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

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

СИМУЛЯЦИЯ И ВЕРИФИКАЦИЯ СТАТИЧЕСКИХ SDL-СПЕЦИФИКАЦИЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ С ПОМОЩЬЮ ПРОМЕЖУТОЧНОГО ЯЗЫКА REAL

Препринт
142

Новосибирск 2007

В настоящей работе описана система SRPV (SDL/REAL Protocol Verifier) для моделирования, анализа и верификации статических SDL-спецификаций коммуникационных протоколов, которая базируется на трансляции языка SDL в язык Basic-REAL (bREAL). Эта система включает: транслятор из языка SDL в язык bREAL, конвертор из языка описания свойств SDL-спецификаций в подъязык логических bREAL-спецификаций, систему моделирования bREAL-спецификаций и систему верификации этих спецификаций. В качестве примера описано применение системы SRPV к анализу и верификации протокола «Касса-Пассажир».

Работа частично поддержана грантами РФФИ № 07-07-00173а и 06-01-00464а.

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

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

SIMULATION AND VERIFICATION OF STATIC SDL SPECIFICATIONS OF DISTRIBUTED SYSTEMS BY HELP OF THE INTERMEDIATE LANGUAGE REAL

Preprint
142

Novosibirsk 2007

The present paper describes system SRPV (SDL/REAL Protocol Verifier) for modelling, analysis and verification of static SDL-specifications of communication protocols. This system is based on translation of SDL language to Basic-REAL (bREAL) language. This system includes: translator from SDL language to bREAL language, converter from properties description language for SDL-specifications to logical specifications sublanguage of bREAL, modelling system for bREAL-specifications, and verification system for these specifications. As an illustrative example, an application of the SRPV system to analysis and verification of "Passenger and Slot-Machine" protocol is described.

1. ВВЕДЕНИЕ

Последние годы заметно возрастает роль формальных методов, применяемых для разработки распределенных систем, таких как, например, коммуникационные протоколы. Это связано с тем, что для современных распределенных систем усложняется документирование, анализ и верификация. Для преодоления указанных трудностей используются языки выполнимых спецификаций SDL, Estelle, LOTOS, принятые в качестве стандарта международной организацией ITU (International Telecommunication Union). Среди этих языков наиболее активно на практике применяется язык SDL [1, 23]. Верификация выполнимых спецификаций, представленных на языке SDL, заключается в проверке корректности их ключевых свойств и является известной открытой проблемой 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 [9, 12, 13, 17, 21, 22]. Для представления свойств SDL-спецификаций использовались логики линейного времени LTL и ветвящегося времени CTL [17, 10], метрическая временная логика [16], логика TLA+ [14], алгебра процессов CRL [13]. Однако использование этих формализмов для представления свойств распределенных систем, функционирующих в реальном времени, приводит к громоздким формулам, особенно в случае иерархических SDL-спецификаций.

Для доказательства свойств моделей распределенных систем применяется метод проверки моделей (model checking method), который успешно используется для систем с конечным числом состояний [2]. Однако его применение[7, 8] в случае бесконечного или параметризованного числа состояний связано с преодолением значительных трудностей. Непосредственное применение этого метода для доказательства свойств SDL-спецификаций требует предварительного преобразования спецификаций и часто приводит к значительным трудностям ввиду громоздкости моделей этих спецификаций. Для преодоления этих трудностей в [8] используется модельный язык IF.

Идея нашего подхода к проблеме верификации SDL-спецификаций состоит в разработке модельных языков, ориентированных на верификацию, которые были бы комбинированными, т.е. включали подъязыки выполни-

Работа частично поддержана грантами РФФИ № 07-07-00173а и 06-01-00464а.

мых и логических спецификаций. Первоначальная версия комбинированного языка спецификаций REAL91, которая базируется на SDL и логике CTL [2], была представлена в [3], где описан его формальный синтаксис и неформальная семантика. Следующая версия REAL92 этого языка представлена в [4, 18], где дан набросок формальной семантики. Последняя версия REAL — Basic-REAL (bREAL) была представлена в [6, 19, 20], где описана полная структурная операционная семантика выполнимых спецификаций в виде систем переходов, которая позволила доказать важные семантические свойства. Упрощенная версия этого языка, названная Elementary-REAL, представлена в [5]. Некоторые эксперименты по верификации спецификаций на языке bREAL методом проверки моделей приведены в [20].

Цель настоящей работы — описать систему SRPV (SDL/REAL Protocol Verifier) для моделирования, анализа и верификации статических SDL-спецификаций распределенных систем, которая базируется на трансляции языка SDL в язык bREAL. Данная работа состоит из девяти разделов. В разд. 2 дается обзор языка SDL. Разд. 3 посвящен обзору языка bREAL. В разд. 4 описан транслятор из языка SDL в язык bREAL. В разд. 5 представлен язык описания свойств SDL-спецификаций, а также конвертор из этого языка в подъязык логических bREAL-спецификаций. Система моделирования bREAL-спецификаций представлена в разд. 6, а система верификации этих спецификаций — в разд. 7. Разд. 8 посвящен применению системы SRPV к анализу и верификации протокола «Касса-Пассажир». В разд. 9 обсуждаются достоинства системы SRPV и перспективы ее развития.

2. ОБЗОР ЯЗЫКА SDL

SDL (Specification and Description Language) — язык спецификации и описаний — разработан бывшим Международным консультативным комитетом по телеграфии и телефонии (CCITT). Язык предназначен для описания структуры и функционирования систем реального времени, в частности, сетей связи. SDL построен на базе модели конечного автомата по объектно-ориентированной схеме.

Язык SDL имеет как статические, так и динамические конструкции, которые описывают порождение и уничтожение экземпляра процесса. В данной работе описывается только статическое подмножество языка SDL без динамических конструкций.

Самый общий объект, описываемый на SDL, называется *системой*. Все остальные объекты находятся на более низких уровнях иерархии определе-

ний. Все, что не вошло в описание системы, называется окружением (внешней средой) системы. Каждая система должна иметь уникальное имя.

Важной особенностью языка SDL является концепция типов данных, в основу которой положена алгебраическая модель. Все виды данных, используемые в конкретной системе, рассматриваются как компоненты единого типа. Элемент типа данных называется значением. Все значения типа данных определяют множество, на котором задаются операторы. Константа (или литерал в SDL) является 0-местным оператором; n-местный оператор (где $n \geq 1$) определяется своей сигнатурой, состоящей из имени оператора, типа результата и типов параметров. Действие оператора задается алгебраическими правилами — аксиомами. Совокупность множества значений типа данных, операторов со своими сигнатурами и аксиом образует алгебраическую систему. В любой SDL-системе существуют предварительно определенные типы (сорта в SDL), такие как целые и вещественные числа, символы, строки. Другие сорта, например различные виды массивов, генерируются по шаблонам с помощью уже определенных сортов, операторов и аксиом.

В SDL определены две синтаксические формы описания систем. Одна форма — текстовая, совпадающая с формой описания обычных языков программирования, другая — графическая, в которой система описывается в виде диаграмм, состоящих из графических символов. Текстовая форма описания является более богатой, графическая — более наглядной.

Система состоит из одного или нескольких *блоков*, соединенных между собой и с окружающей средой *каналами*, по которым передаются *сигналы*. Из окружения система получает внешние сигналы и в окружение возвращает ответы, запуск системы возможен только по сигналу извне. Каждый блок и канал в системе имеют уникальное имя.

Каналы бывают одно- или двусторонними. При каждом канале должны быть указаны имена всех сигналов, которые этот канал может передавать. При сигнале могут быть указаны имена сортов, таким образом, сигнал может нести с собой значения указанных сортов. Несколько сигналов могут быть объединены в один список, такому списку присваивается уникальное имя. Один сигнал может входить в разные списки.

Средствами SDL обеспечивается многоуровневое описание системы. По мере «спуска» от уровня к уровню либо детализируются описания уже имеющихся в системе объектов, либо вводятся новые объекты. Блок может быть разбит на более мелкие единицы — подблоки, которые, в свою очередь, сами являются блоками. Подблоки соединяются «новыми» каналами между собой и с рамкой блока. Будем называть «старыми» каналы, входя-

щие или выходящие из блока. В разбиении блока для «старых» каналов должны быть указаны имена «новых» каналов, которые подсоединены к старым. Это подсоединение происходит таким образом, что каждый сигнал, поступивший по «старому» каналу, должен передаваться только по одному «новому» каналу. «Новый» канал не может передавать сигнал, который не передавался по «старому» каналу. Аналогичными средствами осуществляется разбиение канала на подканалы и новые блоки.

На самом нижнем уровне иерархии определений блоки содержат *процессы*, являющиеся функциональными компонентами системы и определяющие ее поведение. Внутри блока *маршруты* связывают процессы между собой и с рамкой блока. При графическом изображении маршруты, которые связывают процессы с рамкой блока, должны либо начинаться от той точки, в которой в блок входит канал, либо оканчиваться в той точке, в которой из блока выходит канал. К одному входному/выходному каналу могут быть присоединены начальные/конечные точки нескольких маршрутов. Распределение сигналов по маршрутам, присоединенным к некоторому каналу, происходит таким образом, что по каждому маршруту должен передаваться хотя бы один сигнал, поступающий по каналу, а каждый сигнал, поступающий в канал, должен передаваться хотя бы по одному присоединенному маршруту. Кроме того, маршрут не может передавать сигнал, который не передавался по каналу.

Описание процесса состоит из трех частей: заголовка, декларативной части и тела процесса. Заголовок процесса содержит имя, число экземпляров и список формальных параметров с указанием их типов. Формальные параметры — это переменные, используемые в теле процесса. Число экземпляров — пара целых чисел, первое из которых указывает количество создаваемых экземпляров процессов в момент инициации системы, второе задает максимальное число экземпляров процессов, которые могут существовать одновременно в блоке.

Декларативная часть процесса содержит описания констант, типов, переменных, экспортируемых переменных, переменных обозревания, входных и выходных сигналов, списков сигналов, таймеров, процедур и макрокоманд. Тело процесса описывает действия, которые совершает процесс под влиянием входных сигналов. Процесс либо находится в одном из своих состояний, либо совершает переход. При этом каждое состояние должно иметь свое уникальное по отношению к этому процессу имя. Процесс имеет одну стартовую вершину, за которой следует переход. Этот переход совершается не под влиянием входного сигнала, а в результате возникновения процесса. Дальнейшие переходы из состояния в состояние возможны толь-

ко под воздействием входных сигналов, исключение составляют переходы, входящие в конструкцию «непрерывный сигнал».

Все сигналы, пришедшие в порт процесса, образуют в нем очередь. Процесс, находясь в одном из своих состояний, обращается к очереди сигналов. Если очередь пуста, то процесс ждет. Если не пуста, то для каждого состояния процесса однозначно указано, как должен реагировать процесс на любой сигнал, который стоит первым в очереди. Возможны три ситуации:

- указано, какой переход должен совершить процесс. Тогда процесс удаляет из очереди сигнал и начинает указанный переход;
- указано, что восприятие сигнала должно быть отложено до того, как процесс войдет в следующее состояние. Тогда процесс оставляет сигнал на своем месте в очереди и переходит к обработке следующего сигнала. Это действие называется сохранением сигнала и описывается с помощью ключевого слова ***save***;
- нет никакого указания на то, как должен реагировать процесс на сигнал. В этом случае сигнал удаляется из очереди, и процесс переходит к обработке следующего сигнала.

Описание каждого перехода процесса начинается с ключевого слова ***state*** и завершается одним из следующих ключевых слов: ***join***, ***stop*** либо ***endstate*** (если последнее опущено, то словом ***state***).

При переходе процесс может выполнить любой оператор языка Паскаль, а также такие действия, как принятие решения (***decision***); запрос на создание другого процесса; экспортно-импортную операцию; безусловный переход к другой последовательности действий; установку и сброс таймера.

Последовательность действий, выполняемых процессом во время перехода, может разветвляться. Для этого осуществляется проверка истинности некоторого выражения в конструкции ***decision***, которая состоит из условия и не менее двух вариантов. После каждого варианта указывается последовательность действий, которую должен выполнить процесс в данном случае. Если после нескольких вариантов должна быть выполнена одна и та же последовательность действий, то эти варианты объединяют вместе. Также возможен только один вариант ***else***, охватывающий все значения выражения (стоящего в условии), не учтенные во всех остальных вариантах.

Один экземпляр процесса может передавать различным экземплярам других процессов значение некоторой переменной с помощью экспортно-импортной операции. Для этого при описании переменной он должен указать, что ее значение в дальнейшем будет экспортироваться. Если другой процесс хочет получить экспортированное значение переменной, то он

должен в своей декларативной части указать, что намеревается импортировать это значение. Это делается с помощью операторов ***export / import*** или с помощью оператора ***view***.

Безусловный переход к другой последовательности действий осуществляется конструкцией ***join***, аналогичной оператору ***goto*** в языке Паскаль. С ее помощью указывается, что следующим должна выполняться последовательность действий этого же самого процесса, перед которым стоит указанная в конструкции ***join*** метка.

Назначение языка SDL — описание систем реального времени. Комплекс, реализующий систему, должен обладать средствами управления временем, например, часами, которые показывают абсолютное время в согласованных единицах времени. В языке предусмотрена стандартная функция ***now***, значением которой является значение текущего момента абсолютного времени, и имеется возможность описания таймеров и установки в них любого времени, указанного в принятых единицах. Реализуется эта возможность посредством оператора ***set***. Когда абсолютное время в системе станет равным установленному в таймере, в порт процесса будет установлен сигнал от таймера, имя которого совпадает с именем самого таймера. Чтобы процесс мог воспринять сигнал от таймера, этот сигнал должен быть указан в теле процесса в качестве входного. С момента установки таймера и до момента восприятия сигнала от таймера он считается активным. Перевод таймера в неактивное состояние («сброс таймера») осуществляется оператором ***reset***.

3. ЯЗЫК BASIC-REAL

Язык bREAL состоит из подъязыков выполнимых и логических спецификаций.

3.1. Представление систем и их свойств

Подъязык выполнимых спецификаций имеет иерархическую структуру, нижним уровнем которой является понятие процесса. Процессы могут объединяться в блоки, а блоки — строиться из подблоков. Для описания взаимодействия между такими двумя объектами, как процесс, блок или внешняя среда, используются каналы. Содержимым каналов являются сигналы, возможно, с параметрами. Каналы могут иметь различные структуры — очереди, магазины, мультимножества. Время жизни сигнала с параметрами в канале может быть ограничено числом запросов на чтение этого сигнала

и/или непосредственно интервалом времени (например, число запросов не более 5REQuests и время — в течение 6min34sec).

Процесс языка bREAL подобно процессу языка SDL описывает последовательность таких действий, как изменение переменных, чтение сигналов из каналов, запись сигналов в каналы, очистка каналов. В отличие от SDL с каждым действием в bREAL ассоциируется временной интервал, который задает продолжительность выполнения действия. Допускается использование оператора недетерминированного перехода. В языке bREAL предлагается новая концепция времени — асинхронные мультичасы, синхронизированные посредством линейных неравенств на их скорости хода. Множество поведений выполнимой спецификации можно ограничить посредством условий справедливости. В результате мы будем рассматривать только так называемые справедливые поведения, т.е. такие поведения, в которых каждое из условий справедливости реализуется бесконечно часто.

Языком представления свойств служит подъязык логических спецификаций bREAL, который значительно расширяет средства логики ветвящегося времени CTL[6] за счет использования временных интервалов и средств динамических логик [8]. Формулы этого языка строятся из предикатов с помощью булевских операций, кванторов по выделенным переменным, а также двух видов модальностей. Первый вид модальностей — это модальности по моментам времени из мультиинтервала, семантика которых означает «для всех моментов времени» и «существует момент времени». Второй вид модальностей — это модальности по возможным последовательностям действий (т.е. поведениям) выполнимых спецификаций, семантика которых означает «для всех поведений» и «существует поведение». Предикаты бывают четырех видов:

– отношения между значениями переменных и параметров сигналов;
– локаторы управляющих состояний в процессах;
– контроллеры пустоты и переполнения для каналов/шин;
– чекеры наличия/готовности сигналов в каналах/шинах.

3.2. Основы синтаксиса

Всякая спецификация (как выполнимая, так и логическая) носит иерархическую структуру и состоит из: заголовка, шкалы, контекста, схемы, подспецификаций.

Заголовок спецификации определяет ее имя и вид — блок или процесс, формула или предикат.

Шкала спецификации есть конечное множество линейных равенств и неравенств, в которых в качестве переменных выступают неинтерпретированные единицы времени, а в качестве коэффициентов — целые положительные числа, расширенные символами «сейчас» (NOW) и «бесконечность» (INF). Каждая единица времени, которая встречается в шкале, есть такт независимых часов, а сама шкала задает совокупность ограничений для синхронизации хода часов таким образом, что все равенства и неравенства шкалы выполняются.

Для формального описания семантики времени обычно применяется подход, близкий к модели фиктивного такта. Для границ интервала используются следующие обозначения: AFTER — для левой открытой, UNTIL — для правой открытой, FROM — для левой замкнутой, UPTO — для правой замкнутой.

Контекст спецификации есть конечное множество определений типов и описаний переменных и каналов. Каждый объект, описанный в контексте, имеет только одно описание, хотя идентификаторы у разных объектов могут совпадать.

В языке Basic-REAL есть как стандартные, так и нестандартные структуры каналов. Список стандартных структур каналов содержит ограниченные и неограниченные по емкости очереди, стеки и мультимножества (а значит и элементарный буфер в качестве частного случая одноэлементного мультимножества).

Всякий объект, объявляемый в контексте, должен содержать атрибут локации (location), определяющий место использования — в схеме самой спецификации (значение OWN) или в определенной подспецификации (имя подспецификации в качестве значения).

Схема выполнимой спецификации состоит из условий справедливости и диаграммы.

Диаграмма блока состоит из маршрутов, каждый из которых является маршрутом канала и связывает имя подблока с именем другого подблока или со средой. Диаграмма блока может иметь как графическую, так и линейную формы. Графическая форма диаграммы блока — это размеченный граф, вершины которого помечены именами подблоков или символом внешней среды, а дуги соответствуют каналам. В графическом синтаксисе диаграмм блоков каналы изображаются сплошными стрелками, а связанные ими имена процессов и среда заключаются в прямоугольные рамки. Линейная форма диаграммы блока — это просто перечисление в некотором порядке смежных вершин и связывающих их дуг описанного графа.

Диаграмма процесса в Basic-REAL представляет собой некоторое обобщение понятия программы и состоит из переходов, каждый из которых, в свою очередь, состоит из управляющего состояния («метки»), тела, временного интервала и (недетерминированного) скачка («перехода») на следующие управляющие состояния. Тело перехода определяет одно из следующих действий:

- прочитать определенный сигнал из входного канала и одновременно занести значения его параметров в определенные переменные;
- записать определенный сигнал в выходной канал и одновременно передать ему в качестве значений параметров значения определенных переменных;
- в соответствии с определенной программой изменить значения программных переменных процесса.

Любое из этих действий исполняется мгновенно, но исполнить его можно только в интервале времени, предусмотренном для этого перехода, т.е. когда с момента передачи управления состоянию, которое метит переход, пройдет время, укладывающееся во временной интервал перехода.

Диаграмма процесса может иметь как графическую, так и линейную формы. Графическая форма диаграммы процесса дублирует линейную форму диаграммы процесса и может отсутствовать в спецификации процесса, так как играет вспомогательную роль. Линейная форма диаграммы процесса — это полное описание всех переходов. Неформально говоря, процесс выполняется недетерминировано. С внешней средой процесс обменивается сигналами с параметрами через входные/выходные каналы. Это означает, что в канале, связанном с внешней средой, возможно возникновение и исчезновение сигнала.

Схема логической спецификации состоит из списка систем и диаграммы. В списке систем спецификации перечислены те выполнимые спецификации, свойства которых описывает эта логическая спецификация; ограничений на число выполнимых спецификаций в списке систем в языке Basic-REAL нет, но все они наследуют контекст логической спецификации. Системы позволяют ограничить рассмотрение истинности/ложности логической спецификации только на их конфигурациях.

Диаграмма предиката в Basic-REAL может быть

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

В отношениях разрешается использовать расширенные имена переменных и параметров сигналов выполнимых спецификаций в списке систем.

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 state

где **state** (состояние) — identifier (имя состояния).

Контроллеры могут быть как контроллерами пустоты, так и полноты каналов.

Контроллер пустоты канала:

EMP channel или **channel IS EMPTY**

где **channel** — identifier (имя канала).

Чекер присутствия сигнала в канале:

signal IN channel

где **signal** — identifier (имя сигнала), **channel** — identifier (имя канала).

Диаграмма формулы строится из имен предикатов с помощью пропозициональных связок (~ — «отрицание», & — «конъюнкция», * — «дизъюнкция», ‘=>’ — «импликация», ‘<=>’ — «эквивалентность»), кванторов над кванторными переменными («для любого / квантор всеобщности» AL или «существует / квантор существования» EX), модальностей по моментам времени («всегда» AT и «иногда» ET) и модальностей по поведениям («для всех» EACH и «для некоторых» SOME), круглых скобок. Пропозициональные связки, кванторы и скобки используются обычным образом. Динамические и временные модальности используются в качестве модального префикса только в следующих комбинациях:

EACH множество поведений AT мультиинтервал,

EACH множество поведений ET мультиинтервал,

SOME множество поведений AT мультиинтервал,

SOME множество поведений ET мультиинтервал,

где множество поведений задается в виде теоретико-множественного выражения над именами выполнимых спецификаций из списка систем при помощи операций объединения, пересечения и разности. Семантика задан-

ного таким образом множества поведений — это множество всех поведений, которое получается в результате подстановки вместо каждого имени выполнимой спецификации множества ее справедливых поведений и выполнения всех теоретико-множественных операций в соответствии с порядком скобок и приоритетов.

Например, конструкцию

EACH process1 \ process2 ET FROM 2min34sec UNTIL (x+y)*24sec}

можно прочитать следующим образом:

«для любого справедливого поведения process1, которое не является справедливым поведением process2 в некоторый момент времени, начиная с 2min34sec, но прежде чем истечет (x+y)*24sec».

Диаграмма формулы завершается точкой.

Подспецификации — это спецификации подблоков и подформул, имена которых используются в диаграмме самой спецификации.

В языке Basic-REAL иерархия многоуровневая — блок может содержать другие блоки любой вложенности, а формула — подформулы.

4. ТРАНСЛЯЦИЯ СТАТИЧЕСКОГО ПОДМНОЖЕСТВА ЯЗЫКА SDL В ЯЗЫК BASIC-REAL

4.1. Общая схема транслятора

Процесс трансляции проходит в две основных стадии (рис. 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 bREAL-программе

Генерация bREAL-текста (выходного файла), соответствующего внутреннему представлению. По заданным для всех конструкций языка bREAL-шаблонам происходит построение bREAL-текста.

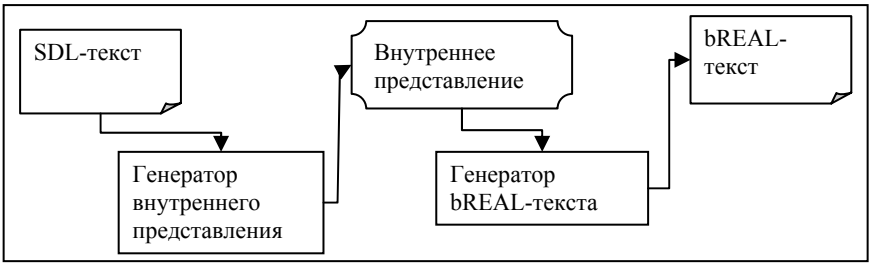

Рис. 1. Схема работы транслятора

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

4.2. Внутреннее представление входных спецификаций

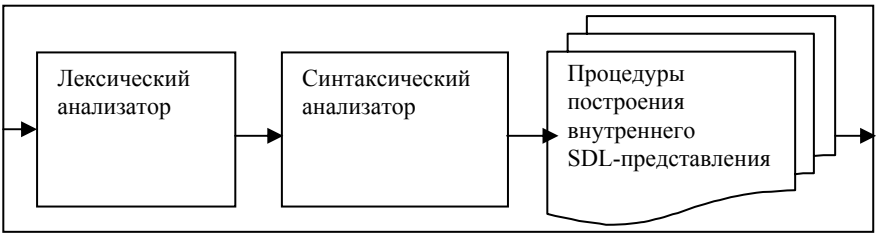

Рис. 2. Генерация внутреннего представления

Текст SDL-программы подается на вход лексическому анализатору, осуществляющему разбивку текста на лексемы. Синтаксический анализатор выделяет синтаксически законченные конструкции и вызывает процедуры генерации внутреннего представления. Затем происходит соединение каналов, создание списков сигналов, с которыми работают процессы, и ряд других действий, которые завершают построение внутреннего представления (рис. 2).

Во время выполнения этого этапа происходит проверка входного текста на его синтаксическую правильность. А также, частично, на его соответствие подмножеству языка SDL, которое может быть переведено в bREAL. В

процессе построения внутреннее представление максимально возможно приближается к построению программ на языке bREAL. Во время этого процесса происходит окончательная проверка представимости входного текста в выполнимых спецификациях языка bREAL.

Объект самого высокого уровня — это служебный “глобальный” блок, с точки зрения внутреннего представления он является обычным SDL-блоком и содержит предопределенные объекты с глобальной областью видимости, такими как, например, предопределенные типы данных.

Непосредственным потомком этого блока является *блок-система*, потомками которой являются блоки и далее — процессы. Таким образом иерархия внутреннего представления практически полностью совпадает с иерархией языка SDL. С точки зрения внутреннего представления процесс — это блок, имеющий граф состояний.

Атрибутами блока являются следующие компоненты.

- Список типов. Типы «Глобального блока» являются предопределенными, их определение задается транслятором до начала работы.
- Список переменных, констант.
- Список сигналов и список списков сигналов.
- Список каналов и временный список соединений.

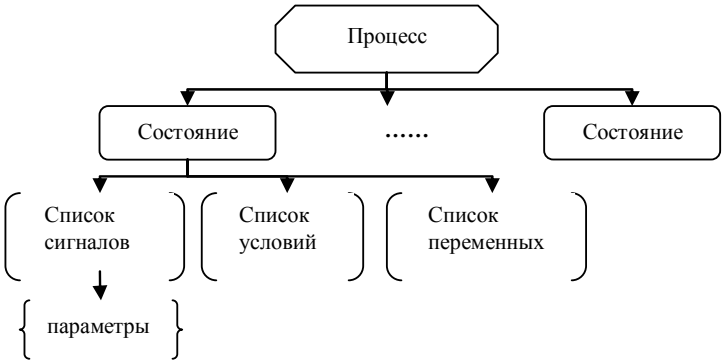

Рис. 3. Иерархия внутреннего представления процесса

Процессы (рис. 3) имеют информацию, в которой содержится граф состояний и список условий, если таковые имеются. При трансляции могут

использоваться ряд служебных свойств, как например — список сигналов, которые может получать процесс.

4.3. Построение bREAL-представления

Блоки/процессы, типы данных и переменные языка SDL имеют прямые аналоги в bREAL, поэтому эта часть процесса трансляции не представляет никакой сложности.

4.3.1. Каналы

Построение ***каналов*** происходит в два этапа: сначала создаются списки ***каналов*** и ***соединений***, в которых запоминается декларативная информация. Затем, после построения дерева блоков, происходит обработка ***соединений***: проверяется непротиворечивость указанной в них информации и связывания их с конкретными объектами внутреннего представления (блоками/процессами или каналами). Это необходимо, так как SDL допускает использование объектов до их определения.

Семантика каналов в SDL и bREAL имеет ряд существенных отличий: SDL-каналы могут разветвляться или сходиться на границе блоков, в то время как bREAL-каналы присоединяются строго один к одному. Двусторонние каналы преобразуются в два: с множеством сигналов, которые передаются в одном и в противоположном направлении. Возможность разветвления SDL-каналов преодолевается следующим образом: каналы, которые разделяются или соединятся в один, преобразуются таким образом, чтобы образовавшиеся в результате этой операции каналы были односторонними.

Пример 1. *Трансляция двусторонних каналов* (рис. 4).

Пусть в спецификации, написанной на языке SDL, существует канал, описанный следующим образом:

```
SIGNAL
  coin    (/* nominal */ integer),
  station (/* station */ integer),
SIGNALROUTE slot
  FROM Passenger TO Slotmachine WITH coin;
  FROM Slotmachine TO Passenger WITH station;
```

Тогда при трансляции он разобьется на два разнонаправленных канала, и его описание будет выглядеть так:

```
INN UNB QUEUE CHN slot
 FOR coin
 WITH PAR p1 OF integer.
INN UNB QUEUE CHN slot_inv
 FOR station
 WITH PAR p1 OF integer.
  FROM Passenger CHN slot  TO Slotmachine.
 FROM Slotmachine CHN slot_inv  TO Passenger.
```

Создается дополнительный канал slot_inv, противонаправленный каналу slot и способный передавать сигнал station.

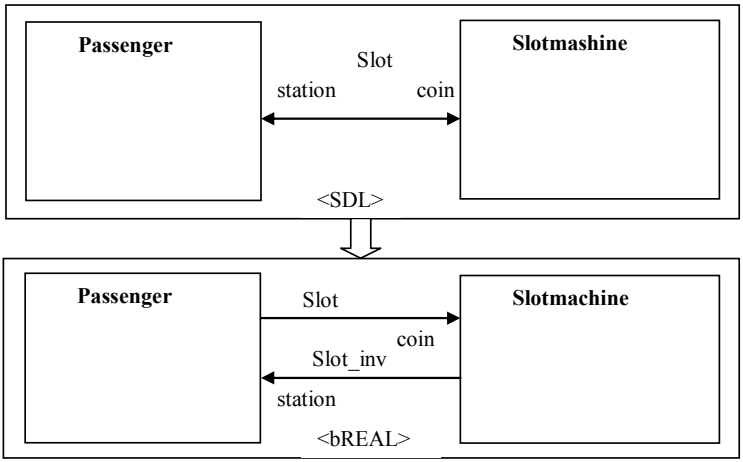

Рис. 4. Пример преобразования двусторонних каналов языка SDL в каналы языка bREAL

4.3.2. Чтение сигнала

Зафиксируем SDL-состояние. Для каждого оператора **INPUT** в этом состоянии создается bREAL-переход с тем же именем. Сработает то из них, сигнал которого придет первым. Если оператор **INPUT** имеет разрешающее условие, перед состоянием, содержащим оператор **READ**, вставляется bREAL-переход с оператором **WHEN condition**, с нулевым временем ожидания и переходом к чтению сигнала.

Дополнительно, для каждого, не сохраненного и не ожидаемого сигнала (сигнала для которого нет оператора **INPUT**, но который может находиться во входном порту процесса) добавляются альтернативы, читающие этот сигнал с нулевым временем ожидания. Таким образом, если первым сигналом в канале будет один из этих сигналов, он будет отброшен, как этого и требует семантика языка SDL.

Пример 2. *Трансляция SDL-состояния в bREAL-переходы, помеченные одним состоянием.*

Пусть есть SDL-состояние **state1** и процесс, которому оно принадлежит, может принимать сигналы **sig1**, **sig2**.

```
STATE state1;
  INPUT sig1;
  NEXTSTATE state2;
  INPUT sig2;
  NEXTSTATE state3;
ENDSTATE;
```

Тогда bREAL-код, реализующий это состояние, будет выглядеть следующим образом:

```
TRANSITION state1:
  READ sig1 from channel
  FROM NOW TO INF
JUMP state2.
TRANSITION  state1:
  READ sig2 from channel
  FROM NOW TO INF
JUMP state3.
```

4.3.3. Тело состояния

Последовательность операторов, образующих тело SDL-состояния, переводятся в один или несколько bREAL-переходов по следующим правилам.

- Каждый SDL-оператор **OUTPUT** переходит в соответствующий ему bREAL-оператор **WRITE**, помещаемый в новый переход.
- Тело оператора **TASK** (список присваиваний), по возможности, помещается в один bREAL-переход (это невозможно сделать, если выражения содержат операторы экспорта/импорта или обозревае-

мые переменные). При этом, если текущий bREAL-переход содержит оператор **READ** или **WRITE**, создается новый переход.

- Если SDL-выражения содержат операторы **EXPORT**, **IMPORT** или **VIEW**, перед bREAL-переходом, использующим это выражение, вставляется группа служебных переходов, реализующих эти конструкции (разд. 4.3.5).
- Если левая часть оператора присваивания является переменной, объявленной в секции **VIEWED** (обозреваемой), после этого выражения вставляются служебные переходы, осуществляющие действия, необходимые для отправки нового значения этой переменной адресатам (разд. 4.3.5).
- Для SDL-оператора **DECISION** создается группа альтернативных bREAL-переходов, помеченных одним состоянием (разд. 4.3.4).
- SDL-операторы SET, RESET переходят в bREAL-операторы WRITE (разд. 4.3.6).

4.3.4. Оператор DECISION

Операторы ветвления языков SDL и bREAL имеют значительные отличия. Кроме того, ограничение на единственность операторов в bREAL-переходе создает дополнительные проблемы при трансляции этих конструкций.

Условный оператор SDL имеет следующий вид:

```
DECISION условие;
  { {вариант:}+ переход }+
  [ELSE: переход]
ENDDECISION;
```

где

условие — выражение произвольного типа,

вариант — ограничение на множество значений ответа: константа (или множество констант) или один или несколько интервалов (открытых или закрытых),

переход — последовательность произвольных команд (за исключением оператора **INPUT**) языка SDL.

Условный оператор bREAL имеет вид:

```
WHEN выражение-условие программа
```

или

IF выражение-условие THEN программа [ELSE программа] FI

где

выражение-условие — обычное выражение, результат которого имеет логический тип,

программа — последовательность из одного или нескольких операторов bREAL.

Для каждого SDL-варианта создается собственный альтернативный переход — начало ветвления. Оно содержит оператор IF или WHEN с условием, полученным из SDL-условия. Выражение-условие для bREAL-операторов IF или WHEN строится следующим образом:

- если вариант — константа, то выражение-условие выглядит как *условие = константнта* ;
- если вариант — открытый интервал, то выражение-условие выглядит как *условие { < | > | <= | >= } граница*;
- если вариант — закрытый интервал, то выражение-условие выглядит как *(условие <= правая-граница) AND (условие >= левая граница)*;
- если список вариантов, то выражение-условие выглядит как *выражение-вариант { OR выражение-вариант }*+;
- выражение-вариант для варианта-ELSE строиться как отрицание дизюнкции всех вариантов оператора DECISION.

Кроме того создаются состояния и для конца условия (end_decision).

REAL-состояние начала ветвления выглядит так:

```
TRANSITION begin_decision_N
      EXE SKIP
      FROM NOW TO INF
JUMP state_name_T.

TRANSITION state_name_T
      WHEN условие вариант
      FROM NOW TO INF
JUMP state2.
```

Оно срабатывает, только если условие из оператора **WHEN** истинно и осуществляет переход к последовательности состояний, реализующих тело варианта. Эти состояния строятся по обычным правилам, описанным выше.

Кроме того, если **DECISION** не содержит варианта **ELSE**, будет добавлено служебное bREAL-состояние с **ELSE-**выражением. Если этого не сделать, процесс попал бы в ситуацию тупика, при условии, если выражение **условие** не удовлетворяло бы ни одному **варианту**.

Проиллюстрируем трансляцию оператора DECISION на примере.

Пример 3.*Трансляция оператора DECISION.*

Следующий фрагмент SDL-состояния содержит условие типа integer и 2 варианта:

```
STATE normal;
. . . . .
  DECISION r;
      ( = 0 ): NEXTSTATE normal;
      ( = 1 ): NEXTSTATE restore;
ENDDECISION;
```

Этот фрагмент будут реализовывать следующие bREAL-переходы.

```
TRANSITION begin_decision_1
EXE SKIP
FROM NOW TO INF
JUMP normal_1.

TRANSITION normal_1
WHEN (r=0)EXE SKIP
FROM NOW TO INF
JUMP normal.

TRANSITION normal_1
WHEN (r=1)EXE SKIP
FROM NOW TO INF
JUMP restore.
TRANSITION normal_1
WHEN (NOT ((r=0) OR (r=1)))
EXE SKIP
FROM NOW TO INF
JUMP end_decision_1.

TRANSITION end_decision_1
EXE SKIP
FROM NOW TO INF
JUMP end_of_process.
```

4.3.5. 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 bREAL нет, поэтому эта конструкция реализуется другими средствами.

В процесс, использующий оператор **VIEW**, добавляются служебные переменные, по одной для каждой переменной, которую этот процесс собирается обозревать. И между этим процессом и процессом, предоставляющим свои переменные для обозрения, проводиться служебный канал **view-channel** для передачи значений переменных.

В процессе, предоставляющем свои переменные для обозрения, после каждого выражения, в левой части которого стоит обозреваемая переменная, вставляются переходы, помещающие в канал **view-channel** сигнал с новым значением переменной.

В процессе-обозревателе перед переходом, использующим оператор **VIEW**, вставляется следующий код:

```
TRANSITION state_view:
  READ viewvar_имя_переменной(viewvar_имя_переменной)
FROM view_channel
  FROM NOW TO INF
JUMP state_view.

TRANSITION state_view:
  WHEN EMP(view_channel)
  EXE SKIP
  FROM NOW TO INF
JUMP state_with_view_expression.
```

А сам оператор **VIEW** заменяется в выражении на служебную переменную **viewvar_имя_переменной**.

Первое состояние этого фрагмента будет срабатывать до тех пор, пока в канале **view_channel** есть сигналы **viewvar_имя_переменной** (новые значения этой переменной). Если в канале не окажется требуемого сигнала, это будет означать, что переменная **viewvar_имя_переменной** содержит текущее значение обозреваемой переменной и сработает вторая альтернатива состояния.

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

IMPORT заменяется по правилам оператора **VIEW** таким же bREAL-шаблоном. А оператор **EXPORT** переходит в **WRITE**, аналогичный тому, что вставляется после присваиваний, изменяющих значение обозреваемой переменной.

4.3.6. Таймеры

Механизм таймеров реализуется следующим образом: в блок, которому принадлежит процесс, использующий таймеры, добавляются служебные процессы (по одному на каждый таймер), которые эмулируют работы SDL-таймера. Это делается следующим образом: SDL-оператор **SET** переводится в bREAL-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 переход 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.

Пример 4. *Служебный процесс-таймер.*

```
PR VAR delay OF integer;

TRANSITION inactive:
  READ set(delay) FROM input;
  FROM NOW TO INF
JUMP active;

TRANSITION inactive:
  READ reset FROM input;
  FROM NOW TO INF
JUMP inactive;

TRANSITION active:
  READ set(delay) FROM input;
```

```
  FROM NOW TO INF
JUMP active;

TRANSITION active:
  READ reset FROM input;
  FROM NOW TO INF
JUMP inactive;
TRANSITION active:
  EXE IF (delay <= 0) THEN SKIP ELSE ABRT
  FROM NOW TO INF
JUMP timeout;

TRANSITION active:
  EXE IF (delay > 0) THEN delay = delay - 1;
  SKIP;
  ELSE ABRT
  FROM 1 sec UPTO 1 sec;
JUMP active;

TRANSITION timeout:
  WRITE timeout INTO output;
  FROM NOW TO INF
JUMP inactive;
```

4.4. Генерация выходного bREAL-текста

Результатом работы этого этапа является REAL-текст, соответствующий входной SDL-программе.

Каждый объект внутреннего REAL-представления имеет функцию, обеспечивающую генерацию текста, в соответствии с некоторым шаблоном, заложенным в нее перед началом компиляции. Этот шаблон не зафиксирован и может быть легко изменен. Шаблон представляет собой некоторый текст, включающий в себя ряд спецсимволов, означающих какой-либо элемент внутреннего представления, который может присутствовать в этом объекте. Кроме того, спецсимволами являются команды форматирования текста, такие как перевод строки, структурный отступ и т.д.

Построение REAL-текста начинается вызовом функции MakeRealText для глобального блока (блока самого верхнего уровня), в процессе работы

этой функции произойдет рекурсивное развертывание всего внутреннего представления.

Пример 5. *Шаблон для bREAL-состояния.*

TRANSITION <name><\n>
<indent><action><\n>
<interval>
<\indent><\n>
JUMP <next>.<\n>

<name>	будет заменено именем конкретного состояния
<\n>	перевод строки
<indent> <\indent>.	означает, что следующий текст будет сдвинут вправо, до спецсимвола
<action>	будет вызвана функция построения REAL-команды этого состояния
<next>	заменится именем следующего состояния

Если <name> = “state1”, <action>=”EXE SKIP”, <interval>=”FROM NOW UPTO INF”, <next>=”state2”, то генератор построит текст:

TRANSITION state1
  EXE SKIP
  FROM NOW TO INF
JUMP state2.

5. КОНВЕРТОР ФОРМУЛ ЛОГИЧЕСКИХ СПЕЦИФИКАЦИЙ

5.1. Язык для описания логических SDL-спецификаций

В языке SDL нет средств для описания логических конструкций, подобных логическим спецификациям языка bREAL. Но существует необходимость описания свойств проверяемой SDL-спецификации. Для этого будет использоваться язык PL (Property Language), аналогичный подъязыку логических спецификаций bREAL и позволяющий представить свойства SDL-спецификации. Как и в подъязыке логических спецификаций bREAL, формулы этого языка строятся из предикатов с помощью булевских операций,

кванторов по выделенным переменным. Предикаты бывают тех же четырех видов (разд. 3).

5.1.1. Основы синтаксиса

Логическая спецификация, написанная на определенном выше языке, имеет иерархическую структуру и состоит из:

- заголовка,
- контекста,
- схемы,
- подспецификаций.

Заголовок спецификации определяет ее имя и вид — формула.

Контекст спецификации, так же как и в разд. 2.1, есть конечное множество определений типов и описаний переменных, каналов и маршрутов. Каждый объект, описанный в контексте, имеет только одно описание. Типы данных используются те, которые определены для языка SDL, и те, которые определил пользователь. Кроме того, в любом определении типа можно использовать ранее определенные в контексте типы.

Описания структуры сигналов, каналов и маршрутов те же, что и в языке SDL.

Схема логической спецификации также состоит из списка систем и диаграммы. В списке систем перечислены те выполнимые спецификации, свойства которых описывает эта логическая спецификация.

Диаграмма предиката совпадает с диаграммой предиката в подъязыке логических спецификаций с поправкой на правила синтаксиса и семантики языка SDL. Исключение составляют чекеры из языка bREAL, поскольку сигналы в SDL хранятся в очереди процесса, а не в канале. Диаграмма формулы также строится из имен предикатов с помощью пропозициональных связок, модальностей по моментам времени, модальностей по поведениям и круглых скобок. После диаграммы формулы должна стоять точка с запятой.

Подспецификации — это спецификации подформул, имена которых используются в диаграмме самой спецификации. Предикаты являются элементарными спецификациями и не имеют подспецификаций. Имя каждой подспецификации должно быть описано в контексте спецификации, и явно присутствовать в схеме спецификации. Аналогично, множество определений типов каждой подспецификации содержит все определения типов самой спецификации. Если какой-либо объект описан в контексте спецификации с указанием имени какой-либо подспецификации в качестве его ло-

кации, то этот же объект должен быть описан и в этой подспецификации с другой локацией.

5.2. Конвертация формул

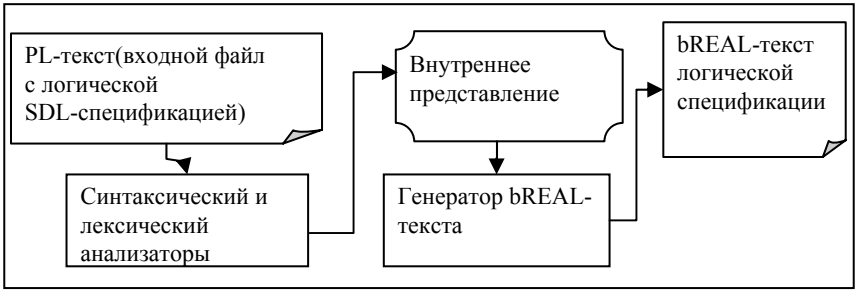

Рис. 5. Диаграмма работы конвертора

Синтаксический анализатор осуществляет синтаксический и лексический разбор спецификаций языка PL.

Синтаксический анализатор написан с помощью программ Flex(Lex) (генератор лексических анализаторов) версии 2.5.4 и Bison(Yacc) (генератор синтаксических анализаторов) версии 1.25.

На вход синтаксического анализатора подается имя файла с логической спецификацией, написанной на языке PL (.PSL-файл). Результатом выполнения модуля является следующее:

- в случае успешного завершения разбора — сообщение “Синтаксический разбор завершен успешно” и внутреннее представление спецификации, которое подается на вход модуля конвертации;
- в случае наличия синтаксической ошибки в спецификации — сообщение вида:
  “Строка N. Синтаксическая ошибка в или перед Name”, где
  N — номер строки, содержащей ошибку,
  Name — имя соответствующей нераспознанной лексемы.

Модуль конвертации получает внутреннее представление PL-спецификации (логической SDL-спецификации) и преобразует его в текст логической bREAL-спецификации, которая используется в модуле проверки моделей подсистемы верификации (рис. 5).

Процесс конвертации проходит в два этапа.

1. Генерация внутреннего представления. Во время этого этапа происходит проверка входного текста, является лм он синтаксически правильной логической спецификацией. Результатом является внутреннее представление.
2. Генерация bREAL-текста (выходной логической спецификации).

Текст PL-спецификации идет на вход лексическому анализатору, осуществляющему разбивку текста на лексемы. Синтаксический анализатор выделяет синтаксически законченные конструкции и вызывает процедуры генерации внутреннего представления. Во время выполнения этого этапа происходит проверка входного текста на его синтаксическую корректность. А также, частично, на его соответствие подмножеству языка SDL, которое может быть переведено в bREAL.

Заголовки формул, схема логической спецификации, описание предикатов имеют прямые аналоги в языке bREAL.

Описания каналов и маршрутов конвертируются так же, как и в разд. 4.3.1.

Предикат с чекером на проверку нахождения сигнала в очереди процесса конвертируется следующим образом. Поскольку в языке bREAL сигнал до его получения находится в канале, а не в очереди процесса, как в языке SDL, то происходит следующая проверка.

1. Определяется, какие каналы или маршруты способны передавать сигнал.
2. Для каждого из полученных каналов или маршрутов создается новый предикат, в теле которого описан чекер для сигнала и соответствующего канала.
3. В тело формулы вместо одного исходного предиката добавляется дизъюнкция всех новых предикатов.

Это значит, что если в процессе выполнения программы на языке bREAL требуемый сигнал появится в одном из каналов, то для языка SDL этот сигнал будет находится в порту процесса, что и требуется.

Конвертор формул логических спецификаций является консольным приложением. Для запуска приложения необходимо в консольной строке набрать:

Converter.exe имя_psl_файла.psl *имя_prl_файла.prl* ,

где имя_psl_файла.psl — имя файла, в котором содержится логическая спецификация SDL-программы, а необязательный параметр *имя_prl_файла.prl* — имя выходного файла, где будет содержаться логическая bREAL-

спецификация. Если этот параметр отсутствует, то конвертор генерирует файл out.prl.

6. СИСТЕМА МОДЕЛИРОВАНИЯ ВЫПОЛНИМЫХ BREAL-СПЕЦИФИКАЦИЙ

6.1. Архитектура системы

Система моделирования выполнимых спецификаций — экспериментальная система, предназначенная для моделирования и тестирования выполнимых спецификаций языка Basic-REAL.

Архитектура системы представлена на рис. 8. Система состоит из следующих модулей:

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

Процесс верификации происходит следующим образом. Пользователь посылает выполнимую спецификацию на синтаксический анализатор, который, сделав синтаксический и лексический разборы, 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 моделирования 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 моделирования 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 моделирования 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 Basic-REAL. Выполнимая спецификация располагается в файле с расширением ".RL", история выполнения записывается в файл "HISTORY.TXT".

6.2. Компоненты системы

Рассмотрим основные компоненты системы (рис. 6).

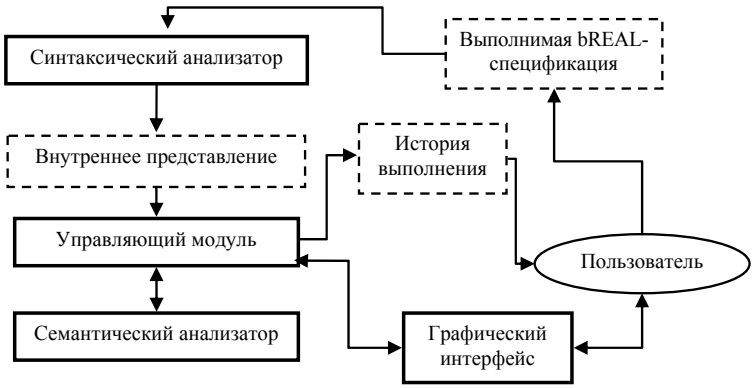

Рис. 6. Структура системы моделирования и тестирования выполнимых спецификаций Basic-REAL

6.2.1. Управляющий модуль

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.

6.2.2. Синтаксический и семантический анализаторы

Синтаксический анализатор осуществляет синтаксический и лексический разборы выполнимых спецификаций языка bREAL.

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 Basic-REAL. Так как спецификация состоит из блоков, процессов, подблоков, то делается разбор и выполнение семантических правил по уровням.

Для 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.

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

6.2.3. Графический интерфейс

Наглядное представление всего, что происходит при моделировании выполнения выполнимой спецификации языка Basic-REAL, осуществляется посредством графического интерфейса.

Графический интерфейс имеет удобный пользовательский интерфейс. Вид графического интерфейса отражают рисунки, приведенные ниже.

На рис. 7 представлен вид Блока. По заголовку диалогового окна “Block: passenger_and_machine” можно определить, что данный диалог отображает структуру Блока “passenger_and_machine”. Мы видим четыре спи-

ска, которые называются “Block&Processes”, “Variables”, “Channels” и “Channel content”. Список “Block&Processes” содержит дерево, описывающее систему. Дерево состоит из имен блоков и процессов, которые принадлежат этим блокам. В списке “Variables” представлены переменные данного блока. Этот список отображает имя переменной (поле “Name”), ее тип (поле “Type”) и значение (поле “Value”). Если переменная была только объявлена, но не проинициализирована, то в поле “Value” будет сообщение “Not initialising”.

Следующий список “Channels” отображает имя канала (поле “Name”), имена процессов, подблоков (или ENV — если внешняя среда), которые взаимодействуют посредством этого канала (поле “From” — имя процесса или блока, для которого данный канал является выходным, поле “To” — имя процесса или блока, для которого данный канал является входным). По сути, список “Channels” является представлением диаграммы блока (см. синтаксис языка Basic-REAL). Окно «Channel Content» отображает имена сигналов в канале, а в скобках — параметры соответствующего сигнала. Если выбранный канал пуст, то в окне “Channel Content” будет только сообщение “No any Signal/Channel empty”.

Если выбрать какое-нибудь имя процесса из дерева “Block&Processes”, то появляется возможность отследить все переходы выбранного процесса (рис. 8). Появляется окно “List of States”, где перечисляются имена состояний, активное выделяется красным цветом. В окне “List of Variables” представлен список, состоящий из имен всех переменных выбранного процесса. Как и в случае с блоком, список отображает имя переменной, ее тип и значение. В окне “List of transitions” отображен список всех переходов данного процесса. Кнопка “Transition Choice”, которая задает тип недетерминированного выбора: “Random” — случайным образом, “User” — пользователь сам осуществляет выбор.

Кнопка “Step” осуществляет пошаговое выполнение моделирования, “Run/Stop” — автоматическое.

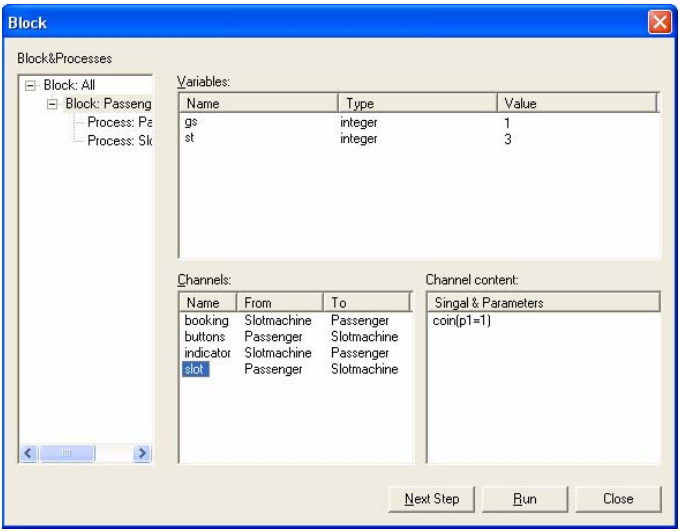

Рис. 7. Состояние блока

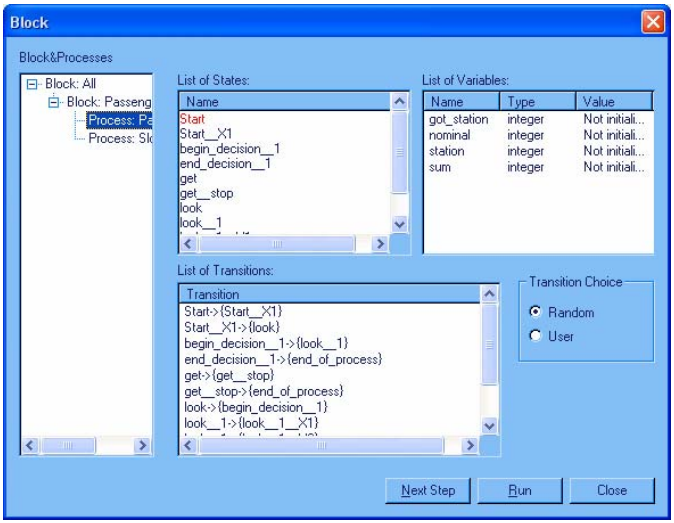

Рис. 8. Состояние процесса

7. СИСТЕМА ВЕРИФИКАЦИИ ВЫПОЛНИМЫХ СПЕЦИФИКАЦИЙ

7.1. Теоретический базис

7.1.1. Синтаксис и семантика логических спецификаций

Синтаксис

Пусть AP — множество предикатов. Тогда *диаграмма формулы* определяется следующим образом.

1. Если p — предикат, то p — диаграмма формулы.
2. Если F_1 и F_2 — диаграммы формул, то $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$, $F_1 \Leftrightarrow F_2$, EACH AT F_1, EACH ET F_1, SOME AT F_1, SOME ET F_1 — диаграммы формул.
3. Других диаграмм формул нет.

Семантика

Рассмотрим модель $M = (CF, R, P)$, где

CF — множество конфигураций выполнимой спецификации;

R — тотальное бинарное отношение на CF :

$$R \subseteq CF \times CF, \quad \forall\, CNF_1 \in CF \;\; \exists\; CNF_2 \in CF \;\; : \;\; (CNF_1, CNF_2) \in R;$$

P — отображение в 2^{AP} , которое сопоставляет каждой формуле множество конфигураций, где она истинна.

Поведение спецификации — счетная последовательность конфигураций $(CNF_0, CNF_1, CNF_2, ...)$, таких, что

$$\forall\; i = 0,1,2,3,... \;\; (CNF_i,\; CNF_{i+1}) \in R\,.$$

Поведение выполнимой спецификации называется справедливым тогда и только тогда, когда каждое из условий справедливости выполняется бесконечно часто в конфигурациях, которые встречаются в этом поведении.

Для любой конфигурации CNF и любой логической спецификации SPC тот факт, что конфигурация принадлежит множеству истинности логической спецификации, обозначается $CNF \models SPC$, а его отрицание — $CNF \neq SPC$. Чтобы сократить описание семантики логических спецификаций, зафиксируем конфигурацию $CNF = (V, C, S)$. Отношение $CNF \models$ определяется индукцией по структуре диаграммы формулы логической спецификации SPC .

Базис индукции: SPC — предикат.

Если SPC — отношение, то его диаграмма (в префиксной записи) имеет вид $\mathrm{Rel}(t_1,...,t_2)$, где Rel — символ отношения, а $t_1,...,t_2$ — выражения, построенные из символов операций, расширенных имен переменных и параметров. Тогда $CNF \models SPC \Leftrightarrow$ значения выражений $t_1,...,t_2$ в конфигурации CNF удовлетворяют отношению, соответствующему Rel.

Если SPC — локатор, то его диаграмма имеет вид AT state, где state — некоторое состояние или расширенное имя состояния. Тогда $CNF \models \mathrm{AT\ state} \Leftrightarrow S(\mathrm{state}) = True$.

Если SPC — контроллер, то его диаграмма имеет вид EMP channel или OVF channel, где channel — некоторый канал. Тогда, соответственно, $CNF \models$ EMP channel $\Leftrightarrow$ C(channel) — пустой канал, $CNF \models$ OVF channel $\Leftrightarrow$ C(channel) — полный канал.

Если SPC — чекер, то его диаграмма имеет вид signal IN channel , где signal — некоторый сигнал, а channel — некоторый канал. Тогда $CNF \models$ signal IN channel $\Leftrightarrow$ C(channel) содержит сигнал signal с некоторым значением параметра.

Шаг индукции.

Если диаграмма спецификации SPC есть пропозициональная комбинация, то ее значение определяется естественным образом.

Если диаграмма имеет вид $\neg A$, где A — диаграмма формулы, то $CNF \models SPC \Leftrightarrow CNF \neq SPA$, где SPA отличается от SPC только диаграммой, которая есть A.

Если диаграмма спецификации SPC есть $\forall$ variable A (или $\exists$ variable A), где variable — кванторная переменная, а A — диаграмма формулы, то $CNF \models SPC \Leftrightarrow$ для каждой (соответственно, некоторой) конфигурации CNF', отличающейся от CNF не более, чем интерпретацией переменной variable, выполняется $CNF' \models SPA$, где SPA отличается от SPC только диаграммой, которая есть A.

Если диаграмма спецификации SPC есть M1 M2 A, где M1 — модальность EACH или SOME, M2 — модальность AT или ET, а A — диаграмма формулы, то пусть SYS — выполнимая спецификация (блок или процесс) из списка систем спецификации SPC. Тогда модальность
EACH означает «для всех справедливых поведений» SYS,

SOME означает «существует справедливое поведение» SYS,
AT означает «для всех следующих конфигураций в поведении»,
ET означает «для некоторой следующей конфигурации в поведении».

Ввиду особой значимости для спецификации свойств протоколов опишем следующие диаграммы "$A \Rightarrow$ EACH AT B", "$A \Rightarrow$ EACH ET B", "$A \Rightarrow$ SOME AT B", "$CNF \models (A \Rightarrow$ SOME ET $B)$". В соответствии с общим определением (и слегка упростив терминологию), их семантикой будет:

$CNF \models (A \Rightarrow$ EACH AT $B) \Leftrightarrow$ если $CNF \models A$, то при любом справедливом поведении блока, которое начинается с CNF, для любой конфигурации CNF' из этого поведения верно $CNF' \models B$;

$CNF \models (A \Rightarrow$ EACH ET $B) \Leftrightarrow$ если $CNF \models A$, то при любом справедливом поведении блока, которое начинается с CNF, для некоторой конфигурации CNF' из этого поведения верно $CNF' \models B$;

$CNF \models (A \Rightarrow$ SOME AT $B) \Leftrightarrow$ если $CNF \models A$, то при некотором справедливом поведении блока, которое начинается с CNF, для любой конфигурации CNF' из этого поведения верно $CNF' \models B$;

$CNF \models (A \Rightarrow$ SOME ET $B) \Leftrightarrow$ если $CNF \models A$, то при некотором справедливом поведении блока, которое начинается с CNF, для некоторой конфигурации CNF' из этого поведения верно $CNF' \models B$.

7.1.2. Синтаксис и семантика μ-исчисления

Кратко представим стандартный подход к проверке моделей формул μ-исчисления.

Системы Переходов

Системой Крипке T назовем тройку (S, Act, →), где S — множество состояний, Act — множество действий, а → — подмножество множества $S \times Act \times S$. Если $<S_1, a, S_2>$ принадлежит →, то будем писать $S_1 \xrightarrow{a} S_2$.

Синтаксис μ-исчисления

Пусть мы имеем счетные непересекающиеся алфавиты пропозициональных констант $C = \{P, Q, \ldots\}$, пропозициональных переменных $V = \{X, Y, \ldots\}$ и символов действий $Act = \{a, b, \ldots\}$.

Формулами μ-исчисления назовем следующие выражения:

P, где P — пропозициональная константа;

X, где X — пропозициональная переменная.

Если F, F_1, F_2 — уже построенные формулы μ-исчисления, то формулами μ-исчисления также являются:

$\neg F_1$, $F_1 \vee F_2$,

$[a]F$, $<a>F$, $\mu x.F$, $\nu x.F$.

Других формул нет.

Семантика μ-исчисления

Рассмотрим модель $M = (S, I)$, где S — множество состояний, а интерпретация $I = (P, Q)$, где P — множество конфигураций, в которых истинна логическая спецификация, $Q : Act \to 2^{S \times S}$ — интерпретация действий.

$M, S \models F$ означает, что в модели M в состоянии S выполняется формула F. Если модель ясна из контекста, то можно писать $S \models F$.

$S \models F \Leftrightarrow S \in H(F)$, где семантическая функция $H(F) \to S$ определяется следующим образом:

$H(p) = P(p)$.

$H(\neg F) = S \backslash H(F)$.

$H(F_1 \wedge F_2) = H(F_1) \cap H(F_2)$

$H(F_1 \vee F_2) = H(F_1) \cup H(F_2)$

$H(\mu x.F(x)) = \cap\{L \subseteq CF \mid H(F(x))[x \leftarrow L] = L\}$

$H(\nu x.F(x)) = \cup\{L \subseteq CF \mid H(F(x))[x \leftarrow L] = L\}$

Причем $H(x)[x \leftarrow L] = L$ а $H(p)[x \leftarrow L] = H(p)$.

Проблема проверки моделей для μ-исчисления состоит в следующем. Имея формулу F и структуру Крипке T, мы должны найти состояния $s \in S$, на которых формула F верна. В общем случае проблема проверки моделей состоит в проверке данной формулы в некоторых состояниях (так называемая локальная проблема проверки моделей) и в поиске множества состояний, в которых данная формула верна (так называемая глобальная проблема проверки моделей).

7.2. Архитектура системы

Система верификации выполнимых спецификаций — экспериментальная система, предназначенная для верификации выполнимых спецификаций языка bREAL.

Архитектура системы представлена на приведенном ниже рис. 9. Система состоит из следующих модулей:

- управляющий модуль,

- синтаксический анализатор,
- конструктор моделей,
- модуль перевода логических спецификаций в μ-исчисление,
- модуль проверки моделей.

Система верификации по bREAL-спецификации, состоящей из выполнимой bREAL-спецификации и логической спецификации, позволяет провести верификацию свойств специфицируемой системы. Для верификации предполагается использовать метод проверки моделей (model checking method), а именно, раннее разработанный В.Е. Козюрой модуль проверки моделей на μ-формулах. В модуле проверки моделей осуществляется проверка формул программных логик с неподвижными точками на конечных системах переходов. Неформально говоря, такие системы переходов генерируются по операционной семантике программных систем, а формулы специфицируют их свойства, так что можно говорить о верификации свойств соответствующих программных систем.

В системе реализовано два варианта применения метода проверки моделей для верификации bREAL-спецификаций:

- верификация спецификаций с конечным числом переходов модели, не ограниченных по времени;
- верификация спецификаций с бесконечным числом переходов модели, ограниченных временем.

Первый случай есть классически определенная задача метода проверки моделей. Во втором случае, когда делается проверка зависимого от времени свойства (например, что ничего плохого не случится, скажем, в первые 10 часов), то модель можно обрезать, когда часы спецификации исчерпают лимит времени. Это делается определением того, что специальная переменная “time” достигает заданного значения.

Для того чтобы можно было использовать модуль проверки моделей для bREAL-спецификаций, система верификации по выполнимой спецификации строит модель, переводит логическую спецификацию в μ-формулу и получает значения констант. Далее все эти данные передаются модулю проверки моделей, который проводит верификацию и выдает результат — состояния, в которых данная формула верна, либо ALL_STATES, если формула верна во всех состояниях.

Более подробно процесс верификации происходит следующим образом. Полученная от пользователя bREAL-спецификация передается синтаксическому анализатору, который, сделав ее синтаксический и лексический разборы, переводит 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. Модель передается модулю проверки моделей, а список состояний — модулю перевода.

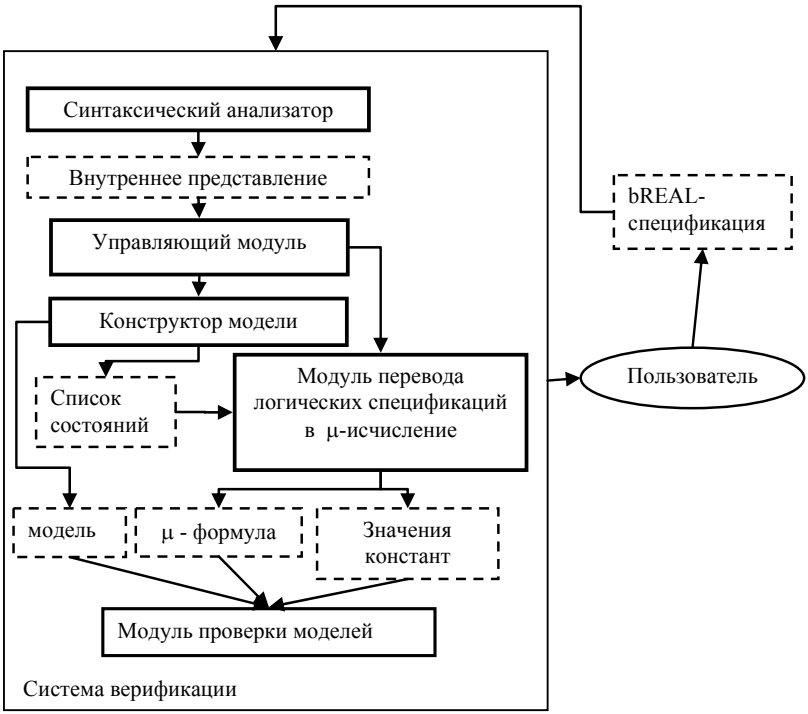

Рис. 9. Структура системы верификации Basic-REAL спецификаций

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 верификации выполнимых спецификаций реализована на языке C++ в среде Microsoft Visual Studio 6.0 для Windows. Объем исходного кода

системы — 1100 Кбайт, исполняемого — 350 Кбайт. Система работает под следующими операционными системами: Windows 95/98/NT/2000/XP.

7.3. Компоненты системы

7.3.1. Управляющий модуль

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.

7.3.2. Синтаксический анализатор

Синтаксический анализатор является несколько модернизированным синтаксическим анализатором системы моделирования выполнимой спецификации языка Basic-REAL. А именно, добавлен синтаксический и лексический разборы логических спецификаций.

На вход синтаксического анализатора подается имя файла с выполнимой спецификацией Basic-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 Basic-REAL (.RLO-файл), которое подается на вход управляющего модуля;
- в случае наличия синтаксической ошибки в спецификации — сообщение 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.

7.3.3. Конструктор моделей

Данный модуль, получив от управляющего модуля внутреннее представление выполнимой спецификации, осуществляет выполнение операционной семантики и конструирует модель. Действия, которые делает конструктор моделей, подобны действиям семантического анализатора из системы моделирования и тестирования выполнимых спецификаций Basic-REAL.

Выходами данного модуля являются:

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

Модель — перечисление переходов системы.

Файл “MODEL.TXT” имеет следующий формат:

12 act1 23
13 act1 25
15 act2 3
. . .

где состояния обозначаются только числами.

Файл “CHANGE.CNG” представляет из себя таблицу, в которой каждая строчка описывает состояние и имеет следующий формат:

7.3.4. Модуль перевода логических спецификаций в μ-исчисление

Данный модуль является реализацией алгоритма перевода логических спецификаций в μ-исчисление. Полученная после синтаксического разбора логическая спецификация передается Управляющим модулем на вход данного модуля. Также для осуществления перевода необходимы данные о состояниях — список состояний (файл CHANGE.CNG, содержащий полный пронумерованный список всех состояний модели). Модуль осуществляет перевод логических спецификаций в μ-исчисление и имеет следующие выходы:

- μ-формула — файл FORMULA.TXT;
- значения констант — файл CON_VAL.TXT.

Форматы этих файлов описаны в разд. 7.3.5.

7.3.5. Модуль проверки моделей

Модуль проверки моделей проводит верификацию систем, использует метод проверки моделей (model checking method).

Подсистема проверки моделей состоит из синтаксического анализатора формул μ-исчисления, блока вычисления предикатов и собственно алгоритмической части проверки формулы на модели. Синтаксический анализатор дает на выходе таблицу разбора, необходимую для проведения прямого алгоритма проверки формулы μ-исчисления на данной модели. Входная модель, получаемая из блока генерации модели, используется без изменений. Для вычисления предикатов используется описание мест, полученное на этапе генерации модели.

Входными данными являются:

- модель (MODEL.TXT);
- μ-формула (FORMULA.TXT);
- значения констант (CON_VAL.TXT).

Файл FORMULA.TXT содержит μ-формулу, в соответствии со следующими правилами.

- В конце формулы должна стоять точка.
- После модальности нужно поставить '}' . Например: <a>(x & p)} или <a>x}.
- Должна быть внешняя неподвижная точка.
- Все под неподвижной точкой нужно брать в скобки mX().
- Обозначения : m : m ; n : v.

Файл со значениями констант (CON_VAL.TXT) пишется перечислением (с новой строки в скобках) состояний, где данная константа верна.

Например:

p (1 2 3 4 5 6 7 8 9 10)
q (1 2 3)
t (100)

8. ПРИМЕР СПЕЦИФИКАЦИИ ПРОТОКОЛА СИСТЕМЫ “КАССА-ПАССАЖИР”

8.1. Описание протокола

Рассмотрим следующий пример: протокол обслуживания “хорошего” пассажира автоматической кассой. Автоматическая касса хранит полученные от пассажиров деньги и имеет следующие комплектующие

- клавиатура с названиями станций, с кнопкой возврата и выдачи билета;
- щель для монет, которые находятся в обращении;
- электронный индикатор для высвечивания суммы;
- лоток для возврата монет;
- окно выдачи билетов.

“Хороший” пассажир знает нужную ему станцию, имеет достаточно денег и может:

- нажимать кнопки клавиатуры;
- опускать монеты в щель;
- видеть показания индикатора;
- забирать из лотка монеты;
- получить в окне билет.

Неформально протокол обслуживания “хорошего” пассажира автоматической кассой состоит в следующем. Сеанс начинается с того, что пассажир нажимает кнопку клавиатуры с нужной ему станцией. Получив от пассажира название станции, касса высвечивает на индикаторе цену билета. Далее начинается следующий цикл: пассажир смотрит на индикатор и, если на нем светится ненулевая сумма, выбирает монету из кошелька и опускает ее в щель-монетоприемник, а касса, получив монету, вычитает ее номинал из суммы денег, которую еще надо получить от пассажира, и высвечивает новое значение суммы на индикаторе. Пассажир может прервать этот цикл в любой момент, нажав на клавиатуре кнопку возврата, и касса в таком случае должна вернуть всю полученную от пассажира сумму через лоток. Однако “хороший” пассажир не пользуется этой возможностью, а, когда на индикаторе высвечена нулевая сумма, он нажимает на клавиатуре кнопку выдачи билета. Касса, получив такую команду с клавиатуры, печатает билет с указанием станции, а пассажир забирает этот билет в билетном окне. После того как касса вернула всю сумму пассажиру или выдала билет, она завершает сеанс обслуживания пассажира. После того как пассажир забрал

всю возвращенную сумму из лотка или получил билет, он завершает сеанс со своей стороны.

Свойство протокола, которое мы хотим специфицировать, неформально говоря, состоит в следующем: при соблюдении описанного протокола касса выдаст билет пассажиру до нужной ему станции.

8.2. Спецификация на языке SDL

Опишем протокол на упомянутом ранее языке спецификаций SDL.

```
SYSTEM All;
 BLOCK Passenger_Slotmachine;
  SIGNAL
   coin    (/* nominal */ integer),
   light   (/* sum */     integer),
   ticket  (/* station */ integer),
   station (/* station */ integer),
   request;
  SIGNALROUTE buttons
   FROM Passenger TO Slotmachine WITH station, request;
  SIGNALROUTE slot
   FROM Passenger TO Slotmachine WITH coin;
  SIGNALROUTE indicator
   FROM Slotmachine TO Passenger WITH light;
  SIGNALROUTE booking
   FROM Slotmachine TO Passenger WITH ticket;

  PROCESS Passenger;
   DCL
    sum, nominal  integer,
    station       integer,
    got_station       integer;
   START;
    TASK station     := 3,
         nominal     := 1;
    OUTPUT station(station);
   NEXTSTATE look;
   STATE look;
    INPUT light(sum);
     DECISION sum;
```

```
      (<=0):
        OUTPUT request;
        NEXTSTATE get;
      ELSE:
        OUTPUT coin(nominal);
        NEXTSTATE look;
    ENDDECISION;
  ENDSTATE;
  STATE get;
    INPUT ticket(got_station);
    STOP;
  ENDSTATE;
ENDPROCESS;

PROCESS Slotmachine;
  NEWTYPE price
    array (integer, integer)
  ENDNEWTYPE;
  DCL
    sum      integer,
    nominal  integer,
    station  integer,
    expenses price;
  START;
    TASK
      expenses(1) := 10,
      expenses(2) := 15,
      expenses(3) := 20;
  NEXTSTATE get_station;
  STATE get_station;
    INPUT station(station);
    TASK sum := expense(station);
    OUTPUT light(sum);
    NEXTSTATE proc;
  STATE proc;
    INPUT coin(nominal);
      TASK sum := sum - nominal;
      OUTPUT light(sum);
      NEXTSTATE proc;
```

```
      INPUT request;
        DECISION sum;
          (>0):
            OUTPUT light(sum);
            NEXTSTATE proc;
          ELSE:
            OUTPUT ticket(station);
            NEXTSTATE get_station;          ENDDECISION;
    ENDSTATE;

  ENDPROCESS;
ENDBLOCK;
ENDSYSTEM;
```

Спецификации процессов Slotmachine и Passenger в графической форме даны на рис. 10 и 11.

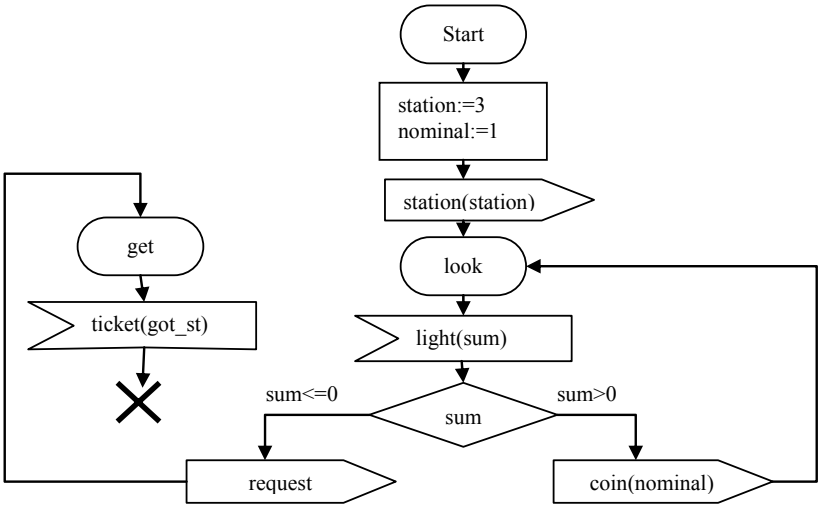

Рис. 10. Процесс Passenger

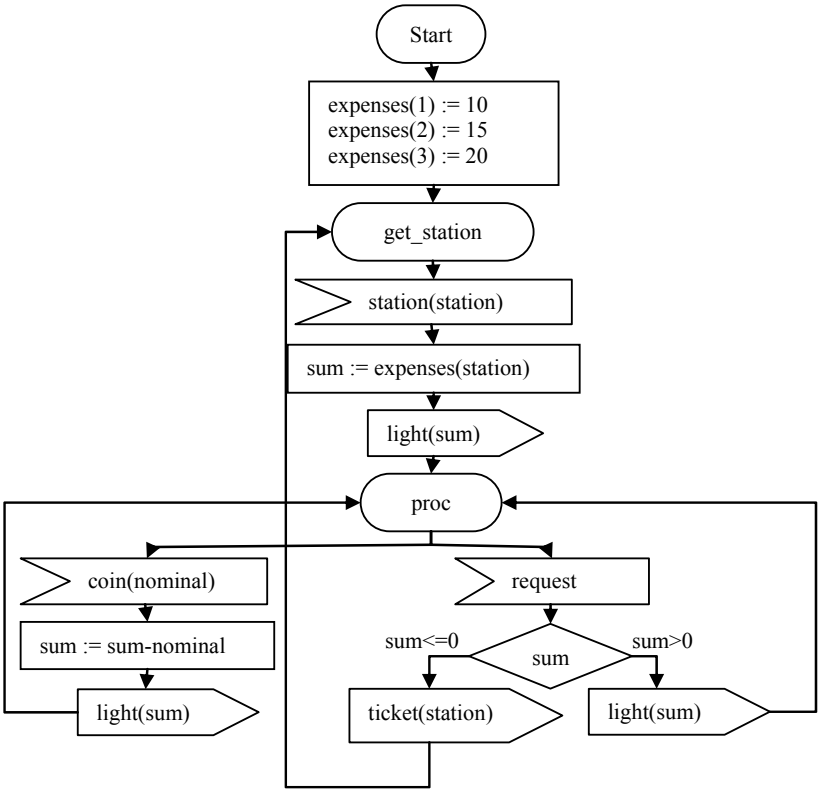

Рис. 11. Процесс Slotmachine

8.3. Выполнимая спецификация на языке bREAL

Приведем спецификацию на языке bREAL, полученную в результате трансляции.

```
/* Sdl to bREAL Translater*/
All: BLOCK
  Passenger_Slotmachine: BLOCK
  INN UNB QUEUE CHN booking
   FOR ticket
```

```
 WITH PAR p1 OF integer.
INN UNB QUEUE CHN indicator
 FOR light
 WITH PAR p1 OF integer.
INN UNB QUEUE CHN slot
 FOR coin
 WITH PAR p1 OF integer.
INN UNB QUEUE CHN buttons
  FOR request;
  FOR station
  WITH PAR p1 OF integer.

 FROM Slotmachine CHN booking  TO Passenger.
 FROM Slotmachine CHN indicator  TO Passenger.
 FROM Passenger CHN slot  TO Slotmachine.
 FROM Passenger CHN buttons  TO Slotmachine.

Passenger: PROCESS
 PR VAR got_station OF integer.
 PR VAR station OF integer.
 PR VAR sum OF integer.
 PR VAR nominal OF integer.

 TRANSITION Start
 EXE station:=3;
 nominal:=1;
 FROM NOW TO INF
 JUMP Start__X1.

 TRANSITION Start__X1
 WRITE station(station) INTO buttons
 FROM NOW TO INF
 JUMP look.

 TRANSITION look
 READ light(sum) FROM indicator
 FROM NOW TO INF
 JUMP begin_decision__1.
```

```
TRANSITION begin_decision__1
EXE SKIP
FROM NOW TO INF
JUMP look__1.

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

TRANSITION look__1__X1
WRITE request INTO buttons
FROM NOW TO INF
JUMP get.

TRANSITION look__1
WHEN (NOT (sum<=0))EXE SKIP
FROM NOW TO INF
JUMP look__1__X2.

TRANSITION look__1__X2
WRITE coin(nominal) INTO slot
FROM NOW TO INF
JUMP look.

TRANSITION end_decision__1
EXE SKIP
FROM NOW TO INF
JUMP end_of_process.

TRANSITION get
READ ticket(got_station) FROM booking
FROM NOW TO INF
JUMP get__stop.

TRANSITION get__stop
EXE ABRT
FROM NOW TO INF
JUMP end_of_process.
```

```
END; /* Passenger */
 Slotmachine: PROCESS
TYPE price IS
 integer ARRAY OF integer.
PR VAR expenses OF price.
PR VAR station OF integer.
PR VAR nominal OF integer.
PR VAR sum OF integer.

TRANSITION Start
EXE expenses[1]:=10;
expenses[2]:=15;
expenses[3]:=20;
FROM NOW TO INF
JUMP get_station.

TRANSITION get_station
READ station(station) FROM buttons
FROM NOW TO INF
JUMP get_station__X1.

TRANSITION get_station__X1
EXE sum:=expenses[station];
FROM NOW TO INF
JUMP get_station__X2.

TRANSITION get_station__X2
WRITE light(sum) INTO indicator
FROM NOW TO INF
JUMP proc.

TRANSITION proc
READ coin(nominal) FROM slot
FROM NOW TO INF
JUMP proc__X1.

TRANSITION proc__X1
EXE sum:=sum-nominal;
FROM NOW TO INF
```

```
JUMP proc__X2.

TRANSITION proc__X2
WRITE light(sum) INTO indicator
FROM NOW TO INF
JUMP proc.

TRANSITION proc
READ request FROM buttons
FROM NOW TO INF
JUMP begin_decision__1.

TRANSITION begin_decision__1
EXE SKIP
FROM NOW TO INF
JUMP proc__1.

TRANSITION proc__1
WHEN (sum>0)EXE SKIP
FROM NOW TO INF
JUMP proc__1__X3.

TRANSITION proc__1__X3
WRITE light(sum) INTO indicator
FROM NOW TO INF
JUMP proc.

TRANSITION proc__1
WHEN (NOT (sum>0))EXE SKIP
FROM NOW TO INF
JUMP proc__1__X4.

TRANSITION proc__1__X4
WRITE ticket(station) INTO booking
FROM NOW TO INF
JUMP get_station.

TRANSITION end_decision__1
EXE SKIP
```

```
  FROM NOW TO INF
  JUMP end_of_process.

 END; /* Slotmachine */
END; /* Passenger_Slotmachine */
END; /* All */
```

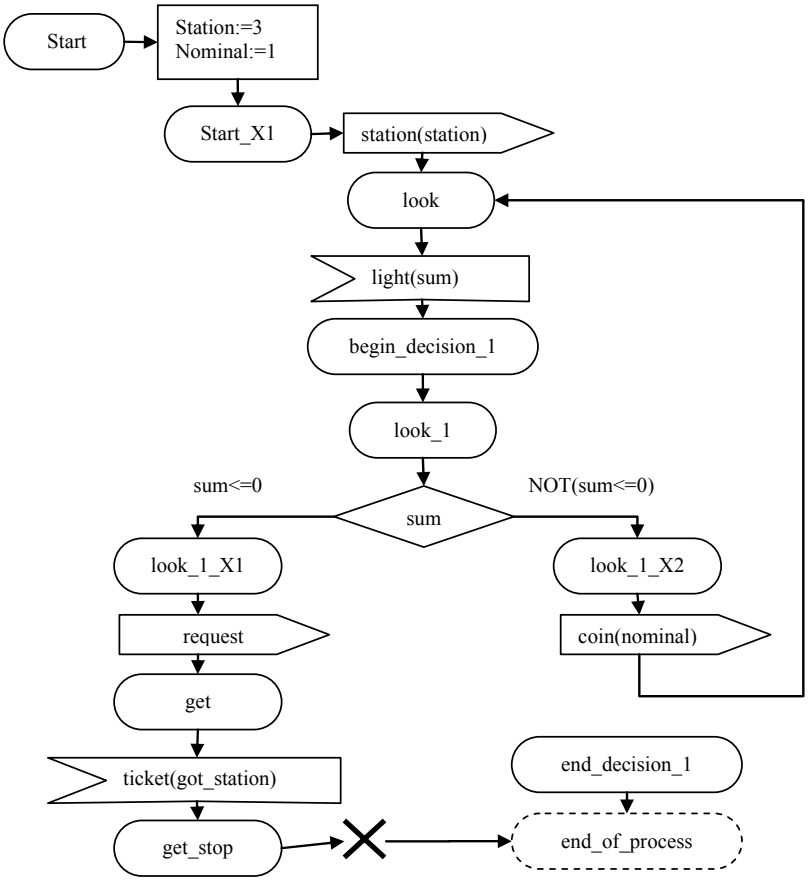

Рис. 12. Процесс Passenger после трансляции в REAL

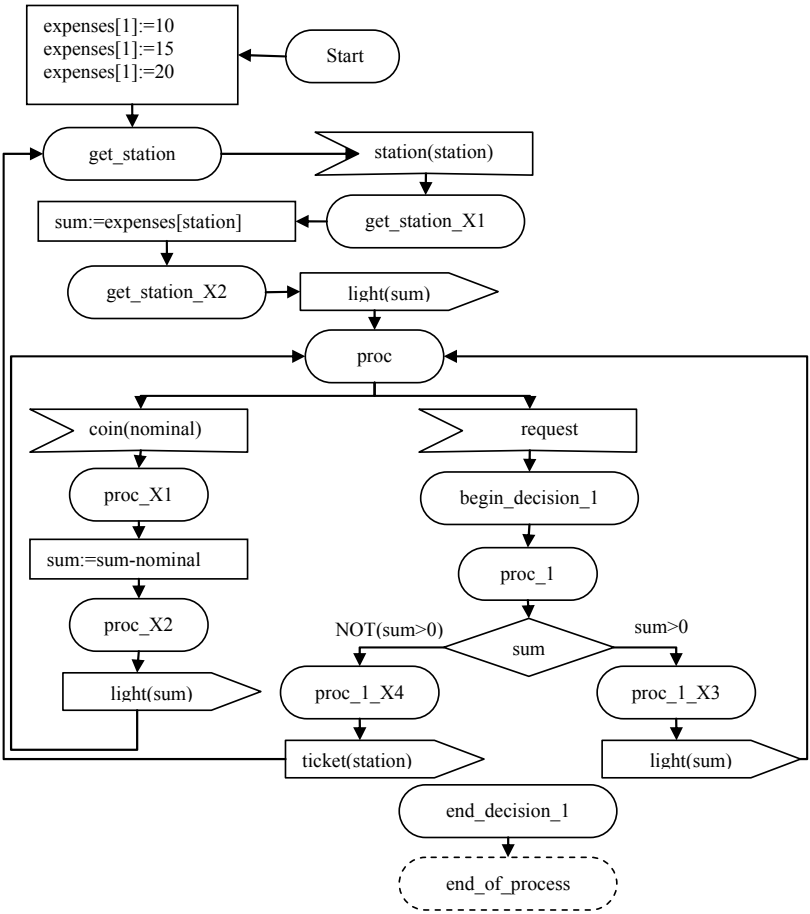

Рис. 13. Процесс Slotmachine после трансляции в REAL

8.4. Логическая спецификация для SDL-программы

Теперь перейдем к спецификации неформального свойства, описанного выше. Напомним его еще раз: при соблюдении описанного выше протокола касса выдаст билет каждому пассажиру до нужной ему станции. Представим его в виде формулы (в фигурных скобках указаны пояснения для каждого элемента спецификации).

```
FORM property;
   SIGNAL
     coin    (/* nominal */ integer),
     light   (/* sum */     integer),
     ticket  (/* station */ integer),
     station (/* station */ integer),
     request;

   SIGNALROUTE buttons
     FROM Passenger TO Slotmachine WITH station, request;
{объявление канала, соответствующего клавиатуре}
   SIGNALROUTE slot
     FROM Passenger TO Slotmachine WITH coin;
{объявление канала, соответствующего щели-монетоприемнику}
   SIGNALROUTE indicator
     FROM Slotmachine TO Passenger WITH light;
{объявление канала, соответствующего индикатору}
   SIGNALROUTE booking
     FROM Slotmachine TO Passenger WITH ticket;
{объявление канала, соответствующего билетному окну}
   BLOCK Passenger_Slotmachine;

(AL Slotmachine.expenses.(AL Passenger.station. ((start_of_machine &
start_of_passenger &
  no_commands_on_buttons &
 no_information_on_indicator &
  no_ticket_in_booking & no_money_in_slot)
        =>(EACH ET ( end_of_passenger & get_ticket)))));
(при любых ценах билетов в кассе, до любой нужной пассажиру станции,
истинность конъюнкции условий
касса готова к работе с пассажиром (предикат start_of_machine),
пассажир намеревается купить билет (предикат start_of_passenger),
ни одна кнопка клавиатуры не запала (предикат no_commands_on_buttons),
на индикаторе не высвечена никакая информация
(предикат no_information_on_indicator),
```

в билетном окне не торчит никакого билета (предикат **no_ticket_in_booking**),
приводит к истинности конъюнкции условий:
пассажир отошел от кассы (предикат **end_of_passenger**),
станция, указанная на билете, та самая, которая нужна этому пассажиру
(предикат **get_ticket**).}

{далее следуют спецификации предикатов}
PRED start_of_machine;
AT Slotmachine.Start;
PRED start_of_passenger;
AT Passenger.Start;
PRED no_commands_on_buttons;
buttons IS EMPTY;
PRED no_information_on_indicator;
indicator IS EMPTY;
PRED no_ticket_in_booking;
booking IS EMPTY;
PRED no_money_in_slot;
slot IS EMPTY;
PRED end_of_passenger;
AT Passenger.end_of_process;
PRED get_ticket;
Passenger.station=Passenger.got_station;

{конец спецификации формулы property}

8.5. Логическая спецификация на языке bREAL

Эта логическая bREAL-спецификация получена путем конвертации логической SDL-спецификации.

```
/*this file from convertor*/
property : FORM

1 min = 60 sec;
1 sec <= 60 sec;
1 ing <= 1 min <= 20 ing;

  INN UNB QUEUE CHN booking
        FOR ticket
        WITH PAR p1 OF integer.
  INN UNB QUEUE CHN buttons
        FOR request;
```

```
        FOR station
        WITH PAR p1 OF integer.
  INN UNB QUEUE CHN indicator
        FOR light
        WITH PAR p1 OF integer.
  INN UNB QUEUE CHN slot
        FOR coin
        WITH PAR p1 OF integer.
. Passenger_Slotmachine
(AL Slotmachine.expenses.
(AL Passenger.station.
((start_of_machine & start_of_passenger & no_commands_on_buttons &
no_information_on_indicator & no_ticket_in_booking & no_money_in_slot) => (EACH
ET (end_of_passenger & get_ticket))))).

end_of_machine : PRED
AT Slotmachine.end_of_process

end_of_passenger : PRED
AT Passenger.end_of_process

get_ticket : PRED
Passenger.station=Passenger.got_station

no_commands_on_buttons : PRED
buttons IS EMPTY

no_information_on_indicator : PRED
indicator IS EMPTY

no_money_in_slot : PRED
slot IS EMPTY

no_ticket_in_booking : PRED
booking IS EMPTY

start_of_machine : PRED
AT Slotmachine.Start

start_of_passenger : PRED
AT Passenger.Start
```

8.6. Описание эксперимента

Так называемый «сквозной » эксперимент заключался в том, что спецификация на языке SDL, описывающая протокол «Касса-Пассажир», подаётся на вход транслятора из SDL в bREAL.

Затем свойство, которое нужно проверить, выражается на новом языке логических спецификаций для SDL и конвертируется в свойство на языке bREAL. После этого производится порождение модели (путем исполнения всех возможных поведений спецификации).

Создаются три вспомогательных файла, в которых подробно описано построение модели.

Файл *model.txt* содержит получившуюся модель, выраженную как перечисление переходов системы.

На рис. 14 получившаяся модель показана в виде графа.

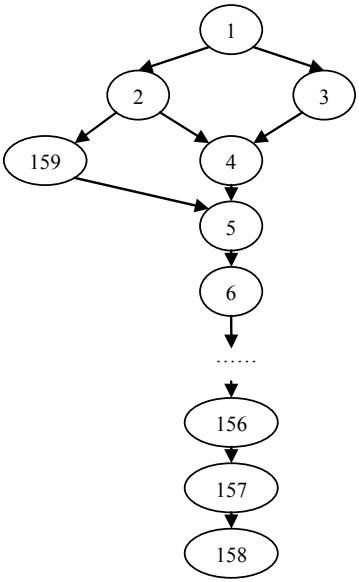

Рис. 14. Модель протокола Касса-Пассажир

В файле *change.cng* подробно описано, в каком состоянии была система в каждом из переходов. В таблице 1 приведены те переходы, которые показаны на рис. 14.

В файле *con_val.txt* отображены состояния проверяемого свойства и предикатов логической спецификации:

/*проверяемое свойство истинно на следующих переходах*/
true (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159)
/*предикат end_of_passenger истинен на следующих переходах */
end_of_passenger (158)
/*предикат get_ticket истинен на следующих переходах */
get_ticket (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159)
/*предикат no_commands_on_buttons истинен на следующих переходах */
no_commands_on_buttons (1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 153 154 155 156 157 158)
/*предикат no_information_on_indicator истинен на следующих переходах */
no_information_on_indicator (1 2 3 4 5 6 7 9 10 11 12 13 14 16 17 18 19 20 21 23 24 25 26 27 28 30 31 32 33 34 35 37 38 39 40 41 42 44 45 46 47 48 49 51 52 53 54 55 56 58 59 60 61 62 63 65 66 67 68 69 70 72 73 74 75 76 77 79 80 81 82 83 84 86 87 88 89 90 91 93 94 95 96 97 98 100 101 102 103 104 105 107 108

109 110 111 112 114 115 116 117 118 119 121 122 123 124 125 126 128 129 130 131 132 133 135 136 137 138 139 140 142 143 144 145 146 147 149 150 151 152 153 154 155 156 157 158 159)
/*предикат no_money_in_slot истинен на следующих переходах */
no_money_in_slot (1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 18 20 21 22 23 24 25 27 28 29 30 31 32 34 35 36 37 38 39 41 42 43 44 45 46 48 49 50 51 52 53 55 56 57 58 59 60 62 63 64 65 66 67 69 70 71 72 73 74 76 77 78 79 80 81 83 84 85 86 87 88 90 91 92 93 94 95 97 98 99 100 101 102 104 105 106 107 108 109 111 112 113 114 115 116 118 119 120 121 122 123 125 126 127 128 129 130 132 133 134 135 136 137 139 140 141 142 143 144 146 147 148 149 150 151 152 153 154 155 156 157 158 159)
/*предикат no_ticket_in_booking истинен на следующих переходах */
no_ticket_in_booking (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 157 158 159)
/*предикат start_of_machine истинен на следующих переходах */
start_of_machine (1 2 159)
/*предикат start_of_passenger истинен на следующих переходах */
start_of_passenger (1 3)

Проверяемое свойство модели автоматически переводится в формулу μ-исчисления. Эта формула записывается в файл formula.txt:

m y((start_of_machine & start_of_passenger & no_commands_on_buttons & no_information_on_indicator & no_ticket_in_booking & no_money_in_slot) - (m x((<a>true} & [a]x}) * (end_of_passenger & get_ticket))))).

Формула подается на вход подсистеме проверки моделей, вместе с порождённой моделью.

Результат в файле «result» («ALL STATES») показывает, что формула истинна во всех состояниях модели (т.е. во всех конфигурациях, возникающих при «исполнении» выполнимой спецификации).

8.7. Описание некорректного эксперимента

Для того чтобы эксперимент был полным, необходимо проверить, как система работает с заведомо некорректной спецификацией.

Из приведенной выше спецификации на языке SDL уберем следующую строку:

NEXTSTATE get;

в описании процесса Passenger.

Этот фрагмент отвечает за переход к получению билета в случае, если пассажир полностью оплатил его стоимость.

В данном примере проверяемое свойство будет таким же, что и в предыдущем эксперименте. Система должна показать, что проверяемое в логической спецификации свойство не будет верным для получившейся спецификации.

На рис. 15 представлена получившаяся модель в виде графа. Как видно, модель зацикливается, не происходит нужного перехода.

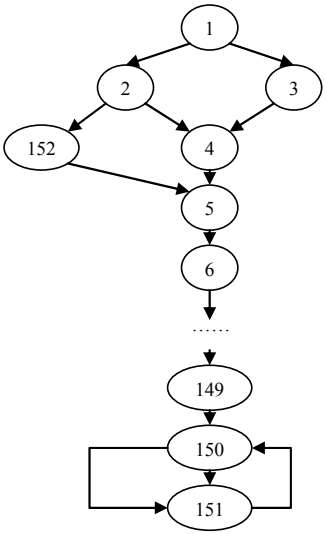

Рис. 15. Модель некорректного протокола Касса-Пассажир

В файле con_val.txt , где представлены состояния предикатов, предикат end_of_passenger ни в одном из переходов не становится истинен, что означает, что пассажир в предлагаемой модели никогда не закончит свою работу.

И наконец, содержимое файла result показывает, что данная формула верна не на всех переходах, а лишь на нескольких.

Из всего этого можно заключить, что в случае некорректной спецификации система покажет, на каком переходе произошла незапланированная ситуация, к какому результату приведет ошибка.

8.8. Эксперимент с помощью графического интерфейса системы моделирования выполнимых Basic-REAL-спецификаций

Графический интерфейс системы моделирования позволяет визуально проследить ход выполнения построения модели и работу проверяемой программы на языке Basic-REAL. Пользователь на любом шаге может узнать данные обо всех блоках, процессах, сигналах и других компонентах системы. Это очень важно для выявления узлов, в которых программа начинает действовать не так, как было задумано пользователем.

Эксперимент проводился на двух спецификациях. Первая представлена выше (разд. 8.3), которая была получена с помощью SRT-транслятора из исходной SDL-спецификации. Вторая спецификация содержит намеренно добавленные ошибки, которые не позволят получить требуемый результат.

1. Эксперимент с правильной Basic-REAL-спецификацией.

На рис. 16 показан старт системы моделирования. Все каналы пусты, все переменные еще не инициализированы. Все процессы находятся в состоянии Start (на рисунке данные по процессу Passenger).

На рис. 17 показано итоговое состояние процесса Passenger. По замыслу спецификации, этот процесс после окончания работы должен был завершиться — что мы и видим: процесс не находится ни в одном из состояний. На правильное завершение работы указывают и значения переменных:

- переменная sum равно 0 — это значит, что пассажир заплатил всю сумму за билет;
- переменная got_station проинициализирована (что по спецификации возможно было только после получения сигнала ticket) и равна переменной station — это значит, что пассажир получил билет до той станции, номер которой он ввел в начале работы с кассовым аппаратом.

На рис. 18 показано итоговое состояние процесса Slotmashine. По спецификации этот процесс после окончания работы с пассажиром должен перейти в состояние ожидания следующего пассажира. На представленном рисунке процесс находится в состоянии get_station — это значит касса ждет ввода номера станции. Переменные процесса Slotmachine также имеют верные значения:

- переменная station равна переменной station в процессе Passenger — это значит, что касса обрабатывает билет именно до той станции, которую ввел пассажир;
- переменная sum равна 0 — значит за билет были выплачены все деньги.

В итоге мы визуально отследили ход выполнения программы и убедились, что результат соответствует требуемому.

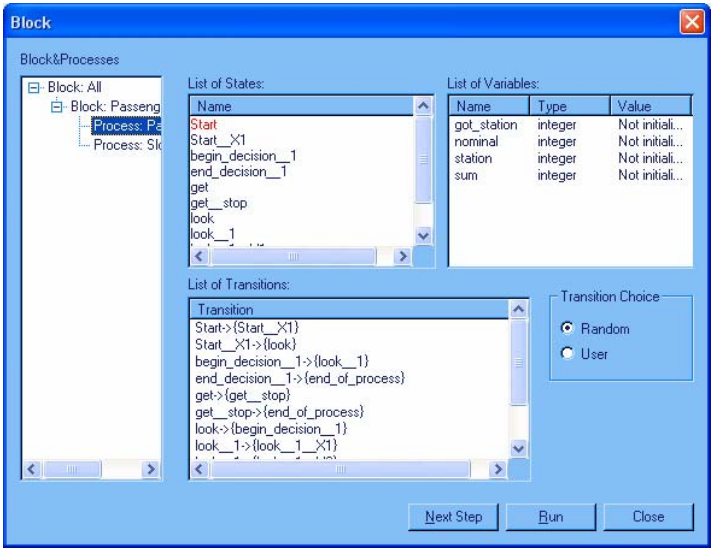

Рис. 16. Старт системы моделирования

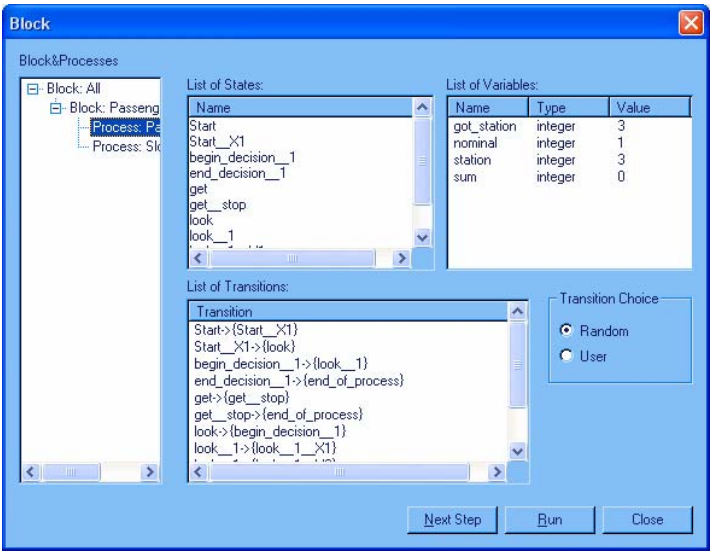

Рис. 17. Итоговое состояние процесса Passenger

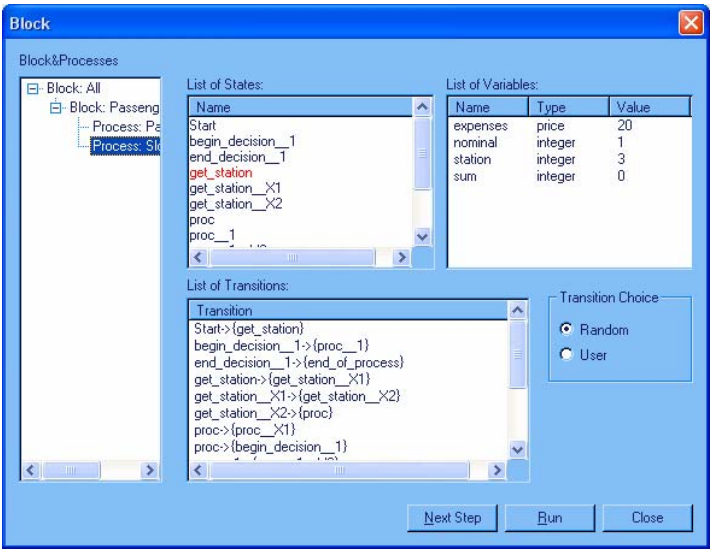

Рис. 18. Итоговое состояние процесса Slotmachine

2. Эксперимент с некорректной спецификацией.

В случае если некорректен синтаксис программы, то система моделирования просто не запустится. Ошибки на этом этапе можно отследить с помощью системы верификации.

Если же ошибка допущена в логике, то система моделирования покажет, где именно программа ведет себя неправильно. Для эксперимента удалим из спецификации следующие строки:

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

Эти строки ответственны за переход к получению билета, если сумма была выплачена полностью.

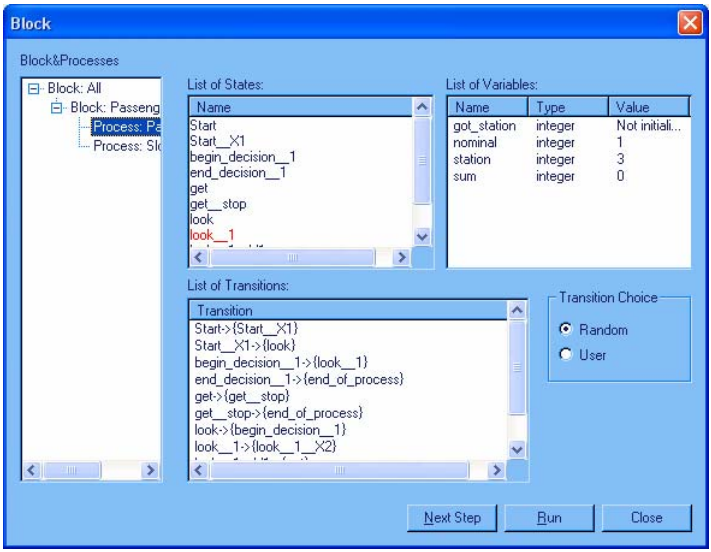

Рис. 19. Итоговое состояние процесса Passenger

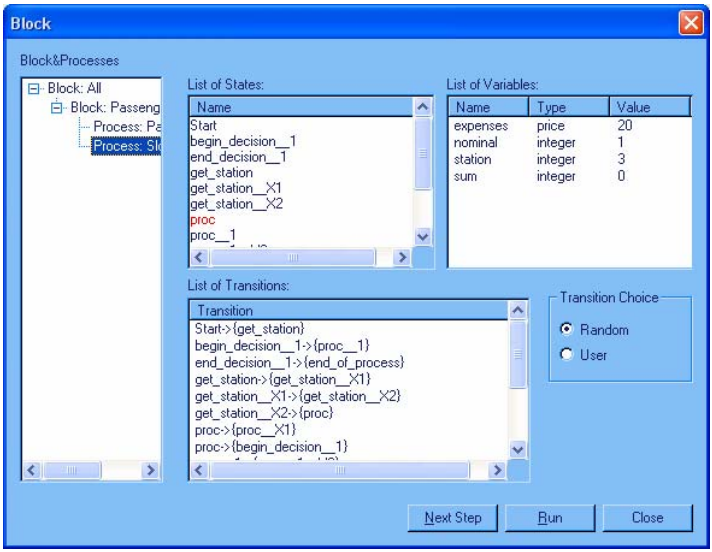

Рис. 20. Итоговое состояние процесса Slotmachine

На рис. 19 видно, что несмотря на то, что сумма пассажиром была выплачена полностью (sum=0), билет он не получил (got_station not initialising). Более того, процесс passenger находится в состоянии look_1 и все время возвращается к нему (если продолжить выполнение программы дальше, то через несколько шагов процесс Passenger вернется в состояние, отображенное на рисунке).

На рис. 20 процесс Slotmachine находится в состоянии proc, поскольку не получает от процесса Passenger сигнала get_ticket().

Налицо некорректная логика выполнения программы. И очень просто отследить, в какой именно момент программа повела себя не так, как задумано (при получении процессом Passenger сигнала о том, что сумма выплачена полностью, процесс не шлет сигнал get_ticket).

9. ЗАКЛЮЧЕНИЕ

Достоинство описанной системы SRPV, предназначенной для моделирования, анализа и верификации статических SDL-спецификаций распределенных систем, обусловлены как достоинствами языка bREAL, так и эф-

фективным методом трансляции статических SDL-спецификаций в bREAL. Среди достоинств языка bREAL следует упомянуть простой синтаксис, допускающий графическое представление выполнимых спецификаций, компактную полную структурную операционную семантику, а также выразительный подъязык логических спецификаций. Описанный эксперимент с протоколом «Касса-Пассажир» подтверждает достоинство системы SRPV.

Предполагается расширить язык bREAL и систему SRPV с целью трансляции динамических конструкций SDL, что позволит проводить моделирование и анализ динамических распределенных систем, которые могут использовать неограниченное число экземпляров процессов.

СПИСОК ЛИТЕРАТУРЫ

1. Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. — М.: Радио и связь, 1993.
2. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. — М.: МЦНМО, 2002.
3. Непомнящий В.А., Шилов Н.В. Язык спецификаций систем и свойств взаимодействующих процессов реального времени // Методы теоретического и системного программирования. — Новосибирск, 1991. — С. 32–45.
4. Непомнящий В.А., Шилов Н.В. Real92: Комбинированный язык спецификаций для систем и свойств взаимодействующих процессов реального времени // Программирование. — 1993. — № 6. — С. 64–80.
5. Непомнящий В.А., Шилов Н.В., Бодин Е.В Спецификация и верификация распределенных систем средствами языка Elementary-REAL // Программирование. — 1999. — № 4. — С. 54–67.
6. 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.
7. Bosnacki D. et al. Model checking SDL with Spin, TACAS/ETAPS 2000 // Lect. Notes Comput. Sci. — 2000. — Vol. 1785. — P. 363–377.
8. Bozga M., Graf S., Ober Il., Ober Iu., Sifakis J. The IF toolset, Proc. SFM-RT 2004 // Lect. Notes in Comp. Sci. — 2004. — Vol. 3185. — P. 237–267.
9. Broy M. Towards a formal foundation of the specification and description language SDL // Formal Aspects of Computing. — 1991. — Vol. 3, N. 1. — P. 21–57.
10. Cavalli A.R., Horn F. Proof of specification properties by using finite state machines and temporal logic // Proc. of 7-th IFIP Conf. on Protocol Specifications, Testing, and Verification. — 1987. — P. 221–233.
11. Cleaveland R., Klein M., Steffen B. Faster model checking for modal mucalculus // Lect. Notes Comput. Sci. — Vol. 663. — P. 410–422.

12. Eschbach R. et al. On the formal semantics of SDL-2000: A compilation approach based on an abstract SDL machine // Lect. Notes Comput. Sci. — 2000. — Vol. 1912. — P. 242–265.
13. Gammelgaard A., Kristensen J.E. A correctness proof of a translation from SDL to CRL // Proc. of the 6th SDL Forum. — 1993. — P. 205–219.
14. Gibson P., Mery D. Telephone feature verification: translating SDL to TLA+. — Report CRIN, Nancy, Dec. 1996.
15. Harel D. First-order dynamic logic // Lect. Notes Comput. Sci. — 1979 — Vol. 68.
16. Leue S. Specifying real-time requirements for SDL specifications | A temporal logic-based approach // Proc. 15-th IFIP Internat. Symp. on Protocol Spec. Test. and Verif. — Warsaw, 1995. — P. 19–34.
17. Mery D., Mokkedem A. CROCOS: An integrated environment for interactive verification of SDL specifications // Lect. Notes Comput. Sci. — 1993. — Vol. 663. — P. 343–356.
18. Nepomniaschy V.A., Shilov N.V. Real92: A combined specification language for systems and properties of real-time communicating processes // Lect. Notes Comput. Sci. — 1993. — Vol. 735. — P. 377–393.
19. Nepomniaschy V.A., Shilov N.V., Bodin E.V. A new language Basic-REAL for specification and verification of distributed system models. — Novosibirsk, 1999. — 39 p. — (Rep. / A.P. Ershov's Institute of Informatics Systems; N 65).
20. Nepomniaschy V.A., Shilov N.V., Bodin E.V., Kozura V.E. Basic-REAL: integrated approach for design, specification and verification of distributed systems // Lect. Notes Comput. Sci. — 2002. — Vol. 2335. — P. 69–88.
21. Orava F. Formal semantics of SDL specifications // Proc. of 8-th IFIP Internat. Symp. on Protocol Spec. Test., and Verif. — 1988. — P. 143–157.
22. Prinz A., Lowis M. Engineering the SDL formal language definition, Proc. FMOODS 2003 // Lect. Notes Comput. Sci. — 2003. — Vol. 2884. — P. 47–63.
23. Specification and Description Language (SDL). — CCITT, Recommendation Z.100, 1988.

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

СИМУЛЯЦИЯ И ВЕРИФИКАЦИЯ СТАТИЧЕСКИХ SDL-СПЕЦИФИКАЦИЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ С ПОМОЩЬЮ ПРОМЕЖУТОЧНОГО ЯЗЫКА REAL

Препринт
142

Рукопись поступила в редакцию 11.12.06
Рецензент Н.В. Шилов
Редактор З. В. Скок

Подписано в печать 10.04.07
Формат бумаги 60 × 84 1/16 Объем 4.0 уч.-изд.л., 4.4 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2», г. Бердск, 49-а, оф. 7, тел./факс 8 (241) 5 38 77