УДК 004.415.53, 004.4'24

В.Н. Негода, В.А. Фолунин

АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ ВРЕМЕННЫХ ПРОТОТИПОВ ПРОГРАММ ЛОГИЧЕСКОГО УПРАВЛЕНИЯ

Негода Виктор Николаевич, *доктор технических наук, окончил радиотехнический факультет Ульяновского политехнического института, профессор кафедры «Вычислительная техника» Ульяновского государственного технического университета. Имеет статьи, монографии и авторские свидетельства в области проектирования встроенных систем контроля и управления. Область научных интересов – автоматизация проектирования логического управления техническими системами. [e-mail: nvn@ulstu.ru].*

Фолунин Владимир Александрович, *аспирант кафедры «Вычислительная техника» УлГТУ, окончил УлГТУ. Имеет статьи в области автоматизации тестирования. Область научных интересов – автоматизация тестирования реализаций алгоритмов и структур данных. [e-mail: v.folunin@ulstu.ru].*

Аннотация

В статье рассматриваются механизмы автоматизации тестирования прототипов программ логического управления, оказывающиеся полезными в условиях, когда прототипирование выполняется в сжатые сроки, охватывает множество функций и выполняется большим количеством разработчиков. Описаны специфические особенности задач логического управления, а также содержание процессов прототипирования и автоматизации тестирования прототипов. Показаны способы формализации и декомпозиции задач логического управления с использованием неоднородных функций многозначной логики, упрощающие разработку инфраструктуры для тестирования прототипов. Приведён пример автоматического создания генератора тестов по спецификации функции фильтрации помех двоичного сигнала.

Ключевые слова: проектирование систем логического управления, автоматизация тестирования программ, быстрое прототипирование, разработка программ на основе моделей и тестов.

AUTOMATION OF TESTING OF TEMPORAL PROTOTYPES OF LOGICAL CONTROL PROGRAMMES

Viktor Nikolaevich Negoda, *Doctor of Engineering, graduated from the Faculty of Radioengineering of Ulyanovsk Polytechnic Institute; Professor at the Department of Computer Engineering of Ulyanovsk State Technical University; an author of articles, monographs, and certificates of authorship in the field of computer-aided design of embedded control and management systems; interested in computer-aided design of technical systems with logical control. e-mail: nvn@ulstu.ru.*

Vladimir Aleksandrovich Folunin, *Postgraduate Student at the Department of Computer Engineering of Ulyanovsk State Technical University; graduated from the Faculty of Information Systems and Technologies of Ulyanovsk State Technical University; an author of articles in the field of testing automation; interested in testing automation for implementation of algorithms and data structures. e-mail: v.folunin@ulstu.ru.*

Abstract

The paper considers the means of testing automation for prototypes of logical control programs, which are of use in conditions of short-time prototyping involving a variety of functions performed by a large number of developers. The authors describe specific features of logical control tasks as well as the contents of prototyping process and prototype testing automation process. The ways of logical control tasks formalization and decomposition using heterogeneous functions of multi-valued logic, which simplify the development of the prototype testing infrastructure are shown. The authors also give an example of automated construction of the test generator based on the specification of binary glitches filtration function.

Key words: logical control systems design, software testing automation, rapid prototyping, model-driven development, test-driven development.

Введение

Создание временных прототипов при разработке программного обеспечения (ПО) нацелено на уточнение спецификаций требований и оценку рисков в части наиболее критических аспектов проектирования [1–3]. Важным критериальным параметром проектного процесса является время разработки ПО.

Тестирование ПО является трудоёмким процессом [4]. Его автоматизация становится рентабельной при обеспечении повторности использования тестов, что характерно для регрессионного тестирования. В этой связи тестирование временных прототипов, уточняющих спецификации требований и оценивающих риски тех или иных проектных решений, обычно выполняется вручную. Однако задачам логического управления присущи такие особенности, которые обеспечивают рентабельность применения средств автоматизации тестирования даже для временных прототипов этапа анализа требований. Дополнительные затраты времени, возникающие в связи с автоматизацией тестирования, предлагается компенсировать тремя способами:

- распараллеливанием действий, связанных с разработкой и тестированием прототипов;
- такой формализацией задач логического управления, которая ускоряет реализацию требуемой функциональности и сокращает время генерации тестов;
- повторным использованием сценариев тестирования в ходе проектирования системы и реализации проектных решений.

Ниже рассматривается специфика задач логического управления, а также проектный процесс, базирующийся на активном прототипировании и автоматизации тестирования.

Специфика задач логического управления

Для многих задач логического управления характерно наличие следующих обстоятельств, усложняющих проектирование:

- большое количество переменных и функциональных зависимостей, фигурирующих в задаче (такие спецификации могут вовлекать многостраничные формулы, таблицы решений, объёмные диаграммы состояний с сотнями функций переходов);
- широкие возможности выбора способов реализации функций логического управления, порождающих различные значения потребляемой памяти и времени обработки данных;
- порождение довольно большого количества ветвей программы;
- наличие расхождений между предполагаемым и реальным поведением управляемых агрегатов;
- необходимость учёта асинхронного поведения агрегатов объекта управления в реальном времени, что зачастую требует моделирования этого поведения в целях тестирования [5];
- широкий спектр систем программирования, используемых для реализации функций логического управления, что характерно для распределенных систем управления; при этом могут использоваться различные языки высокого уровня, ассемблеры, специальные языки релейно-контактных схем [6, 7], специальные языки автоматного программирования [8, 9], а также сочетания различных языков.

Важным обстоятельством, позволяющим упростить разработку, является возможность формализации функциональных связей на основе теории функций двузначных и многозначных переменных и логики предикатов. Это создаёт предпосылки для интеграции технологии разработки на основе моделей (MDD – Model-Driven Development) [10], технологии разработки на основе тестов (TDD – Test-Driven Development) [11] и технологии тестирования на основе данных (DDT – Data-Driven Testing) [12]. Технология MDD даёт возможность активно вовлекать формальные спецификации функций логического управления в процесс автоматической генерации тестов, благодаря чему создаются условия для достаточно быстрой и точной модификации тестов при изменении реализуемых функций. Технология TDD обеспечивает быструю реализацию прототипов за счет явной спецификации функциональных требований в формате тестов. Технология DDT, благодаря отделению тестовых данных от тестируемого кода, позволяет сохранять тесты прототипов для последующего использования при тестировании рабочих сборок.

В условиях недостаточной формализации исходных спецификаций реализуемых функций перечисленные выше особенности разработки обусловливают необходимость активного прототипирования на ранних её этапах с целью уточнения функциональных требований и учёта в этих требованиях реальных свойств агрегатов объекта управления. Функциональные требования находят наиболее точное представление в функциональных тестах. Кроме того, процесс функционального тестирования способен в значительной мере продемонстрировать логику создаваемой системы и дать разработчикам и представителям заказчика основания для ее уточнения.

Содержание проектных процессов прототипирования программ логического управления и автоматизации тестирования прототипов

Анализ требований, выполняемый с использованием временных прототипов, представляет собой итерационный процесс порождения уточнённых спецификаций требований, содержащий цикл создания и тестирования временных прототипов.

Особенностью подхода, предлагаемого авторами, являются формализация и декомпозиция алгоритмов функционирования системы логического управления, а также распараллеливание работ по созданию временных прототипов и их тестированию. Этот подход осуществляется в проектном процессе, представленном на рисунке 1.

В ходе интервьюирования представителей заказчика выполняется не только модификация спецификаций требований, но и формирование перечня задач прототипирования. Принятие решения о дополнении перечня

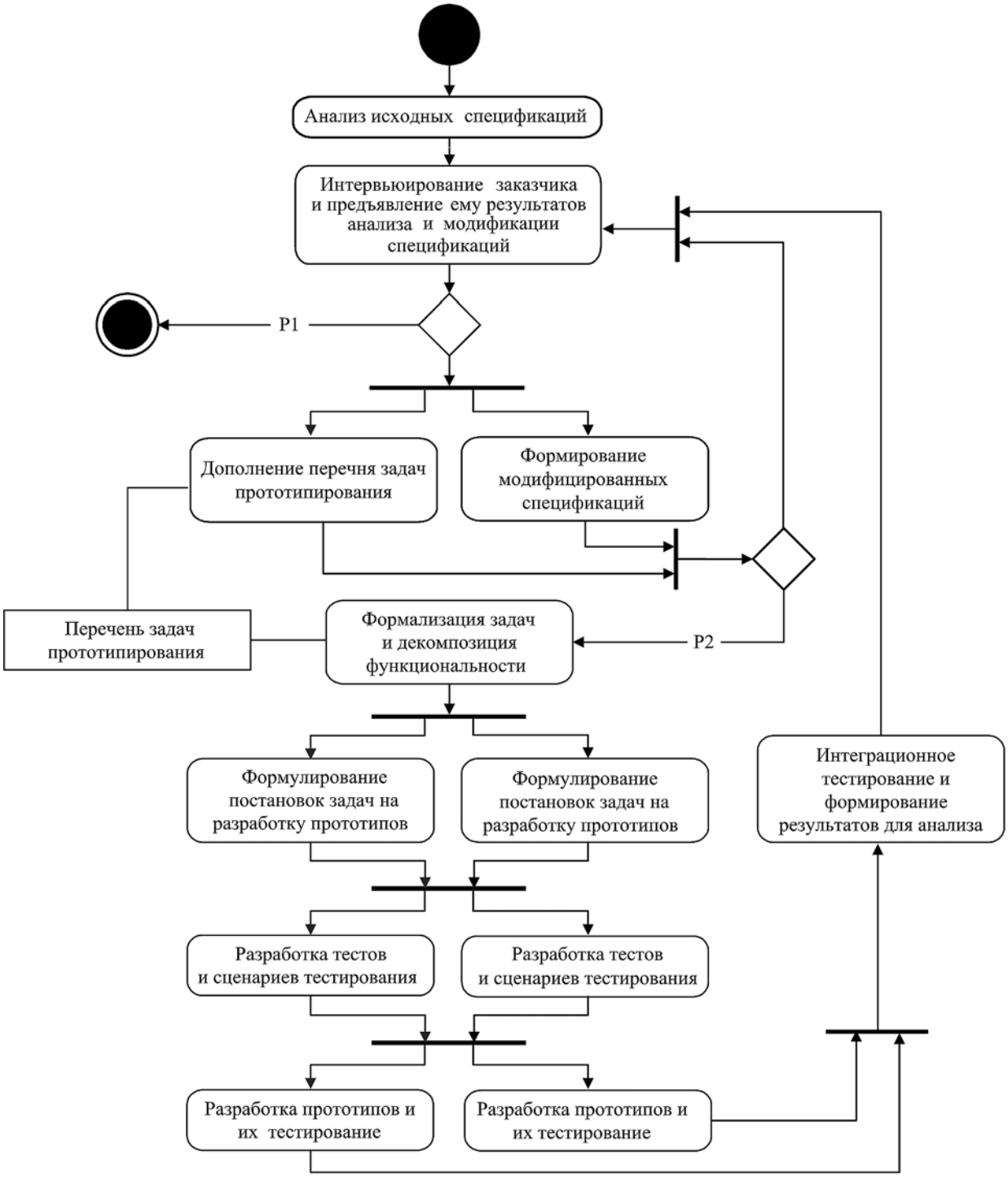

Рис. 1. Процесс уточнения спецификаций требований на основе быстрого прототипирования и автоматизации тестирования

происходит на основе оценки двух видов рисков: а) риск неодинакового понимания функциональных и параметрических требований в сознании разработчиков и заказчиков; б) риск выполнимости требований. Выход из цикла уточнения требований выполняется при снижении этих рисков до допустимого уровня (предикат P1).

Если перечень задач сформирован (предикат P2), то выполняются формализация задач и декомпозиция функциональности, в качестве который выступают как функциональные зависимости, так и процессы с состояниями. Уровень детализации декомпозиции при этом определяется четырьмя основными факторами: возможностью декомпозировать функциональность системы, сложностью прототипов и их тестов, необходимостью выполнить жёсткие параметрические ограничения реального времени, а также возможностями организовать параллельную разработку прототипов. Уровень распараллеливания на разных этапах разработки от формулировки задач на про-

тотипирование до собственно тестирования может быть различным. Декомпозиция процесса создает условия для вовлечения в наиболее простые работы большего числа исполнителей, нежели для более сложных работ, например формализации.

Процесс тестирования реализуется через запуск прототипа на множестве тестов, каждый из которых осуществляет элементарную проверку функциональной зависимости между исходными данными программной функции и её результатом.

В ходе автоматического тестирования используются три основных механизма создания потока тестовых пар вида (Input, Output):

- итератор, осуществляющий обход пар, предварительно созданных либо вручную, либо автоматически;
- генератор тестовых пар;
- имитатор поведения объекта управления.

Сценарии тестирования могут представлять собой различные виды деклараций данных и процедур, основными из которых являются:

- спецификации данных на языке системы, поддерживающей технологию DDT;
- исходные тексты на языках, используемых в средствах поддержки тестирования;
- настройки систем моделирования, используемых для имитации поведения объекта.

Важной особенностью рассматриваемого процесса является возможность применения различных средств автоматизации тестирования. В частности, перспективным представляется использование средств автоматизации тестирования, включаемых в состав систем автоматизации проведения турниров по спортивному программированию. Процессы проведения турниров и раннего прототипирования имеют важные общие свойства:

- потребность в быстром создании тестов, обладающих полнотой покрытия требований к реализуемым функциям;
- сжатые сроки реализации набора задач и их решений;
- решения, разрабатываемые в ходе спортивного программирования, в большинстве случаев не являются законченными приложениями, что отождествляет их с прототипами, корректно реализующими заданную функциональность;
- временные прототипы могут разрабатываться на различных языках программирования, что поддерживается в системах автоматизации турниров.

В условиях использования разноязыковых средств программирования временных прототипов интеграционное тестирование целесообразно реализовать через организацию программных каналов между программными процессами исполнения прототипов и процессом поддержки интеграции.

Формализация задач логического управления и декомпозиция функциональности

Наиболее применяемыми формализмами для задач логического управления являются булевы функции и конечные автоматы [13]. Многие функциональные зависимости, фигурирующие в этих задачах, наиболее естественным образом представляются неоднородными функциями многозначной логики (НФМЛ) [14]. Задачи регулирования в системах логического управления решаются обычно на основе моделей дискретных регуляторов: двухпозиционных, многопозиционных, ПИД-регуляторов [15].

Исходная декомпозиция системы логического управления обычно задаётся группами агрегатов, каждая из которых связана с какой-то частью технологического процесса, например, подсистема включения, выключения и охлаждения силовых трансформаторов, подсистема управления вытяжной вентиляцией, подсистема управления гидравлической насосной станцией и т. п. Широко распространённым техническим решением является создание для каждой такой подсистемы своего шкафа управления с размещением в нем контроллера, модулей ввода-вывода и пульта автономного управления для поддержки конфигурирования и диагностики с участием обслуживающего персонала.

При использовании технологии объектно-ориентированного моделирования функциональность каждой подсистемы может декомпозироваться по прецедентам [3] и даже акторам. В качестве акторов, кроме человека-оператора и обслуживающего персонала, выступают агрегаты объекта управления, например, датчик давления, датчик положения, датчик протечки, кнопка пуска, джойстик ручного позиционирования и т. п. К прецедентам при этом будут относиться, например, такие: конфигурирование регулятора, диагностика состояния процесса, ручное управление с пульта, пуск агрегата и т. п.

Большинство прецедентов реализуется в виде процесса, предполагающего смену нескольких состояний. Естественной моделью процесса в системах логического управления является конечный автомат, активно используемый в так называемом автоматном программировании [13, 16], или реактивная система [17].

В задачах логического управления широко распространено сочетание двух механизмов выдачи управляющих сигналов: при переходе из одного состояния в другое и во время пребывания в состоянии. Это соответствует модели смешанного автомата Мура-Мили, называемого С-автоматом [16].

Декомпозиция конечных автоматов обычно строится на основе группировки состояний, т. е. выделения подграфа диаграммы состояний. На основе объединения нескольких состояний могут быть созданы композитные состояния, например, состояние вывода всех элементов агрегата в исходное положение, состояние перехода в аварийный режим.

В работе [18] рассмотрены несколько вариантов структурно-функциональной организации автоматных моделей, порождаемых в ходе декомпозиции исходного абстрактного автомата.

Исходный автомат представляется следующей спецификацией:

$$FSM = (X, Y, S, fSsx, fYs, fYsx, s\_0),$$

где X, Y – множество состояний входов и выходов автомата; следует иметь в виду, что элементы этих множеств являются чаще всего векторами, компонентам которых в системе логического управления соответствуют дискретные входы и выходы;

$S = \{s\_0, s\_1, \ldots\}$ – множество внутренних состояний автомата, которые в системах логического управления обычно кодируются натуральными числами;

$fSsx$: $S \times X \rightarrow S$ – функция переходов;

fYs: $S \rightarrow Y$ – функция выходов, зависящих только от текущего состояния, т. е. функция выходов в части модели Мура;

$fYsx$: $S \times X \rightarrow Y$ – функция выходов, зависящих от текущего внутреннего состояния и состояния входов, т. е. в части модели Мили;

s_0 – начальное состояние.

Традиционное автоматное программирование базируется на таком множестве элементов формальной спецификации FSM, которое является спецификацией диаграммы состояний. Переход от классического определения С-автомата, представленного выше спецификацией FSM, к автоматной диаграмме основан на двух продуктивных идеях:

а) для каждого состояния определяются такие функции перехода и выхода, которые реализуются без использования кода состояния; это есть декомпозиция функции перехода $fSsx$ и функции выхода fYs, которая по сути дела базируется на обобщении разложения Шеннона, аналогичном приведённому в работе [19];

б) для каждой дуги перехода определяется логическая функция, истинность которой однозначно определяет новое состояние и значение $fYsx$.

Реализация только первой из указанных идей порождает систему множеств функций перехода и выхода, не зависящих от состояния s:

$$sfSsx = \{fSx[s\_0](x), fSx[s\_1](x), \ldots\},$$

$$sfYs = \{fY[s\_0], fY[s\_1], \ldots\},$$

$$sfYsx = \{fYx[s\_0](x), fYx[s\_1](x), \ldots\},$$

где $x \in X$.

Отличие формата записи системы функций от традиционно используемых в математике обусловлено тем, что спецификация должна играть роль входных данных для системы автоматической генерации программ и тестов. При этом должна существовать возможность использовать для составления спецификаций любой текстовый редактор. Правила именования функций ориентированы на упрощение грамматического разбора, который проводится либо в целях генерации тестов и/или кодов прототипов, либо в целях перевода спецификации на языки сторонних инструментальных систем, обстоятельный обзор которых представлен в работе [20].

Каждая функция из приведённой выше системы в общем случае является НФМЛ. Для декомпозиции автоматной модели можно иметь три прототипа, каждый из которых реализует одну из функций $sfSsx$, $sfYs$, $sfYsx$. В случае, когда множество X очень велико, данные функции также подвергаются декомпозиции. В качестве структурообразующей основы при этом целесообразно брать векторную структуру элемента множества X и использовать разложение Шеннона по аргументам, соответствующим компонентам вектора.

Например, пусть каждый элемент множества X представляется вектором вида:

$$vX = <UP, DOWN, LEFT, RIGHT, MODE>.$$

Первые 4 компонента представляют 4 сигнала с датчиков положения, а последний компонент – состояние 5-позиционного переключателя. Мощность множества X при этом равна 80. При выделении в качестве основы декомпозиции компонента $MODE$ мы получим 5 подмножеств множества X по 16 элементов в каждом. При этом для декомпозиции системы функций $sfSsx$, $sfYs$, $sfYsx$ потребуется сначала представить каждую из них изоморфной системой функций от переменных UP, $DOWN$, $LEFT$, $RIGHT$, $MODE$, а затем применить к ним разложение Шеннона по переменной $MODE$. Таким образом, функция вида $sfSsx[s\_i](X)$ изоморфно отображается в систему из 5 функций:

$$\{sfSsx\_vX[s\_i](MODE = m, UP, DOWN, LEFT, RIGHT) \mid m = \{0\ldots4\}\}.$$

Постфикс $\_vX$ в именах функций отражает факт, что векторизация аргумента функции $sfSsx[s\_i](x)$ породила сначала функцию $sfSsx[s\_i](MODE, UP, DOWN, LEFT, RIGHT)$, а затем разложение Шеннона по первому аргументу породило уже систему из 5 функций. В этой системе для каких-то функций могут быть безразличными переменные, представляющие некоторые значения сигналов датчиков положения. Это обстоятельство обычно приводит к существенному уменьшению сложности реализации и тестирования соответствующих программ.

Несмотря на то, что в результате разложения все аргументы полученной изоморфной системы функций становятся двоичными, результат разложения будет иметь тип НФМЛ для случая, когда мощности множеств значений функций $sfSsx$, $sfYs$, $sfYsx$ больше 2.

Одновременная реализация двух представленных выше идей порождает следующую декомпозицию автомата, являющуюся по сути спецификацией диаграммы состояний:

$$SD = (X, Y, S, V, s\_0),$$

где X, Y, S, s_0 – множества входов, выходов, состояний и начальное состояние;

$V = \{(s, Ys, Us) \mid s \in S\}$ – множество спецификаций узлов диаграммы автомата;

$Ys \subset Y$ – подмножество выходов, фигурирующих в состоянии (выходы в части автомата Мура);

$Us = \{(Pi, Sxi, Yxi) \mid i \in \{1, 2, \ldots\}\}$ – множество

троек, каждая из которых описывает одну дугу, выходящую из состояния s;

Pi – спецификация предиката, соответствующего условию перехода i-й дуги;

Sxi – состояние, в которое ведёт i-я дуга;

Yxi – выходы в части автомата Мили, формирующиеся при переходе по i-й дуге.

Если предикаты содержат предметные переменные только с двумя значениями, то их формализация (и, возможно, декомпозиция) строится на базе булевых функций. При наличии многозначных предметных переменных используются НФМЛ. Однако это не исключает использования булевой алгебры в представлениях НФМЛ. Рассмотрим это на примере.

Пусть набор предикатов переходов имеет вид:

$$P(leadOne(b\_1, b\_2, b\_3, b\_4) = k),$$

где $leadOne$ – функция определения номера левой единицы в булевом векторе;

$b_$i – бинарные входы;

$k \in \{0..4\}$ – значение функции.

Функция $leadOne$ относится к классу НФМЛ, поскольку её значность равна 5. В формальной спецификации функцию $leadOne$ можно задать следующим множеством пар соответствия значений функции конъюнкциям литералов (отрицание представляется знаком !, конъюнкция – знаком &&):

$$\{(!b\_1 \;\&\&\; !b\_2 \;\&\&\; !b\_3 \;\&\&\; !b\_4, 0), (b\_1, 1), (!b\_1 \;\&\&\; b\_2, 2), (!b\_1 \;\&\&\; !b\_2 \;\&\&\; b\_3, 3), (!b\_1 \;\&\&\; !b\_2 \;\&\&\; !b\_3 \;\&\&\; b\_4, 4)\}.$$

Для автоматизации генерации теста целесообразно использовать представление конъюнкций троичными векторами:

$$\{(0000,0), (1\text{---},1), (01\text{--},2), (001\text{-},3), (0001,4)\}.$$

Эта спецификация изоморфна предыдущей в силу наличия взаимно-однозначного соответствия множества троичных векторов множеству конъюнкций. Спецификация задаёт разбиение наборов значений аргументов на классы эквивалентности, каждый из которых представляется троичным вектором. Это гарантирует полноту тестирования. Кроме того, спецификация образует набор входных данных для интерпретатора подобных таблиц НФМЛ, при наличии которого в составе системы автоматизации тестирования создание теста занимает считанные минуты.

Пример создания тестов по спецификации тестируемой функции

Рассмотрим программную функцию, обеспечивающую устранение импульсных помех двоичного сигнала. В общем случае на вход функции поступает двоичная последовательность длины N, а также предельная длина помехи M. Помехами считаются все подпоследовательности, которые имеют длину, меньшую или равную M, и состоят из одинаковых значений, не совпадающих с доминирующим значением всей последовательности. Результатом функции является исходная последовательность, в которой все подпоследовательности-помехи заменены на повторения доминирующего значения соответствующее число раз.

Для упрощения дальнейшего изложения примем, что M равно 3, а доминирующим значением последовательности является логический нуль. Кроме того, пусть последовательность всегда заканчивается нулём.

Спецификация рассматриваемой функции с использованием элементов грамматики регулярных выражений может иметь следующий вид:

$$\{(0^*, 0^*), (1\text{^}n0^* \mid (n <= 3), 0\text{^}n0^*), (1\text{^}n0^* \mid (n > 3), 1\text{^}n0^*)\}.$$

Здесь необходимо сделать несколько пояснений.

Прежде всего, из-за отсутствия явных ограничений на длину входной последовательности L в спецификации функции невозможно явно указать всё множество входных двоичных векторов. В связи с этим правила, описанные в спецификации, применяются не ко всей входной последовательности, а к её префиксу; символ * отражает необходимость применения того же набора правил для оставшейся части последовательности. Генерация тестовых данных при такой форме спецификации тестируемой функции может выполняться итеративно либо рекурсивно.

Так как результат обработки подпоследовательностей, состоящих из единиц, зависит только от их длин, данный факт должен быть отражён в спецификации в виде

```
1 registerGen(argc, argv);
2 int tokensCount = rnd.next(1, 100);
3 while (tokensCount--) {
4    int tokenType = rnd.next(1, 3);
5    switch (tokenType) {
6       case 1:
7          fprintf(in, «0»);
8          fprintf(out, «0»);
9          break;
10      case 2:
11         int n = rnd.next(1, 3);
12         for (int i = 0; i < n; i++)
13            fprintf(in, «1»);
14         fprintf(in, «0»);
15         for (int i = 0; i < n; i++)
16            fprintf(out, «0»);
17         fprintf(out, «0»);
18         break;
19      case 3:
20         int n = rnd.next(4, 100);
21         for (int i = 0; i < n; i++)
22            fprintf(in, «1»);
23         fprintf(in, «0»);
24         for (int i = 0; i < n; i++)
25            fprintf(out, «1»);
26         fprintf(out, «0»);
27         break;
28   }
29 }
```

Листинг 1. Программа генерации тестов, созданная по спецификации тестируемой функции

нескольких правил, отличающихся только уточнениями (в данном случае это второе и третье правило). Запись $x\hat{}n$ обозначает значение x, повторяющееся n раз. После символа | следует уточнение, то есть дополнительное условие, связанное со значениями используемых в спецификации переменных.

Показанная спецификация функции может быть (в том числе автоматически) транслирована в программу генерации тестовых входных и выходных данных, показанную в листинге 1. В качестве вспомогательного средства используется библиотека Testlib [21], широко применяемая для автоматизации тестирования решений в турнирах по спортивному программированию.

Как упомянуто ранее, использование символа * в спецификации требует инкрементального подхода к генерации тестовых пар, поэтому в генераторы, создаваемые на основе таких спецификаций, прежде всего включается цикл (строки 2, 3, 29), последовательно добавляющий случайное количество (tokensCount) фрагментов входных и выходных данных. В тело цикла включается конструкция switch (строки 4–6, 10, 19, 28), осуществляющая выбор одного из правил случайным образом.

Внутри опций switch включаются команды, выполняющие запись фрагментов входных и выходных данных в соответствии с конкретным правилом спецификации (строки 7–9, 11–18, 20–27). В файл in записываются входные данные тестовой пары, в файл out – эталонные выходные данные.

В том случае, если в спецификации используется запись $x\hat{}n$, формируется случайное значение n, а затем цикл вывода символа x соответствующее число раз.

Заключение

Производительность и качество труда разработчиков временных прототипов программ логического управления и средств их тестирования может быть существенно повышена за счёт использования формальных моделей для спецификации задач логического управления, декомпозиции функций создаваемой системы, поддержки независимой разработки прототипов большим числом программистов, автоматическим тестированием этих прототипов с частичной реализацией функций комплексного тестирования. Для значительной части функций логического управления имеется возможность трансформировать их формальные спецификации либо в тестовые сценарии, либо в спецификации тестовых пар входных и выходных данных.

СПИСОК ЛИТЕРАТУРЫ

1. Gordon V.S, Bieman J.M. Reported Effects of Rapid Prototyping on Industrial Software Quality // Software Suality Journal. 1993. Vol. 2, № 2. pp. 93–108.

2. Анализ и управление рисками, связанными с информационным обеспечением человеко-машинных АСУ технологическими процессами в реальном времени / Д.Ю. Учаев, Ю.М. Брумштейн, И.М. Ажмухадедов, О.М. Князева, И.А. Дюдиков // Прикаспийский журнал: управление и высокие технологии. – 2016. – № 2 (34). – С. 82–97.

3. Гома Х. UML. Проектирование систем реального времени, распределенных и параллельных приложений. – М. : ДМК Пресс, 2002. – 704 с.

4. Майерс Г., Баджетт Т., Сандлер К. Искусство тестирования программ. – 3-е изд. – М. : Диалектика, 2016. – 272 с.

5. Журавлев С.С., Окольнишников В.В., Рудометов С.В. Инструментальные средства отладки и тестирования программ управления АСУ ТП // Актуальные проблемы гуманитарных и естественных наук. – 2016. – № 2. – С. 49–54.

6. Контроллеры DirectLOGIC. Основы программирования. – URL: https://www.plcsystems.ru/catalog/DirectLOGIC_2/doc/BaseRLL.pdf (дата обращения: 10.05.2017).

7. Петров И.В. Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования / под ред. проф. В.П. Дьяконова. – М. : СОЛОН-Пресс, 2004. – 256 с.

8. Finite State Machine Language 0.1.2. – URL: http://finite-state-machine-language.soft112.com (дата обращения: 10.05.2017).

9. Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А. Язык State Machine – расширение языка Java для эффективной реализации автоматов // Информационно-управляющие системы. – 2005. – № 1. – С. 16–24.

10. Parviainen P., Takalo J., Teppola S., Tihinen M. Model-Driven Development. Process and practices. – URL: http://www.vtt.fi/inf/pdf/workingpapers/2009/ W114.pdf (дата обращения: 10.05.2017).

11. Бек К. Экстремальное программирование: разработка через тестирование. – СПб. : Питер, 2003. – 224 с.

12. Data Driven Testing. – URL: https://docs.microsoft.com/ru-ru/windows-hardware/drivers/taef/data-driven-testing (дата обращения: 10.05.2017).

13. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации алгоритмов. – СПб. : Наука, 2000. – 780 с.

14. Негода В.Н. Неоднородные функции многозначной логики в программировании задач логического управления // Информатика, моделирование, автоматизация проектирования : сб. науч. тр. – Ульяновск : УлГТУ, 2010. – С. 20–46.

15. Олсон Г., Пиани Дж. Цифровые системы автоматизации и управления. – СПб. : Невский диалект, 2001. – 557 с.

16. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. – СПб., 2008. – 167 с.

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

18. Негода В.Н. Унификация проектных решений при автоматном программировании систем логического управления // Автоматизация процессов управления. – 2014. – № 3. – С. 103–111.

19. Горбатов А.В. Характеризационная теория синтеза функциональных декомпозиций в k-значных логиках. – М. : Изд-во физико-математ. литературы, ООО «Издательство АСТ», 2000. – 336 с.

20. Обзор подходов к верификации распределенных систем / И.Б. Бурдонов, А.С. Косачев, В.Н. Пономаренко, В.З. Шнитман. – М. : Российская Академия Наук. Институт системного программирования (ИСП РАН), 2003. – 51 с.

21. Testlib. – URL: https://github.com/MikeMirzayanov/testlib (дата обращения: 10.05.2017).

REFERENCES

1. Gordon V.S, Bieman J.M. Reported Effects of Rapid Prototyping on Industrial Software Quality. *Software Suality Journal,* 1993, vol. 2, no. 2, pp. 93–108.

2. Uchaev D.Iu., Brumshtein Iu.M., Azhmukhadedov I.M., Kniazeva O.M., Diudikov I.A. Analiz i upravlenie riskami, sviazannymi s informatsionnym obespecheniem chelovekomashinnykh ASU tekhnologicheskimi protsessami v realnom vremeni [Analysis of Possibilities for Using of Amorphous Microwires in Magnetiompedance Sensors Developed for Data Measuring and Control Systems]. *Prikaspiiskii zhurnal. Upravlenie i vysokie tekhnologii* [Caspian Journal. Management and High Technologies], 2016, no. 2 (34), pp. 82–97.

3. Goma H. *UML. Proektirovanie sistem realnogo vremeni, raspredelennykh i parallelnykh prilozhenii* [Design Concurrent, Distributed, and Real-Time Applications with UML]. Moscow, DMK Press Publ., 2002. 704 p.

4. Mayers G., Badzhett T., Sandler K. *Iskusstvo testirovaniia programm*. 3-e izd. [The Art of Software Testing. 3d Edition]. Moscow, Dialektika Publ., 2016. 272 p.

5. Zhuravlev S.S., Okolnishnikov V.V., Rudometov S.V. Instrumentalnye sredstva otladki i testirovaniia programm upravleniia ASU TP [Tools for Debugging and Testing of Software for Automated Technical Process Management System (ATPMS)]. *Aktualnye problemy gumanitarnykh i estestvennykh nauk* [Current Problems of Liberal and Natural Sciences], 2016, no. 2, pp. 49–54.

6. Kontrollery DirectLOGIC. Osnovy programmirovaniia. Available at: https://www.plcsystems.ru/catalog/DirectLOGIC_2/doc/BaseRLL.pdf (accessed: 10.05.2017).

7. Petrov I.V. *Programmiruemye kontrollery. Standartnye iazyki i priemy prikladnogo proektirovaniia*. Pod red. prof. V.P. Diakonova [Programmable Controllers. Standard Language and Applied Designing Techniques. Edited by Professor V.P. Diakonova]. Moscow, Solon-Press Publ., 2004. 256 p.

8. Finite State Machine Language 0.1.2. Available at: http://finite-state-machine-language.soft112.com (accessed: 10.05.2017).

9. Korneev G.A., Shamgunov N.N., Shalyto A.A. Iazyk State Machine – rasshirenie iazyka Java dlia effektivnoi realizatsii avtomatov [State Machine–Java Language Expansion for Effective Performance of Automatic Devices]. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2005, no. 1, pp. 16–24.

10. Parviainen P., Takalo J., Teppola S., Tihinen M. *Model-Driven Development. Process and Practices*. Available at: http://www.vtt.fi/inf/pdf/workingpapers/2009/ W114.pdf (accessed: 10.05.2017).

11. *Bek K. Ekstremalnoe programmirovanie: razrabotka cherez testirovanie* [Extreme Programming: Developing Through Testing]. St. Petersburg, Piter Publ., 2003. 224 p.

12. *Data Driven Testing*. Available at: https://docs.microsoft.com/ru-ru/windows-hardware/drivers/taef/data-driven-testing (accessed: 10.05.2017).

13. Shalyto A.A. *Logicheskoe upravlenie. Metody apparatnoi i programmnoi realizatsii algoritmov* [Logical Control. Methods for Hardware and Software Implementation of Algorithms]. St. Petersburg, Nauka Publ., 2000. 780 p.

14. Negoda V.N. Neodnorodnye funktsii mnogoznachnoi logiki v programmirovanii zadach logicheskogo upravleniia [Heterogeneous Functions of Multi-Valued Logic in Logical Control-Oriented Programming]. *Informatika, modelirovanie, avtomatizatsiia proektirovaniia. Sb. nauch. tr.* [Informatics, Modeling, Computer-Aided Design. Proceedings]. Ulyanovsk, UlSTU Publ., 2010, pp. 20–46.

15. Olsson G., Piani G. *Tsifrovye sistemy avtomatizatsii i upravleniia* [Digital Automation and Control Systems]. St. Petersburg, Nevskii dialect Publ., 2001. 557 p.

16. Polikarpova N.I., Shalyto A.A. *Avtomatnoe programmirovanie* [Off-line Programming]. St. Petersburg, 2008. 167 p.

17. Shalyto A.A., Tukkel N.I. SWITCH-tekhnologiia – avtomatnyi podkhod k sozdaniiu programmnogo obespecheniia “reaktivnykh” system [SWITCH-Technology is an Autonomous Approach to Creation of Software for Supporting Reactive Systems]. *Programmirovanie* [Programming], 2001, no. 5, pp. 45–62.

18. Negoda V.N. Unifikatsiia proektnykh reshenii pri avtomatnom programmirovanii sistem logicheskogo upravleniia [The Unification of Design Solutions at Automata-based Programming of Logic Control Systems]. *Avtomatizatsiia protsessov upravleniia* [Automation of Control Processes], 2014, no. 3, pp. 103–111.

19. Gorbatov A.V. *Kharakterizatsionnaia teoriia sinteza funktsionalnykh dekompozitsii v k-znachnykh logikakh* [Characterization Theory of Functional Decomposition Synthesis in k-valued Logics]. Moscow, Izd-vo fiziko-matemat. literatury Publ., Izdatelstvo AST, LLC Publ., 2000. 336 p.

20. Burdonov I.B., Kosachev A.S., Ponomarenko V.N., Shnitman V.Z. *Obzor podkhodov k verifikatsii raspredelennykh sistem* [Review of Approaches to Distributed System Verification]. Moscow, Rossiiskaia Akademiia Nauk. Institut sistemnogo programmirovaniia (ISP RAS) Publ., 2003. 51 p.

21. *Testlib*. Available at: https://github.com/MikeMirzayanov/testlib (accessed: 10.05.2017).