УДК 004.05

Сравнение технологий автоматного программирования и Event-B

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

Показано, что спецификация на языке Event-B представима автоматной программой в виде недетерминировнной композиции простых условных операторов, что соответствует узкому подклассу автоматных программ. Спецификация в Event-B является монолитной. Для построения спецификации нет других средств композиции, кроме уточнения, реализующего расширение ранее построенной спецификации.

Сравнение технологий автоматного программирования и Event-B проводится на примере двух задач. Предыдущие решения задачи управления движением на мосту в системе Event-B являются сложными и громоздкими. Предложено более простое решение с верификацией программы в инструменте Rodin. Эффективность методов верификации в Event-B подтверждена нахождением трех нетривиальных ошибок в нашем решении.

Ключевые слова: *автоматное программирование, Event-B, уточнение (refinement), требования, дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование.*

1. Введение

Event-B [10] – это метод формальной спецификации и верификации систем в программной и системной инженерии, успешно используемый при разработке производственных систем управления, особенно в железнодорожном транспорте. Сила метода Event-B в том, что этот метод гарантирует обнаружение многих серьезных ошибок с помощью доказательства теорем, что нереализуемо другими известными методами. Несмотря на большое число разнообразных приложений, существует ряд факторов, сдерживающих широкое внедрение подхода Event-B, что отмечается в обзоре [11].

Автоматное программирование [1, 2, 5, 6, 9] ориентировано на класс реактивных систем, реализующих взаимодействие с внешним окружением системы и реагирующих на определенный набор событий (сообщений). Автоматная программа определяет конечный автомат в виде гиперграфовой композиции сегментов кода. Технология автоматного

программирования предлагает комплекс методов для разработки, оптимизации и верификации автоматных программ.

В настоящей работе проводится сравнение технологий автоматного программирования и Event-B. Определена модель спецификации на языке Event-B в языке автоматного программирования. Спецификация Event-B отображается в недетерминированную композицию простых условных операторов. Таким образом, язык Event-B оказывается узким подъязыком автоматного программирования, в котором уточнение (refinement) – единственное средство декомпозиции спецификаций. Тогда как автоматное программирование кроме недетерминированной композиции допускает параллельную композицию процессов, гиперграфовую композицию по управляющим состояниям, подпроцессы, объектно-ориентированную и аспектно-ориентированную композиции.

Сопоставление технологий проводится также на двух примерах, которые в руководствах по Event-B приводятся для обучения технологии Event-B. Для этих примеров построены автоматные программы, которые затем были закодированы в виде спецификации Event-B и верифицированы в инструменте Rodin [22].

Во втором разделе настоящей статьи дается описание языка и технологии автоматного программирования. В третьем разделе описывается подход Event-B и его моделирование в автоматном программировании. В четвертом разделе технологии Event-B и автоматного программирования иллюстрируются на примере управления движением на перекрестке. В следующем разделе для примера управления движением автомобилей на мосту иллюстрируются уточнения, применяемые в Event-B. Детально описывается разработка автоматной программы, исправляющей недостатки реализации в Event-B. Обзор работ в шестом разделе. В заключении основные выводы по сравнению технологий. В Приложении 1 представлена спецификация на языке Event-B автоматной программы управления движением на мосту.

2. Автоматное программирование

2.1. Классы программ

Методы программной инженерии, доказавшие свою эффективность, не всегда успешно применимы для всех программ. Причина здесь в различиях архитектур программ. Это ставит задачу классификации программ, то есть построения системы классов программ и разработку адекватной технологии программирования для каждого класса программ. Теория

программ каждого класса должна определять методы спецификации, верификации, моделирования и эффективной реализации программ.

Генеральная классификация [4] определяет: невзаимодействующие программы (или программы-функции), реактивные системы (или программы-процессы), языковые процессоры и операционные среды. Имеются другие классы.

2.2. Программы-функции (невзаимодействующие программы)

Программа принадлежит классу программ-функций, 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 собран в конце программы, то такая программа относится к классу невзаимодействующих программ. Программа обязана всегда завершаться, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Следовательно, программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов.

Гиперфункция с несколькими ветвями [1, 7, 8] – наиболее общая форма программы-функции. Гиперфункция с одной ветвью определяет обычную функцию. Каждая *ветвь* гиперфункции определяет независимый выход из программы и набор результатов по этой ветви. Наборы результатов по разным ветвям могут различаться. Набор результатов ветви может быть пустым. Гиперфункцию можно закодировать в виде программы-функции с одной ветвью.

В программе на языке Си только один результат. Дополнительные результаты поставляются через аргументы-указатели. В некоторых языках несколько результатов оформляются в виде типа «кортеж» (**tupple**). В языках WhyML [24] и Java дополнительные выходы из программы реализуются с помощью исключений. Результатами выхода по исключению являются параметры исключения. В языке предикатного программирования Р [1] выход по ветви гиперфункции реализуется оператором перехода следующего вида:

#<имя ветви гиперфункции>

Спецификация программы-функции представляется двумя предикатами: *предусловием*, ограничивающим значения аргументов программы, и *постусловием*, определяющим связь между значениями аргументов и результатов. Для гиперфункции предусловие и постусловие определяются по каждой ветви. Корректность программы-функции относительно спецификации реализуется доказательством формул корректности, генерируемых на базе логики Хоара [16].

2.3. Программы-процессы

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 конечный автомат в виде гиперграфа с набором управляющих состояний в качестве вершин и набором сегментов в качестве ориентированных гипердуг. Гиперграфовая структура автоматной программы является продолжением аппарата гиперфункций. *Состояние* автоматной программы определяется значениями набора переменных, модифицируемых в программе, за исключением локальных переменных.

Программа-процесс состоит из следующих частей:

- требований к реактивной системе;
- набора автоматных программ;
- секций, определяющих состояние автоматных программ и программы-процесса в целом.

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 другие. Разработка требований должна проводиться в соответствии со стандартом ISO/IEC/ IEEE 29148 [23].

2.4. Язык автоматного программирования

Язык автоматного программирования может быть построен расширением некоторого *базисного* языка для класса программ-функций. Это может быть любой язык императивного или функционального программирования. Таким способом определено автоматное расширение [2, 9] языка предикатного программирования P [1, разд.10].

Определим конструкции языка автоматного программирования.

Секция состояния автоматной программы представляется конструкцией:

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

Секция помещается перед автоматной программой. Имя секции может отсутствовать. Секция может быть построена расширением другой секции при наличии **extends** <Имя секции>.

Автоматная программа определяется следующей конструкцией:

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

Аргументы могут отсутствовать. Произвольный <Сегмент кода> представляется 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.

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

Структура управления автоматной программы аналогична используемой в языке Фортран, применяемом в основном для задач вычислительной математики, но не для реактивных систем.

Сегмент кода следующего вида:

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

может быть записан в виде правила [6]:

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

Сегмент **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 другие.

2.5. Технология автоматного программирования

Управляющие состояния и соответствующие им сегменты представляют различные стадии функционирования реактивной системы и, таким образом, определяют естественное структурирование автоматной программы. Некоторые управляющие состояния заложены уже в требованиях. Специализация для каждого управляющего состояния уменьшает число связей между объектами, что принципиально понижает сложность реализации реактивной системы. Отметим, что сложность экспоненциально зависит от числа переменных в состоянии и числа связей между ними.

Достаточно часто автоматная программа строится простым переписыванием функциональных требований. Их удобнее формулировать в виде логических правил [6].

Для получения более эффективной программы применяются эквивалентные трансформации, существенно меняющие структуру автоматной программы [5]. Гиперграфовая композиция является предельно гибкой. Показано, что любую автоматную программу можно представить композицией двух сегментов. Гиперграфовая композиция трех независимых процессов позволила упростить программу управления лифтом [2].

В дополнение к описанным методам иногда полезно использовать методы объектно- и аспектно-ориентированного программирования.

Технология автоматного программирования также представлена сводом правил [2].

Циклы в автоматной программе имеют другую природу по сравнению с циклами императивной программы. *Не рекомендуется использовать циклы типа* ***while*** *и* ***for*** *для конструирования автоматной программы*.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая, однако, будет значительно сложнее аналогичной функциональной или императивной программы, построенной обычными средствами. Поэтому *не следует использовать автоматное программирование для программ-функций*. Сегменты автоматных программ могут содержать достаточно крупные фрагменты кода, относящиеся к программам-функциям. *Рекомендуется выносить такие фрагменты из автоматных программ и программировать независимыми функциями*.

2.6. Спецификация и верификация автоматных программ

Методы верификации, успешно применяемые для программ-функций, в частности логика Хоара-Флойда [16, 14], непригодны для верификации автоматных программ. Эти методы могут применяться лишь для фрагментов автоматных программ, соответствующих программам-функциям.

Спецификация программы-процесса определяется общими инвариантами и инвариантами управляющих состояний. *Общий инвариант* формулируется для автоматной программы в секции состояния непосредственно перед автоматной программой. Общий инвариант должен быть истинным в начале каждого сегмента. Общий инвариант может быть представлен темпоральной формулой.

Допустим, состояние автоматной программы определяется набором переменных v. Для текущего сегмента построим функцию F(v) таким образом, чтобы формула $v' = F(v)$, где v' – значение набора переменных в конце сегмента, была точной спецификацией текущего сегмента. Текущий сегмент сохраняет общий инвариант Inv(v), если из истинности Inv(v) в начале сегмента следует его истинность в конце сегмента. Сохранность общего инварианта для текущего сегмента гарантируется проведением следующего доказательства:

$$Inv(v)\ \&\ Sinv(v) \vdash Inv(F(v))$$

где Sinv(v) – инвариант управляющего состояния.

Истинность инварианта управляющего состояния M доказывается для каждого оператора перехода #M.

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

Инварианты управляющих состояний часто являются слабыми или отсутствуют, то есть оказываются тождественно истинными. Инварианты управляющих состояний и общие инварианты принципиально отличаются от инвариантов циклов императивной программы.

Доказательство истинности всех инвариантов программы-процесса и отсутствия дедлоков позволяет избежать многих ошибок, но не гарантирует полной корректности программы в отличие от верификации программ-функций, где правильность спецификации и доказательство формул корректности гарантирует корректность программы.

3. Event-B и модель Event-B в автоматном программировании

Event-B [10] – это метод формальной спецификации и верификации систем в программной и системной инженерии с использованием нотации теории множеств и логики первого

порядка. Каждая спецификация на Event-B состоит из компонентов двух видов: контекстов и машин. *Контекст* определяет множества и константы – статическую часть спецификации. *Машина* содержит динамическую часть спецификации: переменные, инварианты, события. Значения переменных формируют текущее *состояние* спецификации. *Инварианты* определены на множестве переменных.

Текущее состояние спецификации может быть изменено событием. *Событие* состоит из названия, параметров, охранных условий и действий. *Охранные условия* ограничивают множество возможных состояний, в которых данное событие может произойти. *Действия* модифицируют значения переменных, изменяя текущее состояние спецификации. Инварианты должны сохраняться после модификаций переменных любыми действиями данной машины. Корректность любого изменения состояния необходимо доказывать.

Произвольное событие далее будем представлять в виде оператора:

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

В типичном случае действие реализует присваивание нескольким переменным. Например, { c:=C; d:=D }. Выражения C и D вычисляются и их значения одновременно присваиваются переменным c и d. Эффект более сложного действия можно определить before-after предикатом, связывающим значения исходных и модифицированных переменных.

Все события атомарны и могут произойти, если выполняются их охранные условия. Если одновременно выполняются охранные условия нескольких событий, то только одно из них может произойти в данный момент, причем какое именно событие произойдет, выбирается недетерминированным образом.

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.

Спецификация на Event-B состоит из последовательности машин, в которой очередная машина является *уточнением* (refinement) 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.

В Event-B имеется возможность доказательства отсутствия состояний взаимной блокировки (дедлока), а также доказательства завершения конечных процессов.

Event-B определяет среду для разработки и верификации спецификаций на платформе Rodin [22]. Доказательство генерируемых для спецификации формул корректности поддерживается автоматическими и интерактивными средствами доказательства с применением SMT-решателей. Степень и возможности автоматизации доказательства существенно выше, чем в системах доказательства PVS [21], Why3 [24] и Coq [12]. В частности, автоматизировано применение эквивалентных замен (rewrite). В дереве текущего процесса доказательства явно обозначены позиции, по которым пользователь может продолжить доказательство, выбрав одну из прилагаемых в позиции альтернатив.

Построим новый язык автоматного программирования расширением языка спецификаций Event-B. Некоторые конструкции языка Event-B заменяются другими, более привычными. Будем использовать примитивные типы BOOL, INT, NAT и оператор присваивания <Переменная> := <Выражение>. Предикат $x \in BOOL$ будем записывать в виде x: BOOL, что соответствует описанию типа переменной в стиле языка Паскаль. Не используется событие INITIALISATION. Начальная инициализация переменных реализуется в секции состояния. Например, x: BOOL := false. Вместо конструкции partition со сложной семантикой используется описание типа перечисления **enum** в стиле языка Си.

4. Управление движением на перекрестке

Данная задача представлена в качестве первого примера в руководстве по платформе Rodin [22] для демонстрации базисных конструкций языка Event-B и начальных действий пользователя в системе Rodin.

Содержательное описание. На пешеходном переходе через автомобильную магистраль имеется два светофора. Светофор для пешеходов: красный или зеленый. Светофор для автомобилей: красный, желтый или зеленый. Запрещено движение на красный светофор для пешеходов и автомобилей. Контроллер управляет переключениями светофоров. Должно соблюдаться требование безопасности: зеленый светофор для пешеходов сочетается только с красным светофором для автомобилей.

Анализ требований. Система управления движением, представленная содержательным описанием, не может использоваться на практике. По меньшей мере, необходимо обеспечить зеленый свет в течение определенного фиксированного времени, чтобы пешеходы успели перейти автомобильную магистраль. Поэтому данная задача – всего лишь учебный пример.

Объекты внешнего окружения: светофор для пешеходов и светофор для автомобилей. Сформулируем требования окружения:

RE1: Светофор для пешеходов может быть зеленым или красным.

RE2: Светофор для автомобилей может быть зеленым, желтым или красным.

Зафиксируем требование безопасности:

RS: Если светофор для пешеходов – зеленый, то светофор для автомобилей может быть только красным.

В руководстве [22] сначала рассматривается упрощенная задача. Красный свет представляется константой false, а зеленый – константой true. Желтого света нет. Светофор для пешеходов представлен логической переменной peds_go, а светофор для автомобилей – переменной cars_go.

Машина для упрощенной задачи в руководстве [22] эквивалентным образом определяется в виде следующей автоматной программы.

```
section {
    peds_go: BOOL := false
    cars_go: BOOL := false
    inv  peds_go = false ∨ cars_go = false
}
process PedsCars0 {
  Cycle:    [set_peds_go:] cars_go = false → peds_go := true  |
            [set_cars_go:] peds_go = false → cars_go := true  |
            [set_cars_stop:] cars_go := false  |
            [set_peds_stop:] peds_go := false
          #Cycle
} PedsCars0
```

Легко доказать, что инвариант сохраняется после исполнения каждой альтернативы недетерминированной композиции.

Представим нашу версию для упрощенной задачи. Введем два управляющих состояния:

- Cars – машины движутся, пешеходы стоят и ждут;
- Peds – пешеходы переходят магистраль, машины стоят перед красным светофором.

Автоматная программа представлена ниже.

```
state {
    peds_go: BOOL := false
    cars_go: BOOL := true
    inv  peds_go = false ∨ cars_go = false
}
process PedsCars1 {
   Cars: inv  peds_go = false & cars_go = true
         #Cars | peds_go := true; cars_go := false #Peds
   Peds: inv  peds_go = true & cars_go = false
         #Peds | peds_go := false; cars_go := true #Cars
} PedsCars1
```

Отметим, что сегмент Cars: #Cars | X эквивалентен сегменту Cars: X. Поэтому далее альтернативу вида #Cars будем опускать.

Чтобы закодировать процесс PedsCars1, например, в таком языке, как Java, реализуется кодирование управляющих связей информационными связями. Этот прием ранее был представлен Switch-технологией [3]. Вместо управляющих состояний Cars и Peds вводятся константы Cars и Peds как значения нового типа перечисления State. Программа процесса PedsCars1 переписывается следующим образом:

```
process PedsCars2 {
    type State = enum { Cars, Peds };
    state: State;
    state := Cars;
    loop:
       switch (state) {
          case Cars:  peds_go := true; cars_go := false; state :=Peds; break
          case Peds: peds_go := false; cars_go := true;  state :=Cars
       }
       #loop
} PedsCars2
```

Отметим, что программа PedsCars1 проще, короче и эффективней по сравнению с программой PedsCars2.

Данный способ кодирования управляющих состояний с помощью новой переменной state далее будем использовать при кодировании автоматной программ в виде спецификации Event-B. Например, кодирование PedsCars1 дает следующую автоматную программу, которая далее непосредственно переписывается в спецификацию на языке Event-B.

```
process PedsCars3 {
   Cycle: [Cars_go:] state=Cars → peds_go := true, cars_go := false, state := Peds |
          [Peds_go:] state=Peds → peds_go := false, cars_go := true, state := Cars
          #Cycle
} PedsCars3
```

Спецификация исходной задачи строится в руководстве [22] посредством пары уточнений спецификации упрощенной задачи, представленной выше. Эти уточнения приведены, видимо, в учебных целях. Однако их полезность в данном случае, сомнительна. Они усложняют решение задачи.

Наша автоматная программа для полной задачи отличается от итоговой спецификации в руководстве [22]. Действия контроллера на желтом свете светофора реализуются неодинаково в зависимости от предыдущего состояния светофора. Чтобы учесть такую особенность, введем два разных желтых света: yellowRed и yellowGreen. Автоматная программа приведена ниже.

```
section {
    type COLOURS = enum {red, yellowRed, yellowGreen, green}
    peds_col: COLOURS := red
    cars_col : COLOURS := green
    inv  peds_col = red ∨ peds_col = green & cars_col = red
}
process LightControl {
   Cars: inv  peds_col = red & cars_col = green
         cars_col := yellowGreen #YellowGreen
   YellowGreen: inv  peds_col = red & cars_col = yellowGreen
         cars_col := red; peds_col := green #Peds
   Peds: inv  peds_col = green & cars_col = red
         peds_col := red;  cars_col := yellowRed #YellowRed
   YellowRed:  inv  peds_col = red & cars_col = yellowRed
         cars_col := green #Cars
} LightControl
```

Данная программа была закодирована в виде спецификации на Event-B. Установлено, что без инвариантов управляющих состояний не проходит автоматическое доказательство всех формул корректности спецификации. Автоматическое доказательство реализуется лишь при введении пары инвариантов:

```
@invPeds   (state = Peds)   ⇔   ( peds_go = green ∧ cars_go = red)
@invCars   (state = Cars)   ⇔   ( peds_go = red ∧ cars_go = green)
```

5. Управление движением автомобилей по мосту

Данная задача представлена в качестве первого примера в известной книге Abrial J.-R. по системе Event-B [10]. На базе этого примера определяются основные концепции технологии Event-B.

5.1. Требования

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.

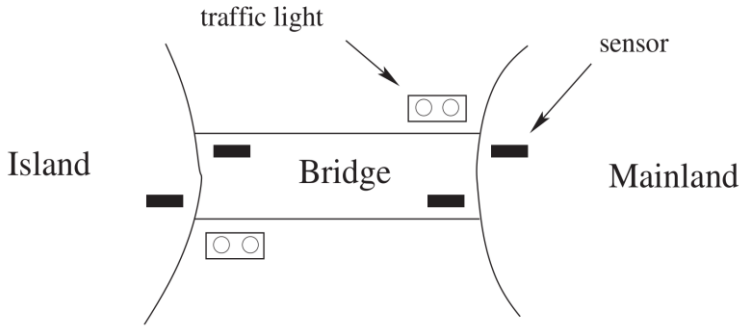

Рис. 1. Схема моста со светофорами и сенсорами.

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.

Детализация и анализ требований. Дадим имена светофорам: mtl – светофор на материке при въезде на мост, itl – светофор на острове при въезде на мост. Сформулируем требования окружения:

RE1: Имеются два светофора с именами mtl и itl.

RE2: Каждый светофор может быть зеленым или красным.

Дадим имена сенсорам: mlOut – при въезде на мост с материка, ilIn – при съезде с моста на остров, ilOut – при въезде на мост с острова, mlIn – при съезде с моста на материк.

RE3: Имеются четыре сенсора с именами: mlOut, ilIn, ilOut и mlIn.

RE4: Каждый сенсор может быть включен (значение on) или выключен (значение off).

RE5: 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, и участок стал свободным от автомобилей. Сформулируем функциональные требования.

RF1: Отключение сенсора mlOut – очередной автомобиль въехал на мост с материка.

RF2: Отключение сенсора ilIn – очередной автомобиль съехал с моста на остров.

RF3: Отключение сенсора ilOut – очередной автомобиль въехал на мост с острова.

RF4: Отключение сенсора mlIn – очередной 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. Одновременное движение с материка на остров и с острова на материк невозможно.

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. Это приведет к неправильному подсчету автомобилей на мосту, что может спровоцировать аварию. Автомобиль может также заехать на мост, а потом съехать с него задним ходом. Наконец, автомобиль может просто поехать на красный свет, если конечно красный свет не сопровождается установкой чугунного шлагбаума.

Поэтому данную задачу построения контроллера управления движением по мосту мы можем рассматривать только как учебную.

5.2. Разработка от простейшей модели

Построение итоговой спецификации определено в книге [10] в виде последовательности уточнений начиная с простейшей модели. Здесь мы покажем начальную модель и первые две ее уточнения.

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. Поскольку число автомобилей на острове ограничено, введем константу d, определяющую максимальное число автомобилей

на острове. Начальная машина содержит два события: уход машины с материка и приход машины на материк. Машину представим в виде следующей автоматной программы.

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

Легко обнаружить, что в модели есть ошибки. Например, нет гарантии, что после действия n:=n+1 не будет превышен лимит автомобилей d. Сила метода Event-B [10] в том, что этот метод позволяет обнаруживать все ошибки такого рода в процессе доказательства формул корректности. В данном случае будет представлена формула корректности: n+1<=d, которая здесь недоказуема. Чтобы исправить данную ошибку, необходимо добавить охранное условие: n<d. Вторая ошибка. Для действия n:=n-1 будет сгенерирована формула корректности n-1 ∈ NAT, которая оказывается недоказуемой. Необходимо вставить охранное условие: n>0. Третья ошибка. В начальный момент работы машины, после исполнения события-инициализации, не выполняется формула корректности: n <= d. Причина в том, что n не инициализирована. Необходимо вставить инициализацию: n:=0. Четвертая ошибка. При анализе отсутствия 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. Исправленная машина представлена ниже.

```
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
}
```

Далее определим уточнение представленной выше начальной модели. В уточненной модели различаются автомобили, находящиеся на мосту и на острове. Уточнение заключается в замене старой переменной 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, определяющийся отношение между старыми и новыми переменными. Появились два новых события: автомобиль переходит с моста на остров (IL_in) и автомобиль въезжает на мост с острова (IL_out).

```
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
}
```

Мы видим, 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, то есть значения переменной n.

Следующая модель строится как уточнение предыдущей модели. Причем это уточнение другого вида. В предыдущей модели уточнялись структуры данных. Здесь же происходит добавление новых структур данных и новых событий.

Вводятся светофоры: mtl – на материке при въезде на мост, itl – на острове при въезде на мост. Вводится инвариант mtl=red ∨ itl=red, обеспечивающий выполнение требования RF6. Два других инварианта введены для упрощения доказательства корректности уточнения: охранное условие некоторого события в уточненной модели должны быть не слабее аналогичного условия в предыдущей модели, а действия в строй и новой моделях должны быть идентичны.

```
section St2 extendes St1{
        type Colour = enum {red, green};
        mtll: 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
    }
```

Новые события Mtl_green и Itl_green определяют вроде бы естественные правила переключения светофоров: если мост стал пустым, то оба светофора меняют свет так, чтобы стало возможным движение в противоположную сторону. Далее, поскольку мост все еще пустой, может сработать другое правило, изменяющее направление движению. Таким образом, данные правила приведут к быстрому миганию светофоров. В книге [10] реализовано следующее решение. Если изменено направление движения, то далее необходимо дождаться появления автомобиля на мосту с противоположной стороны. Данное решение неудовлетворительно. Например. Утром все поехали на остров, где запланирован пикник. Первая партия машин проехала. А остальные окажутся заблокированными до вечера, когда с острова появится первый автомобиль.

Правильное решение следующее. Изменение направления движение возможно, если с противоположной стороны имеется автомобиль, стоящий на сенсоре в ожидании зеленого светофора. Реализация данного решения плохо стыкуется с проведенными уточнениями. Разработку контроллера нужно проводить другим способом.

В последнем уточнении добавляются сенсоры. В книге [10] последняя машина оказалась громоздкой и неадекватной, что серьезно дискредитирует сам метод Event-B.

5.3. Разработка по технологии автоматного программирования

Представим нашу реализацию данной задачи.

Константа d, 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.

Новые переменные. Переменная ml_out = true, если сенсор mlOut включен, то есть на сенсоре mlOut со стороны материка в данный момент находится машина. Переменная il_out = true, если сенсор ilOut включен, т.е. на сенсоре ilOut со стороны острова находится машина. Переменная empty = true в состоянии ожидания первого автомобиля на пустой мост.

Секция состояния для программы-процесса управления движением по мосту представлена ниже.

```
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
}
```

Полная программа-процесс представляется параллельной композицией пяти процессов, определяемых пятью автоматными программами. Для каждого сенсора имеется независимая автоматная программа, управляющая функционированием данного сенсора. Этими программами подсчитывается число автомобилей на мосту и на острове, а также значения логических переменных ml_out и il_out. Управление светофорами реализуется пятой автоматной программой LightControl.

Данная композиция из пяти процессов представляется вполне адекватной. Понятно, что подсчет числа автомобилей на мосту и на острове следует проводить на основе изменений значений сенсоров. И это проще реализовать в программах управления сенсорами. Управление светофорами было бы гораздо труднее реализовать в рамках четырех процессов для сенсоров.

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

- zero – мост пуст, оба светофора красные, и по значениям переменных ml_out и il_out реализуется соответствующая переустановка светофоров;
- right – реализуется движение слева направо, то есть с острова на материк;
- left – реализуется движение справа налево, то есть с материка на остров.

В состояниях right и left проверяется ситуация, когда мост становится пустым. В этом случае реализуется переход в состояние zero. В состоянии left 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.

Очевидно, что после перехода с zero на right или left немедленно произойдет возврат на zero, потому что мост пуст. Мы получим быстрое мерцание светофора с зеленого на красный. Поэтому переход с zero реализуется в новые состояния right0 и left0, где ожидается появление на мосту первой машины. При этом в состоянии left0 при появлении первой машины становится возможной ситуация 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 сработает сегмент кода в состояниях right0 или left0. Подобное, однако, может реализоваться, если приостановить работу процесса LightControl. Во избежание блокировки введен признак empty, При истинном значении empty автомобиль не может покинуть мост.

Далее представлена автоматная программа LightControl.

```
process LightControl {
    zero: inv a=0 & mtl = red & itl = red;
          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;
            if (a=0) #right0 else { empty :=false #right}
    right:  inv mtl = red & itl = green;
            a=0 → itl := red #zero
    left0:  inv mtl = green & itl = red;
            if (a=0) #left0 else { empty := false #left }
    left:   inv mtl = green & itl = red;
            if (a+b=d) {mtl := red; #leftscan}
            else if (a=0) {mtl := red #zero }
            else #left
    leftscan: inv mtl = red & itl = red;
              a=0 → #zero
}
```

5.4. Программы управления сенсорами

Простейшая программа управления сенсором состоит из двух сегментов включения и выключения сенсора: off: #on и on: #off. В нашем случае четыре программы управления сенсорами нагружены разными дополнительными действиями и реализуют переключения при определенных условиях.

На пустом мосту, при a=0, невозможна ситуация «автомобиль покидает мост», то есть переход с on на off. Автомобили могут заехать на мост только при зеленом светофоре. При переключении сенсоров вычисляются значения переменных ml_out и il_out.

Программы управления сенсорами представлены ниже.

```
process ML_out {
    inv a+b<=d
    off: ml_out:=true #on
    on: mtl=green,  a+b<d → a:=a+1, ml_out:=false #off
}
process IL_in {
    inv a+b<=d
    off: a>0, not empty, itl=red →#on
    on: a:=a-1; b:=b+1 #off
}
process ML_in {
    off: a>0, not empty →#on
    on: a:=a-1 #off
}
process IL_out {
    off: b>0 → il_out:=true #on
    on: itl=green → b:=b-1, a=a+1, il_out:=false
}
```

Полная программа управления движением автомобилей на мосту запускается следующей головной программой:

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

5.5. Управление сенсорами через монитор

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, используется монитор для пересчета переменных a и b, управляемый семафорами. Определим секцию для семафоров.

```
section Stcar {
    semal: BOOL:=false
    semabl: BOOL:=false
    semar: BOOL:=false
    semabr: BOOL:=false
    inv semal = TRUE ⇒ a+b < d
}
```

Монитор состоит из двух программ, работающих независимо. Первая программа при движении с материка на остров, вторая программа при движении с острова на материк.

```
process Left_ab {
    loop:  if (semal) { a:=a+1; semal := false};
           if (semabl & a>0) { a:=a-1; b:=b+1; semabl := false} #loop
}
process Right_ab {
    loop:  if (semar & a>0) { a:=a-1; semar := false};
           if (semabr) { a:=a+1; b:=b-1; semabr := false} #loop
}
```

Представим новые версии программ управления сенсорами. Чтобы изменить значения a и b каждая программа устанавливает соответствующий семафор в true и переходит в управляющее состояние wait, где ожидает снятия этого семафора, чтобы продолжить работу.

```
process ML_out {
    inv a+b<=d
    off: ml_out:=true #on
    on: mtl=green, a+b<d → semal:=true, #wait
    wait: not semal → ml_out:=false  #off
}
process IL_in {
    inv a+b<=d
    off: a>0, not empty, not semabl, itl=red → semabl:=true #wait
    wait: not semabl→ #off
}
process ML_in {
    inv itl=green
    off: a>0, not empty → semar := true #wait
    wait: not semar → #off
}
process IL_out {
    inv a+b<=d
    off: b>0 → il_out:=true #on
    on: itl=green →  semabr:=true #wait
    wait: not semabr → il_out:=false #off
}
```

Головная программа реализует запуск семи параллельных процессов:

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

5.6. Опыт верификации в системе Rodin

Программа управления движением автомобилей по мосту, описанная выше, была закодирована в виде спецификации на языке Event-B [10] и верифицирована в инструменте

Rodin [22]. Управляющие состояния программы LightControl закодированы как значения переменной state типа перечисления State:

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

Для кодирования управляющих состояний программ управления сенсорами используются значения следующего типа:

type Sensor = **enum** { off, on, wait }

Сегмент кода автоматной программы может быть закодирован несколькими событиями, от одного до трех. Семь 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 полный интерливинг атомарных сегментов семи автоматных программ.

Чтобы облегчить доказательство формул корректности, инварианты управляющих состояний добавлены к двум основным инвариантам программы.

79 сгенерированных формул корректности инвариантов были доказаны автоматически в системе Rodin. При этом были уточнены некоторые охранные условия. Потребовалось введение дополнительного инварианта: semal = TRUE ⇒ a+b < d .

К сожалению, не были сгенерированы автоматически формулы корректности для проверки на возможность дедлоков. Компонента Rodin проверки на дедлоки оказалась по непонятной причине заблокирована.

Дальнейшая верификация проводилось применением аниматора – компоненты Rodin, реализующей пошаговое исполнение спецификации Event-B с визуализацией текущего состояния, возможностью выбора любого из возможных вариантов исполнения и возможностью вернуться назад в процессе исполнения. С помощью аниматора удалось эффективно перебрать ключевые варианты исполнения и обнаружить следующие ошибки в исходной программе.

Сначала 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**: дедлок в состоянии left0 при b = d. Для исправления ошибки введено дополнительное охранное условие b < d в состоянии zero. **Третья ошибка** была 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.

6. Обзор работ

Построение спецификаций на Event-B реализуется сверху вниз, что для сложных задач приводит к объемным монолитным моделям. В работе [15] предложен механизм, позволяющий проводить разработку моделей снизу, сначала создавая модели более простых компонент, а затем объединяя их в общую модель. Механизм построения снизу интегрирован с техникой уточнений, реализуемой сверху.

Подход иллюстрируется на задаче управления движением автомобилей по мосту. Построена универсальная модель контроллера сенсора. Восемь переменных в состоянии этой модели. Далее четыре экземпляра этих переменных поступают в главную машину управления движением на мосту. Метод уточнений позволяет лишь частично снизить сложность и громоздкость полной модели. Полная модель состоит из пятнадцати машин. Одна из машин длиною в 379 строк. Наша единственная машина содержит 205 строк. В нашей модели всего семь переменных. Связь между процессом LightControl и другими процессами для сенсоров у нас минимизирована двумя переменными ml_out и il_out.

Решение проводить вычисление числа машин на мосту и на острове в центральной части модели, реализованное в руководстве по Event-B [10] и в работе [15], является неудачным, приводящим к сложной реализации. Метод уточнений ситуацию не спасает. Очевидным это становится только в автоматном программировании.

Автоматные методы программирования используются во многих известных языках, таких как UML, SDL, TLA+ [17], Event-B [10] и графических языках. Использование управляющих состояний при разработке программ реактивных систем характерно лишь для автоматного программирования [2, 3], которое доказало свою эффективность на множестве разнообразных приложений.

Существует четыре способа кодирования автоматных программ:

- стиль языка Фортран, используемый в нашем подходе;
- Switch-технология [3];
- рекурсивные вызовы вместо оператора перехода;
- предметно-ориентированный язык (DSL).

Использование меток в TLA+ [17] внешне похоже на управляющие состояния в автоматном программировании. Декларировано, что метки лишь фиксируют участки атомарности в спецификации TLA+.

В Switch-технологии [3] управляющие состояния становятся значениями некоторой переменной, а сегменты кода – альтернативами оператора switch. Показано (Разд. 4), что при таком преобразовании программа становится длиннее, сложнее и менее эффективной. Однако для языков, не содержащих оператора перехода, это лучший способ кодирования автоматных программ.

В третьем способе каждый сегмент кода автоматной программы кодируется независимой рекурсивной функцией. Вместо оператора перехода на другой сегмент вставляется рекурсивный вызов соответствующей рекурсивной функции. Аргументами рекурсивных функций являются переменные состояния автоматной программы. Программа длиннее и сложнее, чем в первых двух способах. Данный способ применяется в алгебрах процессов CCS, CSP и других, а также в языке Erlang [18].

В четвертом способе автомат программы кодируется в специальном предметно-ориентированном языке (DSL). Это, например, языки AbC [20] и Microsoft P [13, 19]. Набор элементарных конструкций в таких языках ограничен. Исполнение программ реализуется через интерпретатор. Имеется также возможность трансляции программы на язык Си.

7. Заключение

В разделе 3 настоящей работы определена модель технологии Event-B в автоматном программировании. С помощью этой модели на примерах задач в разделах 4 и 5 дана демонстрация технологии Event-B [10]. Проведено сравнение с технологией автоматного программирования. Подведем итоги.

Спецификация на языке Event-B представляет собой недетерминированную композицию операторов простого вида: **if** (Охранные условия>) { <Действие> }. Язык Event-B оказывается узким подъязыком автоматного программирования. Для конструирования спецификации Event-B используются лишь недетерминированная композиция и уточнение (refinement). Тогда как автоматное программирование кроме недетерминированной композиции допускает параллельную композицию процессов, гиперграфовую композицию по управляющим состояниям, подпроцессы, объектно-ориентированную и аспектно-ориентированную композиции.

Решения задачи управления движением автомобилей по мосту в руководстве по Event-B [10] и в работе [15], являются сложными и громоздкими, дискредитирующими метод

уточнений. Отметим, что уточнения, представленные в разделе 5.2, не упрощают разработку программы в автоматном программировании. Полезность метода уточнений в автоматном программировании предстоит исследовать в дальнейшем.

Наша программа управления движением по мосту была успешно верифицирована в инструменте Rodin [22] с применением средств автоматического и интерактивного доказательства, по уровню и возможностям выше, чем в системах доказательства PVS [21], Why3 [24] и Coq [12]. С помощью системы Rodin обнаружены три нетривиальные ошибки в нашей программе. Таким образом, систему Rodin следует рекомендовать для верификации критических автоматных программ.

Система правил генерации формул корректности, используемая для спецификаций Event-B, вполне может быть адаптирована для автоматных программ. Следует также предусмотреть интеграцию системы верификации автоматных программ с традиционной системой верификации на базе логики Хоара [16] для применения к фрагментам автоматной программы, относящимся к классу программ-функций.

Список литературы

1. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. — Новосибирск, 2010. — 42с. — (Препр. / ИСИ СО РАН; N 153). URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения: 10.12.2021)
2. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf (дата обращения: 10.12.2021)
3. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1
4. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538. https://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения: 10.12.2021)
5. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. http://persons.iis.nsk.su/files/persons/pages/req_k.pdf (дата обращения: 10.12.2021)
6. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная Информатика. – Новосибирск: ИСИ СО РАН, 2015. – №1. – С. 1-29. URL: http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf (дата обращения: 10.12.2021)

7. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. — Новосибирск, 2012. — 30с. — (Препр. / ИСИ СО РАН. № 164).
8. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. — Новосибирск, 2004. — 52с. — (Препр. / ИСИ СО РАН; N 115).
9. 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 (дата обращения: 10.12.2021)
10. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586p.
11. 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.
12. The Coq Proof Assistant. https://coq.inria.fr/ (дата обращения: 10.12.2021)
13. Desai A., Qadeer S., Seshia S. A.: Programming safe robotics systems: Challenges and ad-vances. In: Leveraging Applications of Formal Methods, Verification and Validation. Ver-ification, 2018, LNCS 11245, pp. 103–119.
14. Floyd R. W. Assigning meanings to programs // Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. AMS, 1967. pp. 19–32
15. 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
16. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
17. Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2021. P.382.
18. Larson J. Erlang for concurrent programming. ACM Queue, 2008, No. 5, pp. 18-23.
19. Liu P., Wahl T., Lal A.: Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers. In: Computer Aided Verification, LNCS 11562, 2019, pp. 386-404.
20. Nicola R. D., Duong, T., Inverso O.: Verifying AbC Specifications via Emulation. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles, ISoLA 2020, Part II, LNCS 12477, pp. 261-279.
21. PVS Specification and Verification System. SRI International. http://pvs.csl.sri.com/ (дата обращения: 05.03.2020)
22. Rodin User's Handbook. Version 2.8 / Jastram M. (editor). 2014. 184p.
23. Systems and software engineering — Life cycle processes — Requirements engineering. ISO/IEC/ IEEE 29148, 2011, 95p.

24. Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr (дата обращения: 10.12.2021)

Приложение 1

Управление движением на мосту. Спецификация Event-B

Переменные ml_out и il_out, которые были определены как логические, здесь представлены значениями констант **on** и **off**.

```
context EnvControl
sets Sensor Colour States
constants d red green on off wait
          zero right0 right left0 left leftscan
axioms
   @d_def d>0
   @col_ty partition(Colour, {red}, {green} )
   @sens_ty partition(Sensor, {on}, {off}, {wait} )
   @states_ty partition(States, {zero}, {right0}, {right}, {left0}, {left}, {leftscan}
)
end

machine CarBridge
sees EnvControl
variables a b mtl itl ml_out il_out state empty
          semal semabl semar semabr
invariants
     @inva a≥0 @invb b≥0
     @inv_mtl mtl∈Colour @inv_itl itl∈Colour
     @inv_ml_out ml_out∈Sensor @inv_il_out il_out∈Sensor
     @inv_1 empty ∈ BOOL
     @inv_2 semal∈ BOOL @inv_3 semabl ∈ BOOL @inv_4 semar∈ BOOL @inv_5 semabr ∈ BOOL
     @inv_st state ∈ States
     @thm_aplus  semal = TRUE ⇒ a+b < d
     @thm_zero   state = zero  ⇒  mtl = red ∧ itl = red ∧ empty = FALSE
     @thm_right0 state = right0  ⇒ mtl = red ∧ itl = green  ∧ empty = TRUE
     @thm_right  state = right    ⇒ mtl = red ∧ itl = green ∧ empty = FALSE
     @thm_left0  state = left0    ⇒ mtl = green ∧ itl = red  ∧ empty = TRUE
     @thm_left   state = left      ⇒ mtl = green ∧ itl = red ∧ empty = FALSE
     @thm_leftscan state = leftscan  ⇒  mtl = red ∧ itl = red ∧ empty = FALSE
     @inv_abd a+b ≤ d
     @inv_main mtl = red ∨ itl = red
events
  event INITIALISATION
  then
     @act1 a:=0    @act2 b:=0  @act3 mtl:=red @act4 itl:=red
     @act5 ml_out:=off  @act6 il_out:=off
     @act7 semal := FALSE @act71 semar := FALSE
     @act8 semabl := FALSE @act81 semabr := FALSE
     @act11 empty := FALSE
     @act12 state := zero
  end
  event Zero1
     where @grd1 state = zero
           @grd2 a=0 //∧ mtl = red ∧ itl = red
```

```
            @grd3 il_out=on
      then
            @act1 itl ≔ green
            @act2 empty ≔ TRUE
            @act3 state ≔ right0
      end
   event Zero2
      where @grd1 state = zero
            @grd2 a=0 //∧ mtl = red ∧ itl = red
            @grd3 b < d
            @grd4 ml_out=on
      then
            @act1 mtl ≔ green
            @act2 empty ≔ TRUE
            @act3 state ≔ left0
      end
   event Right0
      where @grd1 state = right0
            //@grd2 mtl = red ∧ itl = green
            @grd3 a ≠ 0
      then
            @act1 empty ≔ FALSE
            @act2 state ≔ right
      end
   event Right
      where @grd1 state = right
            //@grd2 mtl = red ∧ itl = green
            @grd3 a = 0
      then
            @act1 itl ≔ red
            @act2 empty ≔ FALSE
            @act3 state ≔ zero
      end
   event Left0
      where @grd1 state = left0
            //@grd2 mtl = green ∧ itl = red
            @grd3 a ≠ 0
      then
            @act1 empty ≔ FALSE
            @act2 state ≔ left
      end
   event Left
      where @grd1 state = left
            //@grd2 mtl = green ∧ itl = red
            @grd3 a+b = d
      then
            @act1 mtl ≔ red
            @act2 state ≔ leftscan
      end
   event Left1
      where @grd1 state = left
            //@grd2 mtl = green ∧ itl = red
            @grd3 a = 0
```

```
    then
          @act1 mtl ≔ red
          @act2 empty ≔ FALSE
          @act3 state ≔ zero
    end
  event Leftscan
    where @grd1 state = leftscan
          //@grd2 mtl = red ∧ itl = red
          @grd3 a = 0
    then
          @act1 empty ≔ FALSE
          @act2 state ≔ zero
    end
  event Left_ap      //ab Left Right
    where @grd1 semal = TRUE
    then
          @act1 a≔a+1
          @act2 semal ≔ FALSE
    end
  event Left_ambp
    where @grd1 semabl = TRUE
          @grd2 a > 0
    then
          @act3 semabl ≔ FALSE
          @act1 a≔a−1
          @act2 b≔b+1
    end
  event Right_am
    where @grd1 semar = TRUE
          @grd2 a > 0
    then
          @act2 semar ≔ FALSE
          @act1 a≔a−1
    end
  event Right_bmap
    where @grd1 semabr = TRUE
          @grd2 b > 0
    then
          @act3 semabr ≔ FALSE
          @act1 a≔a+1
          @act2 b≔b−1
    end
  event ML_out_off          //Sensor control
    where @grd1 a+b≤d
          @grd2 ml_out = off
    then
          @act1 ml_out ≔  on
    end
  event ML_out_on
    where @grd2 ml_out = on
          @grd3 mtl=green
          @grd4 a+b<d
    then
```

```
            @act1 semal ≔ TRUE
            @act2 ml_out ≔ wait
      end
   event ML_out_ap
      where @grd1 a+b≤d
            @grd2 ml_out = wait
            //@grd3 mtl=green
            @grd4 semal = FALSE
      then
            @act1 ml_out ≔ off
      end
   event IL_in_off
      where @grd1 a+b≤d
            @grd2 semabl = FALSE
            @grd3 empty = FALSE
            @grd4 itl = red
            @grd5 a > 0
      then
            @act1 semabl ≔ TRUE
      end
   event ML_in_off
      where @grd1 itl=green
            @grd2 semar = FALSE
            @grd3 empty = FALSE
            @grd4 a > 0
      then
            @act1 semar ≔ TRUE
      end
   event IL_out_off
      where @grd1 a+b≤d
            @grd2 b>0
            @grd3 il_out = off
      then
            @act1 il_out ≔ on
      end
   event IL_out_on
      where @grd1 a+b≤d
            @grd3 il_out = on
            @grd4 itl = green
      then
            @act1 il_out ≔  wait
            @act2 semabr ≔ TRUE
      end
   event IL_out_ap
      where @grd1 a+b≤d
            @grd2 il_out = wait
            //@grd3 itl = green
            @grd4 semabr = FALSE
      then
            @act1 il_out ≔ off
      end
end
```