ГОУ ВПО “ЯРОСЛАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. П.Г. Демидова”

На правах рукописи

Кубасов Сергей Валерьевич

Верификация автоматных программ в контексте синхронного программирования

05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук
(в редакции кафедры технологий программирования СПбГУ ИТМО)

Научный руководитель
д. ф.-м. н., проф.
Соколов Валерий Анатольевич

Ярославль – 2008

Содержание

Введение

Актуальность работы С 90-х годов XX века в России развивается автоматное программирование. Профессор А.А. Шалыто предложил использовать switch-технологию (другое название автоматного программирования (АП)) для решения задач логического управления [41]. Ее основная идея в использовании автоматов для кодирования логики программы. Допускается применять другие подходы для решения отдельных подзадач.

Для решения задачи логического управления изобретено немало языков, однако, как показано в книге [41], все они обладают теми или иными недостатками. Технология автоматного программирования значительно упрощает процесс взаимодействия различных участников процесса разработки: Заказчика, Технолога, Разработчика, Программиста, Оператора и Контроллера (пользователя). Программа, заданная в виде графа переходов, должна быть интуитивно понятна всем перечисленным категориям участников. Таким образом решается проблема взаимопонимания.

Со времени своего изобретения технология претерпела некоторые изменения. Были предложены различные модификации switch-технологии, использующие идеи других популярных парадигм программирования, например, объектно-ориентированного программирования. Расширилась область применения. В результате появилось несколько вариаций АП, среди которых уже сложно выявить главное направление.

Для разработки сложных программных систем с применением switch-технологии был создан проект UniMod. В этом инструментальном средстве можно было обнаружить множество полезных функций для разработчика. Это визуальное программирование, отладчик, подсветка синтаксиса и многое другое. К сожалению, не было никакой возможности верификации программ. Более того, заложенные в основу представления об автоматной программе

делали задачу добавления полноценной проверки проблематичной. UniMod не дает четкого определения автоматной программе.

Все эти соображения привели к необходимости разработки новой среды программирования и верификации с упором на функцию проверки. Основная трудность в осуществлении этого замысла заключалась в выборе методов верификации автоматных программ. Хотя задача верификации уже долгое время служит предметом пристального внимания со стороны многих ученых, ее нельзя назвать решенной, верификация же автоматных программ является новым направлением.

Задачи логического управления часто предъявляют критические требования к надежности программного обеспечения. Цифровые устройства сейчас можно встретить практически во всех сферах человеческой жизни. От их бесперебойного функционирования часто зависит жизнь людей. Со временем число и сложность устройств логического управления только возрастают. Поэтому задача верификации не теряет, а, наоборот, усиливает свою актуальность. Ручная проверка — трудоемкое занятие. Гораздо удобнее и надежнее поручить выполнение этой задачи компьютерной программе.

Задача верификации возникла несколько десятилетий назад. К настоящему моменту уже выработаны разнообразные подходы ее решения [16]. Среди ученых, внесших значительный вклад в решение этой задачи, можно выделить Н.А. Анисимова [48], О.Л. Бандман [2, 3], Ю.Г. Карпова [14, 15], И.А. Ломазову [23], В.А. Непомнящего [26], А.К. Петренко [22], Р.Л. Смелянского [32], В.А. Соколова [33], P.A. Abdulla [46], G. Berry [52–55], M.C. Browne [58, 59], K. Cerans [45], E.M. Clarke [62, 63], E.A. Emerson [62, 64, 65], A. Finkel [69, 70], Jr.O. Grumberg [16, 63], G.L. Holzmann [73], K. Jensen [78–80], Z. Manna [81], A. Pnueli [81], Ph. Schnoebelen [70], N. Sidorova [84], J. Sifakis [85, 86]. Рассмотрим некоторые из подходов.

Метод тестирования применяется к готовой системе. На вход подаются

определенные сигналы (последовательности сигналов), контролируются значения на выходе. В целом, можно сказать, что тестировать готовую систему не есть наилучшее решение. Ошибки, обнаруженные на этой стадии, могут потребовать значительных изменений и будут стоить очень дорого. Ошибки лучше находить как можно раньше. Тем меньше будет цена исправления. Модульное тестирование отчасти решает эту проблему. Но, к сожалению, не все ошибки можно обнаружить в отдельных компонентах системы. Некоторые ошибки являются следствием взаимодействия. Еще хуже тот факт, что тестирование в принципе не может гарантировать выполнение данной спецификации. Проверяются только отдельные варианты исполнения системы, но не все.

Дедуктивный анализ связан с формальным доказательством правильности работы программы. Используются системы аксиом и правил вывода. Этот метод действительно может гарантировать выполнение спецификации, но он чрезвычайно трудоемкий. В большинстве своем доказательство приходится проводить вручную. Такую работу может выполнить только специалист. Требуется большой опыт. Само доказательство занимает дни и даже месяцы, причем сложно заранее предсказать, сколько времени оно займет. Не стоит забывать, что даже специалист может допустить ошибку.

Проверка на моделях может быть применена для верификации систем с конечным числом состояний. Большой плюс этого подхода в том, что проверка может быть выполнена полностью автоматически. Результатом проверки является ответ “да” или “нет”, а для случая “нет” строится трасса, демонстрирующая нарушение спецификации.

Темпоральные логики используются для спецификации программных систем. При помощи темпоральных формул удобно описывать свойства, выражающие порядок событий во времени. Спецификация системы может быть описана набором темпоральных формул. Автоматизация доказательства тем-

поральных формул на моделях связана с изобретением Кларка и Эмерсона [64]. Позднее алгоритм проверки был улучшен Кларком, Эмерсоном и Систлой [62].

Алгоритм проверки на моделях связан с обходом всех состояний системы. Часто число состояний слишком велико, чтобы быть представленным в памяти компьютера. Изобретение МакМилланом в 1987 году упорядоченных двоичных разрешающих диаграмм (OBDD) [60] позволило значительно увеличить размеры систем, поддающихся верификации. OBDD позволяют компактно представлять состояния, потребляя меньше памяти.

Несмотря на многочисленные усовершенствования, Model Checking не может быть применен для любой системы. Часто требования к необходимой памяти превосходят все мыслимые пределы или же время проверки недопустимо велико. Все еще существуют практические задачи, которые не могут быть решены методом Model Checking по причине ограниченности вычислительных ресурсов.

Накопленный опыт по верификации разнообразных систем может быть применен для автоматного программирования. У каждого подхода верификации есть свои плюсы и минусы, однако для АП наиболее подходящим методом является проверка на моделях. Автоматная программа уже состоит из автоматов, формальных по своей природе. Достаточно только доопределить, уточнить правила использования и взаимодействия автоматов, чтобы получилась модель, приемлемая для верификации.

Задача верификации автоматных программ находится в процессе исследования. Можно выделить две группы, активно работающих над этой проблемой. В Ярославском государственном университете им. П.Г. Демидова на кафедре теоретической информатики проводятся исследования по моделированию, спецификации и верификации автоматных программ. Результаты работ обсуждаются на семинаре “Моделирование и анализ информационных

систем" и конференциях [20]. В работу вовлекаются студенты [21]. Команда Шалыто А.А., автора технологии автоматного программирования, также достигла определенных успехов в разработке среды для верификации автоматных программ.

В работе [18] предлагается иерархическая модель автоматных программ. Модель рассматривается на примере системы управления кофеваркой. В работе [4] предлагается способ моделирования, спецификации и верификации автоматных программ. Используется верификатор SPIN, логика LTL. Рассмотрен пример системы управления банкоматом. Одним из развитий иерархической модели автоматных программ [18] является модель [9], использующая формализм сетей Петри. Разработка программы выполняется в UniMod, применяется верификатор CPN/Tools [61], рассмотрен пример системы управления кофеваркой. Получено свидетельство об официальной регистрации программы для ЭВМ [31].

На кафедре технологий программирования СПбГУ ИТМО были получены в том числе следущие результаты [13, 17] . В работе [7] рассматривается техника верификации автоматных программ, состоящих из одного конечного автомата. Описывается преобразование автомата в структуру Крипке, выражение темпоральных свойств на языке CTL, верификация по модели (метод "Model Checking") и построение сценария для исходного автомата, если был найден контрпример. Техника верификации демонстрируется на примере универсального инфракрасного пульта для бытовой техники [8]. Применение метода Model Checking также исследуется в работах [5, 6].

Можно заметить, что авторы указанных работ используют Model Checking. Главное различие наблюдается в моделях. Данная работа не является исключением.

Наиболее обстоятельное описание АП можно найти в книге [41]. Она была взята за основу. Было сделано несколько уточнений и ограничений, в резуль-

тате получилась модель автоматной программы.

В указанном источнике предполагается, что входные сигналы автомата могут только бинарными. Во многих подходах разрешается использовать целые и вещественные числа в качестве значений входных сигналов. Логические сигналы оставляют разработчику меньше свободы, но они ближе к решаемой задаче, задаче логического управления.

Язык синхронного программирования Esterel [55] разрабатывался для аналогичного круга задач, что и автоматное программирование. В отличие от АП, в Esterel был изначально заложен прочный теоретический фундамент. В Esterel существует базис — подмножество языка, на котором можно определить все его операторы. Программа на этом языке — это уже модель, готовая для верификации. Существует верификатор Xeve [56] для проверки Esterel-программ.

АП и язык Esterel помимо общей задачи роднит тот факт, что оба они используют бинарные сигналы. Esterel допускает сигналы с дополнительными значениями. Однако значения не используются в процессе верификации, только статус сигнала (“true” — “false” или “присутствует” — “отсутствует”) принимается во внимание.

АП и Esterel удачно дополняют друг друга. АП предлагает средства для визуальной разработки программ, а Esterel обеспечивает прочную математическую основу и средства верификации.

Xeve — это верификатор для Esterel-программ, представляемых в виде конечных автоматов. Xeve предлагает дружественный графический пользовательский интерфейс. Компилятор языка Esterel преобразует программу в систему в систему логических уравнений с защелками (latch), которые неявно определяют конечный автомат. Xeve работает с неявно заданным конечным автоматом. Внутри верификатора автомат описывается с помощью бинарных диаграмм решений (Binary Decision Diagrams).

Верификатор Xeve был разработан совместно институтом Inria[1] и Ecole des Mines de Paris[2] в рамках исследовательского проекта TICK[3]. Технология оказалось многообещающей. Было решено адаптировать ее для промышленного использования. Так появилась компания Esterel Technologies. Сейчас Esterel Techonlogies может гордиться удачными проектами со многими известными заказчиками: Airbus, Elbit Systems, Intertechnique, Eurocopter и др[4].

Цель диссертационной работы Цель диссертации — создать программный комплекс разработки и верификация автоматных программ. Достижение указанной цели было связано с решением следующих подзадач.

- Разработать формальную модель автоматной программы, которая бы подходила для сложившейся практики применения автоматного программирования.
- Разработать метод верификации автоматных программ.
- Разработать и реализовать программные средства построения и верификации автоматных программ. Исследовать работу программной системы при решении практических задач.

Научная новизна Все основные результаты являются новыми.

- Разработана формальная модель автоматной программы, учитывающая наиболее важные идеи автоматного программирования.
- К автоматному программированию был применен синхронный подход. Уточнена временная модель. Поведение программы стало детерминиро-

[1]http://www.inria.fr/index.en.html

[2]http://www.ensmp.fr/

[3]http://www.inria.fr/recherche/equipes/tick.en.html

[4]http://www.esterel-technologies.com/technology/success-stories/

ванным. Разработана вариация автоматного программирования — синхронно-автоматное программирование.

- Разработаны способы верификации синхронно-автоматных программ при помощи существующих программных инструментов языка Esterel. Использован верификатор Xeve.

- Создана программная среда разработки и верификации синхронно-автоматных программ.

Практическая ценность Разработаны методы верификации автоматных программ. Их применение позволит обнаружить многие ошибки, допускаемые в процессе разработки. Возможна проверка программы на соответствие изначальным требованиям технического задания.

Разработана и реализована программная система разработки и верификации синхронно-автоматных программ. Ее применение упрощает и ускоряет разработку и проверку указанного класса программ.

Апробация работы Результаты диссертации докладывались на 7-ой международной конференции и выставке "Системы проектирования, технологической подготовки производства и управления этапами жизненного цикла промышленного продукта (CAD/CAM/PDM-2007)" (Москва, 2007), XIV-ой международной научно-практической конференции "Современные техника и технологии" (Томск, 2008), международной научной конференции "Информация, сигналы, системы: вопросы методологии, анализа и синтеза" (Таганрог, 2008), международной научной конференции "Математика, кибернетика, информатика" (Ярославль, 2008).

Результаты обсуждались на семинаре "Моделирование и анализ информационных систем" кафедры теоретической информатики Ярославского го-

сударственного университета им. П.Г. Демидова (2006–2008 гг.).

Участие в проектах Во время работы над диссертацией автор участвовал в следующих научных проектах:

1. Разработка формальных моделей распределенных систем и исследование их семантических свойств. РФФИ, грант № 07–01–00702.

2. Федеральная целевая программа "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007–2012 годы". Проект № 2007–4–1.4–18–02 "Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода".

Публикации По теме диссертации опубликовано семь научных работ. Из них три опубликованы в изданиях, входивших в перечень ВАК на момент публикации и находящихся в перечне ВАК в настоящий момент.

Личный вклад автора Все результаты исследований, составляющих основное содержание диссертации, получены автором самостоятельно.

Структура и объем диссертации Диссертация состоит из введения, трех глав, заключения, списка литературы и приложений.

В первой главе описывается технология автоматного программирования и закладывается фундамент для разработки средств верификации. Дается краткая история автоматного программирования и обзор основных направлений его развития. Выделяются основные идеи switch-технологии. Обсуждается проблема верификации автоматных программ.

Дается краткое описание синхронного подхода, языка Esterel, верификатора Xeve. Обсуждается возможность применения синхронного подхода к автоматным программам.

Глава 2 описывает среду разработки и верификации синхронно-автоматных программ. Первый раздел посвящен теоретическим аспектам, лежащим в основе. Затем идут: язык описания автоматной программы, верифицируемые свойства, структура среды разработки.

Синхронно-автоматная модель может быть представлена в нескольких различных форматах: XML-структура, UML-диаграммы, программа на языке Esterel. Приводится описание каждого из способов. Отдельный раздел посвящен процедуре верификации. Можно выделить три типа проверяемых свойств: формат модели, синхронность, пользовательские свойства. Основное внимание уделяется проверке пользовательских свойств верификатором Xeve. Программный инструмент TempEst [75] предлагает возможность записи свойств на языке линейной темпоральной логики.

Глава 3 содержит несколько примеров программ, разработанных и проверенных с применение синхронно-автоматного подхода. В заключении главы делаются выводы об удачности применения синхронного подхода и языка Esterel для верификации автоматных программ.

Благодарности Выражаю благодарность сотрудникам кафедры теоретической информатики ЯрГУ, в особенности, научному руководителю В.А. Соколову за помощь в подготовке диссертации, внимание к работе. Е.В. Кузьмину, Д.Ю. Чалому, Р.А. Виноградову спасибо за обсуждение результатов работы, ценные замечания.

Глава 1

Автоматное программирование для решения задач логического управления

1.1. Автоматное программирование или Switch-технология

1.1.1. Возникновение автоматного программирования

Автоматное программирование (АП) возникает в начале 90-х годов прошлого века. Первый проект, в котором оно было опробовано на практике и, фактически, выкристаллизовалось, это система управления дизель-генератором [71]. Проектная документация относится к 1993 году.

Несколько лет исследований завершились написанием серии книг и статей [36]. Из которых наиболее важными следует признать [37, 41].

В автоматном программировании центральное место занимает автомат. Автомат используется в процессе проектирования, реализации, отладки и документирования программы. В АП основное внимание уделяется состоянию объекта. На стадии проектирования для каждого объекта определяется набор допустимых состояний. Предусматриваются все возможные переходы между состояниями. Причиной изменения состояния объекта являются события или «входные воздействия». В процессе изменения состояния возможна генерация «выходных воздействий». Объекты такого типа удобно моделируются конечным автоматом.

Графы переходов позволяют наглядно представить логику программы. Ее несложно понять неспециалисту. Различные участники процесса разработки (Заказчик, Технолог (Проектант), Разработчик, Программист, Оператор

(Пользователь), Контролер) “говорят” на одном языке. Решается проблема взаимопонимания.

1.1.2. Развитие технологии

Автоматное программирование за время своего существования успело пройти несколько стадий [35]. Не претендуя на исчерпывающее описание, ниже приводятся основные направления развития технологии.

Паттерн State Machine

В статье [42] предлагается взгляд на АП как на паттерн объектно-ориентированного проектирования. Паттерны описывают “простые и изящные решения типичных задач”. Коллекционирование паттернов имеет важное значение для программирование в целом. Однажды найденное удачное решение задачи может многократно использоваться в будущем. Накапливается опыт. АП действительно предлагает решение для некоторого класса типичных задач в логическом управлении.

Паттерн State Machine был неоднократно описан в специальной литературе, например, в книге [28], однако авторы статьи [42] полагают, что описание не достаточно удачно. Паттерн предстает в новом виде.

Событийно-управляемые системы

Идея автоматного подхода была опробована на классе систем, управляемых событиями. В основном это относится к интерактивным системам, но в некоторых случаях применимо также к реактивным. Блок управления, реализуемый автоматом, взаимодействует с окружающей средой с помощью событий и набора процедур. События являются входными воздействиями,

инициирующими деятельность автомата. В ответ на воздействие автомат вызывает одну или несколько процедур — инициирует выходные воздействия.

Событийно-управляемые системы можно обнаружить в широком спектре задач. Например, в операционной системе Windows события используются для уведомления приложения о различных действиях пользователя, как нажатие клавиш на клавиатуре, клик кнопки мыши. Процессы широкого класса Unix-подобных операционных систем используют сигналы для уведомления о совершении некоторого события [29]. Нажатие пользователем комбинации клавиш <Ctrl>+<C> в программе эмуляции терминала обычно приводит к отправлению сигнала SIGINT активному процессу. В недрах операционной системы сигналы используются для реализации механизма виртуальной памяти. Бизнес-приложения содержат многочисленные примеры использования событий.

В целом, можно сказать, что потенциальная область применения АП чрезвычайно широка. Автоматный подход может быть эффективно реализован на практически любом языке общего назначения. Особенно удобно кодировать автоматы с помощью оператора выбора. Для языка программирования C++ такой оператор называется switch [34], что послужило причиной возникновения второго названия АП — «Switch-технология».

Достоинством применения АП для событийных систем является централизация логики управления, разбросанной ранее по многочисленным обработчикам событий. Обработчики событий становятся максимально простыми, делегируя всю работу автомату.

Объектно-ориентированное программирование с явным выделением состояний

Популярность объектно-ориентированный методологии программирования сказалась на АП, что привело к появлению объектно-ориентированного программирования с явным выделением состояний [43].

Основная идея этого направления в представлении различных компонентов автоматной программы в виде объектов. В качестве объектов могут рассматриваться автоматы, состояния, переходы, события. Фантазия разработчика позволяет практически все рассматривать как объекты. Объектно-ориентированное программирование приносит с собой целый ряд новых концепций: полиморфизм, инкапсуляция, наследование. Было предложено немало вариантов, оперирующих этими понятиями. Автоматы могут наследовать друг другу, тем самым достигается повторное использование кода. Состояния могут быть сложными. Одни состояния вкладываются в другие, образуется иерархия по вложенности. Здесь можно использовать как идеи наследования, так и более традиционное отношение включения.

Повторное использование кода играет важную роль. Были предложены несколько вариантов библиотек с “заготовками” для разработки автоматных программ. Возможно сохранение и более сложной функциональности в библиотеке.

Стоит отметить несколько работ этого направления: [25, 38]. Реализованы примеры, демонстрирующие применение ООП с явным выделением состояний: система управления турникетом [1], задача “синхронизация цепи стрелков” [10], задача об обедающих философах [27].

UniMod

Развитием ООП с явным выделением состояний стало появление инструментального средства разработки автоматных программ UniMod. Его можно рассматривать как заявку на то, что АП готово для широкого использования. Наиболее удачные идеи развития switch-технологии нашли свое воплощение в этом проекте. Поэтому состояние проекта UniMod может служить показателем зрелости автоматной технологии в целом.

Интересно провести небольшое исследование на тему: “Для чего предназначен UniMod”.

Появление switch-технологии можно ассоциировать с изданием книги [41]. Switch-технология изначально предназначалась для «алгоритмизации и программирования задач логического управления» [41]. В работах [39, 40] АП было распространено на реактивные системы. Со временем были обнаружены другие применения АП. Инструментальное средство UniMod появилось «для поддержки автоматного программирования» [12]. Сложно найти точные указания на то, для построения каких программ был создан проект.

Согласно главному сайту UniMod [68], долгосрочная цель проекта заключается в создании унифицированной методологии процесса разработки приложений, призванной заполнить пропасть, лежащую между фазами проектирования (design) и разработки, существующую в настоящее время. Ключевыми технологиями проекта UniMod называются: модельно-ориентированный подход (Model Driven Architecture), язык UML и универсальные вычисления (universal computing). UniMod адаптирует switch-технологию для использования языка UML в качестве языка описания моделей. UniMod воплощает идею запускаемого, или исполняемого, UML [11]. Ивар Якобсон, один из создателей языка UML, назвал исполняемый UML одним из перспективных направлений ближайшего будущего.

На сайте проекта UniMod [68] разработчики рекомендуют применять этот инструмент для создания:

- автономных приложений на Java с графическим и консольным пользовательским интерфейсом;
- клиент-серверных приложений на Java;
- приложений для платформы Symbian.

Там же на сайте [68] не рекомендуется использовать UniMod для решения задач:

- разработки компиляторов, поскольку для этой области уже существуют специализированные инструменты;
- построения диаграмм классов (Problem domain Class Diagrams design), т.е. не стоит использовать UniMod как обычный UML-редактор.

Реактивные системы, системы реального времени, задачи логического управления не упоминаются. Получается, что из узкопрофильной технологии, нацеленной на задачи логического управления, АП превращается в популярную технологию, предназначенную для решения широкого спектра задач. В UniMod можно писать любые приложения, в том числе широко распространенные сейчас бизнес-приложения, главное, активно использовать автоматы для кодирования сложной логики. Универсализация АП — это одновременно хорошо и плохо. Хорошо потому, что удачную идею теперь можно опробовать на множестве различных задач. А плохо потому, что размывается и так очень непрочное теоретическое основание switch-технологии. По UniMod не заметно, чтобы разработчики в реализации следовали какой-либо формальной модели.

1.1.3. Сложившаяся практика применения

Был выполнен анализ примеров, поставляемых с плагином UniMod [68]. Кроме того, были рассмотрены некоторые UniMod-проекты с сайта [30]. По результатам анализа можно прийти к таким выводам.

Все проекты используют автоматы для реализации логики приложения. Среди объектов выделены источники событий и объекты управления. Однако часть программы, кодируемая традиционным способом, также велика. В некоторых проектах Java-код занимает очень значительную часть программы.

Многие приложения реализуют графический пользовательский интерфейс. Используется стандартная библиотека Java — Swing. Применение графической библиотеки в данном случае оправданно, т.к. она предоставляет множество готовых решений пользовательского интерфейса. Понятно также то, что АП не всегда удачно сочетается с существующими системами. Применение Swing в данном случае более важно, чем верность switch-технологии. Этим можно объяснить значительную долю Java-кода в приложениях.

Тем не менее, в рассмотренных программах существует не мало мест, где применение АП было возможно, но не было использовано авторами проектов. Подтверждением тому служит небольшое число автоматов в программах. Большинство проектов используют один, реже два автомата.

Интеграция АП и ООП не везде прошла удачно. Некоторые проекты демонстрируют формальное использование источников событий. Действительная генерация событий происходит в различных местах программы.

1.2. Проблема верификации автоматных программ

Под верификацией здесь понимается проверка какого-нибудь свойства (свойств) автоматной программы. Проверяемые свойства формулируются для конкретной задачи. Их выполнение не очевидно (иначе нет смысла их проверять).

Проблеме верификации автоматных программ стали уделять внимание сравнительно недавно. В книгах [37, 41] вопрос верификации не поднимается. В то же время, задачи логического управления часто предъявляют критические требования к надежности программного обеспечения. Цифровые устройства широко используются в современной жизни. Им отводятся ответственные роли и цена ошибки в программном обеспечении может быть очень высока.

1.2.1. Подходы к верификации автоматных программ

Задача верификации автоматных программ находится в процессе исследования. Можно выделить два центра научной деятельности. Это кафедра Теоретической информатики Ярославского государственного университета и кафедра Технологий программирования Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Обе группы достигли определенных результатов.

В работе [7] рассматривается техника верификации автоматных программ, состоящих из одного конечного автомата. Применяется модификация языка CTL, что решает “трудности с выполнением композиции автоматов”. Данное изменение также упрощает интерпретацию результатов проверки в том случае, когда найден контрпример. Один автомат — это сравнительно простая модель, поэтому можно ограничиться использованием простых алгоритмов. Описывается преобразование автомата в структуру Крипке,

написание темпоральных свойств на языке CTL, верификация по модели (метод “Model Checking”) и построение сценария для исходного автомата, если был найден контрпример. Техника верификации демонстрируется на примере универсального инфракрасного пульта для бытовой техники [8].

Работа [13] предлагает технологию верификацию автоматных программ, разработанных для UniMod. Если быть точным, то объектом верификации является не вся программа, написанная на UniMod, а только отдельный автомат, взятый изолированно. Автомат, согласно правилам построения программы, может содержать вложенные автоматы, поэтому даже один автомат обладает достаточно большими выразительными возможностями. В качестве верификатора выбран Bogor. Он позволяет расширять свой входной язык за счет описания дополнительных типов данных, к тому же удачно интегрируется с UniMod. Сам процесс верификации заключается в выполнении автоматной программы при различных входных данных. Интерпретацию автоматной программы выполняет все то же инструментальное средство UniMod. Надо заметить, что интерпретация программы, язык Java, очевидно, замедляют процесс проверки. Поэтому такая технология проигрывает по скорости проверки верификаторам, работающим с простыми типами данных. Однако, как утверждают авторы статьи, программы на практике имеют такой размер, что время проверки вполне приемлемо.

На конференцию “Компьютерные науки и технологии” [17] были представлены современные взгляды на проблему верификации автоматных программ. Доклад содержит следующие основные идеи. Верификация программ общего вида является сложной и практически неразрешимой задачей, в то время как для частного случая автоматных программ можно разработать эффективное решение. Большие надежды возлагаются на верификацию на моделях. Существует удачный опыт использования верификатора SPIN. Планируется создание верификатора автоматных программ на основе уже существующего

верификатора, например, SPIN или Bogor.

Среди работ, посвященных верификации автоматных программ необходимо также отметить [6, 24].

Исследования в ЯрГУ уделяют основное внимание методу Model Checking. Разберем несколько работ.

Работа [18] предлагает иерархическую модель автоматных программ. Это одна из первых работ, посвященных формализации понятия автоматной программы. Как уже неоднократно упоминалось, автоматное программирование предлагает технологию построения программ. Однако само понятие автоматной программы не имеет четкого определения. Если мы хотим использовать метод Model Checking для проверки, необходима формальная модель программы. Именно такая модель предлагается. Автор статьи развивает идею вложенных автоматов и объекта управления. За историю автоматного программирования предлагалось немало вариантов комбинирования различных понятий. Можно сказать, что иерархическая модель напоминает взаимодействие автоматов в UniMod. Вложенные автоматы подробно описывались в [41].

Коротко модель можно описать так. Существует главный автомат A_0. Он может управлять своими вложенными автоматами. Каждый вложенный автомат может иметь свои вложенные автоматы и т.д. В результате получается иерархическая система. Различные ветви дерева автоматов не пересекаются. Каждый автомат, кроме A_0, имеет ровно одного родителя. Отдельно существует объект управления, который является источником событий для автомата A_0. Все автоматы имеют возможность читать состояние объекта управления и оказывать на него предусмотренные воздействия. Отношение вложенности позволяет родительскому автомату вызывать вложенные автоматы для обработки какого-нибудь события. Отношение вызываемости напоминает идею процедурного программирования. В статье разбирается пример

управления кофеваркой.

Развитие идеи можно найти в работе [19]. Автоматная программа очень удобна для применения метода проверки на моделях. Предлагается структура Крипке автоматной модели. Для спецификации свойств используется язык темпоральной логики CTL. Описание свойств демонстрируется на примере системы управления кофеваркой.

Работа [4] продолжает изучение иерархической модели автоматной программы. Исследуется возможность спецификации структурных и семантических свойств автоматных программ с помощью логики LTL. Разбирается пример системы управления банкоматом. Верификация выполняется инструментом SPIN.

Работа [9] является еще одним развитием иерархической автоматной модели. UniMod выбран в качестве инструментального средства разработки автоматных программ. Придуман и реализован механизм преобразования диаграмм UniMod в CPN-модель. В качестве верификатора используется система CPN/Tools. Подход демонстрируется на примере системы управления кофеваркой.

Разработана программа, предназначенная для преобразования моделей автоматных программ во внутренний язык программы верификатора CPN/Tools. Поддерживается чтение и сохранение моделей автоматных программ в XML-формате, формате Microsoft Visio и т.д. Получено свидетельство о регистрации программы [31].

Результаты работы регулярно обсуждаются на семинаре “Моделирование и анализ информационных систем”.

1.3. Синхронный подход

Синхронное программирование применяется для создания реактивных систем, приложений реального времени, систем управления [44, 47, 49, 50, 72, 87, 89]. Целевая система представляется синхронной моделью. В первом приближении это черный ящик с набором входных и выходных сигналов, имеющий внутреннее состояние (см. раздел 1.3.4). Синхронный подход накладывает ряд ограничений на среду функционирования системы. Ниже описываются основы синхронного подхода.

1.3.1. Сигналы

Для взаимодействия с окружающий средой применяется единственный способ — сигналы. Входные сигналы устанавливаются окружающей средой и читаются системой. Выходные сигналы генерируются системой, читаются окружающей средой. Обычный сигнал имеет статус присутствия. Сигнал может либо присутствовать, либо отсутствовать. В дополнение каждый сигнал может нести значение определенного типа, например, целое число. Значение сигнала может изменяться только тогда, когда статус сигнала — "присутствует". В остальных случаях значение сигнала не меняется, но оно всегда есть! Существуют также специальные типы сигналов: чистые сигналы и сенсоры. Первые не имеют значения, вторые — статуса. Значение сенсора может свободно меняться.

1.3.2. Временная модель

Синхронный подход использует дискретную временную модель. Каждый отдельный момент времени называется инстентом (от англ. instent). Работа системы в каждый инстент называется реакцией (от англ. reaction). Можно говорить: "реакция системы на входные сигналы". Для системы время течет

как строго упорядоченная последовательность инстентов. Реакция в каждый инстент заключается в вычислении значений выходных сигналов текущего инстента и состояния системы в следующий инстент. Вычисление однозначно определяется входными сигналами и внутренним состоянием системы в текущий инстент. Реакции в разные инстенты непосредственно независимы друг от друга, но связаны историей, которая хранится как состояние системы.

До этого момента мы говорили только о логическом времени. В отношение реального времени возможны две разных реализации: управляемая событиями (рис. 1.1) или таймером (рис. 1.2).

```
<Инициализация>
<Для каждого> <входного события> <повторять>
  <Вычислить выходные данные>
  <Обновить память>
<Конец цикла>
```

Рис. 1.1. Модель синхронной системы, управляемой событиями

```
<Инициализация>
<Для каждого> <срабатывания таймера> <повторять>
  <Считать входные данные>
  <Вычислить выходные данные>
  <Обновить память>
<Конец цикла>
```

Рис. 1.2. Модель синхронной системы, управляемой таймером

1.3.3. Гипотеза нулевой задержки

Считается, что все действия в пределах инстента выполняются мгновенно и одновременно. Длительность любой реакции равна нулю. Каждый инстент

Рис. 1.3. Модель синхронной системы

атомарен, т.е. все действия, запланированные в данном инстенте, выполняются как единое целое.

1.3.4. Модель синхронной системы

Любая синхронная система может быть разбита на две взаимодействующие части (рис. 1.3): функциональную логику и память. Функциональная логика — это правила, по которым вычисляются значения выходных сигналов и состояние в следующий инстент. Вычисление однозначно для данных значений входных сигналов и состояния системы. Вычисление повторяется в каждый инстент, имеет нулевую длительность. Память имеет специальный регистровый тип. В пределах инстента сигналы от памяти остаются постоянными. Новое значение, определяемое в текущем инстенте, ячейки памяти получат только в следующем инстенте. Таким образом удается избежать мгновенно обратной связи между входом и выходом логического блока. Заметим, что логический блок не хранит внутри себя какого-либо состояния между инстентами. Его функция только вычислительная.

1.3.5. Параллельность

Использование параллельных ветвей вычислений часто является удобным при проектировании разных типов систем (реактивных, реального времени

и др.). Поэтому языки синхронного программирования должны поддерживать конструкции языка, позволяющие удобно выражать параллельность — такое решение было принято в 80-х годах прошлого века [50]. В зависимости от области применения, это могут быть диаграммы потока данных, иерархические автоматы или некоторый императивный вид синтаксиса, принятый в инженерном сообществе.

Параллельность является удобным выразительным средством, но она не обязаны быть использована при реализации — параллельные участки программы не обязательно должны соответствовать каким-то физически параллельным элементам, таким как потоки, процессы, аппаратные устройства. Например, компилятор синхронного языка Esterel создает последовательную программу по любому исходному коду.

1.3.6. Структура системы

Крупные системы удобно строить из составных блоков. Синхронное программирование позволяет использовать этот подход. Каждый составной блок является подобием целой системы. У него есть входы и выходы, блок имеет состояние. Пусть X, Y и U — векторы входных, выходных сигналов и внутреннее состояние соответственно. Верхний индекс — номер блока, нижний индекс — номер инстента. f и g — функции от входных сигналов и внутреннего состояния. Тогда можно записать:

$$U^i_{n+1} = f(U^i_n, X^i_n)$$
$$Y^i_n = g(U^i_n, X^i_n).$$

Набор выходных сигналов в текущий инстент (n) и состояние блока в следующий инстент $(n+1)$ вычисляется по набору входных сигналов и состоянию в текущий инстент. Это верно для любого блока (i) и любого инстента (n).

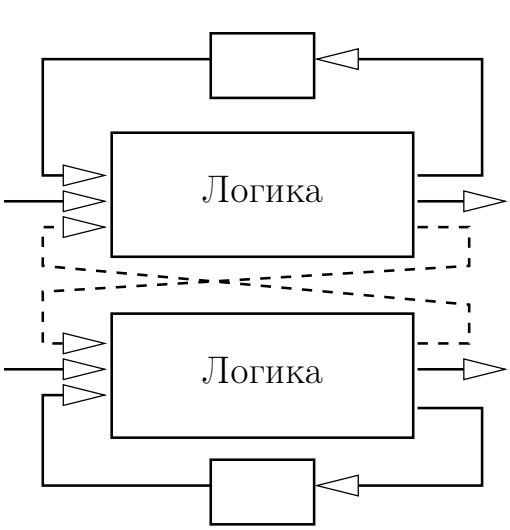

Рис. 1.4. Соединение двух синхронных блоков

Различные блоки соединяются друг с другом посредством входов и выходов. Выходной сигнал одного блока может являться входным сигналом для одного или более других блоков. Следовательно, входной сигнал может быть общим для нескольких блоков. Пусть символ в скобках обозначает номер сигнала в векторе. Например, $Y(k)$ — k-й сигнал вектора выходных сигналов Y. Тогда

$$Y_n^j(k) = X_n^i(l)$$

или

$$X_n^j(k) = X_n^i(l).$$

На рис. 1.4 показана композиция двух синхронных блоков. Пунктирными стрелками отмечены связи между логическими частями разных блоков. Здесь кроется проблема: возможно появление цикла. Таким образом, композиция синхронных блоков в общем случае не является синхронной системой. Существует несколько подходов к решению этой проблемы. Язык Esterel предлагает одно из них.

1.4. Синхронный подход в языке Esterel

1.4.1. Язык Esterel

Язык Esterel относится к семейству синхронных языков [52, 53]. Это императивный язык, предназначенный в первую очередь для описания потока управления [55]. Все характеристики синхронного подхода применимы в равной степени к языку Esterel. Не будем их повторять. Обратим внимание только на те особенности, которые являются специфическими для языка Esterel.

Основным типом данных в языке Esterel является сигнал. Он может принимать два состояния: “присутствует” и “отсутствует”, или “true” и “false” или “0” и “1”. Можно использовать любое из обозначений. Присутствие сигнала является событием, отсюда терминология. Статус любого сигнала в любой реакции однозначно определен. Сигнал признается присутствующим, если он установлен окружением, либо явно генерируется программой. В противном случае, сигнал признается отсутствующим.

Система всегда ведет себя функционально [50]. Для любого допустимого состояния системы и значения входных сигналов однозначно определяется следующее состояние и выходные сигналы. Иначе говоря, каждая реакция является детерминированной. Семантическая проверка программы на языке Esterel возлагается на компилятор. Это сложная задача. Статический анализ может обнаружить мертвое состояние в программе (deadlock), однако динамическая проверка может показать, что это состояние никогда не достигается, поэтому программа имеет шансы оказаться правильной.

Проверка программы на нарушение функциональности поведения называется анализ зависимостей (causality analysis). Ее выполняет компилятор [54, 82].

В языке Esterel предусмотрены операторы параллельного выполнения. Можно организовать сколько угодно параллельных потоков выполнения. Раз-

личные потоки взаимодействуют при помощи сигналов. Сигнал, генерируемый одним из потоков, мгновенно распространяется на все остальные потоки. Сигналы из других потоков рассматриваются наравне с сигналами из окружения. Это входные сигналы. Такое поведение называется синхронной широковещательной рассылкой сигналов (synchronous broadcast). Даже при наличии нескольких потоков выполнения поведение программы остается детерминированным. В начале каждой реакции каждый поток пробуждается в том состоянии, где он был оставлен в предыдущей реакции, выполняется обычный императивный код (вычисление выражений, операторы ветвления и присваивания), и наконец, каждый поток либо завершается, либо останавливается на одном из операторов ожидания до следующей реакции.

Параллельность, присущая языку Esterel, является логической. Некоторые программы удобно описывать, используя параллельные потоки. В то же время, компилятор вовсе не обязан их сохранять. Фактически значение имеет только результат вычислений, каковы будут значения выходных сигналов и состояние в следующий инстент. Компилятор Esterel может построить последовательную программу, эквивалентную исходной, или параллельную, причем исходное разбиение вовсе не ограничивает применение потоков.

Пример разработки протокола на языке HDLC приведен в [51].

Программный пакет для разработки Esterel-программ свободно доступен на сайте института Inrea [74]. Раздел, посвященный языку Esterel: [88]. Зеркало сайта http://esterel.org (в настоящее время сайт не действует), посвященного языку Esterel: [66].

1.4.2. Верификатор Xeve

Верификатор Xeve предназначен для проверки программ на языке Esterel. Он свободно распространяется в наборе с компилятором и другими утилита-

ми для разработки программ [67].

О работе верификатора известно мало. Описание, приводимое ниже, взято из руководства по верификатору [56].

Esterel-компилятор преобразует программу в систему логических уравнений с защелками (latchs), которая неявно определяет конечный автомат. Xeve работает с этим неявно определенным автоматом. Используется библиотека TiGeR, которая предоставляет эффективные структуры и алгоритмы для символьной манипуляции конечными автоматами, используя BDD (Binary Decision Diagrams). На входе верификатор получает набор логических уравнений, представленных в формате BLIF (Berkeley Logical Interchange Format). Xeve предлагает две главные функции верификации.

Первая функция заключается в минимизации конечного автомата на основании понятия бисимуляционной эквивалентности (bisimulation equivalence), которая определяет группы неразличимых состояний. Минимизированный конечный автомат явно генерируется в текстовом формате FC2. При помощи инструмента ATG получившийся граф можно просмотреть в графическом виде.

Вторая функция верификации заключается в проверке статуса выходных сигналов. Можно проверить, что определенный выходной сигнал когда-либо генерируется или всегда отсутствует. Можно также проверить, что данный выходной сигнал всегда генерируется, либо существует состояние, когда сигнал отсутствует. Впрочем, ни каких новых возможностей последняя функция не добавляет. Входные сигналы программы можно зафиксировать, установить их значение в 0 или 1. Если выходной сигнал может быть сгенерирован, минимальная трасса выполнения, демонстрирующая сигнал, сохраняется в формате CSIMUL. Последовательность шагов может быть выполнена графическим симулятором языка Esterel программой Xes. Этот же файл может быть полезен для консольного симулятора.

На первый взгляд кажется, что верификатор имеет слишком ограниченные возможности. Хотелось бы проверять более сложные свойства. Сложные свойства можно проверять, если описать их отдельной программой-наблюдателем (observer) на языке Esterel, которая подсоединяется к основной программе. При нарушении свойства наблюдатель генерирует специальный сигнал. Таким образом, выполнение свойства эквивалентно утверждению, что тревожный сигнал никогда не генерируется.

Более подробное описание Xeve и процедуры проверки можно найти в главе 2, разделы 2.3.3, 2.4.6.

1.4.3. TempEst

Существует приложение, позволяющие описывать проверяемые свойства на языке LTL (Liner Time Temporal Logic). Оно называется TempEst [75]. Особенность в том, что используется линейная темпоральная логика с операторами прошедшего времени. Задание свойств на языке LTL может быть удобнее, чем на языке Esterel. Намерения автора выражаются более явно.

TempEst не является верификатором. LTL-свойства преобразуются в модули на языке Esterel, которые требуется запускать параллельно основной программе. Алгоритм преобразования несложно понять [77]. В принципе, процедура проверки такая же, как при ручном написании свойств на Esterel, только свойства описываются в другом формате.

Более подробно процедура проверки описана в главе 2, раздел 2.3.4.

Глава 2

Среда разработки и верификации синхронно-автоматных программ

2.1. Формальная модель автоматной программы

Назовем реактивную систему, реализованную синхронно-автоматным (С-А) подходом к программированию, реактивной синхронной системой. Для краткости ее будем называть синхронной системой.

Синхронно-автоматная модель описывает синхронную систему. Основные компоненты модели — это синхронные автоматы и синхронные сети. Каждый из них в отдельности сам представляет синхронную систему. Изоморфизм представления достигается за счет унифицированного интерфейса.

2.1.1. Интерфейс синхронной системы

Интерфейс $I = (X, Y)$ синхронной системы определяется парой, состоящей из набора входных сигналов $X = \{ x_1, \ldots, x_m \}$ $(m \geqslant 0)$ и выходных сигналов $Y = \{ y_1, \ldots, y_k \}$ $(k \geqslant 0)$. Как можно видеть, оба списка сигналов могут быть пустыми. Каждый сигнал может принимать значения 0 и 1.

Для обозначения сигналов используются буквы x, y, z. Значения этих сигналов обозначаются так же. В большинстве случаев это не приводит к путанице. Так в записи $z = 0$ речь, очевидно, идет о значении сигнала, а в записи $z = z'$ — о самом сигнале. В том случае, когда нам необходимо, чтобы были равны именно значения сигналов, будем использовать обозначение $\xi(z)$: $\xi(z) = \xi(z')$, или будем оговаривать этот нюанс словами.

2.1.2. Синхронный автомат

Синхронный автомат реализует синхронную систему. Пусть синхронная система имеет интерфейс $I = (X, Y)$, тогда реализующий ее автомат определяется следующим образом:

$$A = (Q, q_1, X, Y, \Phi).$$

Здесь

$$Q = \{ q_1, \ldots, q_n \}$$

— множество внутренних состояний автомата, причем $n \geqslant 1$. q_1 — начальное состояние автомата.

Из множества входных сигналов строится формальный язык L. Он используется для записи условий на переходах.

$$L ::= 0 \mid 1 \mid x_i \mid L \wedge L \mid L \vee L \mid \neg L \mid (L)$$

Константы 0 и 1 обозначают “ложь” и “истину” соответственно. Логические операции $\neg$, $\wedge$, $\vee$ имеют обычный смысл. Все они имеют разный приоритет, убывающий слева направо в порядке перечисления. Скобки используются для переопределения приоритета.

Пусть также

$$R = \{ y_{i_1}, \ldots, y_{i_l} \mid l \in \mathbb{N} \cup \{0\},\ y_{i_j} \in Y \}$$

— множество выходных реакций. Элементы этого множества — списки выходных сигналов (множества выходных сигналов). В списке каждый сигнал присутствует не более одного раза. Списки, отличающиеся только порядком сигналов, считаются одинаковыми. Допускается пустой список.

$$\Phi \colon Q \times L \to Q \times R$$

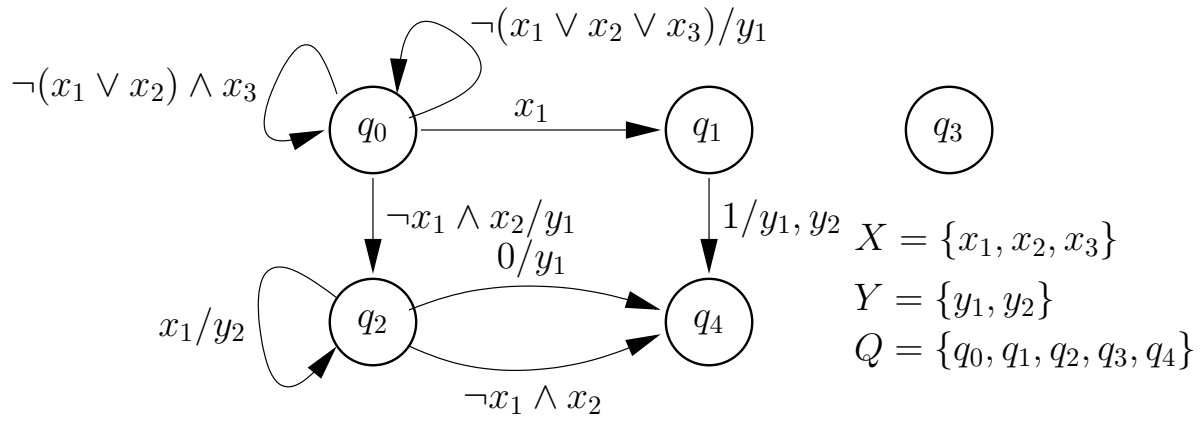

Рис. 2.1. Пример синхронного автомата

— функция переходов. Переход определяется начальным состоянием и логической формулой. Новое состояние может совпадать с начальным, т.е. переход является петлей. Между любыми двумя состояниями (в том числе совпадающими) может быть любое число переходов. Все переходы, выходящие из любого состояния, должны быть помечены ортогональными формулами, чтобы для любой комбинации входных сигналов нашлось не более одного перехода. Однако не обязательно, чтобы каждому набору входных сигналов соответствовал переход. Если перехода нет, по умолчанию считаем, что система не меняет состояние и не генерирует выходные сигналы.

Пример синхронного автомата приведен на рис. 2.1.

2.1.3. Синхронная сеть

Синхронная сеть реализует синхронную систему. Пусть синхронная система имеет интерфейс $I = (X, Y)$, тогда реализующая его синхронная сеть определяется следующим образом:

$$N = (\mathcal{I}, X, Y, G).$$

Синхронная сеть объединяет несколько синхронных систем. Как они реализованы — не имеет значения, доступен только интерфейс. Детали реали-

зации полностью скрыты за интерфейсом. Набор синхронных систем определяется только их интерфейсами:

$$\mathcal{I} = \{ I_1, \ldots, I_n \}, \text{ где } n \geqslant 1.$$

Для формального определения синхронной сети достаточно знать интерфейсы ее компонентов. Однако поведение синхронной сети существенным образом зависит от ее компонентов, которое определяется не только их интерфейсами. Поэтому, когда мы говорим о поведении, мы подразумеваем конкретные синхронные подсистемы, использованные в данной синхронной сети. Более подробно различие будет описано в разделе 2.1.4.

Все, что синхронная сеть делает, — это соединяет различные сигналы внутри себя. Используются входные и выходные сигналы самой сети, а также входные и выходные сигналы ее компонентов. Несколько соединенных сигналов образуют группу. Множество всех групп сигналов обозначается G:

$$G = \{G_1, \ldots, G_f\}, \text{ где } f \geqslant 0.$$

Число групп может быть любым, в том числе нулевым. Все сигналы можно разделить на ведущие и ведомые. К первым относятся входные сигналы интерфейса сети и выходные сигналы ее компонентов, ко вторым — все остальные. В каждую группу должен входить ровно один ведущий сигнал и не менее одного ведомого. Любой из сигналов может входить не более чем в одну группу в пределах данной сети, но может участвовать в двух разных группах, относящихся к разным сетям. Некоторые из сигналов могут не входить ни в одну из групп.

Пример синхронной сети приведен на рисунке 2.2.

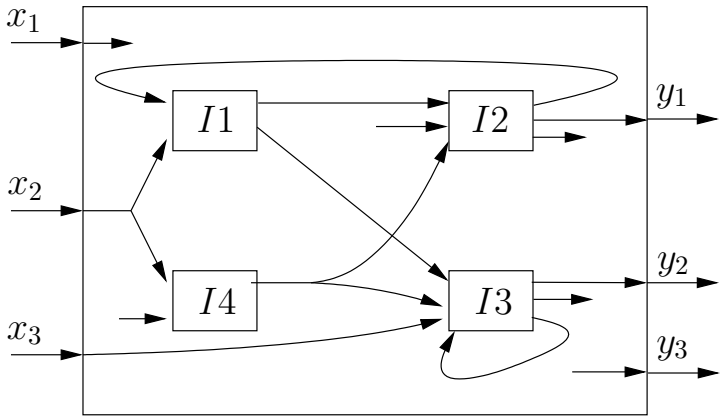

Рис. 2.2. Пример синхронной сети. Здесь $I1$, $I2$, $I3$, $I4$ — некоторые синхронные подсистемы. Синхронная сеть имеет три входных и три выходных сигнала

2.1.4. Типы и экземпляры

Синхронные сети и автоматы описывают типы, модели синхронных систем. Каждый ранее определенный тип может быть использован при определении других типов (только синхронных сетей). Типы отражают статическую структуру соответствующих моделей.

Для описания поведения системы вводится понятие экземпляра. Под экземпляром понимается конкретная синхронная система в процессе выполнения. Экземпляром является вся моделируемая синхронная система в целом, так и ее отдельные подсистемы. Каждый экземпляр имеет тип. Это определенный тип автомата или определенный тип синхронной сети. Один и тот же тип может иметь несколько экземпляров в синхронной системе.

Экземпляр характеризуется текущим состоянием, значениями входных, выходных и внутренних сигналов. Для экземпляров типа автомат состояние определяется значением одной переменной q. Состояние экземпляра синхронной сети определяется состоянием всех его синхронных подсистем.

Экземпляр автомата A обозначим

$$a = (q, \vec{X}, \vec{Y}), \tag{2.1}$$

где q — состояние автомата, $\vec{X}$ и $\vec{Y}$ — вектора значений входных и выходных сигналов соответственно.

Экземпляр синхронной сети N обозначим

$$n = (\vec{R}, \vec{X}, \vec{Y}), \tag{2.2}$$

где $\vec{R} = \{r_1, \ldots\}$ — вектор синхронных подсистем, используемых данным экземпляром синхронной сети. r_i имеет интерфейс I_i. В отличие от определения типа синхронной сети, при определении экземпляра нам необходимо знать не только интерфейс, но и всю используемую синхронную подсистему. Как уже упоминалось, состояние экземпляра синхронной сети включает в себя состояния всех используемых ей синхронных подсистем. $\vec{X}$, $\vec{Y}$ — вектора значений входных и выходных сигналов.

Условимся для обозначения экземпляров использовать строчные буквы, а для обозначения типов — заглавные. Экземпляр и тип будут обозначаться одной и той же буквой в разных регистрах, при этом сохраняются все модификаторы, такие как индексы, штрихи, волны и т.п. Например, в высказывании “Пусть $E_i \in \mathcal{A}$ такой, что $e_i.x_1 = 1 \ldots$” подразумевается, что существует экземпляр e_i типа E_i.

2.1.5. Модель в целом

Синхронно-автоматная модель состоит из автоматов $\mathcal{A} = \{A_1, \ldots\}$ и синхронных сетей $\mathcal{N} = \{N_1, \ldots\}$. Допускается, чтобы $\mathcal{N} = \emptyset$, но $\mathcal{A} \neq \emptyset$.

Мы видим, что простейшая модель состоит из одного автомата. Возможно, это достаточно для построения простой синхронной системы, но обычно приходится задействовать механизм группировки автоматов с помощью сетей.

Для любой модели существует компонент, описывающий всю систему. Назовем его главным объектом модели. В случае модели из одного автомата, он

же является главным объектом модели. Для модели, содержащей сети, главным элементом модели будет сеть, которая явным или косвенным образом включает в себя все остальные компоненты.

Заметим, что ни одна модель не может обойтись без автоматов. Они выполняют основную работу. Благодаря сетям мы можем произвольным образом группировать существующие блоки (построенные синхронные системы).

Вся синхронная система в целом есть экземпляр. Обозначим его $\tilde{r}$, а соответствующий тип — $\tilde{R}$ ($\tilde{R} \in \mathcal{A} \cup \mathcal{N}$). Иначе говоря, $\tilde{r}$ является экземпляром $\tilde{R}$.

2.1.6. Динамика синхронно-автоматной модели

Выполнение синхронно-автоматной модели происходит по законам синхронной системы. Время представляется последовательностью дискретных инстентов. В каждый инстент вычисляется реакция системы. Изначально известно текущее состояние системы и значения входных сигналов. Нужно вычислить следующее состояние и значения выходных и внутренних сигналов. Состояние изменяется при переходе в следующий инстент. Правила "вычисления" будут описаны ниже в разделе 2.1.9.

2.1.7. Иерархические обозначения

Необходимо иметь возможность ссылаться на каждый экземпляр автомата и синхронной сети, составляющий синхронную систему. Важно указать на относительное положение экземпляра в модели. Для этого удобно воспользоваться записью с точкой: $A.Q$ — множество состояний автомата A, $a.x_1$ — первый входной сигнал экземпляра автомата a и т.д. Синхронно-автоматная модель имеет иерархическую структуру, поэтому имена компонентов модели также будут формироваться иерархически: $\tilde{r}.r_2$ — вторая подсистема син-

хронно-автоматной системы (предполагаем, что $\tilde{R} \in \mathcal{N}$), $\tilde{r}.r_2.r_1$ — первая подсистема, указанной выше подсистемы (предполагаем, что $\tilde{r}.R_2 \in \mathcal{N}$) и т.п.

Для краткости конкретный экземпляр будем обозначать e_i, зная, что ему соответствует некоторое полное имя: $e_i = \tilde{r}.r_{i_1}.\ldots.r_{i_l}$, где $l \geq 0$. Тип экземпляра e_i обозначим $E_i = \tilde{r}.r_{i_1}.\ldots.r_{i_{l-1}}.R_{i_l}$.

2.1.8. Состояние и сигналы синхронно-автоматной системы

Текущее состояние синхронной системы $\tilde{r}$ определяется состоянием всех ее автоматов: $(e_{i_1}.q, \ldots)$, где e_{i_j} — экземпляр автомата ($E_{i_j} \in \mathcal{A}$), q — текущее состояние экземпляра e_{i_j}.

Входные и выходные сигналы синхронной системы $\tilde{r}$ соответственно: $\tilde{r}.\vec{X}$, $\tilde{r}.\vec{Y}$.

Внутренние сигналы синхронной системы $\tilde{r}$: $\cup_{e_i \neq \tilde{r}}\{e_i.\vec{X}, e_i.\vec{Y}\}$. Каждый внутренний сигнал играет двойную роль. Во-первых, это входной или выходной сигнал соответствующего экземпляра (назовем его e_i), во-вторых, это внутренний сигнал экземпляра синхронной сети, которая содержит экземпляр e_i.

Следующее состояние синхронной системы: $Next(e_{i_1}.q, \ldots)$, где $E_{i_j} \in \mathcal{A}$. Оператор $Next$ используется для обозначения значения сигналов и состояний автоматов в следующий инстент.

2.1.9. Конструктивная поведенческая семантика синхронно-автоматной модели

Семантика синхронно-автоматной программы определяется семантикой соответствующей Esterel-программы, полученной по известному алгоритму. Для облегчения интерпретации модели описана конструктивная поведенче-

ская семантика языка синхронно-автоматного программирования, оперирующая только в терминах C-A модели. Надо заметить, что она не используется в каких-либо инструментах проверки и ее справедливость не доказывается.

Выполнение синхронно-автоматной системы состоит из последовательности реакций. В каждой реакции синхронно-автоматная программа должна определять значения всех выходных сигналов и следующее состояние при заданном значении входных сигналов (и, разумеется, текущем состоянии). Правила вывода следующего состояния и значений сигналов составляют суть семантики C-A программирования. Семантика описывается через процедуру вычисления реакции системы. Решение о синтаксической корректности программы дается на основании применения указанной процедуры "во всех возможных случаях" (ниже будет дано точное определение).

Процедура вычисления реакции системы

Для работы с сигналами воспользуемся трехзначной логикой. Значение сигнала $z \in \{0, 1, \bot\}$, где 0 — ложь, 1 — истина, $\bot$ — неопределенное значение. Состояние экземпляра автомата a может принимать значения из множества $A.Q \cup \{\bot\}$, где $\bot$ снова обозначает неопределенное значение.

В начале процедуры значения сигналов и состояния автоматов устанавливаются следующим образом.

- Текущее состояние системы известно: $(e_{i_1}.q, \ldots)$, где $E_{i_j} \in \mathcal{A}$, $e_{i_j}.q \in E_{i_j}.Q$.
- Определены значения всех входных сигналов системы: $\tilde{r}.x_i \in \{0, 1\}$.
- Значения всех остальных сигналов не определены: $\tilde{r}.y_i = \bot$; $e_i.x_j = \bot$ и $e_i.y_j = \bot$ для всех $e_i \neq \tilde{r}$.

- Следующее состояние системы тоже не определено: $Next(e_i.q) = \perp$, для всех $E_i \in \mathcal{A}$.

Правила изменения значений

1. Для автоматов

   Пусть $E_i \in \mathcal{A}$.

   а. Пусть переход $f \in E_i.\Phi$ такой, что $f = e_i.q \times l' \to e_i.q' \times r'$, где $e_i.q$ — текущее состояние экземпляра e_i, $l' = 1$. Тогда (1) положим $Next(e_i.q) = e_i.q'$; (2) для каждого $e_i.y_j \in E_i.Y$, такого что $e_i.y_j \in r'$, положим $e_i.y_j = 1$.

   б. Пусть $y \in E_i.Y$. Пусть $F = \{f \in E_i.\Phi \mid f = e_i.q \times l' \to e_i.q' \times r', e_i.q$ — текущее состояние экземпляра e_i, $y \in r'\}$. Пусть $\forall f \in F\ l' = 0$. Тогда положим $y = 0$.

   в. Пусть $\forall f \in E_i.\Phi$ такой, что $f = e_i.q \times l' \to e_i.q' \times r'$, где $e_i.q$ — текущее состояние экземпляра e_i, верно $l' = 0$. Тогда положим $Next(e_i.q) = e_i.q$.

2. Для синхронных сетей

   Пусть $E_i \in \mathcal{N}$.

   а. Пусть $z \in E_i.X$, причем $z \neq \perp$. Пусть $\exists g \in E_i.G$ такой, что $z \in g$. Тогда $\forall z' \in g$ такого, что $z' \neq z$, пусть $\xi(z') = \xi(z)$.

   б. Пусть $r \in e_i.\vec{R}$. Пусть $\exists z \in R.Y$ такой, что $z \neq \perp$. Пусть $\exists g \in E_i.G$ такой, что $z \in g$. Тогда (1) $\forall r' \in e_i.\vec{R}$, $\forall z' \in R'.X$ такого, что $z' \in g$, пусть $\xi(z') = \xi(z)$; (2) $\forall z'' \in E_i.Y$ такого, что $z'' \in g$, пусть $\xi(z'') = \xi(z)$.

в. Пусть $z \in E_i.Y$, причем $\neg\exists g \in E_i.G$ такого, что $z \in g$. Тогда пусть $z = 0$.

г. Пусть $r \in e_i.\vec{R}$, $z \in R.X$, причем $\neg\exists g \in E_i.G$ такого, что $z \in g$. Тогда пусть $z = 0$.

Ограничения на применение правил изменения значений

- Правило 1а нельзя применять дважды для одного перехода f.
- Правило 1б нельзя применять дважды для одного сигнала y.
- Правило 1в нельзя применять дважды для одного экземпляра автомата e_i.
- Правила 2а, 2б, 2в, 2г нельзя применять дважды для одного сигнала z.

Определение синтаксической корректности программы

Если в процессе применения правил делается попытка изменить значение сигнала или состояние автомата, значение которого $\neq 0$, то процедура завершается и программа считается синтаксически некорректной.

Процедура определения реакции синхронно-автоматной системы заключается в применении описанных правил в произвольном порядке. Процедура считается завершенной, если были применены все выполнимые правила. Если после завершения процедуры значения некоторых сигналов или состояний автоматов остались неопределенными, то программа синтаксически некорректна. Синхронно-автоматная программа считается синтаксически корректной, если для любого достижимого состояния и любой комбинации входных сигналов процедура вычисления реакции определяет значения всех сигналов и состояний автоматов.

2.2. Форматы данных

Для описания синхронно-автоматной модели необходим специальный язык. Важное требование к языку разработки — он должен быть удобен для набора на компьютере. Система обозначений, предложенная в разделе 2.1, удобна для восприятия, но сложна для набора на компьютере. Синтаксический анализ таких формул оказывается непростым занятием. Возникает необходимость в создании специального языка для описания синхронно-автоматных программ. Такой язык был создан.

2.2.1. Описание в формате XML

Язык XML удачно подходит для решения данной задачи. Элементы являются строительным блоком языка XML. Они разделяют документ на иерархию областей. Одни элементы служат контейнерами для других. У элементов есть атрибуты. Область применения XML достаточно велика. Он содержит большое число компонентов, которые не были использованы для описания синхронно-автоматной программы. К счастью, язык не навязывает использование дополнительных возможностей, мы можем использовать только то, что действительно необходимо.

Автомат описывает одним элементом “automata”. С элементом связаны атрибуты: “name” — имя автомата, “nodes” — число узлов, т.е. число состояний автомата, “inputs” и “outputs” — число выходных и выходных сигналов. Переходы описываются отдельными элементами “edge”, вложенными в автомат. Переход имеет атрибуты: “start” — номер начального состояния, т.е. откуда выходит дуга, “end” — номер конечного состояния, т.е. куда осуществляется переход, “label” — метка перехода, содержит условие перехода и выходное воздействие, разделенное прямым слешем. Нет необходимости описывать состояния автомата. Достаточно только знать их общее число. По соглашению

первое состояние является начальным.

Синхронная сеть описывает отдельным элементом “net”. У нее есть атрибуты “name”, “nodes”, “inputs”, “outputs”. Их значение такое же, как у элемента автомат. “nodes” обозначает число внутренних подсистем. Подсистемы и группы сигналов описываются вложенными элементами “node” и “group” соответственно.

Все автоматы и синхронные сети содержатся в универсальном контейнере “model”. Его единственный атрибут “top” содержит имя главного элемента модели. Это имя автомата или синхронной сети, описанной внутри элемента “model”.

Описание синхронно-автоматной программы в формате XML выглядит достаточно понятно, если соблюдать определенные соглашения о форматировании тегов. Большой плюс XML в том, что для этого языка существуют парсеры для практически всех популярных языков программированию. Задача лексического, синтаксического анализа, построение своего парсера выглядит устрашающее. Разработка программы занимает много времени. А типичные ошибки в таких программах очень сложно находить. Принимая во внимание все эти факторы, становится понятно, насколько желателен выбор языка, для которого парсер уже разработан. Мы можем сэкономить много времени, используя XML, и скорее проверить теоретическую модель на практике. Так и было сделано.

К сожалению, формат XML подходит для разработки только небольших программ. Причина в том, что описание автоматов в виде набора тегов не наглядно. Когда компонентов в системе много, невозможно их всех держать в голове, а восстановление описания автомата по его теговому описанию требует времени. Формат XML не был отвергнут окончательно. Требуется надстройка, которая будет представлять описание автоматов и синхронных сетей в наглядной форме. В то же время, XML-описание удачно подходит для хра-

нения описания программы.

2.2.2. Описание в формате UML

Визуальным представлением автомата является граф переходов. Язык UML наилучшим образом подходит для решения данной задачи. Практически все компоненты синхронно-автоматной модели находят явные аналоги среди элементов языка UML. Для автомата, его состояний, переходов, меток есть соответствующие конструкции UML. Дополнительное описание является излишним. Используются лишь немногие компоненты UML, существующие для описания автоматов. Необходимо соблюдать правила именования автоматов, состояний. Метка перехода должна иметь установленный формат.

Автоматы и синхронные сети в целом представляются классами UML. Входные и выходные сигналы моделируются с помощью портов класса. Порты являются дочерними элементами класса.

Внутренняя структура синхронной сети описывается диаграммой составной структуры (Composite Structure Diagram). На диаграмме изображается класс синхронной сети со всеми портами-сигналами. Внутри класса располагаются синхронные подсистемы. Они обозначаются элементами часть (Part). Каждая подсистема выставляет все порты-сигналы. Порты синхронной сети и ее составных частей соединяются ассоциациями. Необходимо соблюдать правила именования.

Описание синхронно-автоматной модели набором UML-диаграмм наглядно представляет модель. Тем самым демонстрируется одно из самых важных достоинств, которое заявляет switch-технология — наглядность. Разработка программы может быть выполнена в обычном UML-редакторе. Программ такого типа разного качества и условий распространения достаточно много. Ни каких специфических характеристик от редактора не требуется. Использует-

ся небольшое подмножество элементов UML версии 2.0.

2.2.3. Описание на языке Esterel

Языки описания на основе XML и UML являются высокоуровневыми средствами представления синхронно-автоматной программы. Один из них применяется для хранения и быстрой интерпретации программы, другой — как наглядное преставление программы и удобное средство разработки. Синхронно-автоматная программа реализует синхронную систему, ей присущи многие особенности языка Esterel, поэтому не кажется удивительным, что язык Esterel используется для реализации синхронно-автоматной программы. Синхронно-автоматная программа в конечном итоге преобразуется в программу на языке Esterel.

Между синхронно-автоматной моделью и программой на языке Esterel существует четкое соответствие. Автоматы и синхронные сети реализуются отдельными модулями языка Esterel. Этот язык не имеет конструкции, аналогичной интерфейсу. Интерфейсные сигналы компонентов модели отображаются в просто сигналы модуля. По соглашению, входные сигналы называются *x1*, *x2* и т.д., а выходные — *y1*, *y2* и т.д.

Внутреннее состояние автомата представляется набором локальных сигналов. Каждому состоянию выделяется отдельный сигнал, называемый также как состояние, т.е. *q1*, *q2* и т.д. Сигналы являются взаимоисключающими, причем в любом инстенте обязательно присутствует один из сигналов. Тело модуля автомата заключено в бесконечный цикл, выполняющийся каждый инстент. Тело цикла состоит из двух вложенных switch-подобных конструкций. Внешний оператор выбирает текущее состояние, соответствующее единственному присутствующему сигналу из набора *q1*, *q2*, ... Внутренний оператор выбирает исходящий из данного состояния переход. Если активный

переход найден, то в следующем инстенте генерируется сигнал нового состояния, в противном случае в следующем инстенте сохраняется прежний сигнал состояния. При срабатывания перехода генерируются необходимые выходные сигналы.

Реализация синхронной сети, также как и автомата, содержит бесконечный цикл, срабатывающий каждый инстент. Тело цикла содержит последовательность проверок для каждой группы связанных сигналов. Если ведущий сигнал группы присутствует, то генерируются все ведомые сигналы данной группы. Параллельно с основным циклом запускаются модули, соответствующие подсистемам данной синхронной сети.

Файлы с Esterel программой содержит описания всех автоматов и синхронных сетей, использованных в синхронно-автоматной программе. Помимо перечисленных модулей, в файле присутствует еще один модуль, имя которого совпадает с именем файла. Это точка входа в программу. Обычно этот модуль только вызывает корневой элемент модели и перенаправляет сигналы.

По Esterel-программе, сформированной в соответствии с действующими соглашениями, можно без труда восстановить исходную модель. Синхронно-автоматную программу можно писать сразу на языке Esterel, хотя вряд ли в этом есть смысл.

2.3. Верифицируемые свойства. Язык TempEst

Проверка синхронно-автоматных программ в большей мере выполняется средствами языка Esterel и утилитами, созданными для работы с этим языком. Точнее будет сказать, что проверяется Esterel-программа, построенная по синхронно-автоматной модели. Как было описано ранее, за исключением отдельных моментов, существует однозначное соответствие между моделью

и программой, поэтому результаты проверки Esterel-программы очевидным образом интерпретируются в терминах модели.

Задача проверки синхронно-автоматной программы заключается в выявлении разнообразных ошибок, которые могут появиться на этапе разработки. Можно выделить три класса ошибок.

2.3.1. Нарушение формата модели

Описание синхронно-автоматной модели должно подчиняться определенным правилам. Например, в текущей реализации требуется, чтобы имя автомата состояло из заглавной буквы A и следующего за ним номера. Аналогичные соглашения именования существуют для других элементов модели. Или другой пример: в каждую группу сигналов синхронной сети должен входить ровно один ведущий сигнал. Ошибки нарушения формата модели отлавливаются на этапе интерпретации синхронно-автоматной программы, т.е. чтения набора UML-диаграмм или XML-файла. Обнаружение ошибок этого класса выполняется автоматически, проверка не занимает много времени, причины ошибок несложно понять.

2.3.2. Синхронность

К синхронным свойствам модели относятся детерминированность и единственность генерации каждого сигнала. Свойство детерминированности означает, что при любом наборе входных данных, значения выходных сигналов определяются однозначно. Свойство единственности генерации сигнала означает, что в каждом инстенте любой сигнал либо отсутствует, либо генерируется ровно одним оператором. Выполнение двух операторов, генерирующих один и тот же сигнал, является недопустимым. Выполнение свойства единственности генерации каждого сигнала обеспечивается алгоритмом преобра-

зования синхронно-автоматной модели в программу на языке Esterel, тем не менее, оно все равно проверяется. Средства проверки встроены в компилятор языка Esterel.

К ошибкам нарушения синхронности можно отнести сообщение о существовании циклической зависимости между сигналами, которая делает невозможным однозначное определение значения выходных сигналов. Такое может произойти только при неаккуратном использовании циклических зависимостей в синхронных сетях.

2.3.3. Пользовательские свойства

Проверка пользовательских свойств является наиболее привлекательной частью всей верификации, т.к. она позволяет проконтролировать критические характеристики разрабатываемой системы. Под пользовательскими свойствами понимаются свойства, описываемые пользователем, в отличии от предопределенных характеристик модели, таких как соглашения по именованию элементов.

Проверка пользовательских свойств выполняется верификатором Xeve [56, 57] компании Inria. Он создан для проверки программ на языке Esterel. При этом фактически проверяется Esterel-программа, построенная по синхронно-автоматной модели. Опишем технику проверки (рис. 2.3).

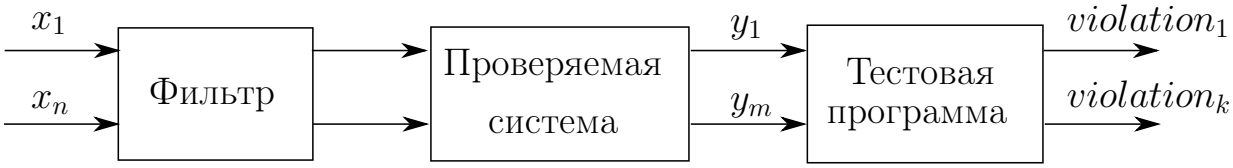

Рис. 2.3. Проверка пользовательских свойств синхронно-автоматной модели

Проверяемая программа представляет собой синхронную систему. У нее есть входные $x_1, \ldots, x_n$ и выходные $y_1, \ldots, y_m$ сигналы. Выходные сигналы проверяемой системы подаются на вход тестовой программы. Тестовая про-

грамма на основании получаемых сигналов может решить, нарушается свойство или нет. Выходных сигналов проверяемой системы не всегда достаточно для описания проверяемого свойства. Можно внедрить тестовую программу в проверяемую систему, или, специально для целей отладки, расширить набор выходных сигналов исходной системы. В любом случае исходная система модифицируется, поэтому встает вопрос об адекватности проверки. Действительно ли результаты проверки для модифицированной системы отвечают на вопросы, поставленные для исходной системы?

Предполагается, что проверяемая система может получать произвольные входные сигналы — проверка осуществляется для всех возможных историй входных сигналов. Иногда необходимо внести дополнительных ограничения на значения входов тестируемой системы. В этом случае пишется дополнительная программа-фильтр, которая отсекает все нежелательные входные сигналы.

До сих пор не было сказано, как тестовая программа сообщает о нарушении свойства. На каждое проверяемое свойство выделяется отдельный выходной сигнал. Этот сигнал генерируется, когда проверяемое свойство нарушается. Задача проверки определенного свойства сводится к вопросу, существует ли история входных сигналов, которая приводит к генерации тревожного сигнала. Для ответа на такие вопросы как раз предназначен верификатор Xeve.

Практически проверка пользовательских свойств реализуется следующим образом. У нас есть программа на языке Esterel, полученная из синхронно-автоматной программы. Пишется код на языке Esterel, который выражает проверяемые свойства. Обычно это несколько модулей. Если необходимо ограничить входные сигналы программы, пишется дополнительная программа-фильтр. Все части кода компилируются в одну программу, которая подается на вход верификатору.

Если набора собственных выходных сигналов синхронной системы недо-

статочно, предусмотрен режим генерации Esterel-программы, который экспортирует все внутренние сигналы, используемые в программе. Мы можем узнать текущее состояние любого автомата, локальные сигналы любой подсистемы, т.е. все, что только есть в программе. Конечно, такой режим изменяет целевую систему, однако результат ее проверки гарантированно будет соответствовать гипотетическому результату проверку исходной системы, если только придерживаться простого правила: использовать дополнительные сигналы только для чтения. При этом мы не должны пытаться генерировать дополнительные сигналы.

2.3.4. TempEst для выражения пользовательских свойств

Проверяемые свойства для программы можно определять не только на языке Esterel. Существует специальный инструмент TempEst [76, 83], предназначенный для описания и интерпретации свойств на языке темпоральной логики. Каждое свойство описывается в отдельном файле. Применяется линейная темпоральная логика с небольшими синтаксическими улучшениями (например, части формулы можно дать имя и повторно использовать его при составлении других формул). Описание формулы транслируется в отдельный модуль языка Esterel, который должен запускаться параллельно основной программе.

Процедура проверки пользовательских свойств с применением инструмента TempEst практически ни чем не отличается от аналогичной проверки, когда свойства выражаются сразу на языке Esterel. Использование языка линейной темпоральной логики, однако, более четко выражает проверяемое свойство, и, возможно, более удобно.

Основная часть языка линейной темпоральной логики TempEst

Опишем язык LTL, используемый в TempEst [75, 77].

Пусть Sig будет фиксированный алфавит используемых сигналов. Существуют специальные сигналы $TRUE$ и $FIRST$. Первый из них присутствует в любом инстенте, второй — только в первом инстенте.

Пусть $\mathcal{PAST}$ — класс темпоральных формул, ссылающихся на прошедшее время. Формулы множества $\mathcal{PAST}$ являются формулами состояния, т.е. значение формулы может быть определено для каждого состояния любого пути. Каждый отдельный сигнал алфавита Sig и каждая константа являются формулой. Если $p, q \in \mathcal{PAST}$, тогда $p \vee q, p \wedge q, \neg p, p \rightarrow q \in \mathcal{PAST}$. Логические операторы $\vee$, $\wedge$, $\neg$, $\rightarrow$ имеют обычный смысл. Язык Esterel не позволяет обращаться к будущему, поэтому используются темпоральные операторы прошедшего времени. Если $p, q \in \mathcal{PAST}$, то $\mathcal{P}\,p, p\,\mathcal{S}\,q, \mathcal{G}\,p, \mathcal{O}\,p, p\,\mathcal{B}\,q \in \mathcal{PAST}$. Операторы имеют традиционное значение. Однако, поскольку при описании линейных темпоральных логик обычно используются операторы будущего времени, ниже приводится краткое описание всех темпоральных операторов.

$\mathcal{P}\,p$

Значение данной формулы равно значение формулы p в предыдущий инстент. В первый инстент значение формулы равно *false*.

$p\,\mathcal{S}\,q$

Значение формулы — *true*, когда формула q принимает значение *true*. Формула сохраняет значение *true*, после того как q — *true*, пока p остается *true*. Во всех остальных случаях значение формулы — *false*.

$\mathcal{G}\,p$

Формула принимает значение *true*, если p — *true* непрерывно с первого инстента. Значение *false* получается начиная с момента, когда p обращается в *false*.

$\mathcal{O}\, p$

Значение формулы — *true*, если p в текущий инстент или когда-то в прошлом принимало значение *true*.

$p\, \mathcal{B}\, q$

Формула определяется через ранее определенные формулы: $p\, \mathcal{B}\, q \equiv (p\, \mathcal{S}\, q) \vee \mathcal{G}\, p$. Значение формулы — *true*, если p непрерывно *true*, начиная с первого инстента, или с последнего инстента, где q было *true*.

Еще одно множество формул обозначается $\mathcal{SAFE}$. Формулы множества $\mathcal{SAFE}$ являются формулами пути. Это также формулы безопасности. Если $p \in \mathcal{PAST}$, то $\mathcal{A}\, p \in \mathcal{SAFE}$. Здесь $\mathcal{A}$ — квантор пути. Формула $\mathcal{A}\, p$ истина тогда и только тогда, когда формула p истина в каждом состоянии пути.

Дополнительные операторы TempEst

TempEst предлагает также несколько дополнительных формул.

$\mathcal{R}(p, q, b, s)$

Здесь $p, q \in \mathcal{PAST}$ — формулы состояния, b — целое неотрицательное число, s — один из сигналов, например, *tick*. Эта формула выражает задержку реакции (bounded response) системы на некоторое воздействие. В ответ на событие p система должна отреагировать событием q в течении не более чем b отсчетов времени. Отсчетом времени считается получение сигнала s. В данном определении под событием понимается инстент, в котором какая-то формула принимает значение *true*, p или q. Вся формула принимает значение *false* в тот инстент, когда приходит b-ый сигнал s, а q имеет значение *false*. При этом, с момента отсчета времени, т.е. когда p — *true*, q принимала только значения *false*. После начала отсчета времени появление других инстентов, когда p — *true*, не играет ни какой роли. Они не изменяют текущий отсчет времени и не начинают дополнительный отсчет времени.

Реализацию данной формулы на языке Esterel можно описать так. Из инстента в инстент мы ждем, когда формула p обратится в *true*. Если в этот же инстент формула q обращается в *true*, то продолжает ждать дальше, иначе входим в состояния ожидания реакции. В данном состоянии мы пребываем до тех пор, пока либо q обратится в *true*, либо мы пронаблюдаем b сигналов s. В первом случае мы выходим из состояния ожидания нормально, т.е. вся формула остается *true*. Во втором случае, формула обращается в *false* на один инстент. Если оба условия выполняются одновременно, мы поступаем как в первом случае.

Практически данная формула представляет интерес, только когда она вкладывается в формулу $\mathcal{A}$, т.е. нам интересно знать, нарушается ли когда-нибудь условие задержки или нет.

$\mathcal{H}(p, q, b, s)$

Здесь снова $p, q \in \mathcal{PAST}$ — формулы состояния, b — целое неотрицательное число, s — один из сигналов, например, *tick*. Эта формула выражает связь между сигналом s и формулой q. Алгоритм, рассчитывающий эту формулу, можно описать так. Ждем, когда p примет значение *true* и будет сохранять его в течение b сигналов s. Затем переходим в специальное состояние, когда приход сигнала s сопоставляется с значением формулы q. Данное состояние заканчивается тогда, когда формула p перестает быть *true*. Каждый инстент, когда приходит сигнал s, необходимо, что бы q обращалась в *true*. Если этого не происходит, вся формула обращается в *false*.

Точно так же, как и предыдущая формула, эта представляет практический интерес, когда находится под действием квантора $\mathcal{A}$.

Особенности языка TempEst

Для описания формул в TempEst на самом деле используются ключевые слова, а не математические символы (таблица 2.1). Например, логическое "или" можно записать либо Or, либо $|$, или оператор $\mathcal{B}$ записывается словом $BackTo$. Язык для TempEst позволяет, а в некоторых случаях настаивает на использовании дополнительных идентификаторов для обозначения частей формул.

Чтобы обозначить формулу при помощи идентификатора id используется синтаксис:

$$id ::= <\text{формула}>.$$

Все это синтаксические удобства. Далее в формулах будут попрежнему использоваться математические символы, т.к. они предлагают более компактную запись. С другой стороны, мы будем прибегать к дополнительным идентификаторам, где необходимо.

Таблица 2.1. Соответствие символьных обозначений, описаний на естественном языке и формул TempEst

Символьное обозначение	Описание на естественном языке	Формула TempEst
$p \vee q$	Or	$p\ Or\ q$
		$p \mid q$
$p \wedge q$	And	$p\ And\ q$
		$p\ \&\ q$
$\neg p$	Not	$Not\ p$
$p \rightarrow q$	$Implies$	$p\ Implies\ q$
		p -> q

$\mathcal{P}\,p$	*Previous*	$Previous\,p$
$p\,\mathcal{S}\,q$	*Since*	$p\,Since\,q$
$\mathcal{G}\,p$	*Past, has-always-been*	$AllPast\,p$
$\mathcal{O}\,p$	*Once*	$Once\,p$
$p\,\mathcal{B}\,q$	*Back-To*	$p\,BackTo\,q$
$\mathcal{A}\,p$	*Always*	$Always\{p\}$
$\mathcal{R}(p,q,b,s)$	*Responds-To*	$\{q\}\ RespondsTo\ \{p\}\ In\ b\ s$
$\mathcal{H}(p,q,b,s)$	*Holding-Yields*	$\{p\}\ Holding\ b\ s\ Yields\ \{q\}$

2.4. Структура среды разработки

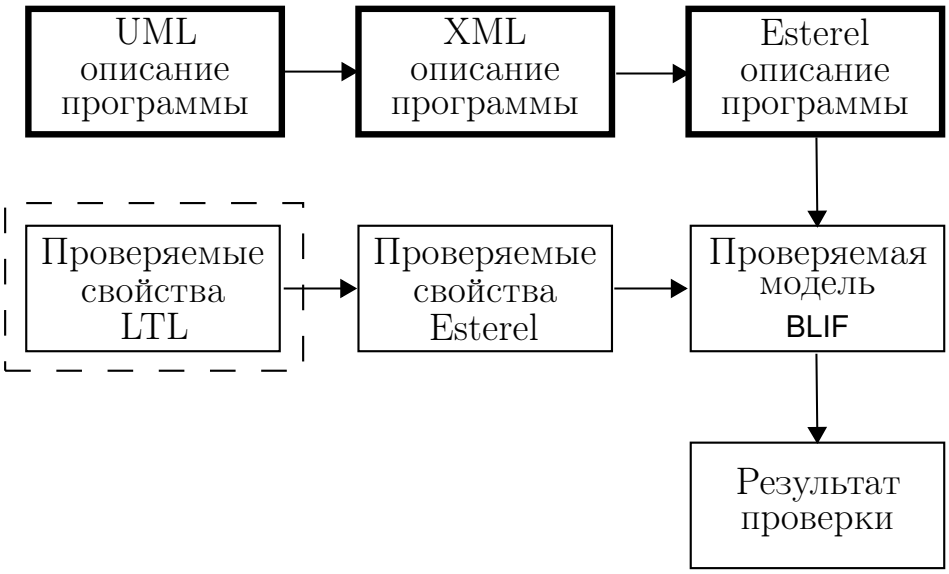

Рис. 2.4. Среда разработки синхронно-автоматных программ. Диаграмма данных

Структуру среды разработки удобно представить с помощью двух диаграмм. На рис. 2.4 показаны пути преобразования данных в процессе работы. Рис. 2.5 содержит соответствующие инструменты, используемые для создания или преобразования данных с рис. 2.4. Каждой стрелке, символизирующей операцию преобразования данных, на рис. 2.4 соответствует прямоугольник инструмента на рис.2.5.

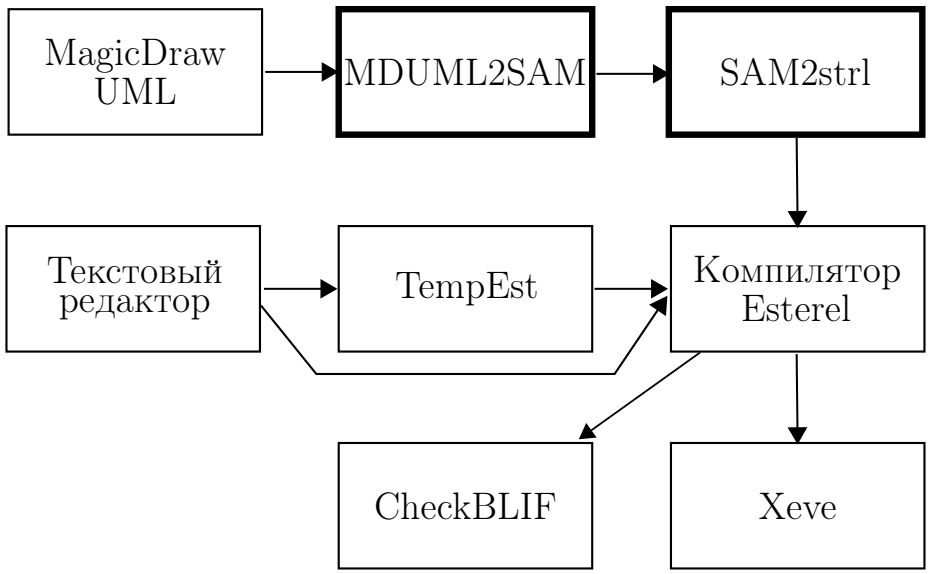

Рис. 2.5. Среда разработки синхронно-автоматных программ. Диаграмма инструментов

Область рисунка, окруженная пунктирной рамкой, является опциональной. Так, на рис. 2.4 компонент «Проверяемые свойства LTL» является необязательным. Проверяемые свойства можно определять сразу на языке Esterel (прямоугольник «Проверяемые свойства Esterel»).

Прямоугольники с жирной рамкой отмечают компоненты, разработанные автором. Все остальные компоненты, соответственно, были заимствованы. Более подробно, что было сделано по каждому компоненту, будет рассказано ниже.

Компилятор языка Esterel, верификатор Xeve и другие инструменты, обеспечивающие основную функцию верификации, спроектированы для работы на нескольких платформах. Дистрибутив компилятора предлагается для Windows NT, Linux, Sun Solaris, IBM AIX, Dec OSF1. Xeve поддерживает платформы: Linux, Sun Solaris, Dec Alpha OSF. В связи с этим при разработке программной среды большое внимание уделалось переносимости. MagicDraw UML, выбранный в качестве UML-редактора, работает на Java-платформе. Скрипты MDUML2SAM, SAM2strl написаны на языке Perl.

Процесс разработки синхронно-автоматных программ является сложным

многоэтапным процессом. Последующее описание преследует две цели. Во-первых, дается обзорное описания процесса разработки синхронно-автоматных программ. Во-вторых, акцентируется внимание на новых инструментах и технологиях, составляющих личный вклад автора.

2.4.1. UML-редактор

Разработка программы начинается в UML-редакторе MagicDraw UML. Его внешний вид приведен на рис. А.1. Используются стандартные возможности инструмента. Мы ограничиваем себя определенным набором диаграмм UML. Выполняется обычное UML-моделирование. Однако, для того чтобы результирующий набор диаграмм воплотился в синхронно-автоматную программу, необходимо соблюдать определенные правила именования компонентов модели и придерживаться соглашений об использования элементов. На данном этапе отсутствует контроль за выполнением правил. Ошибки обнаружат себя на более поздних этапах разработки.

Использование стандартного редактора имеет то преимущество, что нам не пришлось его разрабатывать. Это, конечно, сэкономило много времени. Стандартный редактор также содержит множество возможностей, которые облегчают, ускоряют разработку, например, функции копирования/вставки, отмены. Крайне удобна функция комментирования практически любого элемента модели.

К недостаткам данного решения можно отнести сравнительную тяжеловесность редактора и отсутствие специфических функций. UML-редактор задумывался как универсальный инструмент, поэтому в него заложено гораздо больше возможность, чем реально необходимо для решения нашей задачи. В то же время, в редакторе отсутствуют специфические функции, которые могли быть полезны при разработке именно синхронно-автоматной програм-

мы.

2.4.2. Преобразование из UML-описания в XML-описание

Следующим шагом разработки синхронно-автоматной программы является преобразование набора UML-диаграмм в файл XML. В этот момент выполняется минимальная проверка формата модели. Более тщательная проверка будет выполнена позже. Многие ошибки не являются критическими, например, мы можем без опасений проигнорировать неизвестный тип диаграммы. Существенные нарушения формата модели прерываются процесс генерации. Распознанная модель сохраняется в XML-файле.

Полная проверка модели намеренно откладывается до более поздних этапов. Предполагается, что в будущем можно будет использовать другую программу для рисования диаграмм. Поэтому функции проверки желательно сконцентрировать в одном месте.

Данный этап полностью автоматизирован. Разработан скрипт MDUML2SAM. Он на входе получает файл с UML-диаграммами, а на выходе выдает файл XML. Синтаксис команды приведен в листинге А.5, а пример ее использования приведен в листинге А.7.

2.4.3. Построение Esterel-программы по XML-описанию

Для чтения XML-описания модели используется готовый XML-парсер[1]. При интерпретации модели приходится иметь дело с программными структурами, а не со строками, которые нужно разбирать. Применение готового парсера существенно упрощает программу преобразования. Выполняется полная, насколько это возможно, проверка модели. Если программа прошла проверку, генерируется Esterel-программа. Программа преобразования име-

[1]Пакет Perl XML::Parser

ет несколько режимов генерации. В специальном режиме, предназначенном для проверки модели, экспортируются все внутренние сигналы синхронно-автоматной программы.

Данный этап полностью автоматизирован. Разработан скрип SAM2strl. Он на входе получает файл в формате XML, а на выходе генерирует файл .strl с описанием программы на языке Esterel. Синтаксис команды приведен в листинге А.6, пример ее использования — в листинге А.8. Фрагмент скрипта приведен на листинге А.3.

2.4.4. Проверка синхронных свойств

Файл, созданный на предыдущем этапе, содержит полноценную программу. Вероятно, что в ней присутствуют ошибки. Процедура генерации Esterel-программы практически гарантирует отсутствие синтаксических ошибок. Первая проверка, которую мы выполняем — обычная компиляция программы. Сложно представить случай, когда компиляция будет провалена. Вторая проверка — это проверка на конструктивность. Типичная ошибка, которая может быть обнаружена на этом этапе — существование циклической зависимости между сигналами, которая не позволяет однозначно проинтерпретировать программу. Компилятор Esterel в специальном режиме осуществляет такую проверку.

Реальная проблема, с которой сталкивается разработчик — обнаруженную циклическую зависимость сложно осмыслить. Необходимо основательное понимание синхронных основ языка Esterel.

Данный этап полностью автоматизирован, единственное, что может представлять сложность — это интерпретация найденных ошибок. На листинге А.10 показан пример вызова утилиты Esterel.

2.4.5. Подготовка пользовательских свойств для проверки

Свойства, которые нужно проверить, кодируются отдельными подпрограммами на языке Esterel. Каждое свойство удобно инкапсулировать в отдельном модуле или иерархии модулей. Как правило, модули свойств запускаются параллельно с основной программой. Программа для проверки собирается из нескольких файлов свойств и файла основной программы. Выходной интерфейс программы полностью меняется. Набор выходных сигналов, как правило, состоит только из тревожных сигналов, каждый из которых отвечает за отдельное свойство.

Данный этап требует ручной работы. Единственное, что можно сделать для упрощения разработки — это рекомендовать организацию файлов и модулей для описания свойств.

2.4.6. Проверка пользовательских свойств верификатором Xeve

Верификатор Xeve требует специальный формат входного файла — .blif (Berkeley Logical Interchange Format). Его производит компилятор языка Esterel. Перед использованием верификатора имеет смысл выполнить проверку на конструктивность. Кроме того, сама компиляция может выявить ошибки, допущенные при написании свойств.

Проверка программы верификатором Xeve может выполняться в консольном или графическом режимах. В консольном режиме все параметры указываются через командную строку. Необходимо задать имена выходных сигналов, для которых нужно выполнять проверку. Все остальные параметры не меняются. В графическом режиме те же сигналы требуется отметить в списке. Графический режим не предлагает ни каких дополнительных возможностей, которых нет в консольном режиме, поэтому последний использовать предпочтительнее.

На рис. А.2, А.3 показан внешний вид графической версии верификатора Xeve. Листинг А.11 демонстрирует вызов консольного варианта верификатора CheckBLIF.

Если проверяемое свойство выполняется, то на этом проверку можно закончить. Если проверяемое свойство нарушается, то верификатор генерирует последовательность входных сигналов, приводящих к ошибке. Эту последовательность можно анализировать либо вручную, либо при помощи одного из двух отладчиков Esterel-программ.

Данный этап отчасти автоматизирован. Компиляция и верификация не требуют действий со стороны пользователя, за исключением того, что требуется указать список проверяемых сигналов. Обнаружение ошибки влечет за собой не всегда легкую работу по выяснению ее причин и устранению.

Глава 3

Примеры

Цель данной главы — исследовать применение синхронно-автоматного программирования на реальных примерах. Во-первых, важно понять, насколько удобно синхронно-автоматное программирование для решения задач логического управления. Известно, что такие задачи могут быть решены автоматным подходом, но синхронно-автоматное программирование является модификацией автоматного программирования, поэтому важно знать, как изменились выразительные возможности технологии. Во-вторых, синхронно-автоматное программирование предлагает способ верификации программ. Необходимо исследовать возможности верификации, оценить, насколько удачно эта функция способствует построению более качественных программ. Основной критерий оценки состоит в числе обнаруженных ошибок.

3.1. Пример 1: Арбитр шины

3.1.1. Введение

Пример демонстрирует использование синхронно-автоматного программирования для реализации простого контроллера шины. Пример взят из руководства по верификатору языка Esterel программы Xeve. Главная идея решения связана с особенностями синхронного подхода. Пример интересен с точки зрения декомпозиции синхронной системы на блоки.

3.1.2. Описание задачи

Арбитр шины управляет доступом к разделяемому устройству. Есть несколько пользователей, но только один из них может обращаться к шине в

один момент времени. С каждым пользователем связывается сигнал request. В данном примере их три, однако система легко масштабируется для случая произвольного числа пользователей. Каждому сигналу request соответствует сигнал ask, подтверждающий получения доступа для данного запроса.

Необходимо разработать систему логического управления с тремя входными сигналами request и тремя выходными сигналами ask. Система должна обладать ниже описанными свойствами. В каждый момент времени должно генерироваться не более одного сигнала подтверждения запроса. Если только один пользователь запрашивает доступ к шине, то доступ безусловно предоставляется. Если несколько пользователей одновременно запрашивают доступ к шине, то доступ должен получить только один из них. В случае конкуренции доступ должен предоставляться по справедливому алгоритму. Если какой-то пользователь запрашивает доступ постоянно, то доступ должен быть предоставлен не позднее чем через два инстента (в общем случае через число инстентов, равное числу пользователей минус один).

Алгоритм, решающий данную задачу, указан в первоисточнике [56]. Новизна данной работы в том, что решение адаптировано для синхронно-автоматного подхода.

3.1.3. Реализация

Для решения указанной задачи было использовано четыре автомата и две синхронные сети. Таблицы с описанием основных компонентов системы приведены в приложении Б.

Сеть Net2 представляет всю синхронную систему. Net2 имеет три входных сигнала, через которые пользователи запрашивают доступ к шине, и три выходных сигнала, которые подтверждают предоставление доступа. Внутри Net2 состоит из четырех подсистем. Три подсистемы (Node1, Node2, Node3)

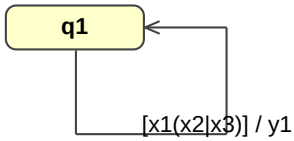

Рис. 3.1. Диаграмма автомата A2

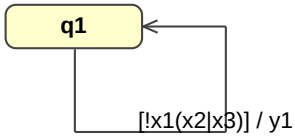

Рис. 3.2. Диаграмма автомата A3

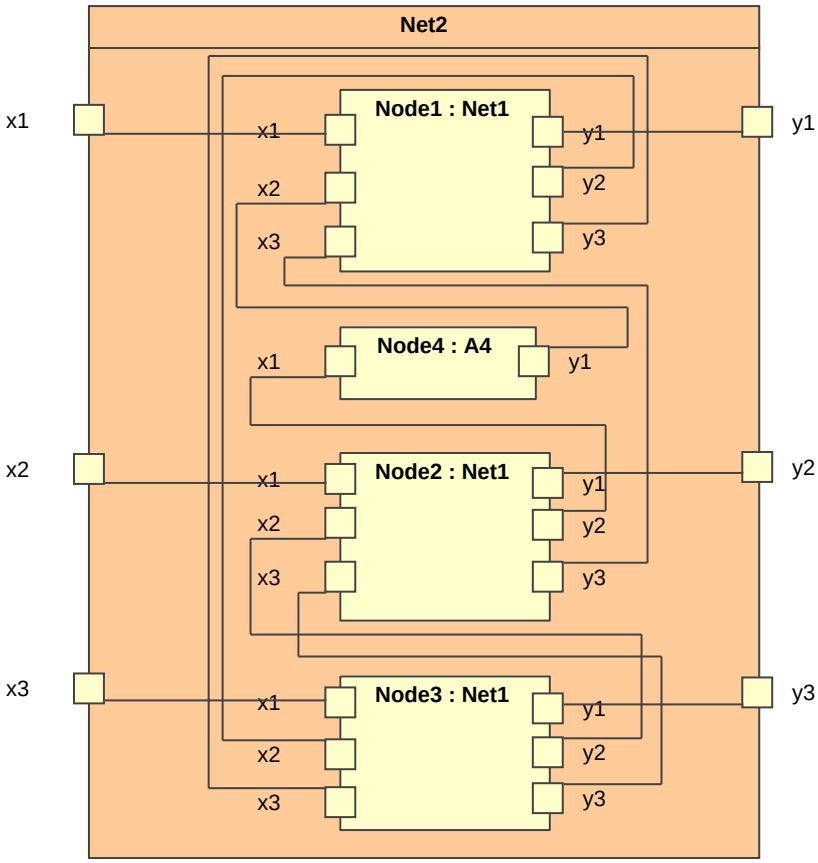

Рис. 3.3. Диаграмма синхронной сети Net2

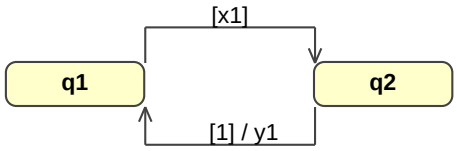

Рис. 3.4. Диаграмма автомата A1

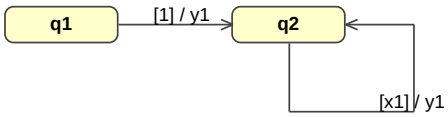

Рис. 3.5. Диаграмма автомата A4

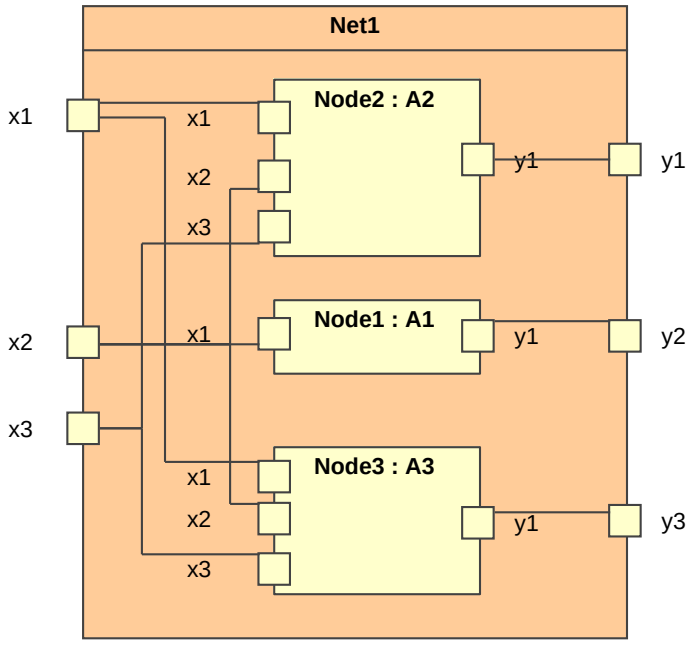

Рис. 3.6. Диаграмма синхронной сети Net1

управляют предоставлением доступа каждый для своего пользователя. Еще одна подсистема (Node4) применяется для инициализации.

Подсистемы Node1, Node2, Node3 реализуются одной и той же синхронной сетью Net1. Net1 имеет входной сигнал запроса доступа (x1) и выходной сигнал предоставления доступа (y1). Для разрешения конфликтных ситуаций вводится понятие токена. Токен ходит по кругу от одного узла к другому. Узел, содержащий токен в данный инстент, имеет больший приоритет, чем все остальные. Если такой узел получает запрос на предоставления доступа к шине, доступ предоставляется именно этому узлу. Для передачи токена используются сигналы x2 и y2.

Если узел с токеном не получает запрос, но запрос получает другой узел, то право предоставления доступа к шине нужно передать. Для этого используется сигнал grand (сигналы x3 и y3). Подсистемы Node1, Node2, Node3 соединяются к кольцо через сигналы x3 и y3. Если узел имеет либо токен, либо сигнал grand, но сам не получает запрос, то он передает сигнал grand следующему по очереди узлу. В результате доступ к шине получает либо узел с токеном, либо ближайший к нему узел в порядке соединения, для которого поступил запрос.

Узел Node4, реализуемый автоматом A4, запускает токен в первый инстент. В первом инстенте токен получает узел Node1. Во всех остальных инстентах Node1 прозрачно пропускает через себя сигнал, идущий от узла Node2.

Сеть Net1 состоит из трех подсистем, каждая из которых занимается вычислением своего выходного сигнала. Node1 (автомат A1) вычисляет сигнал токена (y2), Node2 (автомат A2) вычисляет сигнал подтверждения предоставления доступа (y1), Node3 (автомат A3) вычисляет сигнал grand (y3). Значение каждого выходного сигнала вычисляется независимо. Автоматы A1, A2, A3 очень просты, их поведение должно быть ясно из приведенных диаграмм.

3.1.4. Проверка пользовательских свойств

Были проверены следующие свойства.

Замечание: если указываются краткие имена сигналов, например, x1, y1 вместо полных: Net2.x1, Net2.y1, считаем, что сигналы относятся к главному объекту модели, т.е. эти интерфейсные сигналы всей системы.

- Каждый инстент генерируется не более одного сигнала предоставления доступа (т.е. сигналы y1, y2, y3 являются взаимоисключающими).

  Формула LTL:

$$\mathcal{A}((y1 \wedge y2) \vee (y2 \wedge y3) \vee (y3 \wedge y1))$$

- Если только один пользователь посылает запрос, то этот запрос будет обязательно удовлетворен.

  Формула LTL:

$$\begin{aligned} X1 ::= \ & (x1 \wedge (\neg x2) \wedge (\neg x3)) \rightarrow (y1 \wedge (\neg y2) \wedge (\neg y3)) \\ X2 ::= \ & ((\neg x1) \wedge x2 \wedge (\neg x3)) \rightarrow ((\neg y1) \wedge y2 \wedge (\neg y3)) \\ X3 ::= \ & ((\neg x1) \wedge (\neg x2) \wedge x3) \rightarrow ((\neg y1) \wedge (\neg y2) \wedge y3) \\ & \mathcal{A}(X1 \wedge X2 \wedge X3) \end{aligned}$$

- Если одновременно запрос посылают более одного пользователя, один из них обязательно получит доступ. Проверить для всех возможных комбинаций из двух и трех сигналов.

  Формула LTL:

$$\begin{aligned} S1 ::= \ & (x1 \wedge x2 \wedge (\neg x3)) \rightarrow (y1 \vee y2) \\ S2 ::= \ & ((\neg x1) \wedge x2 \wedge x3) \rightarrow (y2 \vee y3) \\ S3 ::= \ & (x1 \wedge (\neg x2) \wedge x3) \rightarrow (y1 \vee y3) \\ S4 ::= \ & (x1 \wedge x2 \wedge x3) \rightarrow (y1 \vee y2 \vee y3) \\ & \mathcal{A}(S1 \wedge S2 \wedge S3 \wedge S4) \end{aligned}$$

- Если некоторый пользователь постоянно запрашивает ресурс, доступ будет предоставлен не позднее чем через два инстента — доступ будет предоставлен либо в том инстенте, когда получен запрос, либо в следующем, либо в следующем за следующим инстентом. Проверить для всех пользователей.

  Формула LTL:

$$\begin{aligned} S1 ::= &\ \mathcal{R}(x1, y1 \vee (\neg x1), 2, tick) \\ S2 ::= &\ \mathcal{R}(x2, y2 \vee (\neg x2), 2, tick) \\ S3 ::= &\ \mathcal{R}(x3, y3 \vee (\neg x3), 2, tick) \\ &\ \mathcal{A}(S1 \wedge S2 \wedge S3) \end{aligned}$$

Результаты проверки подтверждают, что все перечисленные свойства выполняются.

3.2. Пример 2: Часы-будильник

3.2.1. Введение

Пример демонстрирует возможности синхронно-автоматного программирования и подходы к верификации программ. Часть логики программы реализуется внешними устройствами. Синхронно-автоматная программа и аппаратно-реализуемая часть работают в тесном взаимодействии.

3.2.2. Описание задачи

Пример моделирует работу часов с будильником. Подобные модели часов широко распространены и, вероятно, знакомы читателю. Внешний вид устройства схематично показан на рис. 3.7. Центральное место занимает жидкокристаллический экран, служащий для отображения времени и других значков. Правый верхний угол отдан динамику. С его помощью часы издают

различные мелодии, подают звуковые сигналы, а в некоторых моделях даже сообщают текущее время. Нижнюю часть панели занимают кнопки “Mode”, “Hour”, “Min”, служащие для выбора режима, часа и минуты соответственно. Впрочем, последние две кнопки могут играть разную роль в зависимости от режима.

В рассматриваемом примере устройство обладает такими возможностями: отображение текущего времени, установка времени звонка, выбор мелодии звонка (одна из трех), включение/выключение звонка, установка/отмена режима ежечасного оповещения.

Заметим, что программа моделирует только процесс настройки будильника, задачи отсчета времени, включение звонка и ежечасных оповещений не рассматриваются.

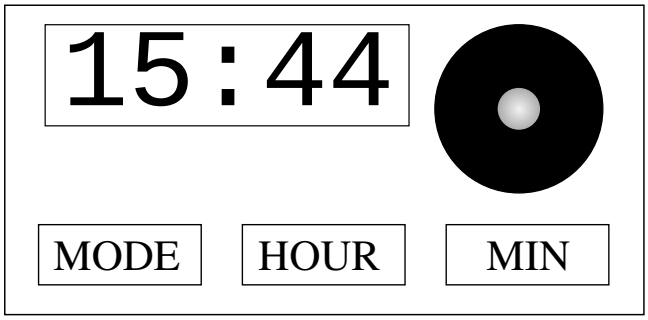

Рис. 3.7. Внешний вид часов с будильником

3.2.3. Реализация

Программа часов с будильником состоит из четырех автоматов и одной синхронной сети. Синхронная сеть Net1 представляет всю систему в целом, это главный объект модели. Автоматы A1–A4 каждый решают свою подзадачу. Они соединяются в рамках синхронной сети Net1. Таблицы с описанием основных компонентов системы приведены в приложении B.

Входные сигналы сети Net1 состоят из сигналов трех кнопок и одного

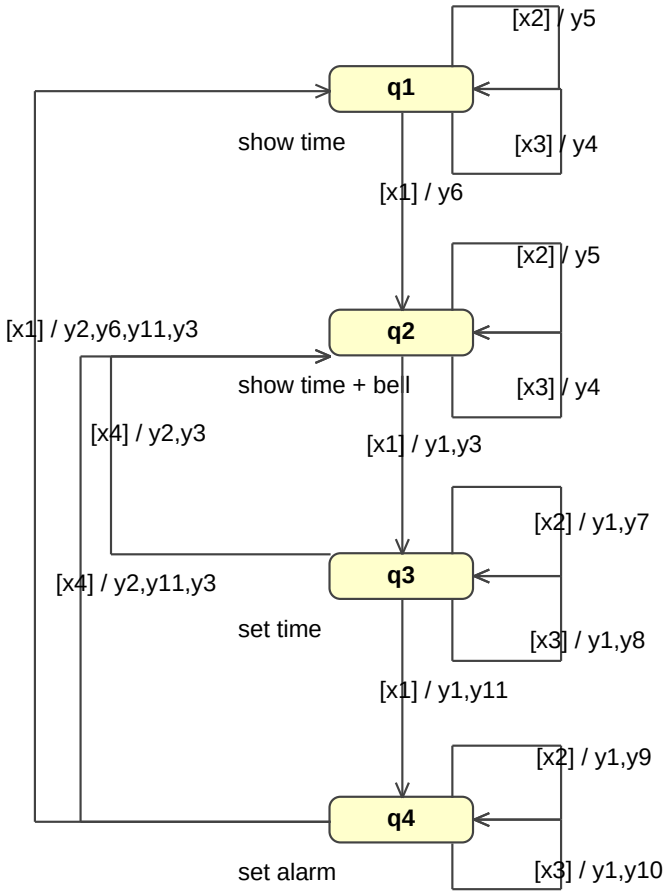

Рис. 3.8. Диаграмма автомата A1

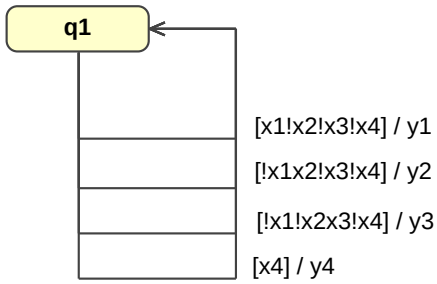

Рис. 3.9. Диаграмма автомата A2

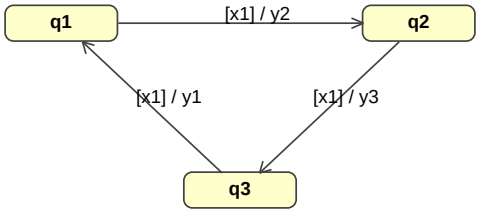

Рис. 3.10. Диаграмма автомата A3

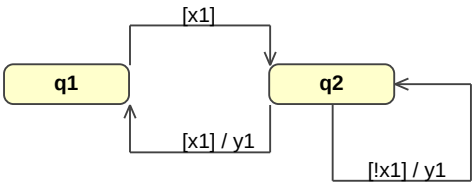

Рис. 3.11. Диаграмма автомата A4

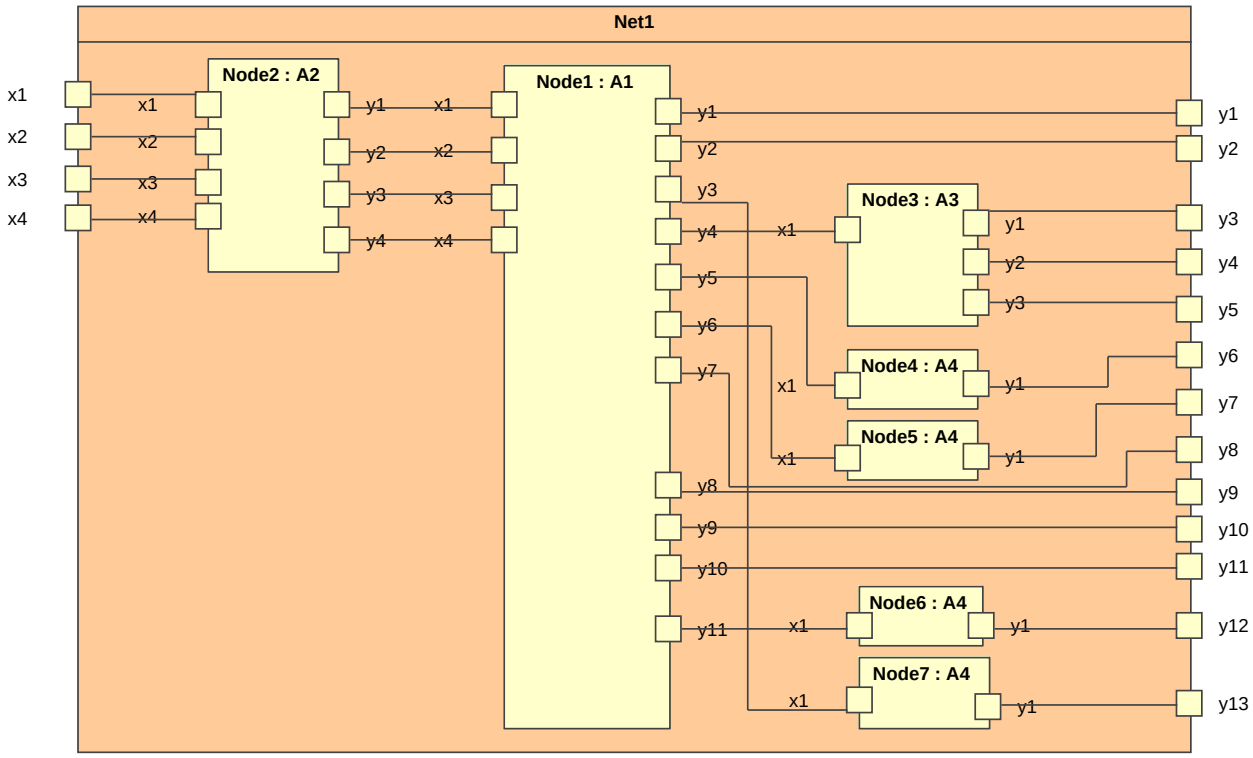

Рис. 3.12. Диаграмма синхронной сети Net1

сигнала таймера. Выходные сигналы сети Net1 выполняют разнообразные функции: вкл./выкл. таймера, включение мелодии будильника, переключения режимов отображения, установка текущего времени и времени будильника. Сама сеть, по сути, ни чего не делает, она только соединяет свои компоненты друг с другом, некоторые сигналы выводятся на интерфейс.

Автомат A1 реализует основную функциональность программы. Он имеет четыре состояния: “время”, “время с установленным будильником”, “настройка текущего времени”, “настройка будильника”. Входные сигналы такие же, как у сети Net1, но взаимоисключающие. В каждый инстент может прийти не более одного входного сигнала. Такое решение было принято для упрощения логики работы устройства. Большинство случаев, когда одновременно приходят несколько сигналов, относятся к ошибочным действиям пользователя. Например, одновременно было нажато две кнопки. Чтобы не усложнять алгоритм управления, специально для фильтрации сигналов был создан от-

дельный модуль A2.

Выходные сигналы A1 разнообразны и приведены таблице B.1. Их особенность состоит в том, что для переключения различных режимов используются кратковременные сигналы, т.е. сигналы, которые длятся один инстент. Например, если нам надо переключить дисплей в режим мигания, мы посылаем соответствующий сигнал, а не поддерживаем этот сигнал все время, пока требуется, чтобы дисплей мигал. Интерфейс Net1 предполагает противоположное поведение. Специально для решения это задачи был разработан модуль A4.

Внутреннее устройство автомата A1 должно быть ясно из рис. 3.8.

Задача автомата A2 в фильтрации входных сигналов автомата A1. Выходные сигналы A2 соответствуют входным сигналам A1, а входные сигналы A2 — входным сигналам Net1. A1 требует, чтобы входные сигналы были взаимоисключающими. Наивысший приоритет имеет сигнал от таймера. Он пропускается всегда, подавляя все остальные сигналы. Если сигнала от таймера нет, пропускаются сигналы кнопок, но только в том случае, если нажата только одна из кнопок. Если нажато несколько кнопок одновременно (сигнала от таймера нет), все сигналы отфильтровываются.

Автомат A3 хранит текущую мелодию звонка, а также запускает мелодию на проигрывание при смене состояния. В реальном будильнике нам обязательно бы потребовался сигнал для проигрывания текущей мелодии, генерируемый в тот момент, когда подошло время звонка. В модели такой сигнал не предусмотрен, т.к. все, что нам требуется — это выбор и хранение мелодии звонка. Поэтому проигрывание связано с переключением состояния.

Автомат A4 используется для реализации простейшего переключателя на два положения. У него есть один входной и один выходной сигнал. Выходной сигнал сохраняет свое значение из инстента в инстент, пока не придет переключающий входной сигнал. Входной сигнал изменяет значение выходного

сигнала на противоположное и фиксирует это значение вплоть до следующего переключения.

Автомат A4 был создан для согласования отдельных сигналов интерфейса автомата A1 с сигналами интерфейса синхронной сети Net1. Синхронная сеть Net1 использует несколько экземпляров автомата A4, таким образом демонстрируется многократное использование разработанного кода.

Работа программы должны быть ясна из приведенных диаграмм и таблиц описания.

3.2.4. Проверка пользовательских свойств

Программа моделирования часов с будильником содержит несколько допущений, выполнение которых не очевидно из кода программы. Очень удачно, что для их проверки мы можем воспользоваться специально разработанными средствами верификации синхронно-автоматных программ. Можно рассмотреть такие свойства.

- Сигналы запуска (y1) и остановки (y2) таймера никогда не генерируются в одном инстенте.

  Формула LTL:

  $$\mathcal{A}(\neg(y1 \wedge y2))$$

- Входные сигналы автомата A1 (x1, x2, x3, x4) являются взаимоисключающими.

Формула LTL:

$$\begin{aligned} X1 &::= Node1.x1 \\ X2 &::= Node1.x2 \\ X3 &::= Node1.x3 \\ X4 &::= Node1.x4 \\ S &::= (X1 \wedge X2) \vee (X1 \wedge X3) \vee (X1 \wedge X4) \vee (X2 \wedge X3) \vee \\ &\quad (X2 \wedge X4) \vee (X3 \wedge X4) \\ &\quad \mathcal{A}(\neg S) \end{aligned}$$

- Сигналы проигрывания мелодии (y3, y4, y5) являются взаимоисключающими.

  Формула LTL:

$$\begin{aligned} S &::= (y3 \wedge y4) \vee (y3 \wedge y5) \vee (y4 \wedge y5) \\ &\quad \mathcal{A}(\neg S) \end{aligned}$$

- Значок установленного будильника (y7 = 1) не показывается только в состоянии q1 автомата A1.

  Формула LTL:

$$\begin{aligned} q1 &::= Node1.q1 \\ q2 &::= Node1.q2 \\ q3 &::= Node1.q3 \\ q4 &::= Node1.q4 \\ S &::= (q1 \wedge \neg y7) \vee ((q2 \vee q3 \vee q4) \wedge y7) \\ &\quad \mathcal{A}(S) \end{aligned}$$

- Таймер выключен, только когда автомат A1 находится в состояниях q3 и q4.

Формула LTL:

$$
\begin{aligned}
q1 &::= Node1.q1 \\
q2 &::= Node1.q2 \\
q3 &::= Node1.q3 \\
q4 &::= Node1.q4 \\
StateOn &::= (\neg y2)\, \mathcal{S}\, y1 \\
S &::= ((q1 \vee q2) \wedge (\neg\, \mathcal{P}\, StateOn)) \vee ((q3 \vee q4) \wedge \mathcal{P}\, StateOn) \\
&\quad \mathcal{A}(S)
\end{aligned}
$$

Сигналы $y1$ и $y2$ выполняют включение и выключение таймера. Это кратковременные сигналы. Вычисляется сигнал $StateOn$. Его значение равно единице тогда и только тогда, когда таймер включен. Включение таймера выполняется мгновенно. В том инстенте, когда пришел сигнал $y1$, таймер считается включенным. В то же время, состояние автомата меняется только в следующий инстент. Для согласования состояние таймера $StateOn$ берется в предыдущий инстент.

- В состояниях q1, q2, q3 автомата A1 дисплей показывает текущее время (y12 = 0), в состоянии q4 — время будильника (y12 = 1).

  Формула LTL:

$$
\begin{aligned}
q1 &::= Node1.q1 \\
q2 &::= Node1.q2 \\
q3 &::= Node1.q3 \\
q4 &::= Node1.q4 \\
S &::= ((q1 \vee q2 \vee q3) \wedge \neg y12) \vee (q4 \vee y12) \\
&\quad \mathcal{A}(S)
\end{aligned}
$$

Результаты проверки подтверждают, что все перечисленные свойства выполняются.

В окончательной версии программы текущее состояние автомата A1, как и любые другие сигналы, не относящиеся к интерфейсным, недоступно. Однако специально для проверки пользовательских свойств разработан особый режим генерации программы на языке Esterel, в котором экспортируются все внутренние сигналы. Используя этот режим, мы можем проверить перечисленные выше свойства.

Свойства можно описывать на языке темпоральной логики для TempEst или отдельным модулем на языке Esterel. Рассмотрим, как можно использовать этот язык для записи свойств. Для примера возьмем свойство, связанное с проигрыванием мелодии (рис. 3.13).

Первая строка — комментарий. Во второй строке указывается имя модуля: *PlayRingSigsAlternativeViolation*. Строки 3–4 определяют интерфейс модуля. Входные сигналы x1–x4 играют роль локальных переменных. Они будут подсоединены к выходным сигналам y3–y5 синхронной сети Net1 соответственно. Выходной сигнал *VIOLATED* генерируется в случае нарушения свойства. Условие генерации сигнала *VIOLATED* записано в строке 6. Мы просто перебираем все возможные пары сигналов. Оператор *present* заключен в бесконечный цикл *loop* (строки 5–10). Проверка выполняется в каждом инстенте.

3.3. Пример 3: Микроволновая печь

3.3.1. Введение

Пример демонстрирует возможности синхронно-автоматного программирования и подходы к верификации программ. Часть функциональности реализуется на аппаратном уровне.

```
 1: % Сигналы проигрывания мелодии являются взаимоисключающими
 2: module PlayRingSigsAlternativeViolation:
 3: input x1, x2, x3;
 4: output VIOLATED;
 5:   loop
 6:     present [x1 and x2 or x1 and x3 or x2 and x3] then
 7:       emit VIOLATED;
 8:     end present;
 9:     pause;
10:   end loop;
11: end module
```

Рис. 3.13. Пример модуля Esterel, выражающего пользовательское свойство

3.3.2. Описание задачи

Данный пример моделирует работу простой микроволновой печи. Пользовательский интерфейс взаимодействия с печью состоит из панели кнопок и дисплея (рис. 3.14). Дверца печи снабжена датчиком открытия.

Основная функция микроволновой печи в разогреве или разморозке продуктов. В режиме разогрева микроволновый генератор работает постоянно, в режиме разморозки микроволновый генератор периодически отключается. Продолжительность обоих режимов можно выбирать с точностью до 10 секунд. Максимальная продолжительность составляет 59 минут 50 секунд.

В микроволновой печи заложена функция кухонного таймера. Можно установить время, через которое печь подаст звуковой сигнал и начнет приготовление, если был выбран режим разморозки или приготовления. “Кухонный таймер”, “разморозка” и “приготовление” являются независимыми режимами, они выполняются в указанном порядке. Время выполнение каждого

Рис. 3.14. Пользовательский интерфейс микроволновой печи

из них может быть выбрано независимо. Если для какого-то режима время равно нулю, он пропускается.

Есть возможность установить мощность приготовления. Она измеряется в процентах от максимальной мощности с шагом 10%. Установка уровня мощности действует также для режима разморозки. Мощность можно установить в 0%. В этом случае оба режима “разморозка” и “приготовление” будут пропущены.

Установка режимов выполняется так. Мы нажимаем одну из кнопок “разморозка”, “приготовление” или “кухонный таймер”, затем устанавливаем время с помощью трех кнопок “10 мин”, “1 мин”, “10 сек”. Далее можно перейти к настройке другого режима или нажать кнопку “старт” для запуска процесса. Если мы раздумали запускать печь, нажатие кнопки “стоп” сбрасывает все настройки.

Уровень мощности настраивается соответствующей кнопкой. Первое нажатие кнопки “мощность” переводит микроволновую печь в режим настройки мощности. На дисплее отображается текущий уровень. Последующие нажатия кнопки уменьшают уровень мощности на 10%.

Во время приготовления дисплей отображает время, оставшееся до завершения текущего этапа. Если во время работы происходит ошибка, соответствующее сообщение появляется на дисплее.

Запрещается открывать дверцу печи во время приготовления. Если она открывается, текущее действие приостанавливается, микроволновый генератор выключается, на дисплее отображается сообщение об ошибке. Но процесс приготовление не сбрасывается, он будет возобновлен, как только дверца печи будет закрыта.

Еще одна ошибка времени работы связана с перегревом. Когда температура внутри устройства превышается некоторый заданный порог, микроволновый генератор выключается, на дисплей выводится сообщение об ошибке. Работа будет возобновлена, как только температура войдет в приемлемые рамки.

Существует режим ускоренного приготовления. Он активизируется нажатием кнопки “старт”, когда печь находится в состоянии бездействия (имеется ввиду, что мы не настраиваем какой-то другой режим и печь не работает). Первое нажатие кнопки “старт” устанавливает время приготовление в “30 секунд”. Каждое последующее нажатие добавляет еще 30 секунд. Приготовление начинается, если в течении 1.5 секунд не была нажата ни одна из кнопок. Автоматически выбирается максимальный уровень мощности.

В режиме бездействия дисплей показывает текущее время. Время можно устанавливать. Для перехода в режим настройки нужно нажать кнопку “часы”. Установка времени выполняется теми же кнопками, которые использовались для настройки времени приготовления. Однако сейчас они действуют

немного по-другому: “1 час”, “10 мин”, “1 мин”. Эффект от нажатия кнопок в режиме настройки часов показан в знаменателе надписи на кнопке. Текущее время может быть установлено с точность до минуты.

Отсчет времени происходит постоянно: во время бездействия, во время приготовления, и даже в процессе настройки текущего времени. Из режима установки времени можно выйти, нажав кнопку “отмена”, тогда все изменения будут аннулированы.

3.3.3. Реализация

Таблицы с описанием основных компонентов системы и диаграммы не включены по причине большого объема.

Периферийные устройства

Синхронно-автоматная программа кодирует только логику управления печью. Хранение и обработка больших целочисленных значений требует иного подхода. Для этих целей решено использовать внешние устройства, такие как контейнеры и регистры. Ниже будет описан интерфейс и поведение соответствующих компонентов.

Регистры. Регистры используются для хранения и модификации целочисленных значений, например, “время приготовления”. Набор входных сигналов используется в основном для модификации счетчика. Обновление выполняется мгновенно, т.е. в том же инстенте. Часто также требуется специальный сигнал для отображения значения счетчика на дисплее. Выходные сигналы отражают текущее состояние счетчика. Обычно важно знать, равен ли он нулю. Рассмотрим используемые виды счетчиков.

Регистр времени (рис. 3.15). Используется для хранения временных значений от 0 секунд до 59 минут 59 секунд с точностью представления 1 се-

кунда. Регистры этого типа используются для хранения значений “кухонный таймер”, “время разморозки”, “время приготовления”. Сигнал “+30 сек” используется только в регистре “время приготовления”. Входные сигналы взаимоисключающие.

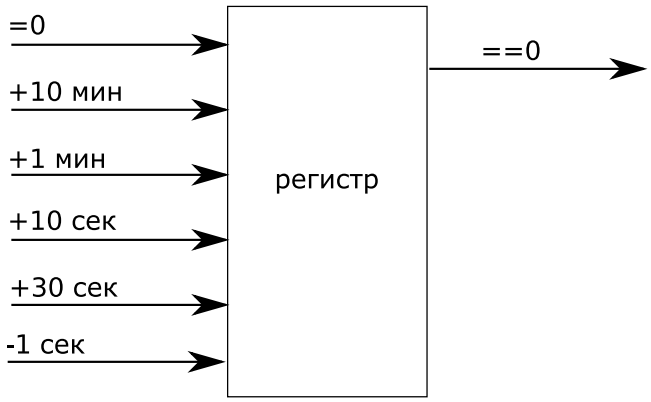

Рис. 3.15. Регистр времени (“кухонный таймер”, “время разморозки”, “время приготовления”)

Регистр “мощность” (рис. 3.16) хранит уровень мощности микроволнового генератора, который используется в процессе приготовления и разморозки. Он может принимать значения от 0% до 100% от максимальной мощности с шагом 10%. Входные сигналы взаимоисключающие.

Рис. 3.16. Регистр мощности

Регистры для установки часов “часы, установка времени”, “часы, отсчет времени” тесно взаимосвязаны друг с другом (рис. 3.17). Часы в микроволновой печи работают постоянно, даже во время их настройки, поэтому необходимы два регистра: один — для хранения текущего времени, которое регулярно обновляется, другой — для устанавливаемого времени — его значение отображается на дисплее в процессе настройки.

У обоих регистров отсутствуют выходные сигналы. В них нет надобности.

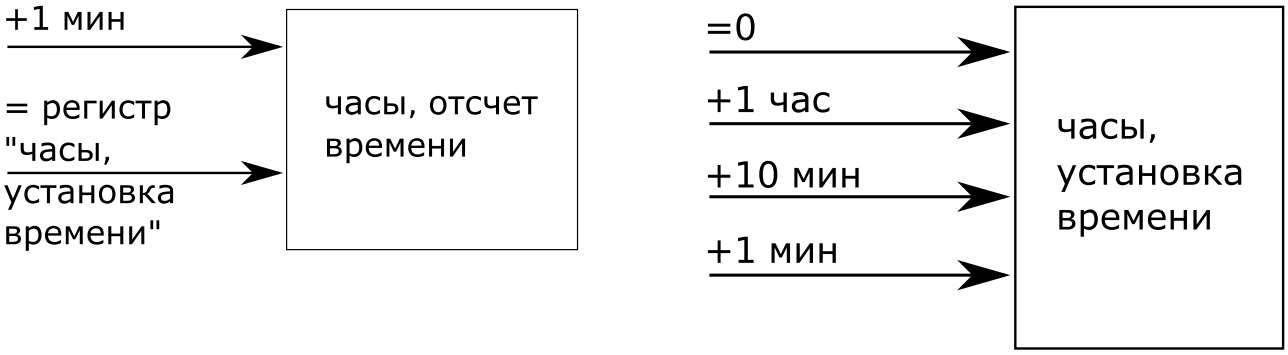

Рис. 3.17. Регистры “часы, установка времени”, “часы, отсчет времени”

Таймеры. Все таймеры устроены примерно одинаково (рис. 3.18). У них есть два входных сигнала “старт” и “стоп”. Первый из них запускает таймер, второй — останавливает. Сигналы взаимоисключающие. Выходной сигнал один: “==0”. Он генерируется, когда значение внутреннего счетчика равно нулю, т.е. когда таймер “сработал”.

Есть два варианта работы таймера. В первом случае таймер срабатывает один раз и сам останавливается. Во втором случае таймер будет работать бесконечно или до явной остановки, генерируя через заданный интервал выходной сигнал.

Рис. 3.18. Универсальный таймер

Таймер “ускоренного приготовления” имеет задержку 1.5 секунды, срабатывает однократно. Таймеры “1 сек” и “1 мин” срабатывают соответственно каждую секунду и каждую минуту. Это многократные таймеры. Таймер “цикл разморозки” многократный.

Компоненты системы

Система микроволновой печи состоит из нескольких компонентов, главный из них — Net1. Синхронная сеть Net1 объединяет все компоненты системы друг с другом. Основная часть логики сосредоточена в автоматах A1 и A3. Автомат A1 отвечает за настройку процесса приготовления (выбор времени приготовления, времени разморозки, мощности микроволнового генератора и т.д.). Автомат A3 отвечает за процесс приготовления. Второстепенные обязанности возложены на автоматы A2, A4, A6, A7. A4 — процесс разморозки, один из режимов приготовления. A2, A6, A7 отвечают за работу внутренних часов. В их обязанности входит реализация режима настройки часов с помощью кнопок передней панели и поддержание времени в актуальном состоянии. Часы работают постоянно, т.е. все время, пока работает программа, реализующая систему управления микроволновой печью. Остальные автоматы выполняют вспомогательные функции. A5 занимается фильтрацией выходных сигналов от кнопок. Нельзя нажимать несколько кнопок одновременно. Если приходят сигналы от нескольких кнопок в одном инстенте, все они подавляются. A8 реализует логическое “или”. Он используется для объединения сигналов, генерируемыми разными подсистемами.

3.3.4. Проверка пользовательских свойств

Большинство проверяемых свойств, описываемых далее, относятся скорее к внутреннему устройству, деталям реализации, чем к некоторым характеристикам, интересным пользователю системы. Проверка свойств, относящихся к реализации, очень важна. В ходе проектирования и реализации делается множество предположений, которые сложно проверить. Использование верификатора Xeve позволяет достичь большей уверенности в правильности кода.

Программа содержит несколько групп сигналов, каждая управляющая

своим внешним объектом. Например, это может быть “регистр мощности”. Управляющие сигналы должны быть взаимоисключающими. Проверены такие группы сигналов.

- Управляющие сигналы регистра “время приготовления” (y1, y2, y3, y4, y5, y6).

  Формула LTL:

$$
\begin{aligned}
S ::= \; & (y1 \wedge y2) \vee (y1 \wedge y3) \vee (y1 \wedge y4)(y1 \wedge y5) \vee (y1 \wedge y6) \vee \\
& (y2 \wedge y3) \vee (y2 \wedge y4) \vee (y2 \wedge y5) \vee (y2 \wedge y6) \vee \\
& (y3 \wedge y4) \vee (y3 \wedge y5) \vee (y3 \wedge y6) \vee \\
& (y4 \wedge y5) \vee (y4 \wedge y6) \vee \\
& (y5 \wedge y6) \\
& \mathcal{A}(\neg S)
\end{aligned}
$$

- Управляющие сигналы таймера “ускоренное приготовление” (y8, y9).
- Управляющие сигналы регистра “мощность” (y11, y12).
- Управляющие сигналы регистра “время разморозки” (y14, y15, y16, y17, y18).
- Управляющие сигналы регистра “кухонный таймер” (y20, y21, y22, y24).
- Управляющие сигналы регистра “часы, отсчет времени” (y26, y27).
- Управляющие сигналы регистра “часы, установка времени” (y29, y30, y31, y32).
- Управляющие сигналы регистра “1 сек” (y36, y37).
- Управляющие сигналы генератора микроволн (y39, y40).
- Управляющие сигналы таймера “цикл разморозки” (y43, y44).

Формула LTL приведена только для первого свойства. В остальных случаях формулы выглядят аналогично.

Следующие проверки в основном относятся к реализации программы.

- Автомат A1 перед переходом в состояние q3 (Микроволновая печь работает) обязательно генерирует сигнал y27 (запустить процесс приготовления).

  Формула LTL:

$$\begin{aligned} Q &::= Node1.q3 \\ Y &::= Node1.y27 \\ S &::= ((\mathcal{P}(\neg Q)) \wedge Q) \rightarrow (\mathcal{P}\, Y) \\ &\quad \mathcal{A}(S) \end{aligned}$$

- Несколько сигналов объединяются с помощью логического “или”. Во время выполнения программы они никогда не должны генерироваться одновременно. Проверим это. Для ссылки на пару сигналов используется имя результирующего сигнала. LTL-формула строится очевидным образом для каждого случая, поэтому LTL-формулы не показаны.

  – Регистр “время приготовления” → “показать на дисплее” (Node1.y7, Node3.y7).

  – Сигнал ошибки (Node1.y10, Node3.y1).

  – Регистр “время разморозки” → “показать на дисплее” (Node1.y19, Node3.y8).

  – Регистр “кухонный таймер” → “показать на дисплее” (Node1.y25, Node3.y9).

  – Генератор микроволн → включить (Node3.y11, Node4.y1).

  – Генератор микроволн → выключить (Node3.y12, Node4.y2).

- Автомат A3 перед переходом в состояние q1 (Начальное состояние) обязательно генерирует сигнал y16 (Выход из режима приготовления). В первый инстент выполняется инициализация, поэтому проверка свойства не выполняется.

  Формула LTL:

$$\begin{aligned} Q ::= \;& Node3.q1 \\ Y ::= \;& Node3.y16 \\ & \mathcal{A}((\mathcal{P}(\neg Q) \wedge Q) \rightarrow \mathcal{P}\, Y) \end{aligned}$$

- Автоматы A1 и A3 работают поочередно. Автомат A1 работает, когда он находится в состоянии отличном от q3. Автомат A3 работает, когда он находится в состоянии отличном от q1. Поэтому, когда A1 находится не в состоянии q3, то A3 — в состоянии q1, и, когда A3 находится не в состоянии q1, то A1 — в состоянии q3.

  Формула LTL:

$$\begin{aligned} S1 ::= \;& Node1.q3 \\ S1 ::= \;& Node3.q1 \\ & \mathcal{A}(((\neg S1) \rightarrow S2) \wedge ((\neg S2) \rightarrow S1)) \end{aligned}$$

- Микроволновый генератор работает только тогда, когда автомат A3 находится в одном из состояний q3, q4. Генератор работает все время, пока автомат A3 находится в состоянии q4. Переключение состояний происходит с задержкой на один инстент, а микроволновый генератор включается и выключается мгновенно. Для согласования состояние генератора берется с задержкой в один инстент.

Формула LTL:

$$\begin{aligned} mw\_on ::= \; & (\neg y40)\, \mathcal{S}\, y39 \\ q3 ::= \; & Node3.q3 \\ q4 ::= \; & Node3.q4 \\ & \mathcal{A}(((\mathcal{P}\, mw\_on) \rightarrow (q3 \vee q4)) \wedge (q4 \rightarrow \mathcal{P}\, mw\_on)) \end{aligned}$$

Формула mw_on true тогда и только тогда, когда работает микроволновый генератор.

- Автомат A3 находится в состоянии q3 тогда и только тогда, когда автомат A4 работает (находится в одном из состояний q2, q3).

  Формула LTL:

$$\begin{aligned} N3 ::= \; & Node3.q3 \\ N4 ::= \; & Node4.q2 \vee Node4.q3 \\ & \mathcal{A}((N3 \rightarrow N4) \wedge (N4 \rightarrow N3)) \end{aligned}$$

- Если автомат A3 находится в состоянии q3, то микроволновый генератор работает все время, пока автомат A4 находится в состоянии q2, и не работает, если A4 находится в другом состоянии.

  Формула LTL:

$$\begin{aligned} prmw\_on ::= \; & \mathcal{P}((\neg y40)\, \mathcal{S}\, y39) \\ N3 ::= \; & Node3.q3 \\ N4 ::= \; & Node4.q2 \\ & \mathcal{A}(N3 \rightarrow ((prmw\_on \rightarrow N4) \wedge (N4 \rightarrow prmw\_on))) \end{aligned}$$

  Формула prmw_on —- true тогда и только тогда, когда работает микроволновый генератор, с задержкой на один инстент.

- Команды включения и выключения режима разморозки взаимоисключающие (Node4.x2, Node4.x3).

- Микроволновый генератор не работает при открытой дверце печи. Микроволновый генератор выключается мгновенно (не включается), если поступает сигнал открытой дверцы печи (x16).

  Формула LTL:

$$\begin{aligned} mw\_on ::= \; & (\neg y40)\, \mathcal{S}\, y39 \\ & \mathcal{A}(x16 \rightarrow (\neg mw\_on)) \end{aligned}$$

- Автоматы A1 и A7 не могут одновременно находится в состояниях q8 и q1 соответственно.

  Формула LTL:

$$\mathcal{A}(\neg(Node1.q8 \wedge Node7.q1))$$

- Режим явного отсчета времени (A6 в q1) включен тогда и только тогда, когда A1 в q1.

  Формула LTL:

$$\begin{aligned} N1 ::= \; & Node1.q1 \\ N6 ::= \; & Node6.q1 \\ & \mathcal{A}((N1 \rightarrow N6) \wedge (N6 \rightarrow N1)) \end{aligned}$$

Результаты проверки подтверждают, что все перечисленные свойства выполняются.

3.4. Выводы

Было рассмотрено три примера применения синхронно-автоматного программирования для решения задач логического управления. Все задачи были успешно декомпозированы на подзадачи. Каждая подзадача решается отдельным элементом синхронно-автоматной программы: автоматом или синхронной сетью. Синхронные сети успешно использовались при разработке

программы. В целом можно утверждать, что технология применима для решения указанного класса задач. Модификация switch-технологии оказалась удачной.

В первом примере, пожалуй, было использовано слишком много компонентов для сравнительно простой задачи. Решение задачи было известно. Оно использует особенности синхронного подхода и языка Esterel в частности. Программа на языке Esterel значительно более компактна, чем решение в UML-диаграммах. С другой стороны, решение синхронно-автоматным подходом легче для понимания.

Декомпозиция задачи на подзадачи в соответствии с синхронно-автоматным подходом имеет неприятную особенность. Каждый компонент синхронной системы стремится инкапсулировать детали реализации. Интерфейс синхронных сетей и автоматов выставляет только набор сигналов. К сожалению, связи между сигналами не могут быть инкапсулированы. Хотя они не видны через интерфейс, они должны учитываться при разработке. Необдуманное использование компонентов может создать циклические зависимости. Esterel-программа, полученная по такой программе, не будет компилироваться. Синхронно-автоматная модель не предлагает стандартных средств для описания таких зависимостей. Выход заключается в использовании комментариев на естественном языке. В том случае, если комментариев нет, единственным безопасным решением будет считать, что каждый выходной сигнал зависит от всех входных сигналов. Если необходимы более слабые условия, придется вникать в детали реализации конкретного модуля.

При разработке примеров проблемы, связанные с циклическими зависимостями сигналов, возникали, но были быстро решены. Верифицирующий компилятор языка Esterel обнаруживает такие зависимости. Программисту предъявляется последовательность сигналов, образующих неразрешимый цикл. Требуется хорошее понимание особенностей синхронной модели

для устранения таких противоречий. Циклические зависимости свидетельствуют об ошибках проектирования.

Средства проверки синхронно-автоматных программ доказали свою полезность. В совокупности было описано и проверено несколько десятков свойств. Использовался язык линейной темпоральной логики с операторами прошедшего времени. Надо заметить, что большинство проверяемых свойств относились к допущениям, принятым в процессе разработки. По техническому заданию было сформулировано намного меньше свойств. Данный факт можно объяснить тем, что постановка задачи дана неформально, в ней мало точных требований. Функция верификации позволила обнаружить и исправить ошибки, допущенные в программе. Практика показала, что процесс разработки программы и верификации свойств разумно выполнять параллельно. Как только оформляется в достаточной мере законченный фрагмент программы, следует подумать о том, какие его свойства можно проверить. Чем раньше будет обнаружена ошибка, тем дешевле обойдется ее исправление.

Заключение

Предложена и реализована среда разработки синхронно-автоматных программ. Предложен метод верификации автоматных программ на основе синхронного языка Esterel. Проверка пользовательских свойств выполняется инструментом Xeve. Вспомогательные проверки (формат модели) выполняются утилитами преобразования, разработанными автором. Возможны два варианта описания пользовательских свойств: отдельный модуль на языке Esterel или язык линейной темпоральной логики.

Разработана формальная модель автоматной программы на основе синхронного подхода. В модели были учтены наиболее важные идеи автоматного программирования. Главный элемент модели — автомат. Синхронная парадигма фиксирует временную модель программы, гарантирует детерминированность поведения, унифицирует способ взаимодействия с окружающей средой. Новый подход разработки автоматных программ назван синхронно-автоматным программированием.

Предложен и реализован алгоритм преобразования синхронно-автоматной модели в программу на языке Esterel. Преобразование сохраняет структуру модели.

Разработан формат XML-файла для хранения синхронно-автоматной модели.

Написана программа автоматического преобразования модели синхронно-автоматной программы, сохраненной в XML-формате, в программу на языке Esterel. Программа преобразования выполняет полную проверку формата модели.

Предложен формат описания синхронно-автоматной программы на основе визуального языка UML.

Разработан набор рекомендаций по использованию редактора UML-диа-

грамм общего назначения для описания синхронно-автоматной программы. Реализована программа автоматического преобразования программы, описанной в виде набора диаграмм, в XML-файл описания модели программы.

Построено несколько примеров синхронно-автоматных программ. Проверены на практике выразительные возможности новой модели и UML-редактора для визуального построения программы. Описаны и проверены пользовательские свойства примеров.

Литература

[1] Альшевский, Ю.А. Механизм обмена сообщений для параллельно работающих автоматов (на примере системы управления турникетом): Проектная документация / Ю.А. Альшевский, М.Г. Раер, А.А. Шалыто // СПбГУ ИТМО. Кафедра «Технологии программирования» [Электронный ресурс]. — Разд. «Проекты». — Режим доступа: http://is.ifmo.ru/, свободный.

[2] Ачасова, С.М. Корректность параллельных вычислительных процессов / С.М. Ачасова, О.Л. Бандман. — Новосибирск: Наука, Сибирское отделение, 1990. — 252 с.

[3] Бандман, О.Л. Проверка корректности сетевых протоколов с помощью сетей Петри // Автоматика и вычислительная техника. — № 6. — 1986. — С. 82–91.

[4] Васильева, К.А. Верификация автоматных программ с использованием LTL / К.А. Васильева, Е.В. Кузьмин // Моделирование и анализ информационных систем. — 2007. — Т.14, № 1. — С. 3–14.

[5] Вельдер, С.Э. Введение в верификацию автоматных программ на основе метода MODEL CHECKING / С.Э. Вельдер, А.А. Шалыто // Научно-технический вестник СПбГУ ИТМО. — 2007. — Вып. 42 (Фундаментальные и прикладные исследования информационных систем и технологий). — С. 33–48.

[6] Вельдер, С.Э. Введение в верификацию автоматных программ на основе метода Model Checking / С.Э. Вельдер, А.А. Шалыто // СПбГУ ИТМО. Кафедра «Технологии программирования» [Электронный ре-

сурс]. — Разд. «Верификация». — Режим доступа: http://is.ifmo.ru/, свободный.

[7] Вельдер, С.Э. О верификации простых автоматных программ на основе метода “Model Checking” / С.Э. Вельдер, А.А. Шалыто // Информационно-управляющие системы. — 2007. — № 3. — С. 27–38.

[8] Вельдер, С.Э. Универсальный инфракрасный пульт для бытовой техники / С.Э. Вельдер, Ю.Д. Бедный // СПбГУ ИТМО. Кафедра «Технологии программирования» [Электронный ресурс]. — Разд. «Проекты». — Режим доступа: http://is.ifmo.ru/, свободный.

[9] Виноградов, Р.А. Верификация автоматных программ средствами CPN/ Tools / Р.А. Виноградов, Е.В. Кузьмин, В.А. Соколов // Моделирование и анализ информационных систем. — 2006. — Т.13, № 2. — С. 4–15.

[10] Гуисов, М.И. Задача Д. Майхилла «Синхронизация цепи стрелков»: Проектная документация / М.И. Гуисов, А.Б. Кузнецов, А.А. Шалыто // СПбГУ ИТМО. Кафедра «Технологии программирования» [Электронный ресурс]. — Разд. «Проекты». — Режим доступа: http://is.ifmo.ru/, свободный.

[11] Гуров, В.С. UniMod— инструментальное средство для автоматного программирования / В.С. Гуров, М.А. Мазин, А.А. Шалыто // Научно-технический вестник. — 2006. — Вып. 30: Фундаментальные и прикладные исследования информационных систем и технологий. — С. 32–44.

[12] Гуров, В.С. Исполняемый UML из России / В.С. Гуров, А.С. Нарвский, А.А. Шалыто // PC Week/RE. — 2005. — № 26. — С. 18–19.

[13] Гуров, В.С. Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора / В.С. Гуров, А.А. Ша-

лыто, Б.Р. Яминов // Международная научно-техническая мультиконференция “Проблемы информационно-компьютерных технологий и мехатроники”. Материалы международной научно-технической конференции “Многопроцессорные вычислительные и управляющие системы” (МВУС-2007). — 2007. — Т.1. — С. 198–203.

[14] Карпов, Ю.Г. Теория автоматов. — СПб.: Питер, 2003. — 208 с.

[15] Карпов, Ю.Г. Формальное описание и верификация протоколов на основе CSS // Автоматика и вычислительная техника. — 1986. — № 6. — С. 21–30.

[16] Кларк, Э.М. Верификация моделей программ. Model Checking / Э.М. Кларк, О. Грамберг, Д. Пелед. — М.: МЦНМО, 2002. — 416 С.

[17] Корнеев, Г.А. Верификация автоматных программ / Г.А. Корнеев, В.Г. Парфенов, А.А. Шалыто // Тезисы докладов Международной научной конференции, посвященной памяти профессора А.М. Богомолова. “Компьютерные науки и технологии”. — Саратов: СГУ, 2007. — С. 66–69.

[18] Кузьмин, Е.В. Иерархическая модель автоматных программ // Моделирование и анализ информационных систем. — 2006. — Т.13, № 1. — С. 27–34.

[19] Кузьмин, Е.В. Моделирование, спецификация и верификация “автоматных” программ / Е.В. Кузьмин, В.А. Соколов // Программирование. — 2008. — № 1. — С. 38–60.

[20] Кузьмин, Е.В. О верификации “автоматных” программ / Е.В. Кузьмин, В.А. Соколов // Актуальные проблемы математики и информатики. Сборник статей к 20–летию факультета ИВТ ЯрГУ им. П.Г. Демидова. — Ярославль: ЯрГУ, 2006. — С. 27–32.

[21] Кузьмин, Е.В. О дисциплине специализации “Верификация программ” / Е.В. Кузьмин, В.А. Соколов // Преподавание математики и компьютерных наук в классическом университете. Доклады II-й научно-методической конференции преподавателей математического ф-та и ф-та ИВТ Ярославского гос. ун-та им. П.Г. Демидова. — ЯрГУ, 2007. — С. 91–101.

[22] Кулямин, В. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия / В. Кулямин, А. Петренко, В. Рубанов, А. Хорошилов // Информационные технологии. — № 8. — 2008. — С. 2–7.

[23] Ломазова, И.А. Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой. — М.: Научный мир, 2004. — 208 с.

[24] Лукьянова, А.П. Моделирование и верификация потоков данных на диаграммах состояний: Бакалаврская работа / А.П. Лукьянова; СПбГУ ИТМО, каф. «Компьютерные технологии». — СПб., 2005. — 60 с.

[25] Наумов, А.С. Объектно-ориентированное программирование с явным выделением состояний: Бакалаврская работа / А.С. Наумов.; СПбГУ ИТМО, каф. «Компьютерные технологии». — СПб., 2004. — 209 с.

[26] Непомнящий, В.А. Практические методы верификации программ // Кибернетика. — № 2. — 1984. — С. 21–28.

[27] Пак, С.В. Задача об обедающих философах / С.В. Пак, А.А. Шалыто // СПбГУ ИТМО. Кафедра «Технологии программирования» [Электронный ресурс]. — Разд. «Проекты». — Режим доступа: http://is.ifmo.ru/, свободный.

[28] Приемы объектно-ориентированного проектирования. Паттерны проек-

тирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. — СПб.: Питер, 2004. — 366 с.

[29] Робачевский, А.М. Операционная система Unix / А.М. Робачевский, С.А. Немнюгин, О.Л. Стесик. — СПб.: БХВ-Петербург, 2007. — 656 с.

[30] Санкт-Петербургский Государственный Университет Информационных Технологий, Механики и Оптики. Кафедра «Технологии программирования» [Электронный ресурс]. — 2002– . — Режим доступа: http://is.ifmo.ru/, свободный.

[31] Свидетельство об официальной регистрации программы для ЭВМ “Система моделирования и анализа автоматных программ”, № 2007611856 / правообладатель Государственное образовательное учреждение высшего профессионального образования “Ярославский государственный университет им. П.Г. Демидова”; авторы Виноградов Роман Александрович, Кузьмин Егор Владимирович, Соколов Валерий Анатольевич. — заявка №2007610513; зарегистрировано в Реестре программ для ЭВМ 7 мая 2007 г.

[32] Смелянский Р.Л. Взаимосвязь программы и вычислительной среды // Вычислительные системы и вопросы анализа решений. — М., МГУ, 1989.

[33] Соколов, В.А. Моделирование распределенных систем и анализ их семантических свойств [Текст]: дис. . . . доктора физико-математических наук : 01.01.09 : утв. 8.12.2006 / Соколов Валерий Анатольевич. — Ярославль, 2006. — 318 с.

[34] Страуструп, Б. Язык программирования C++: Специальное издание:

Пер. с англ. — М.; СПб.: «Издательство Бином» — «Невский Диалект», 2002. — 1099 с., ил.

[35] Шалыто, А.А. Автоматно-ориентированное программирование // Материалы IX Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах». — СПб.: изд-во Политехнического университета, 2005. — С. 44–52.

[36] Шалыто, А.А. Алгоритмизация и программирование задач логического управления. — СПбГУ ИТМО, 1998. — 56 с.

[37] Шалыто, А.А. Логическое управление. Методы аппаратной и программной реализации алгоритмов. — СПб.: Наука, 2002. — 784 с.

[38] Шалыто, А.А. Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов / А.А. Шалыто, Л.А. Наумов // Искусственный интеллект. — 2004. — № 4. — С. 756–762.

[39] Шалыто, А.А. Программирование с явным выделением состояний / А.А. Шалыто, Н.И. Туккель // Мир ПК. — 2001. — № 8. — С. 116–121; № 9. — С. 132–138.

[40] Шалыто, А.А. Switch-технология — автоматный подход к созданию программного обеспечения «реактивных систем» / А.А. Шалыто, Н.И. Туккель // Программирование. — 2001. — № 5. — С. 45–62.

[41] Шалыто, А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. — СПб.: Наука, 1998. — 628 с.

[42] Шамгунов, Н.Н. State Machine — новый паттерн объектно-ориентированного проектирования / Н.Н. Шамгунов, Г.А. Корнеев, А.А. Шалыто // Информационно-управляющие системы. — 2004. — № 5. — С. 13–25.

[43] Шопырин, Д.Г. Объектно-ориентированный подход к автоматному программированию / Д.Г. Шопырин, А.А. Шалыто // Информационно-управляющие системы. — 2003. — № 5. — С. 29–39.

[44] Шопырин, Д.Г. Синхронное программирование / Д.Г. Шопырин, А.А. Шалыто // Информационно-управляющие системы. — 2004. — № 3. – С. 35–42.

[45] Abdulla, P.A. General Decidability Theorems for Infinite-State Systems / Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, Yih-Kuen Tsay // Logic in Computer Science. — 1996. — P. 313–321.

[46] Abdulla P. Verifying Programs with Unreliable Channels / P. Abdulla, B. Jonsson // Proc. Logic in Comp. Science (LICS'93). — 1993. — P. 160–170.

[47] Andre, C. Representation and Analysis of Reactive Behaviors: A Synchronous Approach / C. Andre // Computational Engineering in Systems Applications (CESA), IMACS Multiconference. — Lille, France. — 1996, July. — P. 19–29.

[48] Anisimov, N.A. Two-Levels Formal Model for Protocol Specification Based on Petri Nets / N.A. Anisimov, A.A. Kovalenko, P.A. Postupalski // Proceedings of the IFIP TC6 Int. Symp. "Network Information Processing Systems". — 1993. — P. 143–154.

[49] Barros, T. Formal specification and verification of distributed component systems: PhD thesis: defence date: 25.11.2005 / T. Barros; Universite de Nice; INRIA Sophia Antipolis. — France, 2005. — 158 pages.

[50] Benveniste, A. The Synchronous Languages 12 Years Later: Invited Paper / A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. le Gurnic, R. de Simone // Proceedings of the IEEE. — 2003, January. — Vol. 91, №. 1. — P. 64–83.

[51] Berry, G. Incremental development of an HDLC protocol in Esterel: Technical report 1031 / Gerard Berry, Georges Gonthier. — France: Inria, Sophia-Antipolis, 1989. — 48 pages.

[52] Berry, G. The Esterel v5_91 System Manual: Документация к программному обеспечению Esterel Compiler, version 5.91. Ecole des Mines de Paris (EMP), ARMINES, and INRIA. — 2000, 5 June. — 132 pages.

[53] Berry, G. The Esterel v5 Language Primer, version v5_91: Документация к программному обеспечению Esterel Compiler, version 5.91. Ecole des Mines de Paris (EMP), ARMINES, and INRIA. — 2000, 5 June. — 142 pages.

[54] Berry, G. The constructive semantics of pure Esterel. Draft version 3 [Черновой вариант документации]. — 1999, 2 July. — 160 pages.

[55] Berry, G. The foundations of Esterel / Gerard Berry // Proof, Language, and Interaction: Essays in honour of Robin Milner / edited by Gordon Plotkin, Colin Stirling and Mads Tofte. — MIT Press, 2000. — P. 425–454.

[56] Bouali, A. Xeve: an Esterel Verification Environment : Version v1_3 : Rapport technique №0214 / Amar Bouali ; Inria, Institut National de Recherche en Informatique et en Automatique. — France, 1997. — 23 pages.

[57] Bouali, A. Xeve, and Esterel verification environment // Computer aided verification. — Berlin: Springer, 1998. — P. 500–504.

[58] Browne, M.C. Characterizing finite Kripke structures in propositional temporal logic / M.C. Browne, E.M. Clarke, O. Grumberg // Theoretical Computer Science. — 1988. — Vol. 59, No. 1–2. — P. 115–131.

[59] Browne, M.C. Reasoning about networks with many identical finite-state

processes / M.C. Browne, E.M. Clarke, O. Grumberg // Information and Computation. — 1989. — Vol. 81, No. 1. — P. 13–31.

[60] Bryant, R.E. Graph-based algorithms for boolean function manipulation // IEEE Transactions on Computers. — Vol. 35, №. 8. — P. 677–691.

[61] CPN Tools. Computer Tool for Coloured Petri Nets [Электронный ресурс] / CPN Group, University of Aarhus, Denmark. — 2001– . — Режим доступа: http://wiki.daimi.au.dk/cpntools/cpntools.wiki, свободный.

[62] Clarke, E.M. Automatic verification of finite-state concurrent systems using temporal logic specifications / E.M. Clarke, E.A. Emerson, A.P. Sistla // ACM transactions on Programming Languages and Systems. — Vol. 8, No. 2. — 1986. — P. 244–263.

[63] Clarke, E.M. Model Checking / E.M. Clarke, O. Grumberg, D.A. Peled. — The MIT Press, 1999. — 314 pages.

[64] Emerson, E.A. Branching Time Temporal Logic and the Design of Correct Concurrent Programs. PhD thesis. — Harvard University, 1981.

[65] Emerson, E.A. Temporal and modal logic // Handbook of Theoretical Computer Science, vol. B. — 1991. — P. 995–1072.

[66] Esterel synchronous language web main page [Электронный ресурс]: [Зеркало сайта, посвященного синхронному языку Esterel] / Inria Sophia Antipolis — Méditerranée. — Режим доступа, http://www-sop.inria.fr/esterel.org/files/, свободный.

[67] Esterel verification environment [Электронный ресурс]: [Главная страница раздела, посвященного верификатору Xeve] / Inria Sophia Antipo-

lis — Méditerranée. — Режим доступа, http://www-sop.inria.fr/meije/verification/esterel/index.html, свободный.

[68] Executable UML. UniMod [Электронный ресурс]: [Сайт проекта UniMod] / eVelopers Corporation. — 2003– . — Режим доступа: http://unimod.sourceforge.net/, свободный.

[69] Finkel, A. Reduction and covering of infinite reachability trees // Information and Computation. — Vol. 89, Issue 2. — 1990. — P. 144–179.

[70] Finkel, A. Well-structured transition systems everywhere! / A. Finkel, Ph. Schnoebelen // Theoretical Computer Science. — Vol. 256, № 1–2. — 2001. — P. 63–92.

[71] Functional Description. Warm-up & prelubrication logic. Generator Control Unit. — / copyright Norcontrol as, 1993. — № AO-0054-A/11/10/93.

[72] Halbwachs, N. Synchronous programming of reactive systems, a tutorial and commented bibliography / N. Halbwachs // Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), Canada: LNCS 1427. — Springer Verlag, 1998. — vol. 1427. — P. 1–16.

[73] Holzmann, G.J. Design and Validation of Computer Protocols. — Prentice Hall, New Jersey, 1991. — P. 512.

[74] Inrea Sophia Antipolis — Méditerranée [Электронный ресурс]. — Режим доступа, http://www-sop.inria.fr/, свободный.

[75] Jagadeesan, L.J. A formal approach to reactive system software: A telecommunications application in Esterel / L.J. Jagadeesan, C. Puchol, J.E. Von Olnhausen // Proceeding of the 1th workshop on Industrial-

Strength formal specification techniques. — Washington, DC, USA: IEEE Computer Society, 1995. — P. 132–145.

[76] Jagadeesan, L.J. On the bounded-fairness support in TempEst 1.2: Документация к программному обеспечению TempEst / L.J. Jagadeesan, C. Puchol.. — 1995, 10 March.

[77] Jagadeesan, L.J. Safety property verification of Esterel programs and applications to telecommunications software / L.J. Jagadeesan, C. Puchol, J.E. Von Olnhausen // Proceedings of the 7th international conference on computer aided verification. — Longon, UK, 1995. — P. 127–140.

[78] Jensen, K. Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use. — Springer, 1997.

[79] Jensen, K. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use // Vol. 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. — Springer-Verlag, 1992.

[80] Jensen, K. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use // Vol. 2: Analysis Methods. EATCS Monographs on Theoretical Computer Science. — Springer-Verlag, 1994.

[81] Manna, Z The Modal Logic of Programs / Z. Manna, A. Pnueli // Proceedings of the 6th Colloquium, on Automata, Languages and Programming. — 1979. — P. 385–409.

[82] Potop-Butucaru, D. Compiling Esterel / D. Potop-Butucaru, S.A. Edwards, G. Berry. — Springer, 2007. — 335 pages.

[83] Puchol, C The TempEst program verification toolset: Слайды: Документация к программному обеспечению TempEst. — 1995.

[84] Sidorova, N. Surface measures on paths in an embedded Riemannian manifold. — PhD thesis. — 2003.

[85] Sifakis, J. Building models of real-time systems from application software / J. Sifakis, S. Tripakis, S. Yovine // Proceedings of the IEEE, Special issue on modeling and design of embedded. — Vol. 91, № 1. — 1993. — P. 100–111.

[86] Sifakis, J. Modeling Real-Time Systems-Challenges and Work Directions // Lecture Notes in Computer Science. — Vol. 2211. — 2001. — P. 373–389.

[87] Synchronous programming of reactive systems / Nicolas Halbwachs. — Springer, 1993. — 192 pages.

[88] The Esterel language [Электронный ресурс]: [Главная страница раздела, посвященного языку Esterel] / Inria Sophia Antipolis — Méditerranée. — Режим доступа, http://www-sop.inria.fr/meije/esterel/esterel-eng.html, свободный.

[89] The synchronous approach to designing reactive systems / Axel Poigne, Matthew Morley, Olivier Maffeis, Leszek Holenderski, Reinhard Budde // Formal methods in system design. — Vol. 12, Issue 2. — Hingham, MA, USA: Kluwer Acedemic Publishers, 1998. — P. 163–187.

Приложение А

Среда разработки и верификации синхроно-автоматных программ

Листинг А.1. Пример XML-файла с описанием программы

```
<?xml version="1.0" encoding="UTF-8"?>
<model top="Net1">
  <automata name="A1" nodes="3" inputs="2" outputs="2">
    <edge start="1" end="2" label="1/y1"/>
    <edge start="2" end="3" label="(1x1)|x2!x1/y1,y2"/>
    <edge start="2" end="2" label="0/y1,y2"/>
    <edge start="3" end="1" label="x1/y1"/>
    <edge start="3" end="1" label="not x1"/>
  </automata>
  <automata name="A2" nodes="1" inputs="0" outputs="0">
  </automata>

  <net name="Net1" nodes="3" inputs="2" outputs="2">
    <node name="Node1" type="A1" inputs="2" outputs="2"/>
    <node name="Node2" type="A1" inputs="2" outputs="2"/>
    <node name="Node3" type="A2" inputs="0" outputs="0"/>
    <group name="G1">
      <signal value="x1"/>
      <signal value="Node1.x1"/>
      <signal value="Node2.x1"/>
    </group>
    <group name="G2">
      <signal value="y2"/>
      <signal value="Node1.y1"/>
      <signal value="Node2.x2"/>
    </group>
    <group name="G3">
      <signal value="x2"/>
      <signal value="Node1.x2"/>
    </group>
    <group name="G4">
      <signal value="y1"/>
      <signal value="Node2.y1"/>
    </group>
  </net>
</model>
```

Рис. А.1. UML-редактор MagicDraw с открытой диаграммой автомата

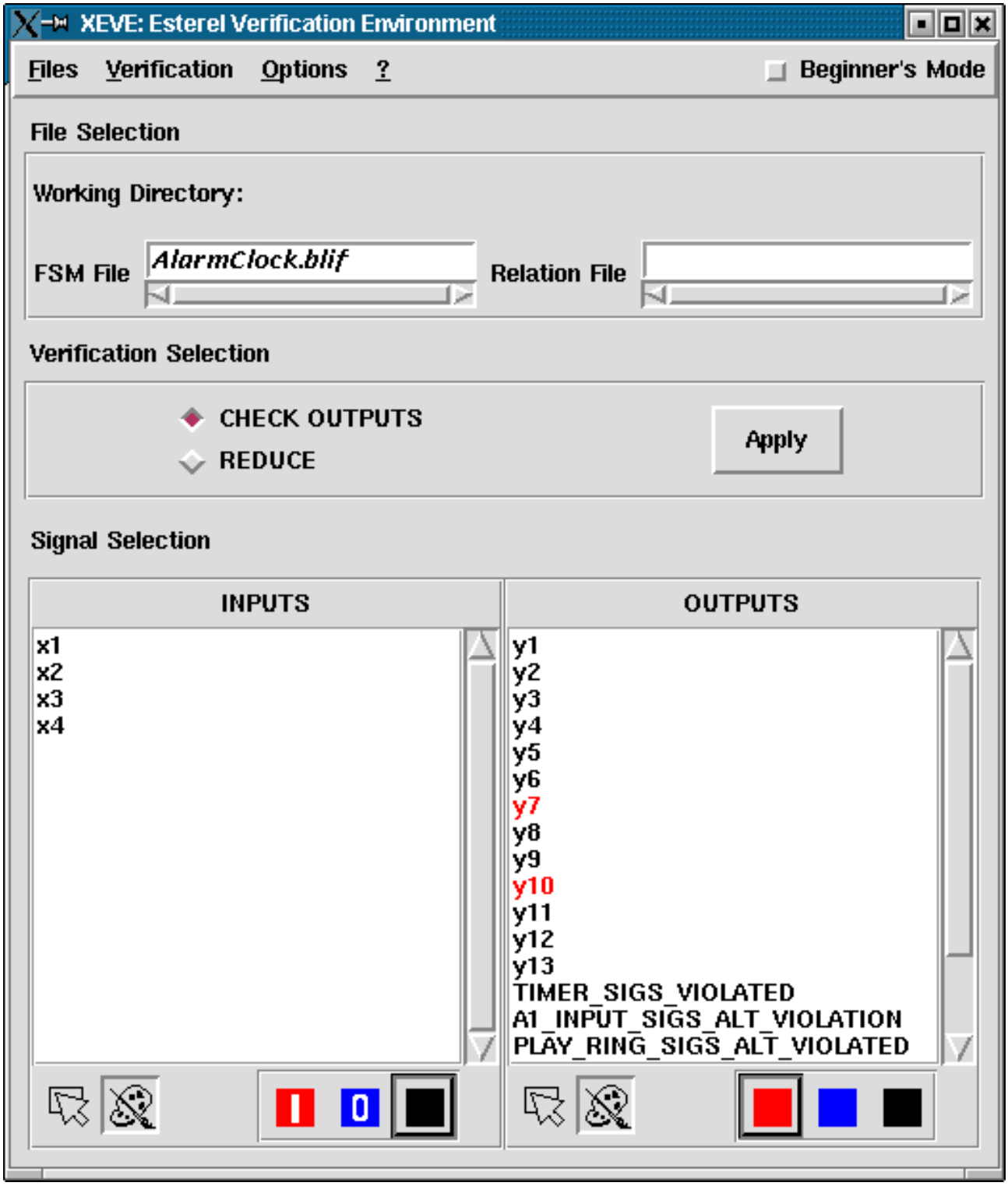

Рис. А.2. Пользовательский интерфейс верификатора Xeve, главное окно

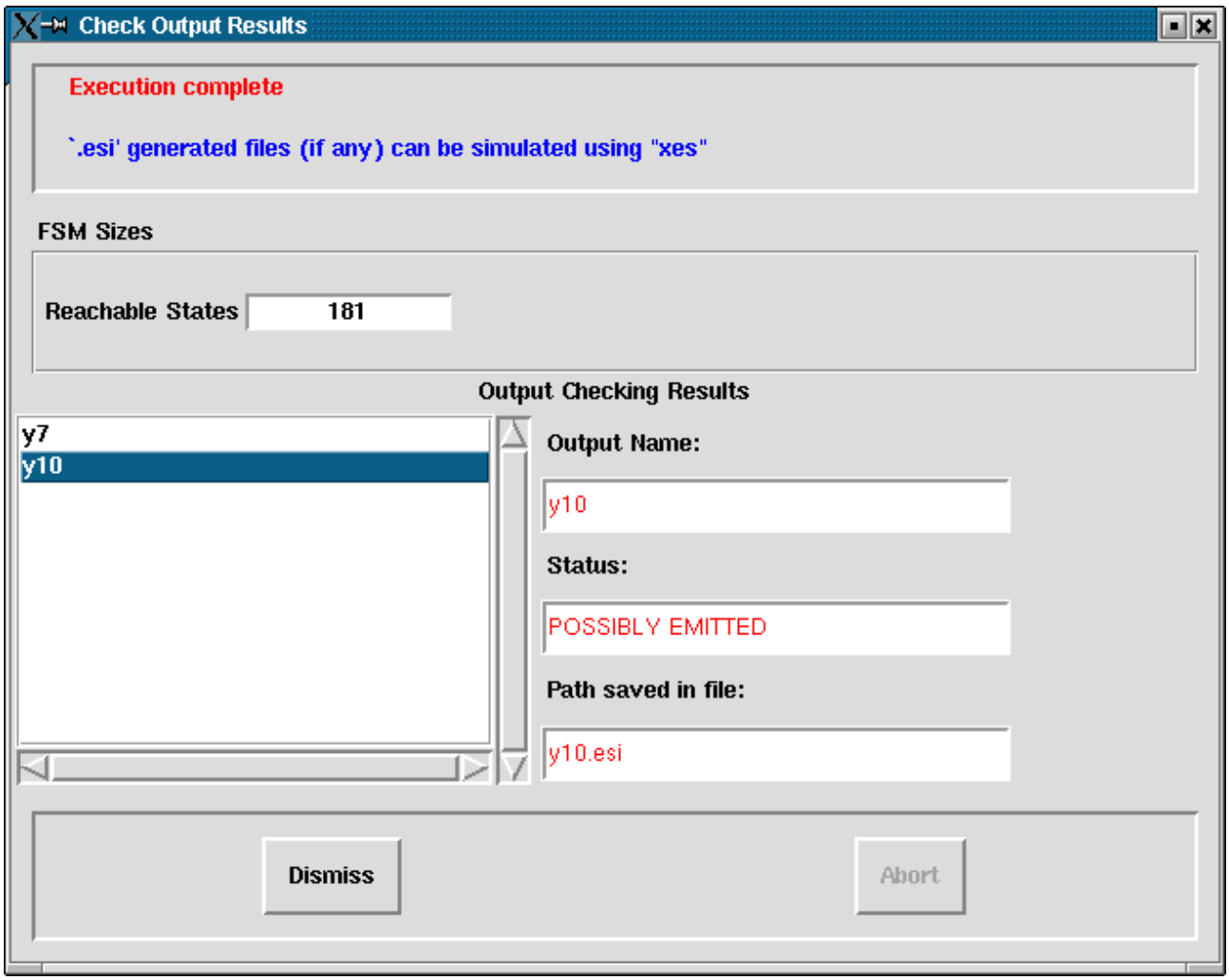

Рис. А.3. Пользовательский интерфейс верификатора Xeve, результаты проверки

Листинг А.2. Пример strl-файла, сгенерированного по XML-описанию программы (часть файла удалена)

```
% ...

module A3:
input x1, x2, x3;
output y1;
  signal q1 in
    %initialization
    emit q1;

    loop
      present % automata states
        case q1 do
          present
            case [ not x1 and ( x2 or x3 ) ] do
              emit y1;
              pause;
              emit q1;
            else
              pause;
              emit q1;
          end present
        else
          pause; % this case  should never be reached
      end present % automata states
    end loop
  end signal
end module

% ...

module Net1:
input x1, x2, x3;
output y1, y2, y3;
  signal % node signals
    Node1_x1, Node1_y1,
    Node2_x1, Node2_x2, Node2_x3, Node2_y1,
    Node3_x1, Node3_x2, Node3_x3, Node3_y1
    in
    [
      run Node1 / A1 [ signal Node1_x1 / x1, Node1_y1 / y1 ];
    ||
      run Node2 / A2 [ signal Node2_x1 / x1, Node2_x2 / x2, Node2_x3
          / x3, Node2_y1 / y1 ];
    ||
      run Node3 / A3 [ signal Node3_x1 / x1, Node3_x2 / x2, Node3_x3
          / x3, Node3_y1 / y1 ];
    ||
```

```
        loop
          [
          %G1 group
          present Node3_y1 then
            emit y3;
          end present;
          ||
          %G2 group
          present x1 then
            emit Node2_x1;
            emit Node3_x1;
          end present;
          ||
          %G3 group
          present x2 then
            emit Node3_x2;
            emit Node2_x2;
            emit Node1_x1;
          end present;
          ||
          %G4 group
          present Node2_y1 then
            emit y1;
          end present;
          ||
          %G5 group
          present x3 then
            emit Node3_x3;
            emit Node2_x3;
          end present;
          ||
          %G6 group
          present Node1_y1 then
            emit y2;
          end present;
          ];

          pause;
        end loop
      ]
    end signal
end module

% ...

module BusArbiter:
input x1, x2, x3;
output y1, y2, y3;
    run Net2;
end module
```

Листинг А.3. Фрагмент программы, генерирующий strl-файл по XML-описанию

```
#!/usr/local/bin/perl
# ...
use strict;
use XML::Parser;
use English;

# ...

sub GenerateAutomata
{
  local (*STDOUT) = shift @_; # redirect stdout
  my ($automata, $Location) = @_;
  $Location .= '.GenerateAutomata';
  foreach my $A (@$automata) {
    &GenerateAutomaton($A, $Location);
  }
}

# ...

sub GenerateNet
{
  my($SAMStr, $net, $Location) = @_;
  $Location .= '.GenerateNet';
  print "\n";
  print "module␣$net->{'name'}:\n";
  print "input␣" . &ListOfValues('x', 1, $net->{'inputs'}) . ";\n"
      if $net->{'inputs'} > 0;
  print "output␣" . &ListOfValues('y', 1, $net->{'outputs'}) . ";\n"
      if $net->{'outputs'} > 0;
  my($IntSigsGen) = &GenerateNetInternalSigsDecl($SAMStr, $net,
     $Location);
  print "␣␣␣␣[\n";
  &GenerateNetNodeInvocation($SAMStr, $net, $Location);
  print "␣␣␣␣␣␣loop\n";
  print "␣␣␣␣␣␣␣␣[\n";
  &GenerateNetGroupList($SAMStr, $net, $Location);
  print "␣␣␣␣␣␣␣␣];\n";
  print "\n";
  print "␣␣␣␣␣␣␣␣pause;\n";
  print "␣␣␣␣␣␣end␣loop\n";
  print "␣␣␣␣]\n";
  print "␣␣end␣signal\n" if not $GenerateVerifSigs and $IntSigsGen;
  print "end␣module\n";
}

# ...
```

Листинг А.4. Описание свойства для проверки на языке линейной темпоральной логики

```
% Если только один пользователь посылает запрос,
% то этот запрос будет обязательно принят.
Signals: x1, x2, x3, y1, y2, y3;

X1 := { (x1 & (Not x2) & (Not x3)) -> (y1 & (Not y2) & (Not y3)) }
X2 := { ((Not x1) & x2 & (Not x3)) -> ((Not y1) & y2 & (Not y3)) }
X3 := { ((Not x1) & (Not x2) & x3) -> ((Not y1) & (Not y2) & y3) }

SingleSig := Always { X1 & X2 & X3 }
```

Листинг А.5. Синтаксис вызова скрипта MDUML2SAM

```
Usage:
   MDUML2SAM [<source> <target>]
```

Листинг А.6. Синтаксис вызова скрипта SAM2strl

```
Usage:
SAM2strl [-s] [-d <doc file>] [<source> <target>]
  -s - Verification mode. Exports internal signals.
  -d - Generate documentation.
```

Листинг А.7. Пример вызова скрипта MDUML2SAM

```
> MDUML2SAM BusArbiter.mdzip BusArbiter.xml
MDUML2SAM
Source = "BusArbiter.mdzip", Output = "BusArbiter.xml"
MagicDraw UML => Synchonous Automata Model XML
Source file parsed.
Output file generated.
>
```

Листинг А.8. Пример вызова скрипта SAM2strl

```
> SAM2strl BusArbiter.xml BusArbiter.strl
SAM2strl
Source = "BusArbiter.xml", Output = "BusArbiter.strl"
in AutomataParse
in AutomataParse
in AutomataParse
in AutomataParse
in NetParse
in NetContentCheck
in NetParse
in NetContentCheck
>
```

Листинг А.9. Пример вызова утилиты tl2strl

```
> tl2strl SingleSig.tl
>
```

Листинг А.10. Пример вызова утилиты Esterel

```
> esterel -main strlmain -causal -B BusArbiter -Lblif BusArbiter.
➥ strl strlmain.strl SingleSig.strl MoreThanOneAskSig.strl
➥ MultipleSigsSimult.strl PersistentReq.strl
>
```

Листинг А.11. Пример вызова утилиты checkblif

```
> checkblif -v -output { SingleSig_VIOLATED
➥ MoreThanOneAskSig_VIOLATED MultipleSigsSimult_VIOLATED
➥ PersistentReq_VIOLATED } BusArbiter.blif
--- xeve: Building the FSM from blif file BusArbiter.blif
--- chkblif: Computing the FSM reachable states
************************************************************
Reachable state computation trace for strlmain
************************************************************
************************************************************
End of RSS trace for strlmain
CPU Time: 0.17 sec.
************************************************************
--- xeve: Reachable states: <<69>> -- BDD nodes: <<678>>
--- chkblif: Output SingleSig_VIOLATED NEVER EMITTED
--- chkblif: Output MoreThanOneAskSig_VIOLATED NEVER EMITTED
--- chkblif: Output MultipleSigsSimult_VIOLATED NEVER EMITTED
--- chkblif: Output PersistentReq_VIOLATED NEVER EMITTED
>
```

Приложение Б

Пример 1. Арбитр шины. Таблицы

Таблица Б.1. Интерфейсные сигналы и состояния автомата A1

q1	Ожидание прихода токена
q2	Токен пришел
x1	Токен (получение токена от предыдущего узла)
y1	Токен (передача токена следующему узлу)

Таблица Б.2. Интерфейсные сигналы и состояния автомата A2

q1	Единственное состояние
x1	Запрос на предоставление доступа
x2	Токен
x3	grand — передача права предоставления доступа
y1	Сигнал подтверждения предоставления доступа

Таблица Б.3. Интерфейсные сигналы и состояния автомата A3

q1	Единственное состояние
x1	Запрос на предоставление доступа
x2	Токен
x3	grand — передача права предоставления доступа
y1	grand — передача права предоставления доступа

Таблица Б.4. Интерфейсные сигналы и состояния автомата A4

q1	Начальное состояние (в этом состоянии автомат проводит только первый инстент)
q2	Основное состояние (в нем автомат проводит остаток жизни)
x1	Токен
y1	Токен

Таблица Б.5. Интерфейсные сигналы синхронной сети Net1

x1	Запрос на предоставление доступа
x2	Токен (получение токена от предыдущего узла)
x3	grand — передача права предоставления доступа
y1	Сигнал подтверждения предоставления доступа
y2	Токен (передача токена следующему узлу)
y3	grand — передача права предоставления доступа

Таблица Б.6. Интерфейсные сигналы синхронной сети Net2

x1	Запрос на предоставление доступа 1
x2	Запрос на предоставление доступа 2
x3	Запрос на предоставление доступа 3
y1	Сигнал подтверждения предоставления доступа 1
y2	Сигнал подтверждения предоставления доступа 2
y3	Сигнал подтверждения предоставления доступа 3

Таблица Б.7. Синхронные подсистемы

A1	Вычисление значения сигнала токена для отдельной линии запроса. Выходной сигнал токена генерируется спустя инстент после получения входного сигнала.
A2	Вычисление сигнала подтверждения предоставления доступа для отдельной линии запроса.
A3	Вычисление значения сигнала grand для передачи разрешения предоставления доступа для отдельной линии запроса.
A4	Запуск токена в первый инстент. Во всех остальных инстентах прозрачно передает значение входного сигнала.
Net1	Полная реализация отдельной линии запроса.
Net2	Вся система арбитра шины в целом.

Приложение B

Пример 2. Часы-будильник. Таблицы

Таблица B.1. Интерфейсные сигналы и состояния автомата A1

q1	Текущее время (устойчивое состояние)
q2	Текущее время + звонок включен (устойчивое состояние)
q3	Установка текущего времени
q4	Установка времени звонка
x1	Кнопка “mode”
x2	Кнопка “minute”
x3	Кнопка “hour”
x4	Сигнал таймера
y1	Включить таймер ожидания
y2	Выключить таймер ожидания
y3	Вкл./выкл. режим мигания дисплея
y4	Переключить на следующую мелодию звонка и проиграть ее
y5	Вкл./выкл. ежечасные оповещения
y6	Вкл./выкл. будильник
y7	Текущее время += 1 минута
y8	Текущее время += 1 час
y9	Время звонка += 1 минута
y10	Время звонка += 1 час
y11	Переключить режим: показывать “текущее время”/”время будильника”

Таблица В.2. Интерфейсные сигналы и состояния автомата А2

q1	Единственное состояние
x1	Кнопка “mode”
x2	Кнопка “minute”
x3	Кнопка “hour”
x4	Сигнал таймера
y1	Кнопка “mode”
y2	Кнопка “minute”
y3	Кнопка “hour”
y4	Сигнал таймера

Таблица В.3. Интерфейсные сигналы и состояния автомата А3

q1	Звонок1
q2	Звонок2
q3	Звонок3
x1	Переключить на следующую мелодию звонка и проиграть ее
y1	Проиграть звонок 1
y2	Проиграть звонок 2
y3	Проиграть звонок 3

Таблица В.4. Интерфейсные сигналы и состояния автомата А4

q1	Выключено

q2	Включено
x1	Переключить состояние
y1	Сигнал состояния

Таблица B.5. Интерфейсные сигналы синхронной сетиNet1

x1	Кнопка “mode”
x2	Кнопка “minute”
x3	Кнопка “hour”
x4	Сигнал таймера
y1	Включить таймер ожидания
y2	Выключить таймер ожидания
y3	Проиграть звонок 1
y4	Проиграть звонок 2
y5	Проиграть звонок 3
y6	Ежечасные оповещения (вкл./выкл.)
y7	Будильник (вкл./выкл.)
y8	Текущее время += 1 минута
y9	Текущее время += 1 час
y10	Время звонка += 1 минута
y11	Время звонка += 1 час
y12	Показывать “текущее время”/”время будильника”
y13	Режим мигания дисплея (вкл./выкл.)

Таблица B.6. Синхронные подсистемы

A1	Заключает в себе основную часть логики часов с будильником.

A2	Фильтрация входных сигналов автомата A1 (сигналы от трех кнопок и сигнал таймера). Только один из четырех сигналов может появиться на выходе.
A3	Хранение и проигрывание мелодии звонка.
A4	Переключатель на два положения.
Net1	Объединяет вместе все составные части модели.