automata-based programming language has been built by extending the Event-B specification language. When developing models in Event-B, it becomes possible to use automata-based methods in addition to the popular refinement method. Automata-based software engineering, supported by deductive verification in Event-B, can be successfully used for the development of control systems in critical infrastructure with a high cost of error.

A model of the Event-B specification in the automata-based programming language is constructed. The Event-B specification is a chain of machine refinements. The machine is defined by a non-deterministic composition of events. An event is the equivalent of a simple conditional statement without else branch. In automata-based software engineering, in addition to non-deterministic composition, a number of other compositions are allowed. The main one is a hypergraphic composition with respect to control states. Parallel process composition, object-oriented and aspect-oriented compositions are also possible. A process can be called from another process. It is possible to send and receive messages. There are different time actions. It is not difficult to rewrite an automata-based program into the Event-B specification.

The automata-based software engineering with Event-B is demonstrated by the example of the problem of traffic control on the bridge from the Event-B system manual. A simpler solution with verification in the Rodin tool is proposed. The effectiveness of Event-B verification methods is confirmed by finding three non-trivial errors in our solution.

Keywords: *automata-based engineering, Event-B, refinement, requirements, program transformation, deductive verification, functional programming*

For citation:

Shelekhov V. I. Automata-based Software Engineering with Event-B, *Programmnaya Ingeneria,* 2022, vol. 13, no. 4, pp. 155—167.

DOI: 10.17587/prin.13.155-167

References

1. **Abrial J.-R.** *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2010, 586 p.
2. ***Rodin*** *User's Handbook*. Version 2.8, Jastram M. (editor), 2014, 184 p.
3. **Butler M. J., Körner P., Krings S., Lecomte T., Leuschel M., Mejia L.-F., Voisin L.** The First Twenty-Five Years of Industrial Use of the B-Method, *Formal Methods for Industrial Critical Systems* (FMICS), 2020, pp. 189—209.
4. **Kharnaukhov N., Perchine D., Shelekhov V.** *The predicate programming language P,* Novosibirsk, ISI SB RAN, 2018, 45 p. (in Russian).
5. **Shelekhov V. I.** Automata-based software engineering: the language and development methods. *Programmnaya Ingeneria,* 2014, no. 4, pp. 3—15 (in Russian).
6. **Tumurov E. G., Shelekhov V. I.** Applying Automata-based Software Engineering for the Lift Control Program, *Programmnaya Ingeneria,* 2017, vol. 8, no. 4, pp. 99—111 (in Russian).
7. **Shelekhov V. I.** Automata-based programming on the base of requirements specification, *Sistemnaya Informatika*, 2015, no. 1, pp. 1—29 (in Russian).
8. **Shelekhov V. I.** Automata-based Program Optimization by Applying Requirement Transformations, *Programmnaya Ingeneria*, 2015, no. 11, pp. 3—13 (in Russian).
9. **Hoare C. A. R.** An axiomatic basis for computer programming, *Communications of the ACM,* 1969, vol. 12 (10). pp. 576—585.
10. **Floyd R. W.** Assigning meanings to programs, *Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science*, AMS, 1967, pp. 19—32.
11. **Shelekhov V. I.** Program Classification in Software Engineering, *Programmnaya Ingeneria,* 2016, vol. 7, no. 12, pp. 531—538 (in Russian).
12. **Liskov B., Zilles S.** Programming with abstract data types, *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages,* SIGPLAN Notices. 9, 1974, pp. 50—59.
13. **Systems** and software engineering — Life cycle processes — Requirements engineering. ISO/IEC/ IEEE 29148, 2011, 95 p.
14. **Shelekhov V. I.** Development of a program for building a suffix tree in the predicate software engineering. Preprint, no. 115, Novosibirsk, ISI SB RAN, 2004. 52 p. (in Russian).
15. **Shelekhov V.** Verification of the heapsort predicate program using inverse transformations, *Sistemnaya informatika*, 2020, no. 16, pp. 75—102 (in Russian).
16. **Owre S., Rushby J. M., Shankar N.** PVS: Specification and Verification System, *CADE-11: Automated Deduction. LNCS,* 1992, vol. 607, pp. 748—752.
17. **Why 3.** Where Programs Meet Provers, available at: http://why3.lri.fr
18. **The Coq** Proof Assistant, available at: http://coq.inria.fr
19. **Shelekhov V. I.** Comparison of automata-based engineering method and Event-B modeling method, *Sistemnaya informatika*, 2021, no. 18, pp. 53—84 (in Russian).
20. **Shalyto A. A.** *SWITCH-technology. Algorithmic and Programming Methods in Solution of Logic Control Problems*, St. Petersburg, Nauka, 1998, 628 p. (in Russian).
21. **Specification** and description language (SDL). ITU-T Recommendation Z.100 (03/93), available at: http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf
22. **Lamport L.** *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2021, 382 p.
23. **Parondzanov V. D.** *DRAKON language,* Short description, Moscow, 2009, 124 p. (in Russian).
24. **Polykarpova N. I., Shalyto A. A.** *Automata-base programming.* Second edition, St. Petersburg, Piter, 2020, 176 p. (in Russian).
25. **Hoang T. S., Dghaym D., Snook C., Butler M.** A Composition Mechanism for Refinement-Based Methods, *22nd International Conference on Engineering of Complex Computer Systems* (ICECCS), 2017, pp. 100—109.
26. **Abrial J.-R.** *The B-book — assigning programs to meanings*, Cambridge University Press, 2005, 816 p.
27. **Milner R.** *Communication and Concurrency,* International Series in Computer Science, Prentice Hall, 1989, 260 p.
28. **Hoare C. A. R.** *Communicating Sequential Processes*, Prentice Hall International, 1985, 260 p.
29. **Larson J.** Erlang for concurrent programming, *ACM Queue*, 2008, no. 5, pp. 18—23.
30. **Nicola R. D., Duong, T., Inverso O.** Verifying AbC Specifications via Emulation, *Leveraging Applications of Formal Methods*, Verification and Validation: Engineering Principles, ISoLA 2020, Part II, LNCS 12477, pp. 261—279.
31. **Liu P., Wahl T., Lal A.** Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers, *Computer Aided Verification*, LNCS 11562, 2019, pp. 386—404.
32. **Desai A., Qadeer S., Seshia S. A.** Programming safe robotics systems: Challenges and advances. *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, 2018, LNCS 11245, pp. 103—119.
33. **Ferg S**. *Event-Driven Programming: Introduction*, Tutorial, History, SourceForge, 2006, 59 p.

ИНФОРМАЦИЯ

Всероссийская научно-техническая конференция

"Многопроцессорные вычислительные и управляющие системы"

МВУС-2022

27—30 июня 2022 г.
Таганрог, Ростовская область

Конференция посвящена 100-летию со дня рождения выдающегося российского ученого, Героя социалистического труда, академика РАН А. В. Каляева и развитию его научных идей.

Целью конференции является представление достижений российских ученых в развитии фундаментальных исследований и прикладных разработок в области многопроцессорных вычислительных и управляющих систем.

Направления работы

- Академик РАН А. В. Каляев и его научная школа
- Архитектура многопроцессорных вычислительных и управляющих систем
- Математическое и системное программное обеспечение многопроцессорных вычислительных и управляющих систем
- Реконфигурируемые вычислительные системы
- Проблемно-ориентированные и встраиваемые вычислительные системы
- Многопроцессорные вычислительные и управляющие системы в системах искусственного интеллекта

Подробности: **https://conf.mvs.sfedu.ru/**