Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В.А. Непомнящий, Е.В. Бодин, С.О. Веретнов

МОДЕЛИРОВАНИЕ И ВЕРИФИКАЦИЯ РАСПРЕДЕЛЁННЫХ СИСТЕМ, ПРЕДСТАВЛЕННЫХ НА ЯЗЫКЕ SDL, С ПОМОЩЬЮ ЯЗЫКА DYNAMIC-REAL

Препринт
156

Новосибирск 2010

В настоящей работе описан программный комплекс SRDSV (SDL/REAL Distributed Systems Verifier) для моделирования, анализа и верификации SDL-спецификаций распределенных систем. Этот комплекс включает транслятор из языка SDL в язык Dynamic-REAL (dREAL), систему автоматического моделирования dREAL-спецификаций и две системы верификации этих спецификаций методом проверки моделей, одна из которых использует известную систему SPIN. Описано применение комплекса SRDSV для анализа и верификации динамической системы управления сетью касс-терминалов.

Эта работа была частично поддержана грантом РФФИ № 07-07-00173.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

V.A. Nepomniaschy, E.V. Bodin, S.O. Veretnov

MODELING AND VERIFICATION OF DISTRIBUTED SYSTEMS SPECIFIED IN THE LANGUAGE SDL WITH THE HELP OF THE LANGUAGE DYNAMIC-REAL

Preprint
156

Novosibirsk 2010

The present paper describes a program complex SRDSV (SDL/REAL Distributed Systems Verifier) for modeling, analysis and verification of SDL-specifications of distributed systems. This complex includes a translator from SDL language into the language Dynamic-REAL (dREAL), a tool for automatic modeling dREAL- specifications and two verifiers of these specifications by model-checking method. One of these verifiers uses the well-known tool SPIN. An application of the complex SRDSV to analysis and verification of a dynamic system for control of booking terminals net is described.

1. ВВЕДЕНИЕ

В последние годы заметно возрастает роль формальных методов, применяемых для разработки распределенных и телекоммуникационных систем. Это связано с тем, что для них усложняется документирование, анализ и верификация. Для преодоления указанных трудностей на практике используется язык выполнимых спецификаций SDL, принятый в качестве стандарта международной организацией ITU (International Telecommunication Union) [1,18]. Верификация выполнимых спецификаций, представленных на языке SDL, заключается в проверке корректности их ключевых свойств и является актуальной задачей современного программирования. Развитие формальной семантики языка SDL необходимо для разработки формальных методов анализа и верификации. Интересные формализмы для описания формальной семантики SDL были предложены в [12,16]. Однако при использовании этих формализмов формальная семантика языка SDL остается сложной. Для доказательства свойств SDL-спецификаций применяется метод проверки моделей [2,3]. Однако непосредсвенное применение этого метода часто приводит к значительным трудностям ввиду громоздкости моделей SDL-спецификаций. Для преодоления этих трудностей используется трансляция SDL-спецификаций в модельный язык IF [11]. С помощью языка IF была проведена верификация используемого на практике коммуникационного протокола Mascara [10, 17].

Идея нашего подхода к проблеме верификации SDL-спецификаций состоит в разработке модельных языков, ориентированных на верификацию, которые были бы комбинированными, т.е. включали подъязыки выполнимых и логических спецификаций. Такой комбинированный язык спецификаций Basic-REAL (bREAL), базирующийся на статическом подмножестве SDL, был представлен в [7,15], где описана полная структурная операционная семантика выполнимых спецификаций в виде систем переходов, которая позволила доказать важные семантические свойства. Упрощенные версии этого языка, названные Real92 и Elementary-REAL, представлены в [5] и [6]. В [8] описана система SRPV (SDL/REAL Protocol Verifier) для моделирования, анализа и верификации статических SDL-спецификаций распределенных систем. В [9] описан язык Dynamic-REAL (dREAL), полученный расширением языка bREAL посредством динамических конструкций порождения и уничтожения экземпляров процессов.

Цель настоящей работы – описать программный комплекс SRDSV (SDL/REAL Distributed Systems Verifier), предназначенный для моделирования, анализа и верификации SDL-спецификаций распределенных систем, и эксперименты с ним. Данная работа состоит из восьми разделов. В разд. 2 дается обзор языка dREAL. Разд. 3 посвящен обзору программного комплекса SRDSV. В разд. 4 описан транслятор из языка SDL в язык dREAL. Системы симуляции и моделирования dREAL-спецификаций представлены в разд. 5, а система верификации этих спецификаций, использующая известную систему SPIN [14], представлена в разд. 6. Разд. 7 посвящен применению программного комплекса SRDSV к верификации динамической системы управления сетью касс-терминалов. В разд. 8 обсуждаются достоинства программного комплекса SRDSV и перспективы развития нашего подхода.

Эта работа была частично поддержана грантом РФФИ № 07-07-00173.

2. ЯЗЫК DYNAMIC-REAL

Язык Dynamic-REAL (dREAL) состоит из языка выполнимых спецификаций для представления распределённых систем и языка логических спецификаций для представления их свойств. Будем обозначать язык выполнимых спецификаций через dREAL.ex, а язык логических спецификаций – через dREAL.log.

2.1. Язык выполнимых спецификаций

Для описания систем в языке dREAL.ex определены две синтаксические формы – текстовая и графическая. Система состоит из блоков, соединенных между собой и с окружающей средой каналами. Средствами языка dREAL обеспечивается многоуровневое описание системы. Блок может содержать другие блоки и процессы, которые работают параллельно и взаимодействуют друг с другом и с внешней средой посредством обмена сигналами через каналы. С каждым действием процесса ассоциируется временной интервал, который задает продолжительность выполнения действия. В отличие от [9], мы используем единственную единицу времени, называемую *tick*. Таким образом, временной интервал содержит, кроме NOW и INF, только время, выраженное в этих единицах. Также добавлена возможность использования специальной служебной переменной TICK (значение глобальных часов) в выражениях.

Каждый экземпляр процесса может порождать или уничтожать экземпляры процессов в своем блоке. В отличие от [9], уничтожение экземпляра процесса осуществляется посылкой специального сигнала *Kill* в соответствующий канал, т.е. тело перехода имеет вид WRITE Kill INTO имя_канала[PId], где PId – уникальный идентификатор экземпляра процесса.

При этом сигналы, уже посланные уничтожаемым экземпляром процесса, не уничтожаются и могут быть прочитаны получателем сигналов, т.е. остаются в его (получателя) входном канале (если сам получатель не был уничтожен), тогда как сигналы, адресованные уничтожаемому экземпляру, пропадают.

Процессы, имеющие экземпляры, не могут выполнять операцию CLEAN (очистку выходного канала), так как это может приводить к уничтожению сигналов, посланных другим экземпляром этого процесса.

По сравнению с [9] в язык выполнимых спецификаций dREAL были добавлены следующие операторы для отладки и тестирования спецификаций:

– `ASSERT (`*`условие`*`)` – это условие должно быть выполнено в данном месте спецификации; при невыполнении условия во время отладки происходит прерывание;
– `PRINT(«`*`строка-сообщение`*`», список_`*`выражений`*`)` – при отладке пользователю выдаётся эта *`строка-сообщение`*; если `список_`*`выражений`* не пуст, строка считается «форматной строкой», причём выражения могут содержать переменные, включая переменную `TICK` (с указанием номера строки исходной спецификации); заметим, что аналогичная форматная строка используется в стандартной библиотеке ввода-вывода языка C.

В спецификации оператор ASSERT(условие) может входить только в переход следующего вида:

```
состояние:
ASSERT(условие)
тело_перехода
интервал
JUMP состояния
```

который можно выразить в виде пары стандартных переходов:

```
состояние:                    WHEN NOT(условие)
WHEN(условие)                 EXE ABRT
тело_перехода                 JUMP состояния
интервал
JUMP состояния
состояние:
```

Формальная семантика языка выполнимых спецификаций dREAL дана в [9].

2.2. Язык логических спецификаций

Подъязык логических спецификаций dREAL (dREAL.log) расширяет конструкции логики ветвящегося времени CTL [2,3] за счет использования временных интервалов и средств динамических логик. Формулы этого языка строятся из предикатов с помощью булевских операций, модальностей по моментам времени из интервала и по возможным поведениям выполнимых спецификаций, а также кванторов (существования и всеобщности) по выделенным переменным и по экземплярам процессов. Предикаты бывают четырех видов: отношения между значениями переменных и параметров сигналов, локаторы управляющих состояний в процессах, контроллеры пустоты и переполнения для каналов, индикаторы наличия и готовности сигналов в каналах.

Заметим, что по сравнению с Basic-REAL в этой версии язык логических спецификаций dREAL.log расширен только посредством кванторов существования и всеобщности по экземплярам процессов:
SOME INSTANCE и EACH INSTANCE.

То есть формула “EACH INSTANCE proc.(f)” истинна, если формула f истинна для всех экземпляров процесса proc, а формула “SOME INSTANCE proc.(f)” истинна, если формула f истинна хотя бы для одного экземпляра процесса proc.

3. ПРОГРАММНЫЙ КОМПЛЕКС SRDSV. ОБЗОР

Программный комплекс SRDSV (SDL/REAL Distributed Systems Verifier) предназначен для моделирования и верификации спецификаций, представленных на языках SDL и dREAL.ex. Для представления свойств SDL-спецификаций мы вводим язык SPL (SDL Property Language), близкий к подъязыку логических спецификаций dREAL.log. Единственное различие состоит в том, что в языке SPL кванторы и модальности применяются к экземплярам SDL-процессов и их поведениям. На рис. 1 представлена схема комплекса, где прямоугольники со сплошной границей означают компоненты системы, а со штриховой – файлы данных.

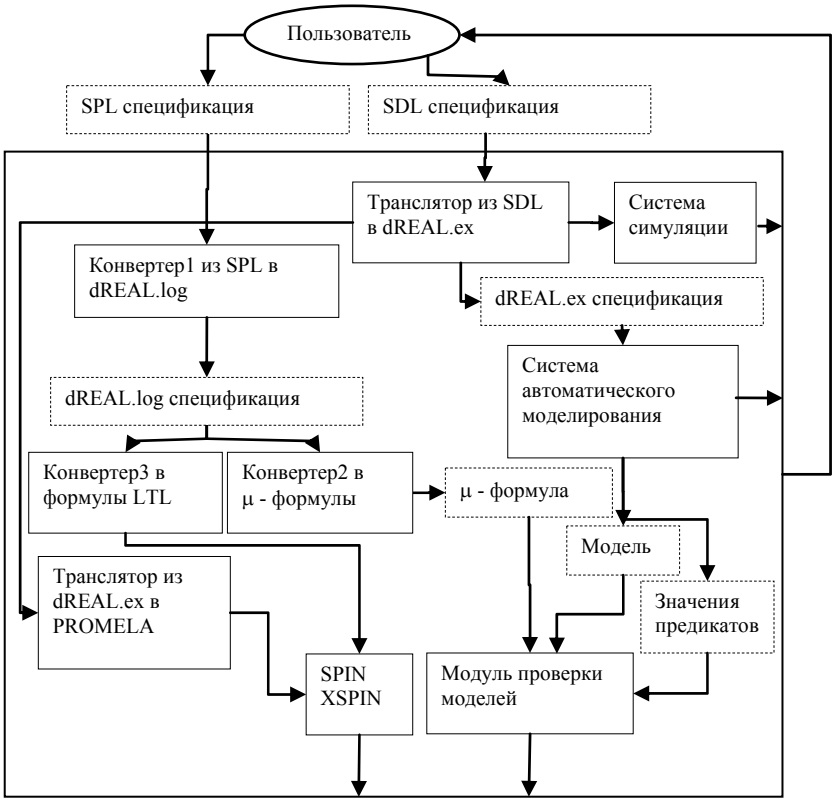

Рис.1. Программный комплекс SRDSV

Комплекс SRDSV состоит из следующих компонентов.

- Транслятор из языка SDL в язык dREAL переводит SDL-спецификацию в эквивалентную dREAL.ex-спецификацию.
- Конвертер1 переводит SPL-спецификацию, описывающую свойства SDL-спецификации, в эквивалентную dREAL.log-спецификацию.
- Система симуляции позволяет визуально проследить пошаговое выполнение процессов dREAL.ex-спецификации.
- Система автоматического моделирования позволяет проводить анализ dREAL.ex -спецификации с помощью пользовательских запросов, а также строить ее модель [8].

– Конвертер2 переводит dREAL.log-спецификации в эквивалентные формулы μ-исчисления [3].
– Модуль проверки моделей [8] проводит верификацию модели, построенной системой автоматического моделирования, относительно свойств, представленных формулами μ-исчисления.
– Транслятор из языка dREAL.ex во входной язык PROMELA системы SPIN [2,14] переводит dREAL.ex-спецификацию в спецификацию на языке PROMELA.
– Конвертер3 переводит dREAL.log-спецификации в эквивалентные формулы логики линейного времени LTL [2,3], если это возможно.
– Система SPIN [2,14] проводит верификацию спецификации на языке PROMELA относительно свойств на языке логики LTL методом проверки моделей [2,3], а система XSPIN [14] используется для графической симуляции PROMELA-спецификации и построения соответствующих этой спецификации MSC-диаграмм.

4. ТРАНСЛЯЦИЯ ПОДМНОЖЕСТВА ЯЗЫКА SDL В ЯЗЫК DYNAMIC-REAL

Транслируемое подмножество языка SDL88 не включает в себя следующие конструкции: GENERATOR, NAMECLASS и некоторые другие по причине отсутствия в языке dREAL необходимых конструкций.

Процесс трансляции проходит в две основных стадии:

1. Генерация внутреннего представления, соответствующего тексту входного файла. Во время этого этапа происходит проверка входного текста на предмет того, что он является синтаксически корректной SDL-спецификацией. Результатом является внутреннее представление, максимально приближенное по структуре к dREAL-программе
2. Генерация dREAL-текста (выходного файла), соответствующего внутреннему представлению. По заданным для всех конструкций языка dREAL-шаблонам происходит построение dREAL-текста.

Результатом работы транслятора является dREAL-текст, соответствующий входной SDL-программе, если она не содержала синтаксических ошибок и представима в языке выполнимых спецификаций dREAL. Также выдаются сообщения об ошибках, показывающие причину, по которой построение выходного файла провести не удалось.

Трансляция большинства статических конструкций SDL в dREAL.ex проводится аналогично [8]. Рассмотрим статические конструкции, трансляция которых потребовала значительных изменений, а также динамические конструкции.

Обозреваемые переменные. Операторы EXPORT/IMPORT.

Язык SDL имеет механизм, позволяющий процессу узнать текущее значение переменной другого процесса. Это делается с помощью оператора VIEW (имя_переменной), который заменяется в выражении, где он использован, значением переменной «имя_переменной» (эта переменная должна быть объявлена специальным образом). Ничего подобного в языке dREAL нет, поэтому поведение этого оператора моделируется следующим образом.

Переменные, описанные как REVEALED, переносятся из процесса, в котором они объявлены, в родительский блок. При этом на каждый тип обозреваемой переменной создается новый тип-массив «имя_типа_переменной_revealed», элементы которого имеют исходный тип, а индексация происходит по PId. Новые переменные в родительском блоке становятся этого типа. При каждом использовании имя переменной заменяется на элемент массива с индексом PId экземпляра процесса.

В процессе, использующем оператор VIEW, каждый оператор VIEW заменяется на элемент соответствующего массива.

SDL-операторы EXPORT/IMPORT также не имеют аналогов в языке dREAL. Их реализация в целом аналогична механизму обозреваемых переменных.

Переменные, описанные как EXPORTED,остаются в процессе как обычные переменные. В родительском блоке же, как и в случае REVEALED, создаются новые типы-массивы и служебные переменные имя_переменной_export на каждую переменную для экспорта. Оператор EXPORT заменяется на присваивание, где в левой части будет элемент массива, а в правой экспортируемая переменная. Оператор IMPORT заменяется так же, как и оператор VIEW.

Оператор CREATE. *Оператор, позволяющий динамически порождать новый экземпляр процесса.*

Порождение процеса в SDL осуществляется с помощью оператора CREATE <имя процесса>. В REAL оператор порождения практически не отличается CREATE PROCESS <имя процесса>. Поэтому трансляция не представляет трудности.

Пример 1. *Трансляция оператора CREATE*

Язык SDL:

```
STATE init;
  INPUT frame;
   CREATE Monitor;
NEXTSTATE state1;
```

Язык dREAL.ex:

```
TRANSITION init
  READ frame FROM s_to_m
  FROM NOW TO INF
 JUMP init_X1.
 TRANSITION init_X1
  CREATE PROCESS Monitor
  FROM NOW TO INF
 JUMP state1.
```

Оператор, уничтожающий экземпляр процесса.

Подобно SDL, уничтожение процесса происходит специальным оператором `ABRT`.

Пример 2. *Уничтожение процесса*

Язык SDL:

```
STATE init;
INPUT <signal_name>;
STOP;
```

Язык dREAL.ex:

```
TRANSITION init
  READ frame FROM m_to_m
  FROM NOW TO INF
  JUMP init_stop.

  TRANSITION init_stop
  EXE ABRT
  FROM NOW TO INF
  JUMP end_of_process
```

Идентификаторы экземпляров процесса

С развитием языка Dynamic-REAL появилась возможность посылать сигналы определенным экземплярам процесса. Сначала по правилам, приведенным выше, строятся каналы и маршруты. Затем в теле программ при отправлении сигнала добавляется PId экземпляра процесса.

Подобно SDL, в dREAL есть предопределённые переменные, содержащие PId.

PId могут быть нескольких видов:

- переменная типа PId, которой заранее было присвоено значение;
- OFFSPRING – PId порожденного процесса;

– PARENT – PId процесса-родителя;
– SELF – PId самого себя;
– SENDER – PId процесса, от которого прошел последний сигнал.

Например, в SDL:

```
OUTPUT frame TO OFFSPRING;
B REAL:
WRITE frame INTO channel[OFFSPRING].
```

Процедуры

Процедуры языка SDL служат для упрощения описания и улучшению обозримости описания. Они состоят из определения процедур и вызовов процедур. Определение процедуры состоит из заголовка и тела.

```
PROCEDURE <имя_процедуры>;[<формальные параметры>];
```

Формальные параметры могут быть только входными (IN) или входными-выходными (IN/OUT):

```
FPAR IN <имя>,...,<имя> <сорт>;
       IN/OUT <имя>,...,<имя> <сорт>;
```

Вызов процедуры в теле основной программы осуществляется оператором

```
CALL <имя_процедуры>[<формальные параметры>];
```

В языке dREAL такие способы описания отсутствуют. Поэтому трансляция макрокоманд реализуется следующим образом. Поскольку рекурсивные вызовы процедур запрещены в транслируемом подмножестве языка SDL, то возможно в месте вызова процедуры вставить копию ее тела. При этом формальные параметры переводятся в переменные процесса, в котором происходит вызов процедуры. Перед телом процедуры вставляется блок, в котором происходит инициализация тех переменных, которые являлись формальными параметрами в процедуре. После тела процедуры вставляется блок, в котором те формальные параметры, что были объявлены в процедуре как IN/OUT, присваиваются соответствующим переменным в вызове процедуры. Аналогичным образом преобразуются и макрокоманды языка SDL.

Таймеры

Механизм таймеров реализуется следующим образом: в блок, которому принадлежит процесс, использующий таймеры, добавляются служебные процессы (по одному на каждый таймер), которые эмулируют работу SDL-таймера, служебные каналы от процесса, использующего таймер, до процесса-таймера. Количество экземпляров процесса-таймера равно количеству экземпляров процесса, в котором описан таймер. Таким образом, для каждого экземпляра процесса существует свой независимый экземпляр процесса-таймера. Также в процесс, использующий таймеры, добавляются служебные переменные типа «имя_таймераTimer_PId» типа PId, в которых хранится PId соответствующего процесса-таймера. Эмуляция работы таймера происходит следующим образом: если процесс использует таймер, то в него добавляются служебные состояния

```
TRANSITION InitTimer
CREATE PROCESS Имя_ТаймераTimer
FROM NOW TO INF
JUMP InitTimerPId.

TRANSITION InitTimerPId
EXE Имя_ТаймераTimer _PId:=OFFSPRING;
FROM NOW TO INF
JUMP Start__rl.
```

В этих состояниях создается экземпляр процесса-таймера и инициализируется служебная переменная Имя_ТаймераTimer _PId.

SDL-оператор SET переводится в dREAL-оператор WRITE с сигналом SET, параметром которого является выражение. Получив этот сигнал, процесс-таймер переходит к альтернативным переходам:

- срабатывает при получении процессом сигнала RESET, после чего процесс переходит в состояние ожидания;
- при получении нового сигнала SET со временем срабатывания, равным полученному при установке выражению, переходит к состоянию, которое отправляет сигнал о срабатывании таймера, после чего процесс переходит в состояние ожидания.

Когда экземпляр процесса, использующий таймеры, прекращает свою работу, то процессу-таймеру посылается сигнал stop_timer, по которому он также прекращает работу.

Пример 3. *Служебный процесс-таймер.*

```
Имя_ТаймераTimer: PROCESS(0,n)
  PR VAR delay OF integer.

  TRANSITION inactive
    READ set(delay) FROM Имя_Таймера_
    FROM NOW to INF
  JUMP active.

  TRANSITION inactive
    READ reset FROM Имя_Таймера
    FROM NOW to INF
  JUMP inactive.

  TRANSITION inactive
    READ stop_timer FROM Имя_Таймера
    FROM NOW to INF
  JUMP stop.

TRANSITION active
    READ set(delay) FROM Имя_Таймера
    FROM NOW TO NOW
  JUMP active.

  TRANSITION active
    READ reset FROM Имя_Таймера
    FROM NOW TO NOW
  JUMP inactive.

TRANSITION active
  WHEN (delay <= 0)
  EX SKIP
  FROM NOW TO NOW
JUMP time_out.

TRANSITION active
  WHEN (delay > 0)
  EX delay := delay - 1;
FROM 1 tick UPTO 1 tick
JUMP active.

TRANSITION active
  READ stop_timer FROM Имя_Таймера
  FROM NOW TO NOW
JUMP stop.

TRANSITION stop
  EX ABRT
  FROM NOW TO NOW
JUMP end_of_process.

TRANSITION time_out
      WRITE timeout
        INTO Имя_Таймера_inv[SENDER]
  FROM NOW TO INF
JUMP inactive.
```

5. МОДЕЛИРОВАНИЕ И СИМУЛЯЦИЯ ВЫПОЛНИМЫХ DREAL-СПЕЦИФИКАЦИЙ

5.1. Система симуляции dREAL.ex-спецификаций

5.1.1. Архитектура системы

Система симуляции выполнимых спецификаций – экспериментальная система, предназначенная для симуляции и тестирования выполнимых спецификаций языка Dynamic-REAL.

Архитектура системы представлена ниже на рис. 2. Система состоит из следующих модулей:

- синтаксический анализатор;
- управляющий модуль;
- семантический анализатор;
- графический интерфейс.

Процесс симуляции происходит следующим образом. Пользователь посылает выполнимую спецификацию на синтаксический анализатор, который, сделав синтаксический и лексический разбор, преобразует ее во внутреннее представление. После этого внутреннее представление передается управляющему модулю. Управляющий модуль, общаясь с семантическим анализатором, осуществляет моделирование спецификации. Управляющий модуль подготавливает данные для графического интерфейса и передает их ему, а также сохраняет историю выполнения. Графический интерфейс представляет данные, получаемые в процессе моделирования и тестирования, в удобном для пользователя виде.

Система симуляции выполнимых спецификаций реализована на языке C++ в среде Microsoft Visual Studio 6.0 для Windows. Система работает под следующими операционными системами: Windows 95/98/NT/2000/XP.

Система симуляции выполнимых спецификаций позволяет осуществлять моделирование выполнения специфицируемой системы в следующих режимах.

1. Режим автоматического выполнения.
2. Режим пошагового выполнения.
3. Режим трассировки, или иначе – возможность остановки системы в соответствии с введенным пользователем запросом.

Во время симуляции спецификации сохраняется ее история выполнения. Система также имеет следующие возможности для осуществления недетерминированного выбора: случайным образом или путем запроса.

Входными данными системы является выполнимая спецификация Dynamic-REAL. Выполнимая спецификация располагается в файле с расширением ".RL", история выполнения записывается в файл "HISTORY.TXT".

5.1.2. Компоненты системы

Рассмотрим основные компоненты системы (рис. 2).

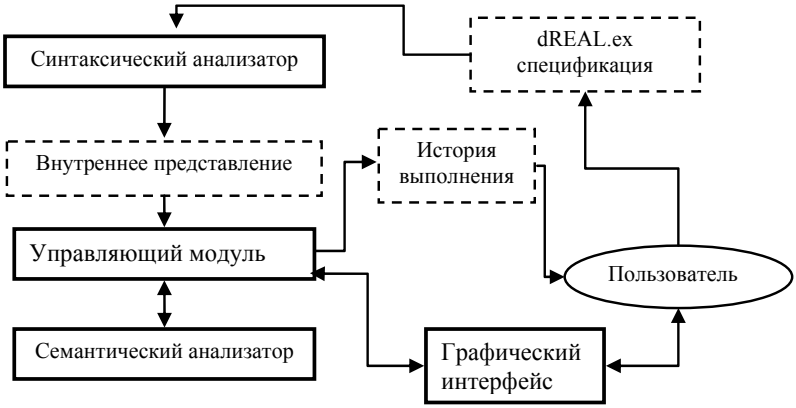

Рис. 2. Структура системы симуляции выполнимых dREAL-спецификаций

Управляющий модуль осуществляет управление и контроль всех процессов, происходящих в системе моделирования выполнимых спецификаций, а также связывает все модули. После того как синтаксический анализатор преобразовал выполнимую спецификацию во внутреннее представление, управляющий модуль получает это внутреннее представление и, преобразовав его, передает семантическому анализатору. Семантический анализатор, выполнив какое-либо действие, сообщает об этом управляющему модулю, и управляющий модуль записывает действие в файл (таким образом сохраняется история выполнения) и передает измененные данные в графический интерфейс, а также осуществляет остановку моделирования и тестирования выполнимой спецификации. После этого управляющий модуль ждет команды от пользователя: либо продолжить выполнение моделирования, либо завершить работу системы. Таким образом, осуществляется так называемый режим трассировки, который позволяет узнавать, что происходит, и какими данными обладает пользователь в определенный момент функционирования специфицируемой системы.

Синтаксический и семантический анализаторы. Синтаксический анализатор осуществляет синтаксический и лексический разборы выполнимых спецификаций языка dREAL. Синтаксический анализатор написан также с помощью программ Flex (Lex) версии 2.5.4 и Bison (Yacc) версии 1.25.

Семантический анализатор осуществляет моделирование выполнения операционной семантики процессов выполнимой спецификации Dynamic-REAL. Так как спецификация состоит из блоков, процессов, подблоков, то выполняются по уровням семантические правила.

Рассмотрим уровень блоков. Так как у блока всего лишь одно правило – правило композиции, то в соответствии с этим правилом семантический анализатор разбивает блок на соответствующие составляющие – процессы и подблоки (если у блока имеются подблоки). Далее, для каждого процесса из активного состояния происходит сравнение текущих данных с условиями каждого правила шага, и, если условия выполняются, происходит срабатывание перехода, и процесс переходит в новое активное состояние. Если ни одно из правил шага не выполняется, то в зависимости от значений данных в активном состоянии выполняются семантические правила заикания, стабилизации, “часы” или “голодание”. Для подблоков делается все то же самое, что и для блоков.

Выполняя какое-либо действие по операционной семантике, семантический анализатор постоянно передает все данные в управляющий модуль.

Графический интерфейс наглядно представляет симуляцию выполнимой спецификации языка Dynamic-REAL и позволяет получить протокол выполнения.

5.2. Система автоматического моделирования выполнимых dREAL-спецификаций

5.2.1. Архитектура системы

Система моделирования выполнимых спецификаций – экспериментальная система, предназначенная для моделирования выполнимых спецификаций языка dREAL.

Архитектура системы представлена на приведенном ниже рис. 3. Система состоит из следующих модулей:

- управляющий модуль,
- синтаксический анализатор,

– конструктор моделей,
– модуль перевода логических спецификаций в μ-исчисление.

Система симуляции по dREAL-спецификации, состоящей из выполнимой dREAL-спецификации и логической спецификации, позволяет провести проверку свойств специфицируемой системы.

Более подробно процесс моделирования происходит следующим образом. Полученная от пользователя dREAL-спецификация передается синтаксическому анализатору, который, сделав её синтаксический и лексический разборы, переводит её во внутреннее представление и передает управляющему модулю. Управляющий модуль разбивает полученные данные на данные от выполнимой спецификации и передает их конструктору моделей, а данные от логической спецификации передает модулю перевода логических спецификаций в μ-исчисление. Конструктор моделей строит модель и создает список состояний.

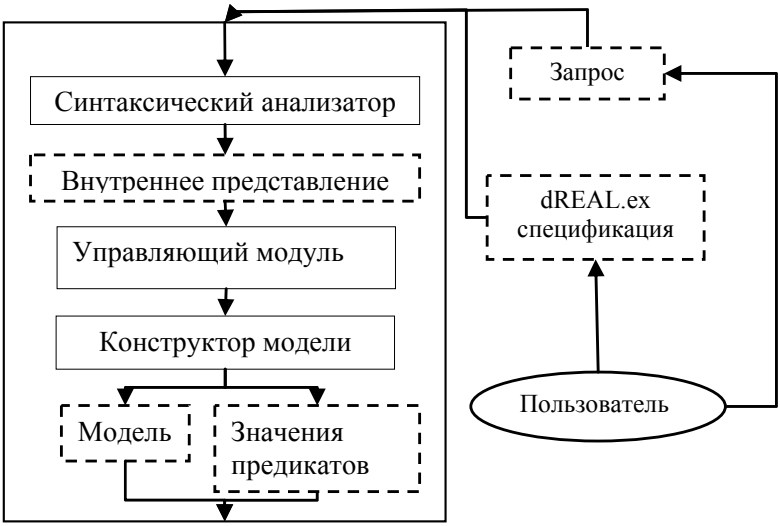

Рис. 3. Структура системы моделирования Dynamic-REAL спецификаций

Модуль перевода переводит логическую спецификацию в μ-формулу и значения констант и передает все эти данные модулю проверки моделей. Модуль проверки моделей проводит верификацию и передает результат пользователю.

Система моделирования выполнимых спецификаций реализована на языке C++ в среде Microsoft Visual Studio 6.0 для Windows. Объем исходного кода системы – 1100 Кбайт, исполняемого – 600 Кбайт. Система работает под следующими операционными системами:
Windows 95/98/NT/2000/XP.

5.2.2. Язык запросов

Чтобы совершить определенное действие при достижении системой определенной конфигурации, создается пользовательский запрос к системе.

Для языка запросов использованы конструкции языка dREAL с добавлением служебных слов, обозначающих команды для системы моделирования, которые представляются конструкцией

WHEN <предикат> EXE <операция>.

Предикат представляется

- отношением (relation) между переменными и параметрами,
- локатором (locator) состояния управления,
- контроллером (controller) пустоты, полноты и одновременного доступа к каналам,
- индикатором присутствия сигнала в канале (checker).

Синтаксис диаграммы отношения:
expression sign_relation expression, где expression – алгебраическое выражение, содержащее расширенные имена переменных и параметров сигналов выполнимых спецификаций, sign_relation – один из знаков отношений (<, <=, >, >=, =, <>).

В локаторах контроля управления также используются расширенные имена состояний процессов из выполнимых спецификаций в списке систем.

Диаграмма локатора: AT *состояние*, где *состояние* – имя состояния. Например, когда нужно выяснить, попадет ли данный процесс P в состояние Q, составляется запрос WHEN P AT P.Q EXE PAUSE.

Контроллеры могут быть как контроллерами пустоты (EMPTY), так и полноты каналов (FULL). Контроллер пустоты канала: EMP *канал* или *канал* IS EMPTY, где *канал* – имя канала. Контроллер полноты канала: FUL *канал* или *канал* IS FULL. Если нужно выяснить состояние канала C в данном процессе, составляется запрос WHEN C IS EMPTY EXE PAUSE или WHEN C IS FULL EXE PAUSE

Индикатор присутствия сигнала в канале: *сигнал* IN *канал*, где *сигнал* – имя сигнала, *канал* – имя канала. Для выяснения, находится ли сигнал S в канале C, используется запрос WHEN S IN C EXE PAUSE.

Операция выражается с помощью служебных слов и выражений:

- STOP – прервать процесс симуляции;
- PAUSE – прервать процесс симуляции до нажатия любой клавиши;
- PRINTLOG (сообщение) – записать сообщение в файл лога;
- MESSAGE (сообщение) – выдать сообщение на экран (симуляция продолжится после нажатия кнопки «OK»).

Сообщение может быть как просто текстом, так и текстом с использованием переменных, как в языке C. Например: (“variable D = %s “,var_name).

1. Для работы с экземплярами процессов используются кванторы SOME INSTANCE (для какого-либо экземпляра данного процесса) и EACH INSTANCE (для всех экземпляров процесса).
2. Прерывание работы системы по истечении заданного времени. Для этого используется конструкция FROM interval_1 TO interval_2.
3. Прерывание работы системы по истечении заданного числа тактов.

Компоненты системы

Управляющий модуль осуществляет управление и контроль всех процессов, происходящих в системе верификации выполнимых спецификаций, а также связывает все модули. После того как синтаксический анализатор преобразовал спецификацию во внутреннее представление, управляющий модуль получает это внутреннее представление и, преобразовав и разделив его, передает конструктору моделей данные, полученные от выполнимой спецификации, а модулю перевода логических спецификаций в μ-исчисление передает данные, полученные от логической спецификации.

Синтаксический анализатор является модернизированным синтаксическим анализатором системы симуляции выполнимой спецификации языка Dynamic-REAL. А именно, добавлены синтаксический и лексический разборы логических спецификаций.

На вход синтаксического анализатора подается имя файла с выполнимой спецификацией Dynamic-REAL и логической спецификацией(“.RL” – файл). Результатом выполнения модуля является:

- в случае успешного завершения разбора – сообщение “Синтаксический разбор завершен успешно”, и выходом синтаксического анализатора является внутреннее представление выполнимой спецификации Dynamic-REAL (“.RLO” – файл), которое подается на вход управляющего модуля;
- в случае наличия синтаксической ошибки в спецификации – сооб-

щение вида:

“Строка N. Синтаксическая ошибка в или перед Name”, где

N – номер строки, содержащей ошибку,

Name – имя соответствующей нераспознанной лексемы, и система моделирования выполнимых спецификаций завершает работу.

Полученное внутреннее представление передается управляющему модулю.

Конструктор моделей, получив от управляющего модуля внутреннее представление выполнимой спецификации, осуществляет выполнение операционной семантики и конструирует модель. Действия конструктора моделей подобны действиям семантического анализатора из системы моделирования и тестирования выполнимых спецификаций Dynamic-REAL.

Выходами данного модуля являются:

- файл “MODEL.TXT”, содержащий модель;
- файл “CHANGE.CNG”, содержащий список состояний.

6. СИСТЕМА МОДЕЛИРОВАНИЯ И ВЕРИФИКАЦИИ, ИСПОЛЬЗУЮЩАЯ SPIN

6.1. Общая схема системы

Эта система состоит из следующих компонентов (см. рис. 4):

- транслятор из языка dREAL.ex в язык PROMELA, который переводит выполнимую dREAL.ex-спецификацию в эквивалентную PROMELA-спецификацию;
- конвертер3, который переводит dREAL.log-спецификацию в эквивалентную формулу логики линейного времени LTL;
- система SPIN, которая верифицирует методом проверки моделей спецификации на языке PROMELA относительно свойств, выраженных в логике LTL;
- система XSPIN, которая проводит графическую симуляцию PROMELA-спецификаций и строит соответствующие MSC-диаграммы.

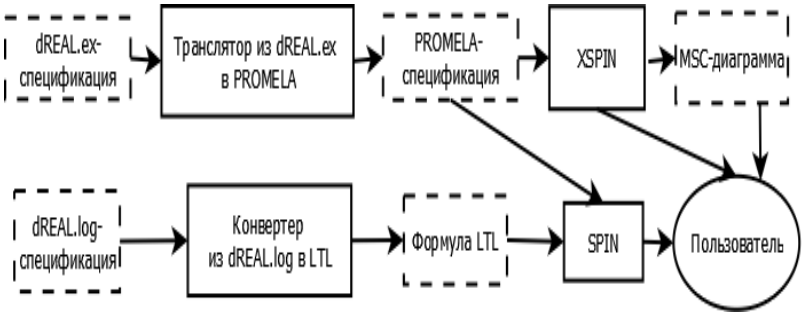

Рис.4. Общая схема системы верификации, использующей SPIN

6.2. Транслятор из языка dREAL.ex в PROMELA

Трансляция в язык PROMELA выполняется следующим образом. Сначала dREAL-спецификация преобразуется во внутреннее представление, близкое к дереву синтаксического разбора, затем проходом по дереву генерируется выходной текст. Чтобы обеспечить соответствие семантики результирующей и исходной спецификации, при создании нового экземпляра процесса порождающий процесс выполняет следующую непрерываемую последовательность действий: запускает экземпляр порождаемого процесса и запоминает его идентификатор (PId), обнуляет элемент глобального массива, соответствующий этому PId, помещает свой PId в элемент массива для процессов-родителей. После этого исполнение порождающего процесса приостанавливается до тех пор, пока не будут выполнены следующие действия в порожденном процессе. Процесс присвоит своей локальной переменной значение PId процесса-родителя. Он установит соответствующие элементы массивов для всех своих входных каналов в соответствующие значения своих локальных переменных типа каналов и сигнализирует через глобальную переменную о своей готовности. Заметим, что семантика языка PROMELA ограничивает число одновременно существующих экземпляров процессов.

Трансляция статических конструкций. Как в dREAL, так и в PROMELA, есть такие понятия, как переменные, массивы, каналы, процессы.

Состояния языка dREAL переводятся в метки языка PROMELA с добавлением префикса «L_».

Каналы и сигналы получают префиксы «C_» и «S_», соответственно.

Для моделирования реального времени (временных интервалов) в результирующую спецификацию добавляется процесс timer(), который увеличивает значение предопределённой глобальной переменной `TICK`. Этот процесс срабатывает, когда среди остальных процессов нет «переходов последней секунды» , которые должны сработать «в этот момент или никогда», то есть «все процессы ожидают наступления своей левой границы интервала» и «все процессы могут выдержать срабатывание часов, то есть никакой переход не станет запрещённым при шаге по времени».

Трансляция динамических конструкций потребовала дополнительных усилий для соответствия семантике результирующей спецификации.

Напомним «специальные» переменные, связанные с динамическими конструкциями:

`SELF` – PId текущего процесса;

`PARENT` – PId процесса, который породил данный процесс;

`OFFSPRING` – последний PId процесса, который был порожден данным процессом;

`SENDER` – PId процесса, от которого было получено последнее сообщение из канала.

Глобальные переменные для экземпляров процессов. Так как в Spin не может быть более 255 процессов, достаточно использовать тип «байт» в качестве типа индекса и значения следующих массивов. Введём два глобальных массива языка PROMELA – `PROC_INIT` и `PARENT`.

Порождение экземпляра процесса. Порождающий процесс выполняет следующую непрерываемую последовательность действий:

- проверяет, что количество уже существующих экземпляров данного (типа) процесса допускает создание нового экземпляра (в противном случае, выполнение завершается);
- порождает экземпляр процесса и запоминает его PId в переменной `aOFFSPRING`;

– обнуляет элемент массива PROC_INIT, соответствующий этому PId;
– помещает свой PId в элемент массива PARENT, соответствующий этому Pid.

После этого исполнение порождающего процесса приостанавливается до тех пор, пока не будут выполнены (тоже как «непрерываемая последовательность») необходимые действия в порожденном процессе, а именно

– процесс установит свою локальную переменную aPARENT из соответствующего элемента массива PARENT;
– процесс установит соответствующие элементы массивов CV_<имя_канала> для всех своих входных каналов в соответствующие значения своих локальных переменных-каналов (это можно рассматривать как «привязку» «входных портов» к «маршрутам»);
– процесс помещает единицу в соответствующий (своему PId) элемент массива PROC_INIT, чтобы сигнализировать о своей готовности.

Во всех экземплярах процессов определены следующие локальные переменные:

aOFFSPRING, aSENDER, aPARENT;

Для обеспечения уникальности PId'ов процессы, завершившие работу, не уничтожаются, а останавливаются в «последнем» состоянии, ожидая наступления условия false.

Транслятор написан на языке Perl.

7. ВЕРИФИКАЦИЯ СИСТЕМЫ УПРАВЛЕНИЯ СЕТЬЮ КАСС-ТЕРМИНАЛОВ

7.1. Система управления сетью касс-терминалов

В общем виде система управления сетью касс-терминалов представлена на языках SDL и dREAL в [9]. Система состоит из терминалов (terminal) и главной машины (main_machine), к которой терминалы обращаются за информацией о наличии и цене билетов до нужной пассажиру станции. По сравнению с [9] эта система имеет следующее ограничение: число терминалов заранее фиксировано. Пассажир (passenger) встает в очередь (queue), а затем подходит к терминалу и нажимает на кнопку начала работы. Терми-

нал сообщает о готовности и ожидает нажатия кнопки с номером станции. Получив номер станции, терминал запрашивает у главной машины стоимость проезда до этой станции. Если же билетов до нее уже не осталось, главная машина сообщает об этом терминалу, который передает это сообщение пассажиру. Если же билеты есть, на индикаторе терминала указывается сумма, которую нужно заплатить. Пассажир опускает необходимую сумму денег. Если денег хватает, то терминал печатает билет до нужной ему станции и выдает сдачу. Если у пассажира не хватает выделенных на покупку билета средств, он прекращает сеанс по своей инициативе. В конце сеанса пассажир завершает свою работу, освобождает терминал и уходит из очереди.

Выполнимая спецификация этой системы на языке SDL представлена на рис. 5, 6, 7, 8.1, 8.2, 9.1, 9.2, контекст спецификации на языке SDL представлен в Приложении 1, а спецификация системы на языке dREAL представлена в Приложении 2.

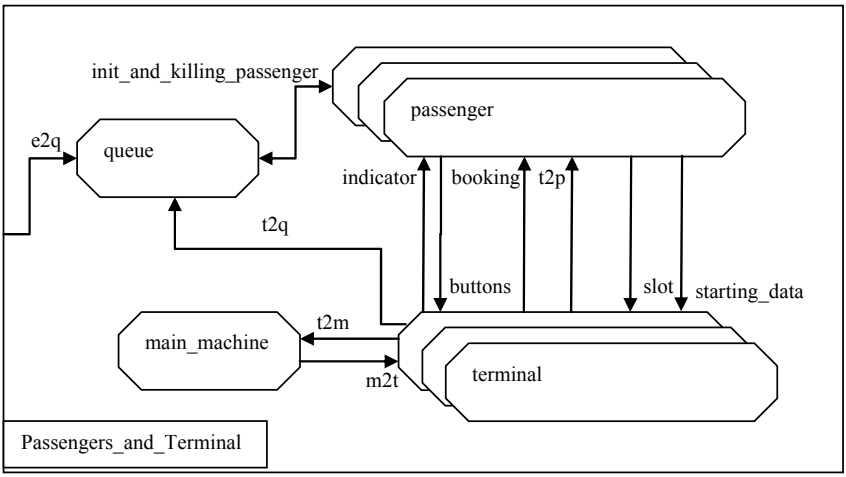

Рис.5. Блок-схема системы управления сетью касс-терминалов

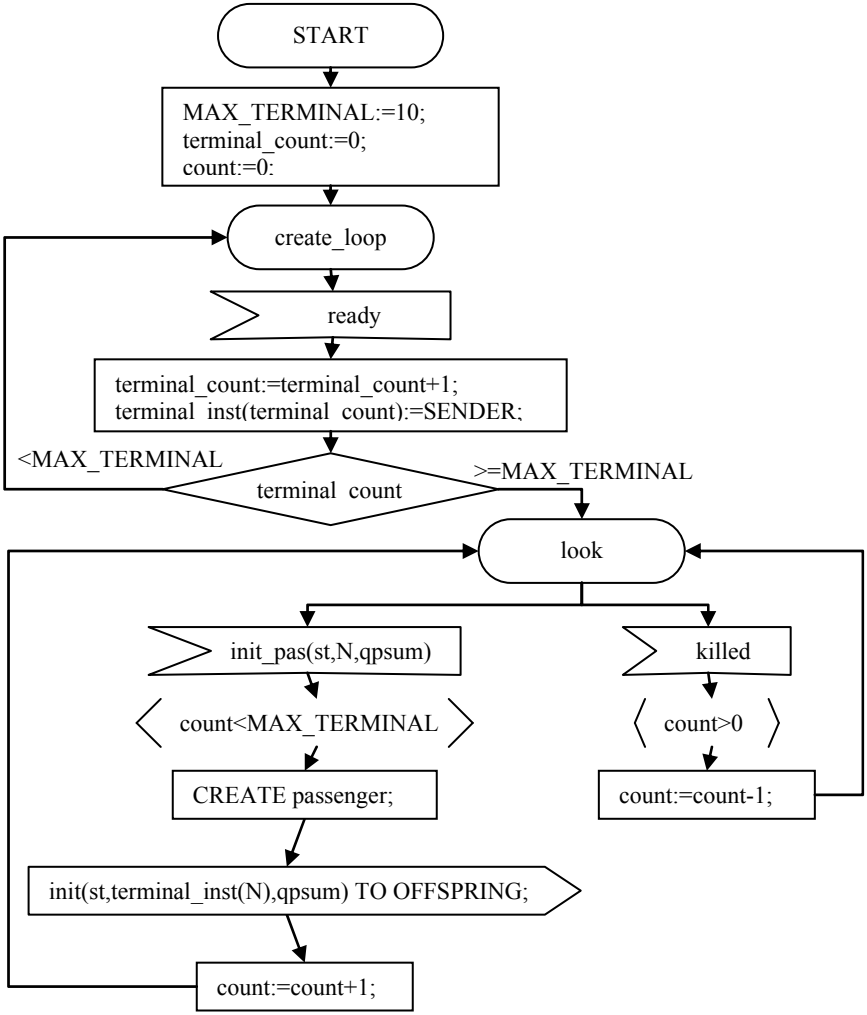

Рис.6. Блок-схема процесса queue

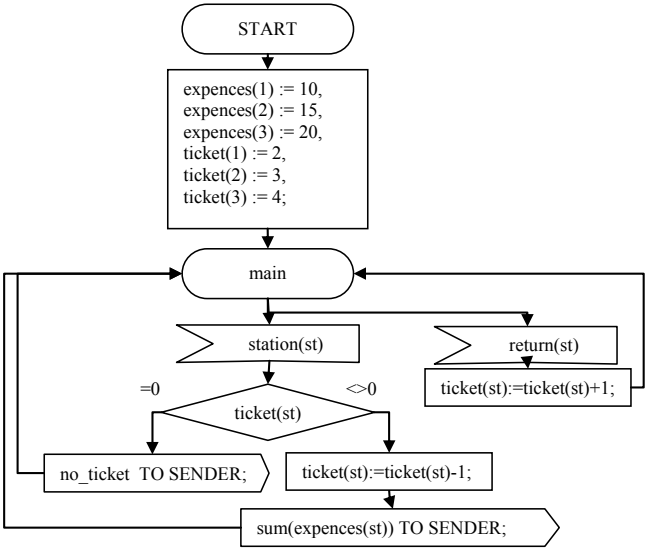

Рис.7. Блок-схема процесса main-machine

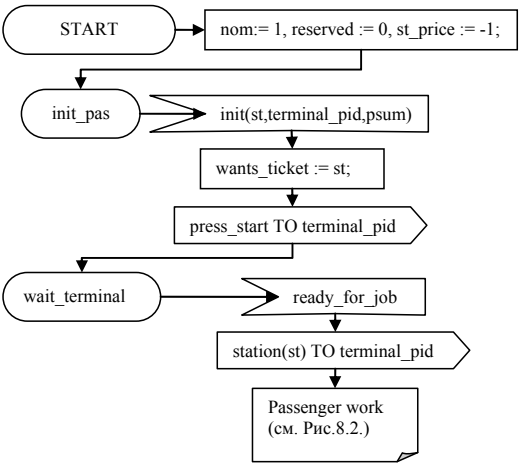

Рис.8.1. Блок-схема процесса passenger

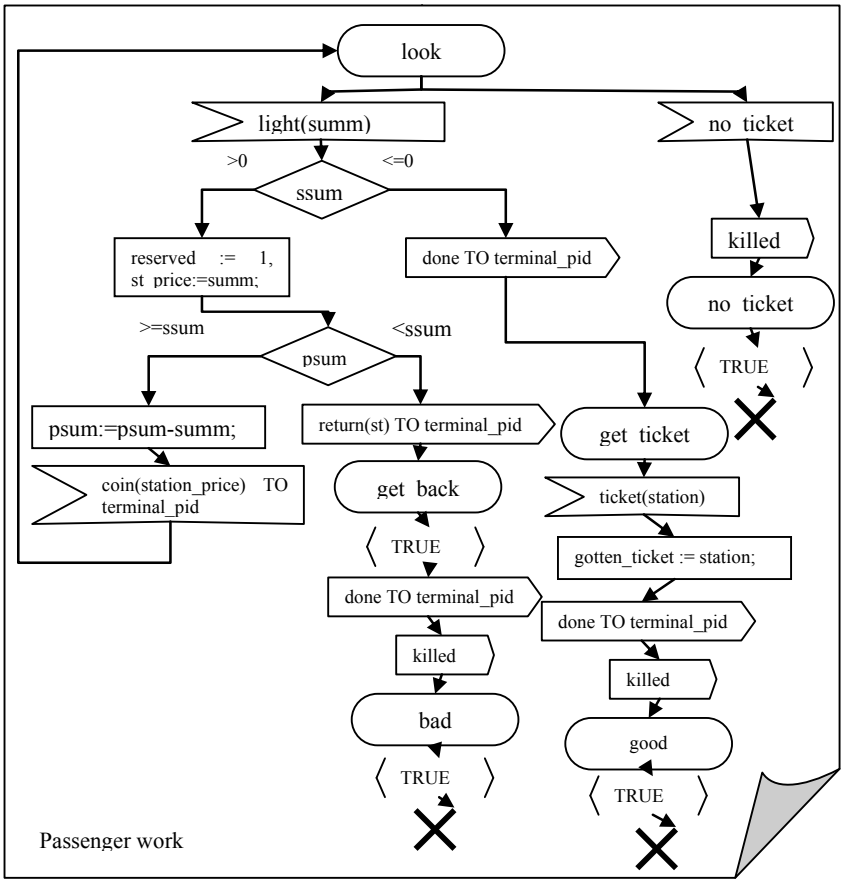

Рис.8.2. Блок-схема фрагмента процесса passenger

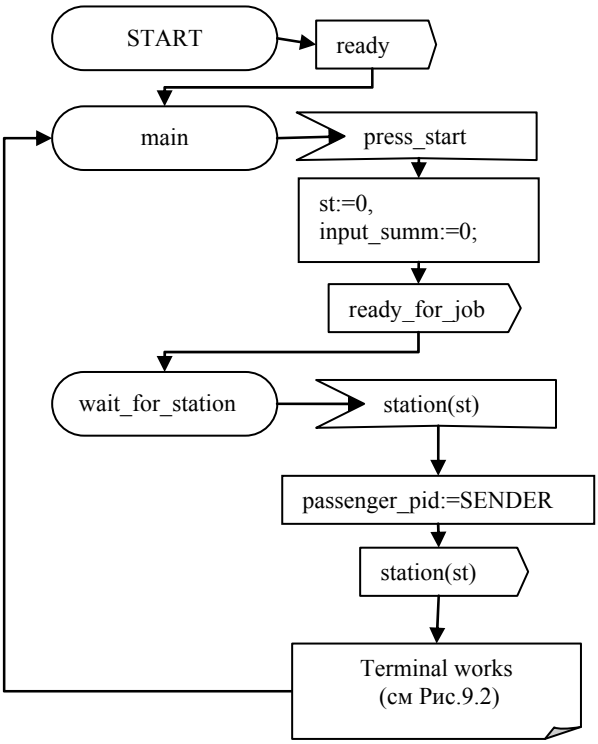

Рис.9.1. Блок-схема процесса terminal

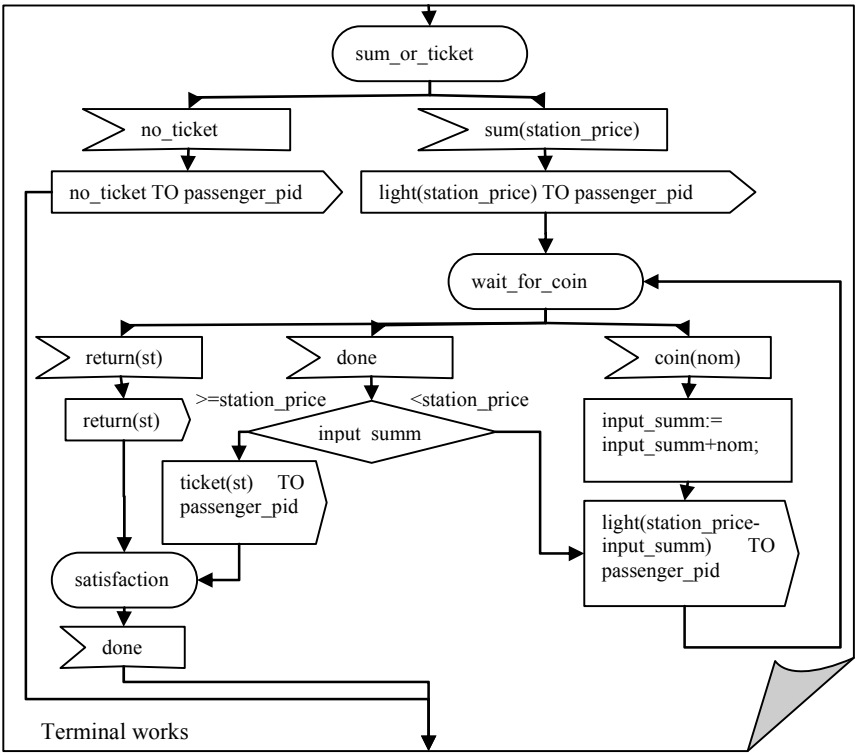

Рис.9.2. Блок-схема фрагмента процесса terminal

Были проведены два эксперимента с программным комплексом SRDSV по анализу системы управления сетью касс-терминалов и её автоматической верификации. Вначале спецификация этой системы на языке SDL и проверяемое свойство на языке SPL были транслированы в язык dREAL.

7.2. Эксперимент с системой автоматического моделирования

В первом эксперименте полученная dREAL-спецификация и соответствующий запрос пользователя были поданы на вход системы автоматического моделирования. Эта dREAL-спецификация была успешно проверена с помощью следующих запросов:

1. Станция на полученном билете совпадает с требуемой станцией.

```
WHEN (passenger.wants_ticket==passenger.gotten_ticket) PAUSE
```

2. Если стала известна стоимость билета до требуемой станции (т.е. достаточно билетов), и хватает денег, то пассажир получит билет (т.е. попадет в состояние good).

```
WHEN (passenger.st_price[pid]>=0 &&
 passenger.psum>= passenger.st_price[pid] &&
 passenger AT passenger.good) PAUSE
```

3. Если же стоимость билета не стала известна (т.е. нет билетов до требуемой станции), то пассажир не получит билет (попадет в состояние no_ticket).

```
WHEN (passenger. st_price[pid]<0 &&
 passenger AT passenger.no_ticket) PAUSE
```

4. Если стала известна стоимость билета (т.е. достаточно билетов до требуемой станции), но не хватает денег, то пассажир не получит билет (т.е. попадет в состояние bad).

```
WHEN (passenger.st_price[pid]>=0 &&
 passenger.psum< passenger.st_price[pid] &&
 passenger AT passenger.bad) PAUSE
```

Эксперимент проводился с 10 терминалами и 15 пассажирами.

7.3 Эксперимент с системой SPIN

Второй эксперимент состоял в трансляции dREAL-спецификации системы управления сетью касс-терминалов в язык PROMELA и верификации ее с помощью системы SPIN.

Из-за того, что в системе SPIN невозможно использовать локальные переменные, используемые в формулах локальные переменные отмечены в SDL-спецификации как REVEALED, что делает их глобальными массивами при переводе в dREAL. Таким образом, локальный массив процесса, не имеющего экземпляров, становится глобальным массивом, а скалярная переменная процесса с экземплярами становится глобальным массивом, индексированным идентификаторами процессов.

Определим вначале следующие функции и предикаты.

`passenger[pid].wants_ticket` – функция, определяющая номер станции, до которой пассажиру нужен билет (т.е. результатом функции является значение переменной `wants_ticket` экземпляра процесса `passenger` с PId'ом `pid`);

`passenger[pid].gotten_ticket` – функция, определяющая номер станции, до которой пассажир получил билет;

Предикат «до нужной пассажиру станции есть билеты»:

```
enough_tickets(pid)   : PRED
 (main_machine.ticket[passenger[pid].wants_ticket]>0);
```

Предикат «у пассажира достаточно денег на билет до нужной ему станции»:

```
enough_money(pid)   : PRED
(passenger[pid].psum >=
main_machine.expences[passenger[pid].wants_ticket]);
```

Предикат «пассажир получил билет до нужной ему станции»:

```
got_good_ticket(pid)   : PRED
(passenger[pid].gotten_ticket ==
passenger[pid].wants_ticket);
```

Предикат «пассажир получил какой-то билет»:

```
got_some_ticket(pid)   : PRED
(passenger[pid].gotten_ticket > 0 );
```

Проверялись следующие три свойства для 5 терминалов и 15 пассажиров.

7.3.1. prop1. Если до нужной пассажиру станции билеты есть, и у него достаточно денег, то он получит билет до этой станции.

Свойство prop1 на языке SPL представляется в виде

```
(ALL PROCESS passenger WITH PID pid
((enough_tickets(pid) && enough_money(pid) =>
(EACH ET got_good_ticket(pid)))));.
```

Это свойство представляется на языке dREAL.log в виде

```
(ALL PROCESS passenger WITH PID pid
((enough_tickets(wants_ticket[pid]) &&
enough_money(pid)=>
(EACH ET ( got_good_ticket(pid))))));.
```

Следующая LTL формула представляет свойство prop1:

```
enough_tickets && enough_money ->
<>(got_good_ticket)
```

Результат эксперимента: свойство подтверждено, и при этом использовано 310МБ памяти и 140 секунд времени.

7.3.2 prop2. Если до нужной пассажиру станции билетов нет, то он не получит никакого билета;

На языке SPL:

```
 (ALL PROCESS passenger WITH PID pid
((NOT enough_tickets(pid) =>
(EACH AT (NOT got_some_ticket(pid))))));.
```

На языке dREAL.log:

```
 (ALL PROCESS passenger WITH PID pid
((NOT enough_tickets(pid) =>
(EACH AT (NOT got_some_ticket(pid))))))).
```

На языке LTL:

```
 ! enough_tickets -> [] ! got_some_ticket
```

Результат эксперимента: свойство подтверждено, и при этом использовано 206МБ памяти и 28 секунд времени.

7.3.3. prop3. Если у пассажира недостаточно денег до нужной ему станции, то он не получит никакого билета.

На языке SPL**:**

```
 (ALL PROCESS passenger WITH PID pid
((NOT enough_money(pid) =>
(EACH AT (NOT got_some_ticket(pid))))))).
```

На языке dREAL.log:

```
(ALL PROCESS passenger WITH PID pid
((NOT enough_money( wants_ticket[pid]) =>
(EACH AT (NOT got_some_ticket(pid))))))).
```

На языке LTL:

```
! enough_money -> [] ! got_some_ticket
```

Результат эксперимента: свойство подтверждено и при этом использовано 700МБ памяти и 6000 секунд времени.

В силу особенности SPIN для доказательства этих свойств утверждались их отрицания, которые опровергались системой SPIN.

ЗАКЛЮЧЕНИЕ

Насколько нам известно, язык Dynamic-REAL является единственным комбинированным языком спецификаций как распределенных систем, так и их свойств. Преимущества языка dREAL обусловлены простым синтаксисом, допускающим графическое представление выполнимых спецификаций, полной компактной операционной семантикой и выразительным языком представления свойств. SRDSV является мощным программным комплексом, который включает транслятор из SDL в dREAL, систему автоматического моделирования dREAL-спецификаций, а также два верификатора. Входной язык этого транслятора включает все базовые конструкции языка SDL (кроме абстрактных типов данных), которые образуют язык SDL-88 и широко применяются на практике [13]. Система автоматического моделирования играет важную роль, так как может успешно применяться и

в тех случаях, когда системы верификации комплекса SRDSV неприменимы из-за громоздких и сложных моделей SDL-спецификаций. Преимущества нашего подхода иллюстрируются верификацией динамической системы управления сетью касс-терминалов. В [4] представлено успешное применение комплекса SRDSV для анализа и верификации коммуникационных протоколов скользящего окна, включая применяемый на практике *i*-протокол.

Предполагается расширить комплекс SRDSV транслятором dREAL-спецификаций в сети Петри высокого уровня и системой верификации методом проверки моделей относительно свойств, представленных в логике ветвящегося времени CTL. Также предполагается расширить язык запросов системы автоматического моделирования с целью обнаружения тупиков и зацикливания.

СПИСОК ЛИТЕРАТУРЫ

1. Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. – М.: Радио и связь, 1993.
2. Карпов Ю.Г. Model checking. Верификация параллельных и распределенных программных систем. – Санкт-Петербург: БХВ-Петербург, 2010.
3. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.
4. Корниенко И.С. Анализ и верификация коммуникационных протоколов, представленных на языке SDL, с помощью языка Dynamic-REAL . – Магистерская диссертация, ММФ Новосибирского государственного университета, 2010.
5. Непомнящий В.А., Шилов Н.В. Real92: Комбинированный язык спецификаций для систем и свойств взаимодействующих процессов реального времени // Программирование. – 1993. – № 6. – С. 64–80.
6. Непомнящий В.А., Шилов Н.В., Бодин Е.В. Спецификация и верификация распределенных систем средствами языка Elementary-REAL // Программирование. – 1999. – № 4. – С. 54–67.
7. Непомнящий В.А., Шилов Н.В., Бодин Е.В. REAL: язык для спецификации и верификации систем реального времени // Системная информатика.– Новосибирск: Наука, 2000. – Вып. 7. – С. 174–224.
8. Непомнящий В.А., Бодин Е.В., Веретнов С.О., Тюрюшкин М.В. Симуляция и верификация статических SDL-спецификаций распределенных систем с помощью промежуточного языка REAL // Препринт ИСИ СО РАН № 142, 2007.
   http://www.iis.nsk.su/preprints/pdf/142.pdf

9. Непомнящий В.А., Бодин Е.В, Веретнов С.О. Язык спецификаций распределенных систем Dynamic-REAL. – Новосибирск, 2007. – (Препр. / ИСИ СО РАН; № 147).
http://www.iis.nsk.su/preprints/pdf/147.pdf
10. Bosnacki D., Dams D., Holenderski L., Sidorova N. Model checking SDL with Spin // Proc. TACAS/ETAPS 2000. –Lect. Notes Comput. Sci. – 2000. – Vol. 1785. – P. 363–377.
11. Bozga M., Graf S., Ober Il., Ober Iu., Sifakis J. The IF toolset // Proc. SFM-RT 2004. – Lect. Notes Comp. Sci. – 2004. – Vol. 3185. – P. 237–267.
12. Eschbach R., Glasser U., Gotzhein R., Prinz A. On the formal semantics of SDL-2000: A compilation approach based on an abstract SDL machine // Proc. ASM 2000. – Lect. Notes Comput. Sci. – 2000. – Vol. 1912. – P. 242–265.
13. Grammes R., Gotzhein R. SDL Profiles – Formal Semantics and Tool Support // Proc. FASE 2007. – Lect. Notes in Comp. Sci. – 2007. – Vol. 4422. – P. 200–214.
14. Holzmann, G.J.: The SPIN model checker. Primer and Reference Manual. – Addison-Wesley, 2004.
15. Nepomniaschy V.A., Shilov N.V., Bodin E.V., Kozura V.E. Basic-REAL: integrated approach for design, specification and verification of distributed systems // Proc. IFM 2002. – Lect. Notes Comput. Sci. – 2002. – Vol. 2335. – P. 69–88.
16. Prinz A., Lowis M. Engineering the SDL formal language definition // Proc. FMOODS 2003. – Lect. Notes Comput. Sci. – 2003. – Vol. 2884. – P. 47–63.
17. Sidorova N., Steffen M. Verifing Large SDL-Specifications Using Model Checking // Proc. SDL 2001. – Lect. Notes in Comp. Sci. – 2001. – Vol. 2078. – P. 403–420.
18. Specification and Description Language (SDL). – CCITT, Recommendation Z.100, 1988.

КОНТЕКСТ SDL-СПЕЦИФИКАЦИИ СИСТЕМЫ УПРАВЛЕНИЯ СЕТЬЮ КАСС-ТЕРМИНАЛОВ

```
    SYSTEM All;
/*объявление блоков, сигналов, каналов*/
  BLOCK Passengers_and_Terminals;
   SIGNAL
/*сигнал с номером станции, до которой
пассажир намерен взять билет*/
/*параметр1: номер станции*/
        station         (integer),

/*сигнал, передающий информацию,
сколько денег ввел пассажир*/
/*параметр1: сумма денег*/
        coin            (integer),

       /*сигнал успешного завершения
работы пассажира*/
        done            ,

       /* сигнал, сообщающий стоимость
билета*/
/*параметр1: цена билета*/
        sum             (integer),

       /*сигнал деинициализации пассажи-
ра*/
        killed          ,

/*сигнал инициации пассажира пассажира
в очереди*/
/*параметр1: номер станции, до которой
хочет ехать пассажир*/
/*параметр1: номер терминала к которому
подошел пассажир*/
/*параметр1: сумма денег для покупки
билета*/
        init_pas        (integer, integer, inte-
ger),

/*сигнал инициации пассажира пассажира
в очереди*/
/*параметр1: номер станции, до которой
хочет ехать пассажир*/
/*параметр1: PId терминала к которому
подошел пассажир*/
/*параметр1: сумма денег для покупки
билета*/
        init            (integer, PId, integer),

       /*сигнал индикации стоимости биле-
та*/
/*параметр1: стоимость билета*/
        light           (integer),

       /*сигнал выдачи билета*/
/*параметр1: номер станции на билете*/
        ticket          (integer),

       /*сигнал готовности терминала к
сеансу работы с текущим пассажиром*/
        ready_for_job,

       /*сигнал отсутсвия билетов до стан-
ции*/
        no_ticket,

       /*сигнал отмены покупки билета*/
/*параметр1: номер станции*/
      return            (integer),

        /*сигнал готовности терминала к
работе*/
        ready,

       /*сигнал начала сеанса работы с
пассажиром*/
        press_start     ;
/*канал для связи внешнего окружения с
системой*/
   SIGNALROUTE e2q
     FROM ENV TO queue WITH init_pas;
/*канал для инициации и деинициализа-
ции пассажира*/
   SIGNALROUTE
init_and_killing_passenger
```

```
    FROM queue TO passenger WITH init;
    FROM passenger TO queue WITH
killed;
/*канал для индикации состояния терми-
нала*/
   SIGNALROUTE indicator
    FROM terminal TO passenger WITH
light, no_ticket,ready_for_job;
/*канал монетоприемника */
   SIGNALROUTE slot
    FROM passenger TO terminal WITH
coin;
/*канал от пассажира к терминалу*/
   SIGNALROUTE starting_data
    FROM passenger TO terminal WITH
press_start;
/*канал ввода данных на терминал*/
   SIGNALROUTE buttons
    FROM passenger TO terminal WITH
station,return,done;
/*канал выдачи билета пассажиру*/
   SIGNALROUTE booking
    FROM terminal TO passenger WITH
ticket;
/*канал от терминала к главной машине*/
   SIGNALROUTE t2m
    FROM terminal TO main_machine
WITH station, return;
/*канал от главной машины к терминалу*/
   SIGNALROUTE m2t
    FROM main_machine TO terminal
WITH sum, no_ticket;
/*канал от терминала к очереди*/
   SIGNALROUTE t2q
    FROM terminal TO queue WITH ready;
/*процесс Очередь*/
   PROCESS queue          REFERENCED;
/*процесс Главная машина*/
   PROCESS main_machine      REFER-
ENCED;
/*процесс Пассажир*/
   PROCESS passenger(0,10)      REFER-
ENCED;
/*процесс Терминал*/
   PROCESS terminal(10,10)      REFER-
ENCED;

 ENDBLOCK Passengers_and_Terminals;
ENDSYSTEM All;
```

DREAL.EX-СПЕЦИФИКАЦИЯ СИСТЕМЫ УПРАВЛЕНИЯ СЕТЬЮ КАСС-ТЕРМИНАЛОВ

```
Passengers_and_Terminals: BLOCK
  TYPE int_arr__revealed IS
   integer ARRAY OF integer.
  TYPE integer__revealed IS
   PId ARRAY OF integer.
  PR VAR expences OF int_arr__revealed.
  PR VAR ticket OF int_arr__revealed.
  PR VAR wants_ticket OF inte-
ger__revealed.
  PR VAR gotten_ticket OF inte-
ger__revealed.
  PR VAR psum OF integer__revealed.
  PR VAR summ OF integer__revealed.
  PR VAR reserved OF integer__revealed.
  PR VAR at_no_ticket OF inte-
ger__revealed.
  INN UNB QUEUE CHN t2q_rl
   FOR ready.
  INN UNB QUEUE CHN m2t_rl

    FOR no_ticket;
    FOR sum
    WITH PAR p1 OF integer.
  INN UNB QUEUE CHN t2m_rl

    FOR return
    WITH PAR p1 OF integer;
    FOR station
    WITH PAR p1 OF integer.
  INN UNB QUEUE CHN booking_rl
   FOR ticket
   WITH PAR p1 OF integer.
  INN UNB QUEUE CHN buttons_rl

    FOR done;
    FOR return
    WITH PAR p1 OF integer;
    FOR station
    WITH PAR p1 OF integer.
  INN UNB QUEUE CHN starting_data_rl
   FOR press_start.
  INN UNB QUEUE CHN slot_rl
    FOR coin
    WITH PAR p1 OF integer.
  INN UNB QUEUE CHN indicator_rl

    FOR ready_for_job;
    FOR no_ticket;
    FOR light
    WITH PAR p1 OF integer.
  INN UNB QUEUE CHN
init_and_killing_passenger_rl
    FOR init
    WITH PAR p1 OF integer,
    WITH PAR p2 OF PId,
    WITH PAR p3 OF integer.
  INN UNB QUEUE CHN
init_and_killing_passenger_rl_inv
    FOR kill.
  INP UNB QUEUE CHN e2q_rl
    FOR init_pas
    WITH PAR p1 OF integer,
    WITH PAR p2 OF integer,
    WITH PAR p3 OF integer.

    FROM terminal CHN t2q_rl  TO queue.
    FROM main_machine CHN m2t_rl  TO
terminal.
    FROM terminal CHN t2m_rl  TO
main_machine.
    FROM terminal CHN booking_rl  TO
passenger.
    FROM passenger CHN buttons_rl  TO
terminal.
    FROM passenger CHN starting_data_rl
TO terminal.
    FROM passenger CHN slot_rl  TO termi-
nal.
    FROM terminal CHN indicator_rl  TO
passenger.
    FROM queue CHN
init_and_killing_passenger_rl  TO passenger.
```

```
  FROM passenger CHN
init_and_killing_passenger_rl_inv  TO
queue.
  FROM ENV CHN e2q_rl  TO queue.

  queue: PROCESS
   TYPE arr IS
    integer ARRAY OF PId.
   PR VAR terminal_count OF integer.
   PR VAR N OF integer.
   PR VAR MAX_TERMINAL OF integer.
   PR VAR st OF integer.
   PR VAR qpsum OF integer.
   PR VAR count OF integer.
   PR VAR terminal_inst OF arr.

   TRANSITION Start__rl
   EXE count:=0;
   terminal_count:=0;
   MAX_TERMINAL:=10;
   FROM NOW TO NOW
   JUMP create_loop__rl.

   TRANSITION create_loop__rl
   READ ready FROM t2q_rl
   FROM NOW TO INF
   JUMP create_loop__X1.

   TRANSITION create_loop__X1
   EXE terminal_count:=terminal_count+1;
   terminal_inst[terminal_count]:=SENDER;
   FROM NOW TO NOW
   JUMP begin_decision__1.

   TRANSITION begin_decision__1
   EXE SKIP
   FROM NOW TO NOW
   JUMP create_loop__1.

   TRANSITION create_loop__1
   WHEN (termi-
nal_count<MAX_TERMINAL)EXE SKIP
   FROM NOW TO NOW
   JUMP create_loop__rl.

   TRANSITION create_loop__1
   WHEN (NOT (termi-
nal_count<MAX_TERMINAL))EXE SKIP
   FROM NOW TO NOW
   JUMP look__rl.

   TRANSITION look__rl
   WHEN (count < MAX_TERMINAL)
   READ init_pas(st,N,qpsum) FROM e2q_rl
   FROM NOW TO INF
   JUMP look__X1.

   TRANSITION end_decision__1
   EXE SKIP
   FROM NOW TO NOW
   JUMP end_of_process.

   TRANSITION look__X1
   CREATE PROCESS passenger
   FROM NOW TO NOW
   JUMP look__X2.

   TRANSITION look__X2
   WRITE init(st,terminal_inst[N],qpsum)
INTO
init_and_killing_passenger_rl[OFFSPRING]
   FROM NOW TO NOW
   JUMP look__X3.

   TRANSITION look__X3
   EXE count:=count+1;
   FROM NOW TO NOW
   JUMP look__rl.

   TRANSITION look__rl
   WHEN (count > 0)
   READ kill FROM
init_and_killing_passenger_rl_inv
   FROM NOW TO INF
   JUMP look__X4.

   TRANSITION look__X4
   EXE count:=count-1;
   FROM NOW TO NOW
   JUMP look__rl.
  END; /* queue */

  main_machine: PROCESS
   TYPE int_arr IS
    integer ARRAY OF integer.
   PR VAR st OF integer.

   TRANSITION Start__rl
```

```
  EXE expences[1]:=10;
  expences[2]:=15;
  expences[3]:=20;
  ticket[1]:=2;
  ticket[2]:=3;
  ticket[3]:=4;
  FROM NOW TO NOW
  JUMP main__rl.

  TRANSITION main__rl
  READ station(st) FROM t2m_rl
  FROM NOW TO INF
  JUMP begin_decision__1.

  TRANSITION begin_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP main__1.

  TRANSITION main__1
  WHEN (ticket[st]=0)EXE SKIP
  FROM NOW TO NOW
  JUMP main__1__X1.

  TRANSITION main__1__X1
  WRITE no_ticket INTO m2t_rl[SENDER]
  FROM NOW TO NOW
  JUMP main__rl.

  TRANSITION main__1
  WHEN (NOT (ticket[st]=0))EXE
ticket[st]:=ticket[st]-1;
  FROM NOW TO NOW
  JUMP main__1__X2.

  TRANSITION main__1__X2
  WRITE sum(expences[st]) INTO
m2t_rl[SENDER]
  FROM NOW TO NOW
  JUMP main__rl.

  TRANSITION end_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP end_of_process.

  TRANSITION main__rl
  READ return(st) FROM t2m_rl
  FROM NOW TO INF
  JUMP main__X3.

  TRANSITION main__X3
  EXE ticket[st]:=ticket[st]+1;
  FROM NOW TO NOW
  JUMP main__rl.
END; /* main_machine */

passenger: PROCESS(0,10)
  PR VAR terminal_pid OF PId.
  PR VAR N OF integer.
  PR VAR input_summ OF integer.
  PR VAR nom OF integer.
  PR VAR st OF integer.
  PR VAR st_price OF integer.
  PR VAR station OF integer.

  TRANSITION Start__rl
  EXE nom:=1;
  reserved[SELF]:=0;
  st_price:=0;
  FROM NOW TO NOW
  JUMP init_pas__rl.

  TRANSITION init_pas__rl
  READ init(st,terminal_pid,psum) FROM
init_and_killing_passenger_rl
  FROM NOW TO INF
  JUMP init_pas__X1.

  TRANSITION init_pas__X1
  EXE wants_ticket[SELF]:=st;
  FROM NOW TO NOW
  JUMP init_pas__X2.

  TRANSITION init_pas__X2
  WRITE press_start INTO start-
ing_data_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP wait_terminal__rl.

  TRANSITION wait_terminal__rl
  READ ready_for_job FROM indicator_rl
  FROM NOW TO INF
  JUMP wait_terminal__X1.

  TRANSITION wait_terminal__X1
  WRITE station(st) INTO but-
tons_rl[terminal_pid]
```

```
  FROM NOW TO NOW
  JUMP look__rl.

  TRANSITION look__rl
  READ light(summ) FROM indicator_rl
  FROM NOW TO INF
  JUMP begin_decision__1.

  TRANSITION begin_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP look__1.

  TRANSITION look__1
  WHEN (summ<=0)EXE SKIP
  FROM NOW TO NOW
  JUMP look__1__X1.

  TRANSITION look__1__X1
  WRITE done INTO but-
tons_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP get_ticket__rl.

  TRANSITION get_ticket__rl
  READ ticket(station) FROM booking_rl
  FROM NOW TO INF
  JUMP get_ticket__X1.

  TRANSITION look__1
  WHEN (NOT (summ<=0))EXE re-
served[SELF]:=1;
  st_price:=summ;
  FROM NOW TO NOW
  JUMP begin_decision__2.

  TRANSITION begin_decision__2
  EXE SKIP
  FROM NOW TO NOW
  JUMP look__2.

  TRANSITION look__2
  WHEN (psum<summ)EXE SKIP
  FROM NOW TO NOW
  JUMP look__2__X2.

  TRANSITION look__2__X2
  WRITE return(st) INTO but-
tons_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP get_back__rl.

  TRANSITION get_back__rl
  WHEN TRUE
  WRITE done INTO but-
tons_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP get_back__X1.

  TRANSITION look__2
  WHEN (NOT (psum<summ))EXE
psum[SELF]:=psum-summ;
  FROM NOW TO NOW
  JUMP look__2__X3.

  TRANSITION look__2__X3
  WRITE coin(summ) INTO
slot_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP look__rl.

  TRANSITION end_decision__2
  EXE SKIP
  FROM NOW TO NOW
  JUMP end_decision__1.

  TRANSITION end_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP end_of_process.

  TRANSITION look__rl
  READ no_ticket FROM indicator_rl
  FROM NOW TO INF
  JUMP look__X4.

  TRANSITION look__X4
  EXE at_no_ticket[SELF]:=1;
  FROM NOW TO NOW
  JUMP look__X5.

  TRANSITION look__X5
  WRITE kill INTO
init_and_killing_passenger_rl_inv
  FROM NOW TO NOW
  JUMP no_ticket__rl.

  TRANSITION no_ticket__rl
```

```
  EXE SKIP
  FROM NOW TO NOW
  JUMP no_ticket__stop__rl.

  TRANSITION get_back__X1
  WRITE kill INTO
init_and_killing_passenger_rl_inv
  FROM NOW TO NOW
  JUMP bad__rl.

  TRANSITION bad__rl
  EXE SKIP
  FROM NOW TO NOW
  JUMP bad__stop__rl.

  TRANSITION get_ticket__X1
  EXE gotten_ticket[SELF]:=station;
  FROM NOW TO NOW
  JUMP get_ticket__X2.

  TRANSITION get_ticket__X2
  WRITE done INTO but-
tons_rl[terminal_pid]
  FROM NOW TO NOW
  JUMP get_ticket__X3.

  TRANSITION get_ticket__X3
  WRITE kill INTO
init_and_killing_passenger_rl_inv
  FROM NOW TO NOW
  JUMP good__rl.

  TRANSITION good__rl
  EXE SKIP
  FROM NOW TO NOW
  JUMP good__stop__rl.

  TRANSITION no_ticket__stop__rl
  WHEN TRUE
  STOP
  FROM NOW TO NOW
  JUMP end_of_process.

  TRANSITION good__stop__rl
  WHEN TRUE
  STOP
  FROM NOW TO NOW
  JUMP end_of_process.

  TRANSITION bad__stop__rl
  WHEN TRUE
  STOP
  FROM NOW TO NOW
  JUMP end_of_process.
 END; /* passenger */

 terminal: PROCESS(10,10)
  PR VAR passenger_pid OF PId.
  PR VAR nom OF integer.
  PR VAR st OF integer.
  PR VAR station_price OF integer.
  PR VAR input_summ OF integer.

  TRANSITION Start__rl
  WRITE ready INTO t2q_rl
  FROM NOW TO NOW
  JUMP main__rl.

  TRANSITION main__rl
  READ press_start FROM starting_data_rl
  FROM NOW TO INF
  JUMP main__X1.

  TRANSITION main__X1
  EXE st:=0;
  input_summ:=0;
  FROM NOW TO NOW
  JUMP main__X2.

  TRANSITION main__X2
  WRITE ready_for_job INTO indica-
tor_rl[SENDER]
  FROM NOW TO NOW
  JUMP wait_for_station__rl.

  TRANSITION wait_for_station__rl
  READ station(st) FROM buttons_rl
  FROM NOW TO INF
  JUMP wait_for_station__X1.

  TRANSITION wait_for_station__X1
  EXE passenger_pid:=SENDER;
  FROM NOW TO NOW
  JUMP wait_for_station__X2.

  TRANSITION wait_for_station__X2
  WRITE station(st) INTO t2m_rl
  FROM NOW TO NOW
```

```
  JUMP sum_or_ticket__rl.

  TRANSITION sum_or_ticket__rl
  READ sum(station_price) FROM m2t_rl
  FROM NOW TO INF
  JUMP sum_or_ticket__X1.

  TRANSITION sum_or_ticket__X1
  WRITE light(station_price) INTO indica-
tor_rl[passenger_pid]
  FROM NOW TO NOW
  JUMP wait_for_coin__rl.

  TRANSITION wait_for_coin__rl
  READ coin(nom) FROM slot_rl
  FROM NOW TO INF
  JUMP wait_for_coin__X1.

  TRANSITION sum_or_ticket__rl
  READ no_ticket FROM m2t_rl
  FROM NOW TO INF
  JUMP sum_or_ticket__X2.

  TRANSITION sum_or_ticket__X2
  WRITE no_ticket INTO indica-
tor_rl[passenger_pid]
  FROM NOW TO NOW
  JUMP main__rl.

  TRANSITION wait_for_coin__X1
  EXE input_summ:=input_summ+nom;
  FROM NOW TO NOW
  JUMP wait_for_coin__X2.

  TRANSITION wait_for_coin__X2
  WRITE light(station_price-input_summ)
INTO indicator_rl[passenger_pid]
  FROM NOW TO NOW
  JUMP wait_for_coin__rl.

  TRANSITION wait_for_coin__rl
  READ done FROM buttons_rl
  FROM NOW TO INF
  JUMP begin_decision__1.

  TRANSITION begin_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP wait_for_coin__1.

  TRANSITION wait_for_coin__1
  WHEN (input_summ<station_price)EXE
SKIP
  FROM NOW TO NOW
  JUMP wait_for_coin__1__X3.

  TRANSITION wait_for_coin__1__X3
  WRITE light(station_price-input_summ)
INTO indicator_rl[passenger_pid]
  FROM NOW TO NOW
  JUMP wait_for_coin__rl.

  TRANSITION wait_for_coin__1
  WHEN (NOT (in-
put_summ<station_price))EXE SKIP
  FROM NOW TO NOW
  JUMP wait_for_coin__1__X4.

  TRANSITION wait_for_coin__1__X4
  WRITE ticket(st) INTO book-
ing_rl[passenger_pid]
  FROM NOW TO NOW
  JUMP satisfaction__rl.

  TRANSITION satisfaction__rl
  READ done FROM buttons_rl
  FROM NOW TO INF
  JUMP main__rl.

  TRANSITION end_decision__1
  EXE SKIP
  FROM NOW TO NOW
  JUMP end_of_process.

  TRANSITION wait_for_coin__rl
  READ return(st) FROM buttons_rl
  FROM NOW TO INF
  JUMP wait_for_coin__X5.

  TRANSITION wait_for_coin__X5
  WRITE return(st) INTO t2m_rl
  FROM NOW TO NOW
  JUMP satisfaction__rl.
 END; /* terminal */
END; /* Passengers_and_Terminals */
/* END; */ /* All */
```

В.А. Непомнящий, Е.В. Бодин, С.О. Веретнов

МОДЕЛИРОВАНИЕ И ВЕРИФИКАЦИЯ РАСПРЕДЕЛЁННЫХ СИСТЕМ, ПРЕДСТАВЛЕННЫХ НА ЯЗЫКЕ SDL, С ПОМОЩЬЮ ЯЗЫКА DYNAMIC-REAL

Препринт
156

Рукопись поступила в редакцию 25.04.10
Редактор Т. М. Бульонкова
Рецензент Н.В. Шилов

Подписано в печать 15.06.10
Формат бумаги 60 × 84 1/16 Объем 2.5 уч.-изд.л., 2.75 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42