МОДЕЛИРОВАНИЕ И АВТОМАТИЧЕСКАЯ ПРОВЕРКА ПО AsmL

О.Р. Набиуллин,
аспирант, Нижегородский филиал ГУ ВШЭ,
nabiulin@tecomgroup.ru
Э.А. Бабкин,
доктор PhD, профессор, заведующий кафедрой информационных систем и технологий, Нижегородский филиал ГУ ВШЭ,
babkin@hse.nnov.ru

Целью данной статьи является описание подхода связанного с использованием формализма Абстрактных Машин Состояний (ASM) и среды SpecExplorer, разработанной в Microsoft Research.

Введение

С усложнением программного обеспечения усложняются и подходы к разработке и анализу корректности создаваемых продуктов. Одним из новых подходов является использование моделей не только для анализа требований и архитектуры, но и для анализа непосредственно продукта. От статических моделей (UML, Booch и т.д.) происходит переход к исполняемым (executable) моделям. Целью данной статьи является описание подхода связанного с использованием формализма Абстрактных Машин Состояний (ASM) и среды SpecExplorer, разработанной в Microsoft Research [1].

Исполняемые модели не являются совершено новым явлением, к таким моделям можно отнести сети Петри [2], модели на языке Promela [3], и многие другие. Абстрактных Машин Состояний (в дальнейшем просто ASM) это математический формализм, разработанный [4]. Одной из имплементаций этого формализма в виде языка программирования является AsmL [5; 6]. По синтаксису этот язык является некоторой смесью Python и Visual Basic. Это объектно-ориентированный, динамический слабо типизированный язык. Отдельно стоит отметить, что AsmL являясь .Net-языком имеет доступ к любому совместимому с .Net коду в т.ч. ко всем стандартным сервисам .Net Framework.

В первой части данной статьи описывается формализм абстрактных машин состояний и его реализация AsmL. Во второй части описывается методология предлагаемая средой SpecExplorer. Третья часть содержит пример AsmL-модели и её детальное описание. В четвертой части авторы приводят заключение и обзор альтернативных инструментов.

Исполняемые спецификации

Формализм абстрактных машин состояний, описанный в [4], является чисто математическим объектом. Однако структура ASM такова, что возможна его реализация в виде языка программирования. Существует несколько альтернативных реализаций, например [7] или [8]. Основной идеей ASM является хранение состояния объекта как некоторого набора значений ассоциированных с локациями (location)[1]. Также ASM определяет механизм частичных обновлений, как основной способ эволюции системы, и математическую нотацию правил, по которым изменяются значения. Одним из ключевых моментов является одновременность выполнения действий.

Среда SpecExplorer, разработанная в Microsoft Research [1], содержит реализацию ASM в виде

1 На данный момент не существует, канонического перевода терминов связанных с ASM. В спорных случаях английский вариант термина приводится в скобках.

языка AsmL [.Net]. С точки зрения синтаксиса AsmL представляет собой некий синтез языков Python, Delphi и Visual Basic, и при этом содержит высокоуровневые конструкции, близкие к математическим (определение множеств; квантификация выражений, pattern matching). С точки зрения парадигмы программирования AsmL является объектно-ориентированным языком, в тоже время допуская как функциональное, так и императивное программирование.

AsmL входит в семейство .Net-языков и имеет доступ к любому .Net-совместимому коду. Тем не менее предполагается, что модели написанные на AsmL минимально пользуются средствами не входящими в язык и его стандартную библиотеку, т.к. использование внешних API сокращает возможности по анализу модели. Более подробно анализ AsmL моделей будет рассмотрен ниже. Помимо AsmL, SpecExplorer предлагает методологию автоматической проверки соответствия модели проектируемой системе.

AsmL [6] это язык для создания спецификаций для программного обеспечения, основанный на формализме абстрактных машин состояний. Он используется для создания понятных (human-readable), исполняемых моделей работы системы таким образом, чтобы обеспечить минимальность и завершенность, с учетом заданного уровня абстракции. Спецификации на AsmL называются *исполняемыми спецификациями*.

Так же, как и традиционные спецификации, исполняемые спецификации являются описанием того, как работают компоненты программного обеспечения. В отличие от традиционных спецификаций исполняемые спецификации имеют единственное, недвусмысленное значение. Это значение проявляет себя в виде абстрактной машины состояний (ASM), математической модели изменения системы, состояния времени выполнения (runtime state).

Спецификации на AsmL могут быть запущены как программы, например, чтобы симулировать поведение некоторой системы или проверить поведение реализации на соответствие спецификации. Однако предполагается, что в отличие от обычных программ, исполняемые спецификации должны быть минимальными. Другими словами, несмотря на то, что они верны в описании, без упущений, всего, что является частью на выбранном уровне детализации, они эквивалентно верны в том оставлении неспецифицированным того, что осталось за рамками этого уровня детализации.

Таким образом, в отличие от программ, исполняемые спецификации ограничивают себя теми ограничениями (constraints) и поведением, какие должны быть присущи всем корректным реализациям системы. Другими словами, исполняемые спецификации должны быть так же ясны относительно свободы данной конкретным реализациям системы, которая описывается, как и относительно ограничений.

В качестве примера, исполняемые спецификации не ограничивают порядок выполнения операций до тех пор, пока он не является важным, в то время как сегодняшний программы реализуют последовательный порядок выполнения операций, как решение уровня реализации (в отличие от уровня проектирования).

Это можно проследить на примере:

```
var A = [3, 10, 5, 7, 1]
indices = {0, 1, 2, 3, 4}

Main()
 step until fixpoint
  choose i in indices, j in indices
     where i < j and A(i) > A(j)
   A(i) := A(j)
   A(j) := A(i)
 step
  WriteLine(A)    // prints [1, 3, 5, 7, 10]
```

Листинг 1 Сортировка

Эта исполняемая спецификация использует абстрактную машину состояний для сортировки через алгоритм обмена.

Машина выполняет последовательные шаги, которые меняют местами значения *A*, элементы которого обозначаются индексами *i* и *j*, такими что *i* меньше *j*, и значения *A*(*i*) и *A*(*j*) не совпадают с порядком сортировки. Это продолжается до тех пор, пока никаких дополнительных обновлений не возникает, то есть, до тех пор, пока последовательность не станет отсортированной. Как последний шаг, печатается отсортированная последовательность. Состояние машины на каждом этапе полностью характеризуется значением последовательности *A* на этом шаге.

Эта спецификация минимальна. Первый момент заключается в том, что выражение choose не говорит о том, как два индекса выбираются, только то, что выбранные значения должны быть различными индексами элементов, нарушающих порядок сортировки. Таким образом, многие алгоритмы сортировки, в том числе и quicksort, и пузырьковая сортировка, будут соответствовать тому, что мы указали.

Кроме того, наш пример не говорит о том, как происходит операция обмена значениями. Значения переменных изменяются, как атомарная транзакция. Это оставляет реализации решение о том, как осуществлять последовательный обмен, например, с использованием копирования во временный элемент.

Методология

Подход, предлагаемый средой SpecExplorer (Microsoft), состоит из следующих шагов:

1. Создание высокоуровневой модели системы с использованием языка AsmL (также возможно использование Spec#, но его рассмотрение выходит за рамки данной работы).

2. Исследование поведения модели путем прогонки по фиксированным (пользовательским) и произвольным (автоматическим) сценариям.

3. Автоматическая генерация сценариев с целью их дальнейшего воспроизведения.

4. Автоматическая проверка продукта на соответствие модели.

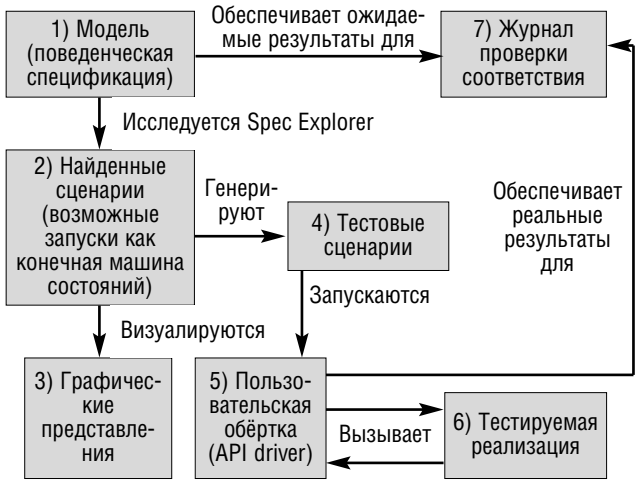

Рис. 1 Методология SpecExplorer

Сквозной пример

В качестве сквозного примера в данной статье будет рассматриваться следующая задача: есть несколько компонентов (программ), которые обладают определенными зависимостями. Требуется запустить их некоторое подмножество из них с учетом зависимостей. Предполагается, что старт компонента может завершиться неудачей. В этом случае необходимо пропустить все компоненты, которые явно или косвенно зависят от компонента, вызвавшего сбой. Исходная задача – сделать подобие windows services, но с учетом специфики предметной области.

Рассмотрим пример. Есть три компонента *A*, *B*, *C*. Исходные зависимости:

A: []
B: [*A*]
C: [*A*, *B*]

Предположим, что, пользователь пытается запустить компонент *C*. Система должна запустить последовательно компоненты *A, B* и *C*. В случае если все компоненты запущены, и пользователь пытается остановить компонент *B*, система должна сначала остановить компонент *C*, так как в его список зависимостей входит *B*, и только потом *B*. В реальной задаче компонентом являются как отдельные программы, так и совокупности программ, но с точки зрения предлагаемой модели данное различие несущественно. Выбранный уровень абстракции предполагает работу программы как взаимодействие с некоторым числом абстрактных компонент. В дальнейшем модель может быть уточнена с учетом этих различий. Механизм итеративного уточнения моделей описан в (Borger E., 2003).

Целью моделирования на первоначальном этапе является фиксация требований к системе в виде формальной спецификации и проверка результата на логическую непротиворечивость. Следующим этапом будет являться автоматическая проверка соответствия модели и реальной программы, что выходит за рамки данной статьи.

Модель

Рассмотрим модель проектируемой системы:

Класс Comp является абстракцией компонента системы. Его характеристиками являются: имя компонента, список зависимостей (имен), а также текущий статус (запущен/нет).

```
class Comp
 var name as String
 const deps as Seq of String
 var running as Boolean = false
 Start()
  require not running
  require forall d in deps holds FindComp(d).running
  choose x in {true, false}
   if x then
    running := x
   else
    throw new Exception(name)
  WriteLine(«Started « + name + « successfully»)
 Stop()
  require running
```

```
    require forall d in FindDependants(name) holds not
FindComp(d).running
    running := false
   Check() as Boolean
    return running
   Depends(name as String) as Boolean
    return name in deps
```

Листинг 2. Класс Comp

Язык AsmL поддерживает программирование по контракту, что выражается с помощью конструкций require. В случае если произойдет вызов метода Comp.Stop в тот момент, когда компонент еще не запущен, система сгенерирует ошибку. Такая ошибка означает, что модель внутренне не согласована, и, либо ограничение установлено неверно, либо существует такая последовательность действий, которая приводит к «незаконному» вызову Stop. Каждый компонент однозначно определяется своим именем, причем предполагается, что двух компонентов, обладающих одним именем нет. Состояние компонента определяется булевой переменной running, означающей запущен компонент в данный момент или нет. Допущение данной модели состоит в том что, предполагается если сбой возникает, то он возникает в момент запуска компонента. Удачно стартовавший компонент в дальнейшем работает стабильно. Как видно из кода функции Comp.Start при запуске компонента возможен сбой. В этом случае генерируется исключение. Предусловиями для запуска компонента являются утверждения:

1. Он не запущен.

2. Все компоненты, от которых он зависит, работают.

Известные компоненты описываются следующей последовательностью:

```
var components as Seq of Comp = [
 new Comp(«core», []),
 new Comp(«mv», [«core»]),
 new Comp(«logo», [«core», «mv»]),
 new Comp(«automation», [«core»]),
 new Comp(«nxos», [«core»])]
```

Листинг 3. Список компонентов

Теперь определим высокоуровневые действия, применимые к системе:

```
[Action]
StartA(name as String)
 let c = FindComp(name)
 step
  forall n in c.deps + [name]
   SetWillStart(n, true)
 step
  Start(name)
```

Листинг 4. Запуск компонента

Запуск компонента – это двухфазная операция. На первой фазе мы помечаем компоненты, предназначенные к запуску, на втором непосредственно стартуем всю цепочку.

```
Start(name as String)
 let c = FindComp(name)
 if not c.running then
  step
   WriteLine(«Starting « + c.name)
  step
   try
    step
     StartSeq(c.deps)
    step
     if GetWillStart(name) then
      c.Start()
     else
      WriteLine(«Skipped « + name + « (failed dependency)»)
   catch
    e as Exception:
     Failed(name)

StartSeq(l as Seq of String)
 require AllowedSequence(l)
 var i = 0
 step while i < Size(l)
  Start(l(i))
  i += 1
```

Листинг 5. Запуск компонента (продолжение)

Как видно из метода Start(name as String) на высоком уровне состояние компонента проверяется во время исполнения (runtime). Вкупе с требованием чтобы Comp.Start вызывался для не запущенных компонентов, обеспечивается проверка утверждения что ни один компонент не будет запущен дважды. В противном случае в процессе автоматического исследования (exploration) модели будет сгенерировано исключение. Причем исключение сгенерированное нарушением контракта всегда останавливает выполнение модели, в отличие от пользовательских исключений, таких как в методе Comp.Start.

Вспомогательная функция AllowedSequence проверяет упорядоченную последовательность компонентов на предмет непротиворечивости, т.е. отсутствие компонентов, которые могли бы быть запущены прежде своих зависимостей.

```
function AllowedSequence(l as Seq of String) as Boolean
 return forall i in Indices(l), j in Indices(l)
  where i < j
  holds not (l(j) in GetDependencies(l(i)))
```

Листинг 6 Функция AllowedSequence

Как можно видеть из кода этой функции язык AsmL допускает формулирование условий практически в математической форме (для любых *i* и *j* из множества Indicies(l), таких что $i < j$, выполняется условие).

Чтобы завершить тему запуска компонентов приведем два оставшихся высокоуровневых метода, которые являются отражением исходных требований к системе. Метод StartAll предназначен для запуска всех известных компонентов. Метод StartDefault предназначен для запуска некоторого подмножества компонентов, которое в реальных условиях будет задаваться конфигурационным файлом.

```
[Action]
StartAll()
 require forall c in components holds not c.running
 step
  forall c in components
   SetWillStart(c.name, true)
 step
  var i = 0
  step while i < Size(components)
   Start(components(i).name)
   i += 1
[Action]
StartDefault()
 require forall c in components holds not c.running
 step
  forall ci in Indices(components) where start_ini(ci)
   SetWillStart(components(ci).name, true)
 step
  StartSeq([components(c).name | c in IndexRange(compo-
nents) where start_ini(c)])
```

Листинг 7. Групповой запуск компонентов

Помимо запуска компонентов система должна предоставлять возможность остановки уже запущенных компонентов причем, в случае если останавливается компонент от которого зависят другие запущенные, остановить нужно всю цепочку.

```
StopSeq(l as Seq of String)
 var i = 0
 step while i < Size(l)
  let c = FindComp(l(i))
  if c.running then
   Stop(l(i))
  i += 1
[Action]
Stop(name as String)
 let c = FindComp(name)
 require c.running
 step
  StopSeq(FindDependants(name))
 step
  c.Stop()
```

Листинг 8. Остановка компонентов

В данной спецификации использовались вспомогательные функции, приведенные в Листинг 9.

```
var start_ini as Seq of Boolean = [
 true,  // core
 true,  // multiviewer
 false, // logo_inserter
 false, // automation
 true]  // nxos

var will_start as Seq of Boolean = [
 false,
 false,
 false,
 false,
 false]

function FindComp(name as String) as Comp
 return the c | c in components where c.name = name

function IsRunning(name as String) as Boolean
 return FindComp(name).running

function GetDependencies(name as String) as Seq of String
 return FindComp(name).deps

function FindDependants(name as String) as Seq of String
 return [c.name | c in components
  where exists dep in c.deps
   where dep = name]

SetWillStart(name as String, val as Boolean)
 let index = the ci | ci in Indices(components) where compo-
nents(ci).name = name
 will_start(index) := val

function GetWillStart(name as String) as Boolean
 let index = the ci | ci in Indices(components) where compo-
nents(ci).name = name
 return will_start(index)
```

```
PrintComponents(l as Seq of Comp)
 Write(«Components: «)
 Write([c.name | c in l])
 WriteLine(«.»)
```

Листинг 9. Вспомогательные функции. Исследование модели

При исследовании модели Spec Explorer был сконфигурирован следующим образом:

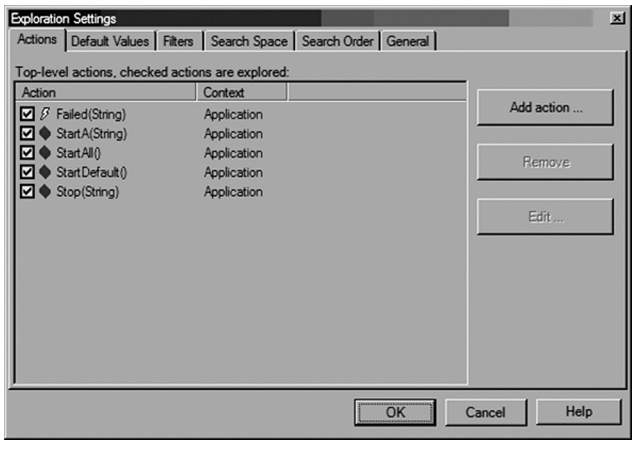

Рис. 2 Настройка действий

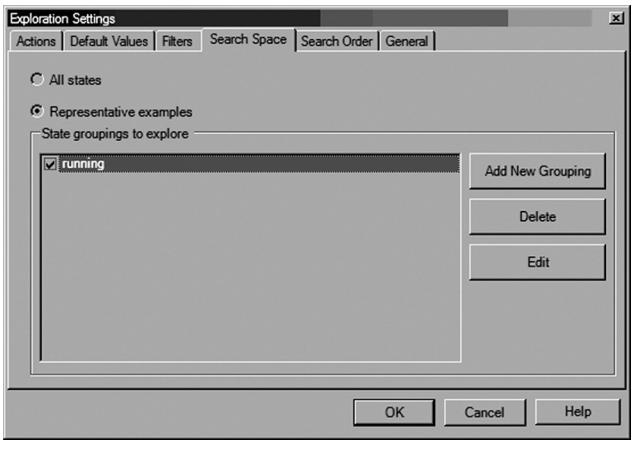

Рис. 3 Настройка области исследования

Группировка running определяется следующим AsmL выражением:

```
[c.name | c in components where c.running]
```

Листинг 10 Группировка состояний

Автоматическое исследование модели привело к следующему графу состояний (*рис. 4*):

Количество компонентов в данном графе сокращено до трех с целью повышения наглядности,

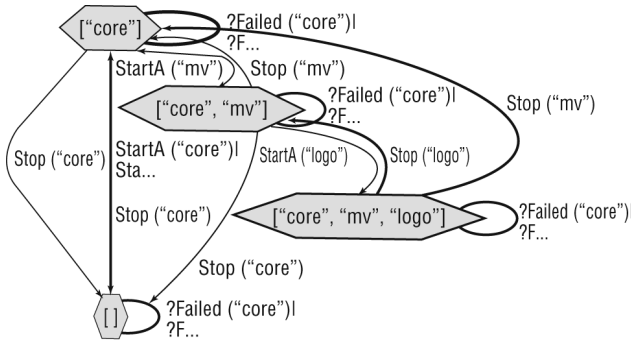

Рис. 4 Граф состояний

в реальной системе участвует большее количество компонентов и результирующий граф слишком велик, чтобы быть приведенным в иллюстрации. Несмотря на небольшое число компонентов граф, представленный на рисунке «Граф состояний», является срезом полного графа. Полный граф состояний, сгенерированный SpecExplorer приводится на *рис. 5* «Полный граф состоянийЦ

Заключение

Приведенные выше выкладки демонстрируют, как формализм ASM может быть применен для фиксации требований и проверки их на непротиворечивость. Построена модель программной системы и продемонстрированы способы автоматического исследования (exploration) этой модели. Выделены некоторые ключевые моменты подхода ASM, такие как одновременность вычисления выражений, последовательные обновления, кодирование состояния с помощью значений переменных. Также продемонстрированы некоторые возможности среды SpecExplorer, разработанной в Microsoft Research. Следующим шагом будет являться построение обертки (API Driver) для реальной системы с целью автоматической проверки соответствия модели системе.

Язык AsmL существует не только в виде части SpecExplorer, но также доступен в виде open-source компилятора, расположенного на ресурсе CodePlex [9].

Существуют также альтернативные реализации ASM такие как CoreASM [7] и его предшественник ASM-SL (реализованный в инструменте AsmWorkbench [8]), а также AsmGofer [10], реализующий семантику ASM как подмножество языка Haskell.

Исследования проводились при поддержке фонда РФФИ, грант 07-07-00058-а. ■

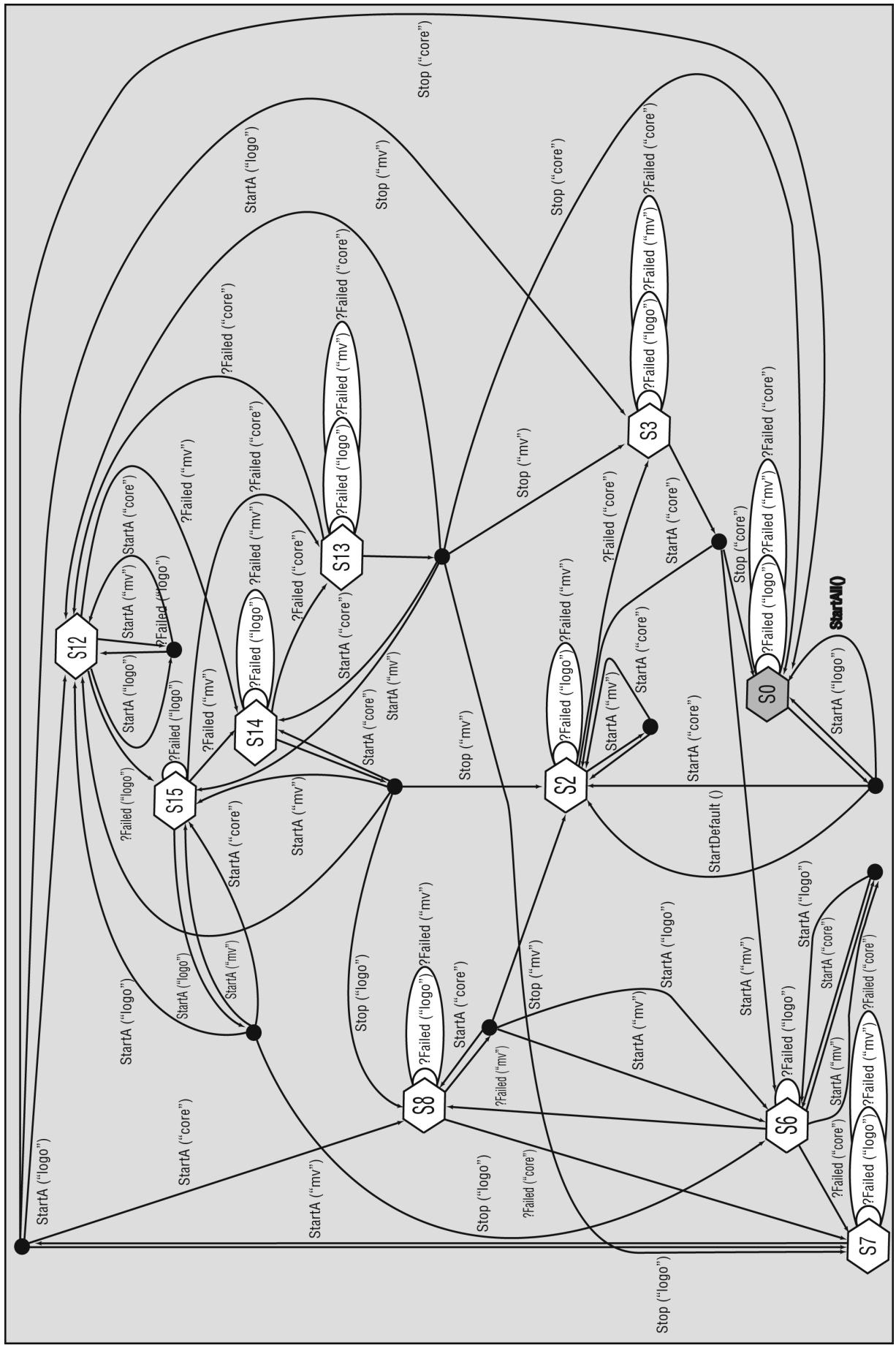

Рис. 5. Полный граф состояний

Литература

1. Foundations of Software Engineering. Online: http://research.microsoft.com/fse/. // 2008.
2. Cheung K, Cheung T, Chow K, A petri-net-based synthesis methodology for use-case-driven system design//Journal of Systems and Software, 2006, 79 Issue 6, C. [772–790].
3. Del Mar Gallardo M, Merino P, Pimentel E, A generalized semantics of PROMELA for abstract model checking//Formal Aspects of Computing, 2004, 16 Issue 3, C. [166–193]
4. Börger E, Stark R, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer-Verlag, 2003, 448 p.
5. Gurevich Y, Rossman B, Schulte W. Semantic Essence of AsmL: Extended Abstract. In Springer Lecture Notes in Computer Science. Vol. 3188. 2004.
6. The AsmL webpage. Online: http://research.microsoft.com/foundations/AsmL/. // 2008.
7. Farahbod R, Gervasi V, Glasser U, Memon M, Design and Specification of the CoreASM Execution Engine // 2006, SFU-CMPT-TR-2006-09, [58]
8. Del Castillo G. The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. Heinz Nixdorf Institut, Universit?t Paderborn. 2001.
9. AsmL Compiler. Online: http://www.codeplex.com/AsmL/. // 2008.
10. AsmGofer project page. Online: http://www.tydo.de/Doktorarbeit/AsmGofer/index.html. // 2008.