Статья опубликована в журнале «Информационно-управляющие системы». 2004. № 3, с. 35-42.

УДК 681.3.07

Синхронное программирование

Д.Г. Шопырин,
аспирант
А.А. Шалыто,
д-р техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики (СПбГУ ИТМО)

В последние годы в Западной Европе при построении ответственных систем широко применяется синхронное программирование, однако на русском языке какие-либо публикации на эту тему отсутствуют. Цель настоящей работы состоит в том, чтобы восполнить указанный пробел.

Введение

Несмотря на широкое распространение универсальных языков программирования, таких как *C++* и *Java*, и технологий программирования на их основе, в мире проводятся работы по созданию специализированных языков и технологий, предназначенных для программирования управления ответственными объектами.

С 1991 года в России [1] для указанного класса систем развивается SWITCH-технология, которая в качестве языка спецификации использует графы переходов. Эта технология была также названа «программирование с явным выделением состояний».

С этого же времени в Западной Европе технологии программирования для управления ответственными объектами развиваются под общим названием «синхронное программирование» [2]. Однако, несмотря на большое количество работ по этой тематике, они практически не известны в России, в частности, в связи с отсутствием публикаций на русском языке.

Цель настоящей работы – восполнить указанный пробел.

Отметим, что SWITCH-технология также является разновидностью синхронного программирования, особенностью которой является использование универсальных языков программирования.

1. Типы программных систем

В работе [3] выделены следующие типы программных систем:

* *преобразующие системы* – это системы, завершающие свое выполнение после преобразования входных данных (например, архиватор, компилятор). В таких системах обычно входные данные известны и доступны на момент запуска системы, а выходные данные доступны после ее завершения;
* *интерактивные системы* – это системы, взаимодействующие с окружающей средой в режиме диалога (например, текстовый редактор). Характерной особенностью таких систем является то, что они могут контролировать скорость взаимодействия с окружающей средой – заставлять окружающую среду «ждать».
* *реактивные системы* – это системы, взаимодействующие с окружающей средой посредством обмена сообщениями в темпе, задаваемом средой.

Применительно к цели настоящей работы, интерес представляют реактивные системы, которые и будут рассматриваться в дальнейшем.

Системы этого класса имеют следующие особенности:

* *время отклика* реактивной системы задается ее окружением;
* поведение реактивных систем *детерминировано;*
* для реактивных систем характерен *параллелизм*.

В силу специфики задач, решаемых реактивными системами, сбой в работе таких систем *недопустим*, так как это может повлечь за собой катастрофические последствия.

Типичным примером реактивной системы является *автопилот*, который не может заставлять ждать окружающую среду. Поведение автопилота должно быть детерминировано. Он должен функционировать параллельно с другими бортовыми системами. Сбой автопилота может привести к катастрофе.

2. Основные идеи синхронного программирования

Объясним основную идею синхронного программирования. Известно, что схемы делятся на синхронные, асинхронные и апериодические (с самосинхронизацией) [4]. При построении вычислительных машин в основном используются синхронные схемы, важнейшей составляющей которых являются синхронные автоматы с памятью. Каждый из таких автоматов содержит две составляющие (Рис. 1):

* автомат без памяти, осуществляющий функциональное преобразование;
* тактируемые элементы памяти, хранящие состояние автомата.

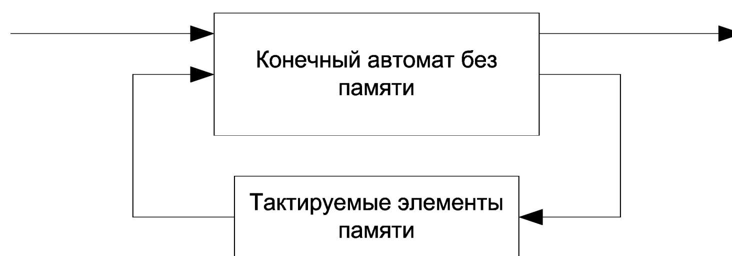


Рис. 1. Синхронный автомат с памятью

Посредством элементов памяти удается обеспечить ***синхронную*** передачу сигналов в обратной связи с выходов автомата – на его входы, устранив состязания сигналов в обратной связи. Это позволяет разделить состояния автомата в последовательные моменты времени. Тактируемые элементы памяти обеспечивают также *синхронизацию* поступающих на вход автомата входных воздействий и сигналов обратной связи.

Один из подходов к программной реализации автоматов с памятью состоит в написании по графу переходов автомата системы реализующих его булевых формул. Причем переменные в левой части каждой формулы соответствуют компоненте вектора состояний в текущий момент времени, а в правой части входные воздействия в текущий момент времени, а компоненты состояний – в предыдущий момент.

При этом задержка на такт моделируется переобозначением переменных соответствующих компонентам состояния для того, чтобы в новом цикле вычислений «предыдущие» значения стали «настоящими». Таким образом, обеспечивается параллельное (синхронное) вычисление всех компонент вектора состояния.

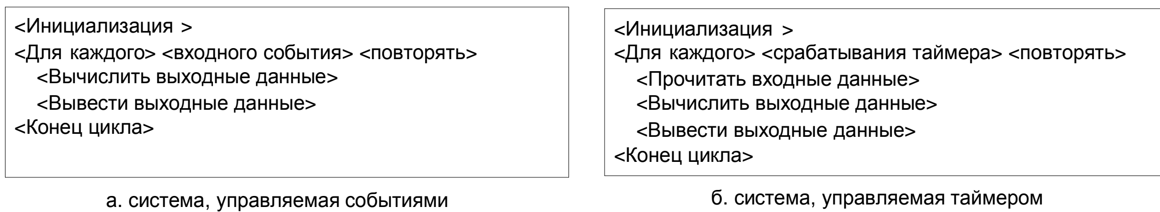
Пусть, например, необходимо синхронно перейти из состояния «00» в состояние «11». При этом для корректной работы этот переход должен осуществляться непосредственно (синхронно), а не асинхронно – через промежуточные состояния «01» или «10». Переобозначение переменных позволяет достичь этого.

В синхронном программировании вводится термин *реакция*. Реакция – это неделимый такт работы системы, в процессе которого все ее компоненты синхронно обрабатывают входные сигналы, изменяют свое состояние и формируют выходные сигналы. Считается, что длительность любой реакции равна нулю.

Таким образом, в синхронном программировании вводится абстрактное дискретное время, каждый момент которого соответствует одной реакции системы.

Существуют две основные модели реактивных систем:

- системы, управляемые событиями (рис. 2, а);
- системы, управляемые таймером (рис. 2, б).



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

а. система, управляемая событиями

```
<Инициализация >
<Для каждого> <срабатывания таймера> <повторять>
   <Прочитать входные данные>
   <Вычислить выходные данные>
   <Вывести выходные данные>
<Конец цикла>
```

б. система, управляемая таймером

Рис. 2. Модели реактивных систем

Будем рассматривать синхронную систему как совокупность компонентов, задаваемых:

- вектором входных переменных $\bar{x}_i$;
- состоянием y_i;
- вектором выходных воздействий $\bar{z}_i$.

Тогда реакция системы в целом есть комбинация реакций всех компонентов системы. При этом для каждого компонента справедливо соотношение:

$$y_i^{n+1} = f(y_i^n, \bar{x}_i^n)$$
$$\bar{z}_i^n = g(y_i^n, \bar{x}_i^n). \qquad (1)$$

Комбинация реакций отдельных компонентов обеспечивается тем, что входные или выходные значения i-го компонента могут являться также входными значениями для j-го компонента:

$$x_i^n(k) = z_j^n(l),$$

или $\qquad (2)$

$$x_i^n(k) = x_j^n(l),$$

где $x_i^n(k)$ означает k-ю координату вектора $\bar{x}_i^n$.

Отметим, что система, заданная таким образом не всегда имеет однозначное детерминированное поведение. Дело в том, что в ней могут возникнуть мгновенные обратные связи. Эта ситуация проиллюстрирована в терминах синхронных конечных автоматов на рис. 3 (мгновенная обратная связь выделена пунктиром).

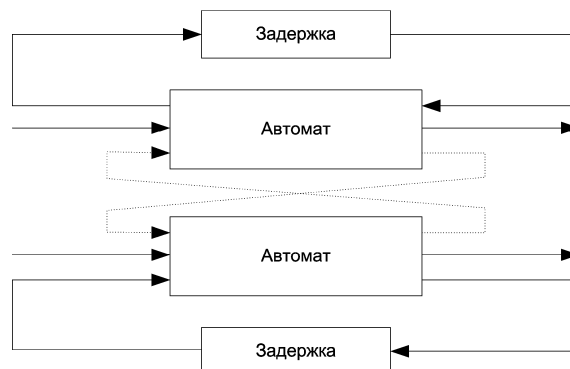


Рис. 3. Мгновенная обратная связь при соединении двух автоматов

Эта проблема может быть решена как минимум четырьмя разными способами [2]:

- каждая реакция системы разбивается на последовательность элементарных *микрошагов*. Концепция микрошагов соответствует общепринятому в программировании мнению, что выполнение программы есть последовательность атомарных действий. Однако в этом случае система перестает быть синхронной, так как невозможен синхронный (без промежуточных состояний) переход из «00» в «11». Кроме того, данный подход приводит к многочисленным противоречивым интерпретациям [5]. Подход используется в *Very High Speed Integrated Circuit Hardware Description Language (VHDL)* [6], *Statecharts* [7] и других системах.
- вводится требование, что система не может содержать мгновенных обратных связей. Например, при их наличии программа не компилируется. Данный подход удобен для программирования систем, тактируемых таймером. Язык программирования *Lustre* [8] использует данный подход.
- вводится требование, что каждой реакции системы соответствует ровно одно детерминированное выражение вида

  $$\{state_{n+1}, output_n\} \leftarrow \{state_n, input_n\}, \qquad (3)$$

  что может иметь место несмотря на наличие мгновенных обратных связей. Такой подход используется в языке *Esterel* [9].
- вводится требование, что каждой реакции может соответствовать ни одного (программа заблокирована), одно или более выражений вида (3). В последнем случае имеет место недетерминированное поведение. Данный подход используется языком *Signal* [10].

3. Синхронные языки программирования

3.1. Целесообразность применения синхронных языков программирования

В общем случае, реактивную систему можно реализовать в виде детерминированного конечного автомата Мили, представленного в виде графа переходов, дуги которого помечены входными и выходными воздействиями. Однако применение автоматов Мили в чистом виде весьма ограничено.

В качестве примера, рассмотрим так называемую задачу `ABRO` [11, 12]: сформировать сигнал `O` как только произойдут каждый из сигналов `A` и `B`. Сигнал `R` сбрасывает текущее состояние системы. Ниже приведена таблица, иллюстрирующая выполнение программы, решающей эту задачу (табл. 1).

Таблица 1

Номер реакции / Сигнал	1	2	3	4	5	6	7	8	9	10	11	12
A	x				x			x	x		x	x
B		x	x		x			x		x		x
R				x			x	x	x			
O		x			x						x	

На рис. 4 приведен автомат Мили, формализующий решение задачи *ABRO*. Состояние, отмеченное двумя концентрическими окружностями, является начальным. Формат метки перехода – `I[/O]`, где `I` – условие перехода, а `O` – сигналы, формируемые во время перехода. Квадратные скобки показывают, что на некоторых переходах выходные воздействия могут не формироваться.

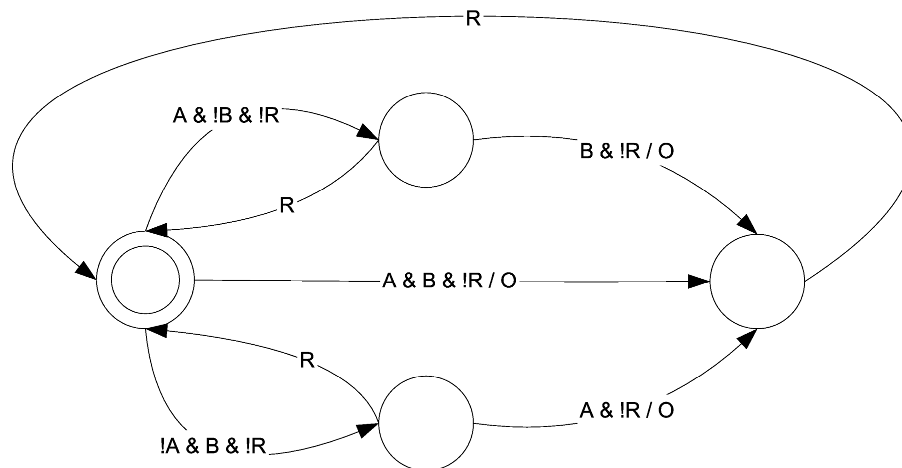


Рис. 4. Автомат Мили для решения ABRO

Теперь, вместо задачи *ABRO*, рассмотрим задачу *ABCRO*, в которой необходимо дождаться трех сигналов – `A`, `B` и `C`. Соответствующий автомат Мили будет содержать восемь состояний. Очевидно, что для решения задачи $A_1A_2..A_nRO$ потребуется автомат Мили, содержащий 2^n состояний. Другими словами, размер программы растет экспоненциально в зависимости от размерности задачи.

Теперь рассмотрим программу на языке *Esterel*, которая решает задачу *ABRO* (компилятор языка *Esterel* можно загрузить по адресу http://www-sop.inria.fr/esterel.org/) :

```
module ABRO:
input A, B, R;
output O;
 loop
  abort
   [await immediate  A || await immediate B];
   emit O
  when immediate R;
```

```
  [await immediate R || pause]
 end;
end module
```

Основные конструкции, использованные в этой программе, описаны в разд. 3.3.

Для решения задачи `ABCRO` достаточно добавить только одну инструкцию. Следовательно, при решении этой задачи размер программы, в отличие от автомата Мили, растет линейно:

```
module ABCRO:
input A, B, C, R;
output O;
 loop
  abort
   [await immediate  A || await immediate B || await immediate C];
   emit O
  when immediate R;
  [await immediate R || pause]
 end;
end module
```

3.2. Язык программирования Lustre

Язык *Lustre* – это декларативный язык программирования, оперирующий *потоками данных* [8]. Этот язык удобен для программирования систем, управляемых таймером.

Потоком данных P называется бесконечная последовательность значений одного и того же типа $p_n, n \in (0, \infty)$, где n – номер реакции системы.

Потоки могут быть следующих разновидностей:

- константы, значения которых постоянны;
- входные переменные, определяемые окружением системы;
- выходные переменные.

Каждая выходная переменная `x` задается выражением вида `x = E`, означающим, что переменная `x` всегда равна выражению `E`. Другими словами, в каждый момент времени справедливо равенство

$$x_n = E_n .$$

Обычные арифметические и логические операторы, операторы сравнения и ветвления, расширены для неявной работы с последовательностями. Кроме обычных операторов, доступны два специальных:

- `pre(x)` – определяет поток, значение которого на каждом шаге n равно значению потока `x` на шаге n-1. Если значение потока `x` на

шаге n-1 не определено, то поток `pre(x)` принимает специальное значение `nil`;

- “`x -> y`” – определяет поток, значение которого на первом шаге равно значению потока `x`, а на всех последующих шагах – значению потока `y`.

Определение переменной может быть рекурсивным – значение переменной может зависеть от ее предыдущих значений.

Для структурирования программ язык *Lustre* предоставляет *узлы* (*node*). Узел – это функция, принимающая несколько входных потоков и возвращающая несколько выходных. Он может использовать локальные потоки. Каждый выходной или локальный поток в узле должен быть определен ровно один раз. Порядок определений не важен.

В качестве примера программы на языке *Lustre*, рассмотрим определение узла, подсчитывающего количество некоторых событий:

```
node COUNT(event: bool)
returns (count: int);
let
  count = (if (event) then 1 else 0) ->
            if (event) then pre(count) + 1
                       else pre(count);
tel.
```

Узел `COUNT` принимает поток `event` типа `bool`, равный *true*, если произошло некоторое событие, и выдает поток `count` типа `int`, равный количеству событий, произошедших с момента начала работы системы.

При первом запуске значение `count` равно единице, если событие произошло, и нулю, если событие не произошло. На всех последующих шагах значение `count` увеличивается на единицу, если событие произошло, и остается неизменным, если событие не произошло.

3.3. Язык программирования Esterel

Язык *Esterel* – это императивный язык программирования, оперирующий *сигналами* [9]. Язык *Esterel* удобен для программирования систем, управляемых событиями. Отметим, что существует графический язык программирования *SyncCharts* [13], семантика которого полностью соответствует семантике языка *Esterel*.

Сигнал – это событие, жизненный цикл которого ограничен одной реакцией системы. Программа может проверять наличие входных и выходных сигналов и *формировать* (emit) выходные сигналы. Сигнал присутствует тогда и только тогда, когда он установлен системой или системным окружением во время текущей реакции. С сигналом может

быть ассоциировано значение, так, например, с сигналом `SPEED` может быть ассоциировано значение, равное текущей скорости.

Система, реализованная на языке *Esterel*, состоит из нескольких вложенных, параллельно работающих потоков выполнения (threads). Потоки взаимодействуют между собой исключительно посредством *сигналов*. Сигнал, сформированный одним из потоков, может быть получен другими потоками (только во время той же реакции).

Программа на языке *Esterel* состоит из последовательности императивных инструкций. Инструкция начинает свое выполнение в момент времени t, выполняется, и завершается в момент $t'\ ^{3} t$. Инструкция называется *мгновенной*, если $t' = t$, и *длительной*, если $t' > t$.

Примером мгновенной инструкции является инструкция `emit S`, формирующая сигнал `S`. Примером длительной инструкции является инструкция `await S`, ожидающая первой следующей реакции, в которой присутствует сигнал `S`.

Инструкции могут выполняться последовательно и параллельно. Выражение `p; q` означает, что `q` начинает выполняться в тот же момент, когда заканчивает выполняться `p`. Выражение `p || q` означает, что `p` и `q` запускаются одновременно (параллельно), разветвляя текущий поток. Выполнение выражения `p || q` завершается в тот же момент, когда завершаются и `p`, и `q`.

Особенностью языка *Esterel* являются *вытесняющие инструкции*. Вытеснение – это отказ некоторому процессу в праве выполняться навсегда (сильная форма) или временно (слабая форма) [14].

Инструкция `abort p when S` относится к сильной форме вытеснения. Процесс `p` вытесняется навсегда (прерывается) в момент выполнения условия `S`.

Инструкция `suspend p when S` относится к слабой форме вытеснения. Процесс `p` временно вытесняется (замораживается) в момент выполнения условия `S`.

Отметим следующие характерные черты вытесняющих инструкций языка *Esterel*:

- телом вытесняющей инструкции может быть любая, сколь угодно сложная, последовательность инструкций;
- вытесняющие инструкции могут вкладываться друг в друга, естественным образом задавая приоритеты.

Более подробно вытесняющие инструкции языка *Esterel* рассмотрены в [11].

Пример программы на языке *Esterel* был рассмотрен в разд. 3.1. В качестве еще одного примера использования этого языка, рассмотрим программу управления микроволновой печью.

Микроволновая печь может иметь несколько режимов приготовления пищи (приготовление картофеля, говядины и т.д.) Параметры работы печи (мощность излучения, мощность гриля и т.д.) изменяются в зависимости от выбранного режима, объема приготовляемой пищи и времени. Процесс приготовления должен быть приостановлен, если открыта дверца печи, и прерван, если нажата кнопка «стоп».

Код, управляющий параметрами работы печи, может быть весьма сложным. Однако ниже приводится сильно упрощенная версия, в которой основное внимание уделяется прерыванию процесса приготовления пищи:

```
module OVEN:
input start, stop, open;
output heat, bell;
  loop
   await start;
   emit bell;
   abort
    suspend
     repeat 5 times
       emit heat;
       pause;
     end repeat;
    when open;
   when stop;
   emit bell;
  end loop;
end module
```

Перечислим сигналы, которые принимает и формирует модуль OVEN.

1. Входные сигналы:
   - `start` – нажата кнопка «старт»;
   - `stop` – нажата кнопка «стоп»;
   - `open` – открыта дверца печи.
2. Выходные сигналы:
   - `heat` – греть;
   - `bell` – издать звонок, уведомляющий о начале/конце работы.

Приведем пример работы этой программы (табл. 2).

Таблица 2

Номер реакции / Сигнал	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Start		x						x						
Stop						x								
Open				x			x				x			
Heat		x	x		x			x	x	x		x	x	
Bell		x				x		x						x

Ниже приведен подробный комментарий для каждой реакции.

1. Ничего не происходит.
2. Нажата кнопка «старт», издан звонок, нагревание.
3. Нагревание.
4. Открыта дверца, нагревание приостановлено.
5. Нагревание.
6. Нажата кнопка «стоп», приготовление прервано, издан звонок.
7. Открыта дверца.
8. Нажата кнопка «старт», издан звонок, нагревание.
9. Нагревание.
10. Нагревание.
11. Открыта дверца, нагревание приостановлено.
12. Нагревание.
13. Нагревание.
14. Приготовление завершено, издан звонок.

3.4. Язык программирования Argos

Язык *Argos* – это графический язык программирования, оперирующий конечными автоматами [15].

Поведение системы в языке *Argos* описывается графом переходов. Программа на рис. 5 по каждому второму событию e1 генерирует событие e2.

Множество состояний на графе переходов обозначается множеством прямоугольников. Прямоугольники помечены строками, идентифицирующими состояния системы. Граф переходов имеет одно начальное состояние, помеченное дугой без начальной вершины.

Множество переходов между состояниями представлено множеством помеченных дуг. Метки переходов состоят из входного и выходного воздействий, записанных в формате `I[/O]`. Входные и выходные воздействия оперируют событиями. Входное воздействие `I` – это условие, которое должно быть выполнено для осуществления перехода. Выходное воздействие `O` – это множество событий, которые должны быть сформированы при осуществлении перехода.

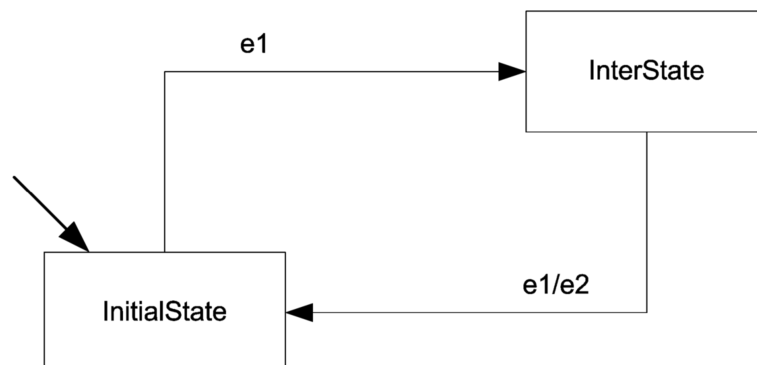


Рис. 5. Простая программа на *Argos*

Преимуществом такого представления является простота проверки детерминизма системы. Поведение системы недетерминировано, если из одного состояния существуют более одного перехода с эквивалентными входными воздействиями, но ведущие в разные состояния и/или с разными выходными воздействиями.

Для моделирования больших систем язык *Argos* предоставляет разнообразные средства структурирования программ. При этом программа разбивается на множество компонентов, состоящих из конечных автоматов.

Компоненты, разделенные пунктирной линией, выполняются параллельно. На рис. 6 показана программа, по каждому четвертому событию e1 генерирующая событие e2.

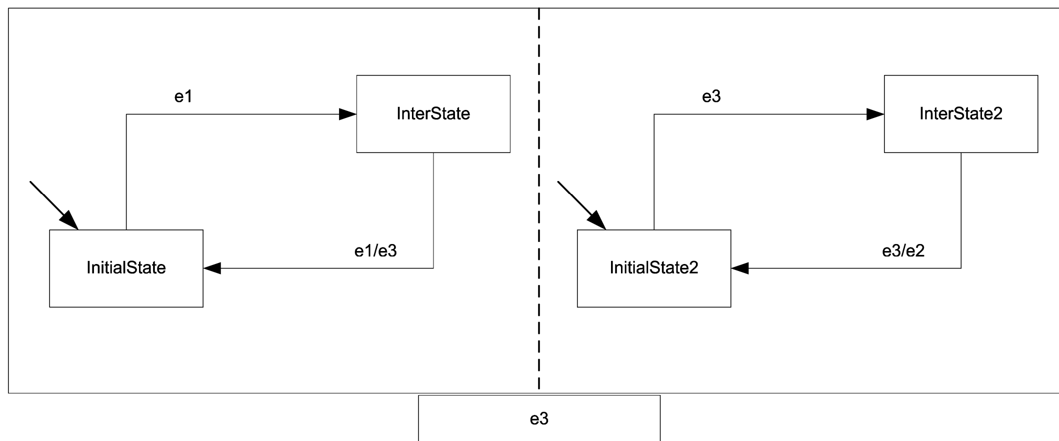


Рис. 6. Параллельное выполнение автоматов

Автоматы могут взаимодействовать между собой посредством событий. Выходные события одного автомата могут быть входными событиями для другого автомата. Язык *Argos* позволяет задать область видимости локального события. На рис. 6 область видимости события e3 ограничена прямоугольником, охватывающим оба автомата.

В состояние автомата может быть вложена подсистема. На рис. 7 приведена программа, демонстрирующая вложение подсистем. Переключение режимов осуществляется событием e0.

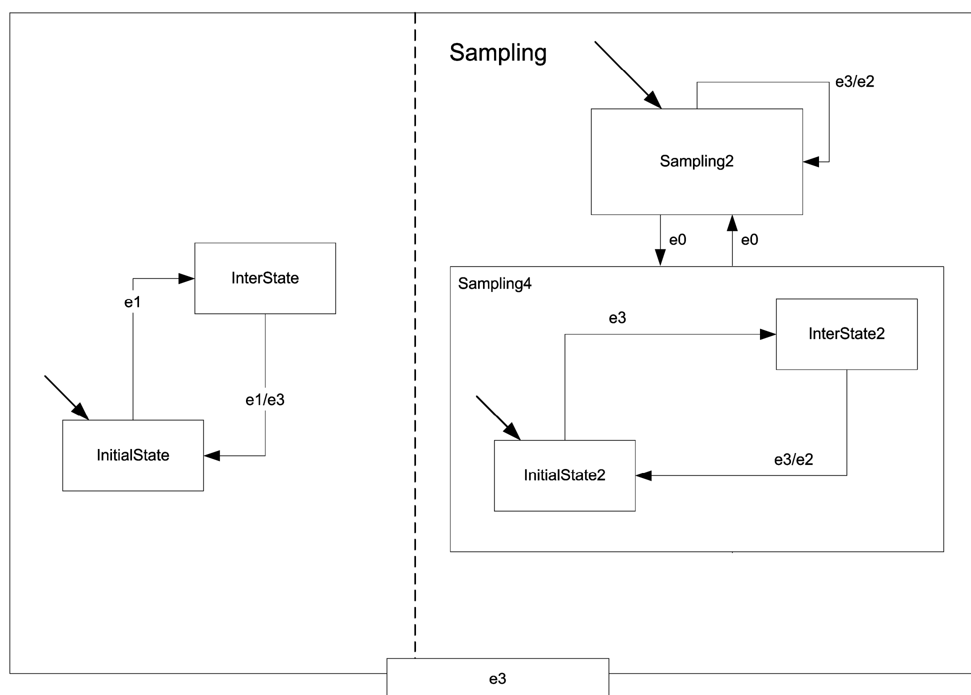


Рис. 7. Вложение подсистем

До первого события `e0` подсистема `Sampling` находится в состоянии `Sampling2`. При получении события `e0` эта подсистема переходит в состояние `Sampling4`. В этот момент создается подсистема `Sampling4` и запускается в своем начальном состоянии `InitialState2`. При повторном получении события `e0` подсистема `Sampling4` уничтожается и подсистема `Sampling` переходит в состояние `Sampling2`.

4. Синхронное программирование и тестирование

Тестирование синхронных программ обычно сводится к проверке выполнения некоторых *условий безопасности* (safety properties) [16]. Условия безопасности формулируются в зависимости от специфики решаемой задачи. Например, микроволновая печь должна быть выключена, если открыта ее дверца.

Поведение синхронной программы почти всегда зависит от окружающей среды. Программа разрабатывается исходя из некоторых предположений о свойствах этой среды. Эти предположения формулируются в виде *условий реалистичности*. Например, про микроволновую печь может быть известно, что сигналы `start` и `open` не могут возникать одновременно.

Условия безопасности и условия реалистичности могут быть выражены в виде *синхронных наблюдателей* (synchronous observers). Синхронный наблюдатель – это модуль, запускаемый параллельно с тестируемой программой. В случае нарушения контролируемого условия наблюдатель формирует соответствующий сигнал.

Наблюдатели могут быть реализованы на том же языке, что и тестируемая программа.

Для многих синхронных языков существуют системы верификации, позволяющие автоматизировать проверку условий безопасности. Например, вместе с компилятором языка *Esterel* поставляется система верификации *Xeve* [17].

Эта система анализирует программу, состоящую из тестируемой программы и ее наблюдателей. Далее, генерируется конечный автомат, каждое состояние которого соответствует состоянию анализируемой программы. Затем проверяется, достижимы ли состояния автомата, в которых нарушаются условия безопасности. Пользователь сам указывает сигнал, наличие или отсутствие которого указывает на нарушение условия безопасности.

Рассмотрим пример, тестирующий модуль `Oven` управляющий микроволновой печью, приведенный в разд. 3.3. Для этого построим программу, состоящую из трех модулей, два из которых являются синхронными наблюдателями (`NoStartWhenOpenAssumption` и `HeatingWhenOpenProperty`), а третий (`OvenTest`) – запускает тестируемый модуль `Oven` параллельно с этими наблюдателями:

```
module NoStartWhenOpenAssumption:
input start, open;
output START_OPEN_INVALID;
  loop
    present start and open then
      emit START_OPEN_INVALID;
    end;
    pause;
  end
end module

module HeatingWhenOpenProperty:
input open, heat, START_OPEN_INVALID;
output HEATING_WHEN_OPEN;
  loop
    present open and heat and not START_OPEN_INVALID then
      emit HEATING_WHEN_OPEN;
    end;
    pause;
  end
end module

module OvenTest:
input start, stop, open, ready;
output heat, bell, HEATING_WHEN_OPEN, START_OPEN_INVALID;
    run Oven
    || run NoStartWhenOpenAssumption
    || run HeatingWhenOpenProperty
end module
```

В данном примере, условие безопасности нарушается, если присутствует сигнал `HEATING_WHEN_OPEN`. Система *Xeve* сообщает, что вышеуказанный сигнал никогда не будет сформирован, и, следовательно, соответствующее условие безопасности никогда не будет нарушено.

Автоматическое тестирование позволяет упростить сертификацию программных систем.

Заключение

В настоящей работе выполнен обзор языков синхронного программирования, позволяющих повысить надежность программного обеспечения по сравнению с традиционными языками и технологиями. Объяснено, почему этот класс языков назван синхронными. Приведены примеры использования этих языков.

Наряду с синхронными языками программирования, созданными в Западной Европе, в России развивается SWITCH-технология, которая также предназначена для построения ответственных систем. С большим числом проектов, построенных с использованием этой технологии, можно ознакомится на сайте http://is.ifmo.ru, раздел «Проекты».

Эти проекты выполняются в рамках «Движения за открытую проектную документацию», что является особенно важным для ответственных систем [18].

Литература

1. **Шалыто А.А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628 с.
2. **Benveniste A. et al.** The Synchronous Languages 12 Years Later. Proceedings of the IEEE, vol. 91, no. 1, January 2003. pp. 64-83.
3. **Harel D., Pnueli A.** On the development of reactive systems. In Logic and Models of Concurrent Systems. NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985. pp. 477-498.
4. **Апериодические** автоматы /Астановский А.Г., Варшавский В.И., Мараховский В.Б. и др. М.: Наука, 1976. 423 с.
5. **Harel D., Naamad A.** The stalemate semantics of statecharts. ACM Trans. Softw. Eng. Methodology, vol. 5, Oct. 1996. pp. 293-333.
6. **IEEE** Standard VHDL Language Reference Manual. IEEE Press, Piscataway, NJ, 1994, pp. 1076-1993.
7. **Harel D.** Statecharts: A visual formalism for complex systems. Sci. Comput. Program., vol. 8, June 1987. pp. 231-274.

8. **Caspi P., Pilaud D., Halbwachs N., Plaice J. A.** LUSTRE: A declarative language for programming synchronous systems. In ACM Symp. Principles Program. Lang. (POPL), Munich, Germany, 1987, pp. 178-188.
9. **Berry G., Gonthier G.** The Esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program., vol. 19, Nov. 1992. pp. 87-152.
10. **Benveniste A., Guemic P.** Hybrid dynamical systems theory and the SIGNAL language. IEEE Trans. Automat. Contr., vol. AC-35, May 1990. pp. 535-546.
11. **Berry G.** The Esterel v5 Language Primer, July 2000. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf. 148.p
12. **Jourdan M., Lagnier F., Raymond P., Maraninchi F.** A Multiparadigm Language for Reactive Systems. In 5th IEEE International Conference on Computer Languages, Toulouse, May 1994, IEEE Computer Society Press. pp. 211-218
13. **Andre C.** Representation and Analysis of Reactive Behaviors: A Synchronous Approach. CESA'96, Lille, France, IEEE-SMC, July 1996.
14. **Berry G.** Preemption in Concurent Systems. Proceedings of FSTTCS 93. Springer Verlag, LNCS 761, 1993. pp. 72-93.
15. **Maraninchi F.** The Argos language: Graphical representation of automata and description of reactive systems. Presented at the IEEE Workshop Visual Lang., Kobe, Japan, 1991. 7 p
16. **Raymond P., Weber D., Nicollin X., Halbwachs N.** Automatic testing of reactive systems. Proc. 19th IEEE Real-Time Syst. Symp., Madrid, Spain, Dec. 1998, pp. 200-209.
17. **Bouali A.** Xeve: An Esterel verification environment. Proc. 10th Int. Conf. Comput.-Aided Verification (CAV '98), vol. 1427, LNCS, Vancouver, BC, 1998. pp. 500-504.
18. **Шалыто А.А.** Новая инициатива в программировании. Движение за открытую проектную документацию //Информационно-управляющие системы. 2003. № 4, с. 52-56.