Статья опубликована в журнале "Надежность", 2007, № 1

Татарчевский В. А.

ПРИМЕНЕНИЕ SWITCH-ТЕХНОЛОГИИ В ЗАДАЧАХ УПРАВЛЕНИЯ ТЕХНОЛОГИЧЕСКИМИ ПРОЦЕССАМИ

В статье рассматривается метод проектирования надежных программных систем логического управления на основе SWITCH-технологии, предлагаются способы упрощения графов переходов программного обеспечения путем введения локального таймера и параллелизма потоков выполнения программы. Также в статье рассматриваются преимущества SWITCH-технологии в плане повышения надежности программного обеспечения, его верифицируемости и модифицируемости.

В настоящее время для построения систем логического управления (ЛУ) широко используется микропроцессорная и микроконтроллерная техника. Большое количество автоматических систем управления технологическими процессами строится на базе промышленных компьютеров и программируемых логических контроллеров (ПЛК). При проектировании таких систем основная часть трудоемкости приходится на разработку программного обеспечения (ПО). От надежности функционирования ПО зависит надежность работы всей системы управления. Однако построение надежных программных систем представляет собой сложную и в общем виде пока нерешенную задачу. Низкая надежность сложных программных систем связана в первую очередь с сильным влиянием «человеческого фактора» [1, 2]. На надежность программных систем непосредственно влияют уровень квалификации программистов, методы организации их работы, методы проектирования, кодирования и тестирования ПО, применяемые в конкретной организации в процессе разработки. В последнее время появился ряд продуктов и связанных с ними методик, предназначенных для повышения эффективности труда программистов, такие, как, например, *IBM Rational Rose* [3] и методика экстремального программирования [4]. Однако в силу различных причин ни один из этих методов не обеспечивает решения всех проблем, возникающих при разработке ПО.

Одной из главных проблем разработки ПО в настоящее время является недокументированность (или недостаточный уровень документированности) программных проектов [5, 6]. В большинстве случаев основным документом, описывающим структуру и поведение программы, является ее исходный текст, что сильно осложняет процессы отладки, модификации и сопровождения ПО. Некоторые надежды на изменение ситуации связаны с появлением графического языка *UML* [7, 8], представляющего собой достаточно мощный инструмент описания структуры и поведения ПО.

Для задач ЛУ особое значение имеет раздел *UML*, описывающий диаграммы состояний. Диаграмма состояний представляет собой, по сути, граф перехо-

дов конечного автомата (КА) с некоторыми отличиями. Они заключаются в том, что на диаграмме состояний не изображаются петли (переходы из состояния в то же состояние), сами состояния являются сложными объектами, имеющими, в общем случае, действие при входе, действие при выходе, деятельность, внутренние переходы и отложенные события. Также диаграммы состояний могут содержать составные состояния, имеющие подсостояния, и параллельные потоки. Стандарт *UML* описывает еще ряд дополнительных возможностей, расширяющих модель КА. В рамках модели диаграмм состояний *UML* можно описывать автоматы Мура, Мили, а также автоматы смешанного типа.

Главное преимущество от применения диаграмм состояний при проектировании систем ЛУ заключается в том, что появляется возможность создания графической документации на ПО, точно и в полной мере отражающей поведение и структуру ПО. Представление логики программы в виде КА позволяет применить для анализа поведения ПО теорию КА. При КА может быть исследован на предмет достижимости всех его состояний, отсутствие тупиковых состояний, может быть проанализирована правильность взаимодействий частей программы. Это позволяет повысить надежность ПО.

Основным методом реализации программ, представленных в виде КА на языке высокого уровня (ЯВУ), является SWITCH-технология (называемая также программированием с явным выделением состояний, или автоматным программированием). Она получила свое название от оператора switch языка С, который является основной структурой при написании программ в SWITCH-технологии.

SWITCH-технология разрабатывается в России с 1991 года. Ведущая роль в разработке технологии принадлежит А.А. Шалыто, профессору Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Впервые SWITCH-технология была описана в книге А.А. Шалыто «SWITCH-технология. Алгоритмизация и программирование задач логического управления» [9] и получила дальнейшее развитие в работах [10–12]. SWITCH-технология находит широкое применение не только при программной реализации ЛУ, но и в других областях, таких, как, например, визуализаторы алгоритмов [13, 14]. Большое количество работ, посвященных SWITCH-технологии, показывает актуальность данного подхода. Более подробно со SWITCH-технологией можно познакомиться на сайте http://is.ifmo.ru.

В данной статье рассмотрен способ повышения эффективности SWITCH-технологии, основанный на введении в модель КА локальных таймеров и параллельных потоков в явном виде.

Локальным таймером в предлагаемой модели является выделенная переменная, увеличивающее свое значение через фиксированные интервалы времени (например, каждую секунду). При входе в состояние значение этой переменной обнуляется (таймер инициализируется). В результате появляется возможность указывать в качестве условия выхода из состояния определенное значение данной переменной, соответствующее необходимому интервалу времени. Это позволяет упростить граф переходов КА. Следует отметить, что наряду с локальными таймерами возможно также применение и глобальных таймеров, которые инициализируются в одном состоянии, а являются условиями выхода в другом состоянии (состояниях).

Наличие параллелизма является особенностью, характерной именно для задач ЛУ. Язык *UML* описывает только один вариант параллелизма в диаграммах состояний, заключающийся в том, что составное состояние может включать в себя несколько КА, работающих параллельно. В данной статье предлагается указывать параллельные потоки на графе переходов в явном виде, используя для изображения разветвлений и слияний потока управ-

ления специальный символ из двух параллельных линий. Также в статье показаны преимущества использования SWITCH-технологии по сравнению с традиционными методами программирования на примере реализации несложной системы управления технологическим процессом.

Рассмотрим пример реализации автомата, использующего потоки и таймеры. Для примера возьмем автомат, описанный в работе [9, стр. 343, рис. 13.1]. Эта схема приведена на рис. 1. В работе [9] на рис. 13.2 приведена схема алгоритма работы устройства управления. Приведем и ее (рис. 2).

Словесное описание работы системы.

1. В исходном состоянии все клапана закрыты, мотор (двигатель) мешалки выключен.

2. Этап 1. При нажатии кнопки «Пуск» осуществляется налив жидкости путем открытия клапана 1 до срабатывания датчика $x3$.

3. Этап 2. Осуществляется налив жидкости путем открывания клапана 2 до срабатывания датчика $x4$.

4. Этап 3. Включается двигатель мешалки на время $t1$.

5. Этап 4. Производится слив смеси путем открывания клапана 3 на время $t2$.

6. После выполнения пункта 5 система возвращается в исходное состояние.

7. При нажатии кнопки «Стоп» во время выполнения пунктов 2 – 4 система переходит к сливу смеси (пункт 5).

Построим по словесному описанию автомат Мура, использующий локальные таймеры (рис. 3).

Заметим, что схема алгоритма на рис. 2 содержит 12 элементов, а эквивалентный ей граф переходов на рис. 3 состоит всего из пяти состояний. При этом за счет введения локального таймера (всего одной переменной на весь автомат) мы избавились от «искусственных» входов $T1$, $T2$, выходов $t1$, $t2$ и двух элементов задержки ФЭЗ1 и ФЭЗ2. Можно заметить, что у данной схемы есть один недостаток – из всех «рабочих» состояний, кроме последнего (из состояний 2 – 4), в состояние 5 ведут переходы с условием $x2$, соответствующим нажатию кнопки «Стоп». С учетом того, что в сложных технологических установках число состояний автомата в рабочем цикле может достигать нескольких десятков (и даже сотен), такие переходы могут сильно загромождать схему, затрудняя ее читаемость. Также подобный «прямой» подход может привести к ошибкам при разработке программы и особенно при ее модификации (из какого-либо состояния забыли провести переход по кнопке «Стоп»). Теперь представим себе, что в процессе разработки было решено использовать в качестве кнопки «Стоп» не нормально-разомкнутую, а нормально-замкнутую (как это и имеет место в стандартных двухкнопочных постах «пуск-стоп»). Для этого необходимо произвести в соответствующих переходах из состояний 2 – 4 замену $X2$ на $\sim X2$ ($X2$ инверсное). При этом небольшое, казалось бы, изменение является «ошибкоопасным», так как замену придется производить во всех рабочих состояниях. Поэтому целесообразно преобразовать граф переходов на рис. 3 в схему с параллельными потоками, изображенную на рис. 4.

Несмотря на то что количество состояний в новой схеме увеличилось до шести, число переходов в ней уменьшилось, а ее наглядность возросла. И, что самое главное, возросла модифицируемость схемы. Теперь можно ввести любое количество состояний рабочего цикла вместо состояний 2–4, не заботясь о корректности срабатывания кнопки «Стоп», а замена условия останова системы (например, замена $x2$ на $\sim x2$) влечет за собой изменение только в одном месте программы.

Сравним реализацию данного алгоритма на языке C при традиционном подходе и при использовании SWITCH-технологии. Код для обоих вариантов приведен в табл. 1, в левом столбце таблицы приведен код для схемы алгоритма по рис. 2, в правом – для «улучшенной» схемы, приведенной на рис. 4.

Следует отметить, что в таблице приведена скорее «условная» реализация, чем реальный код, в нем опущены многие подробности реализации, такие, как объявления переменных и функций, опрос входов автомата и выдача воздействий на выходы. Функция *starttimer( )*, обнуляющая таймерную переменную, должна выполняться только при первом входе в состояние, что довольно просто решается введением дополнительного флага. Чтобы излишне не перегружать код, здесь эта конструкция также не приведена.

Таблица 1

Реализация алгоритма логического управления при традиционном подходе и в SWITCH-технологии

```
//Традиционный подход
while(1) {
  z1 = z2 = z3 = z4 = 0;
  while(~x1);
  if(~x2){
    z1 = 1;
    while(~(x3 || x2));
    z1 = 0;
    if(~x2){
      z3 = 1;
      while(~(x4 || x2));
      z3 = 0;
      if(~x2){
        x2 = 1; t1 = 1;
        while(~(T1 || x2));
        x2 = 0; t1 = 0;
      };
    };
  };
  z4 = 1; t2 = 1;
  while(~T2);
};
```

```
//switch-подход
while(1) {
  switch(state){
    case1: z1 = z2 = z3 = z4 = 0;
      if(x1) state = 2;
      break;
    case2: z1 = 1;
           z2 = z3 = z4 = 0;
      if(x3) state = 3;
      break;
    case3: z1 = z2 = z4 = 0;
           z3 = 1;
      if(x4) state = 4;
      break;
    case4: z1 = z3 = z4 = 0;
           z2 = 1;
           starttimer();
      if(timer >= t_1) state = 5;
      break;
    case5: z1 = z2 = z3 = 0;
           z4 = 1;
           starttimer();
      if(timer >= t_2) state = 1;
      break;
  };
  if((state > 1)&&(state < 5)&& x2)
    state = 5;
};
```

Реализация алгоритма, записанного в виде схемы алгоритма, неочевидна. Даже для столь простой программы она требует от программиста некоторых творческих усилий, интуитивного подхода к написанию текста программы. Алгоритм может быть описан на ЯВУ несколькими разными способами, и выбор одного из них является делом личных предпочтений программиста, его интуиции, квалификации и опыта. В реальном проекте программист вообще не будет составлять схему алгоритма, а составит программу непосредственно на основе ТЗ. Если же документирование ПО в виде схем алгоритмов является обязательным на предприятии, выполняющем разработку, то можно с уверенностью утверждать, что эта схема будет составлена по уже готовой программе, а не наоборот. Внесение в программу даже небольших изменений в данном случае сопряжено с трудностями и высокой вероятностью внесения ошибок в ПО. Отладка ПО в данном случае также сопряжена с значительными усилиями и затратами времени, требует от программиста опыта и интуиции.

Программа, построенная на основе SWITCH-технологии, имеет больший объем, но меньшую структурную сложность. Ее код взаимно-однозначно соответствует графу переходов. Особенностью данного подхода является то, что очень сложно понять логику функционирования программы без графа переходов. Как ни парадоксально, это является достоинством метода. Данное свойство SWITCH-программирования заставляет программиста изготавливать графическую до-

кументацию до написания кода. Впрочем, логику программы можно понять очень легко, если произвести обратную операцию – восстановить граф переходов по SWITCH-программе. В силу их изоморфизма эта операция весьма несложна, имеет абсолютно формальный характер и совершенно не зависит от личностных свойств программиста. Если сравнивать затраты времени в том и другом случае, то выигрыш от применения SWITCH-технологии измеряется в разах, даже с учетом обязательных затрат на построение графа переходов.

Сам процесс кодирования – перевод графа переходов в текст на ЯВУ – является абсолютно формализованным. Все, что требуется, – это создать шаблон программы с функциями ввода-вывода и таймеров, и тогда можно легко и быстро написать программу, в которой будет шесть состояний, 100 состояний или 1000 состояний. При этом ошибку допустить трудно, а исправить легко, поскольку легко проверить соответствие текста программы графу переходов. Более того, в силу формальности преобразования процесс кодирования можно полностью автоматизировать – разработать компилятор графов переходов, тем самым во многом исключить человеческий фактор из процесса кодирования. Примером такого компилятора может служить инструментальное средство Unimod, разрабатываемое российской компанией *eVelopers* [9].

Если граф переходов составлен правильно, то этап отладки при таком методе проектирования отсутствует как таковой. Отметим, что при традиционных методах программирования этап отладки занимает до 50 % времени разработки [2]. Однако представим, что ошибка вкралась в граф переходов и осталась незамеченной вплоть до этапа тестирования. При испытаниях системы выяснилось, что программа зависает или производит неправильные действия на одном из этапов. Поиск ошибки в данном случае предельно прост. Все, что требуется сделать – это вывести на дисплей (собственный дисплей системы управления или дисплей ПК, подключенного к системе по линии связи и т.п.) номер состояния и текущее состояние входов. Таким образом, основная задача отладки – локализация места ошибки – решается мгновенно. При традиционном, интуитивном подходе к программированию локализация места возникновения ошибки зачастую представляет собой головоломку, требующую значительных затрат времени и высокой квалификации программиста.

После того как ошибка локализована, обычно достаточно одного взгляда на граф переходов, чтобы понять причину некорректного поведения программы.

На этом преимущества SWITCH-технологии не заканчиваются. Представим себе, что система управления должна управлять не одной установкой, изображенной на рис. 1, а, допустим, десятью. При этом схема алгоритма усложнится настолько, что привести ее здесь представляется затруднительным. Даже если попытаться представить программу для десяти таких установок в виде одного КА, то он будет иметь в общем случае 5^{10}=9765625 различных состояний, если КА для одной установки содержал пять состояний. Вообще при объединении N конечных автоматов, каждый из которых содержит M_i состояний, результирующий КА будет иметь, в общем случае, количество состояний, описываемое формулой (1):

$$M = \prod_{i=1}^{N} M_i. \qquad (1)$$

Этот эффект носит название «комбинационного взрыва». Он являлся одной из причин, по которой модель КА не получила широкого распространения в практике программирования в 60-е годы, несмотря на то что теория КА была в то время достаточно разработана.

Конечно, применение объектно-ориентированного программирования и многозадачной операционной системы реального времени сократит задачу до разумных размеров, если представить алгоритм управления технологической установкой как самостоятельный объект и создать необходимое количество экземпляров таких объектов. На практике так обычно и поступают, однако все проблемы, связанные с разработкой и отладкой, при этом все равно возрастают.

В случае же представления программы в виде параллельных потоков просто копируется код исходного КА требуемое число раз, не забыв при этом назначить каждому потоку свою переменную состояния, таймер и соответствующие порты ввода-вывода.

При этом результирующий граф переходов будет иметь всего 6*10=60 состояний, а сложность разработки и отладки практически не увеличится. Более того, впоследствии можно будет добавить в программу графы переходов для управления какими-либо другими технологическими объектами без риска повредить ранее написанный код.

В заключение можно отметить, что в данной статье рассмотрены основы применения SWITCH-технологии для реализации систем ЛУ, показаны существенные преимущества данной технологии, которые заключаются в значительном сокращении сроков разработки, значительном сокращении (или вовсе исключении) этапа отладки, достижении полного соответствия программы программной документации и техническому заданию, повышении верифицируемости и модифицируемости ПО. Главное же преимущество SWITCH-технологии состоит в том, что при ее использовании выполняется переход от интуитивных методов разработки к формальным, от вероятностных методов оценки надежности ПО к доказательству надежности ПО, основанному на структуре его графов переходов.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за ценные замечания и редактирование статьи.

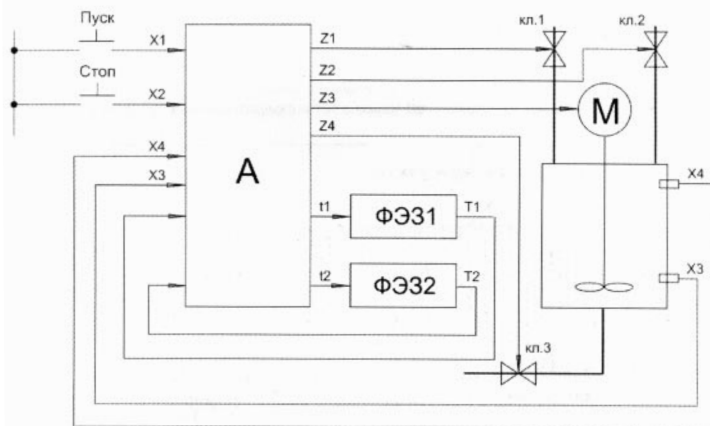

Рисунок 1

Схема автоматизации. ФЭЗ – функциональный элемент задержки, М – мотор, кл. – клапан

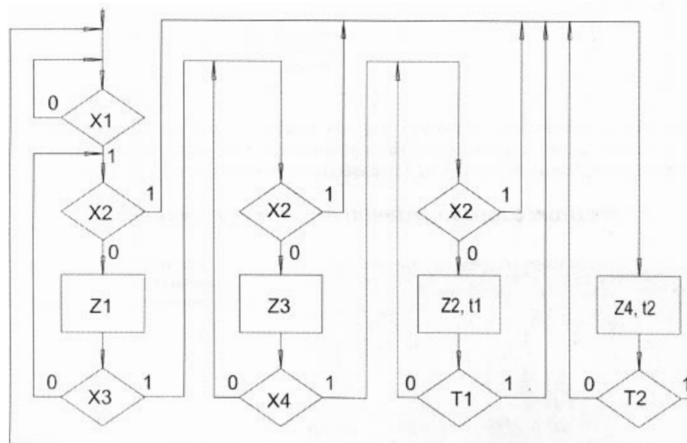

Рисунок 2

Схема алгоритма работы системы управления

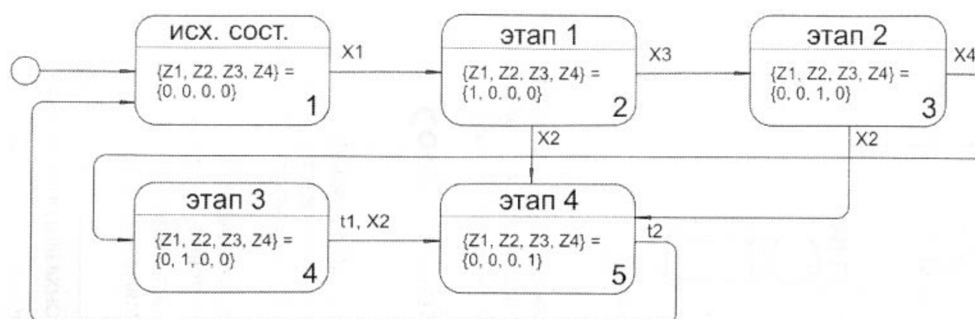

Рисунок 3

Граф переходов системы управления

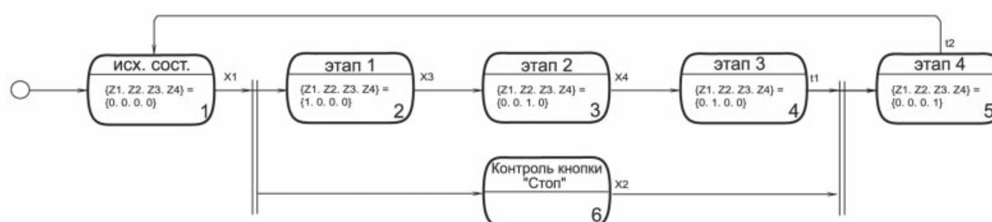

Рисунок 4

Граф переходов системы управления с параллельными потоками

Литература

1. Константин Л. Человеческий фактор в программировании. – СПб.: Символ-Плюс, 2004. – 384 с.

2. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.: Символ-Плюс, 2005. – 304 с.

3. Боггс У., Боггс М. UML и Rational Rose. – М.: Лори, 2001. – 608 с.

4. Бек К. Экстремальное программирование. Библиотека программиста. – СПб.: Питер, 2002. – 224 с.

5. Шалыто А.А. Новая инициатива в программировании. Движение за открытую проектную документацию //Мир компьютерной автоматизации. – 2003, № 5. – С. 67–71.

6. Шалыто А.А. Еще раз об открытой проектной документации //PC Week/RE. – 2005, № 11. – С. 33–34.

7. Фаулер М. UML. – СПб.: Символ-Плюс, 2004. – 192 с.

8. Буч Г., Рамбо Д., Джекобсон А. Язык UML. – Руководство пользователя. М.: ДМК Пресс. СПб.: Питер, 2004. – 432 с.

9. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.

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

11. Шалыто А.А. Алгоритмизация и программирование задач логического управления. – СПб.: СПбГУ ИТМО, 1998. – 56 с.

12. Шалыто А.А. Использование граф-схем и графов переходов при программной реализации алгоритмов логического управления //Автоматика и телемеханика, 1996, № 6. С. 148–158, № 7. С. 144–169.

13. Корнеев Г.А., Казаков М.А., Шалыто А.А. Построение логики работы визуализаторов алгоритмов на основе автоматного подхода // Труды X Всероссийской научно-методической конференции «Телематика-2003». Т.2. – СПб.: СПбГИТМО(ТУ), 2003.

14. Казаков М.А., Шалыто А.А. Использование автоматного программирования для реализации визуализаторов. //Компьютерные инструменты в образовании. – 2004, № 2. – С. 19–33.

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