Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А. А. Чебатуркин, М. А. Мазин

Методы верификации конечных автоматов, взаимодействующих по акторной модели

Санкт-Петербург
2010

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время растет актуальность разработки параллельных программ. Это связано с тем, что дальнейший рост производительности вычислительных систем больше не может достигаться увеличением тактовой частоты процессоров, и главной тенденцией повышения производительности становится увеличение числа ядер в системах. Однако для того, чтобы программа выполнялась эффективно на нескольких ядрах, она должна быть написана с применением техник параллельного программирования.

Универсальные императивные языки программирования, наиболее распространенные на сегодняшний день, хорошо подходят для реализации последовательных алгоритмов, но плохо приспособлены для написания алгоритмов параллельных. Это связано с тем, что программа, написанная на императивном языке, представляется собой последовательность инструкций процессору, изменяющих состояние памяти. Ввиду того, что порядок этих команд фиксирован, выполнение программы не может быть распределено между несколькими процессорами. Поэтому программисты вынуждены самостоятельно разбиваться свою программу на параллельно выполняющиеся потоки.

Автоматическому распараллеливанию хорошо поддаются программы, написанные на функциональных языках программирования. Однако функциональные языки, в отличие от императивных, значительно менее распространены. Тем не менее, некоторые функциональные конструкции в последнее время стали активно проникать в универсальные императивные языки программирования. Например, становится популярным использование в императивных языках замыканий для обработки списков, в некоторых случаях это позволяет выполнять автоматическое распараллеливание по данным.

В связи с этим естественно продолжить адаптацию абстракций параллельных функциональных программ для императивных языков

программирования. К таким абстракциям, в частности, относится акторная модель, реализованная в языке программирования *Erlang*.

Эта модель была предложена К. Хьюиттом, П. Бишопом и Р. Штайгером в 1973 г. Они ввели понятие актора – примитива параллельных вычислений, обладающего собственным потоком выполнения и способного обмениваться сообщениями с другими акторами. Актор обрабатывает сообщения из собственной очереди сообщений и в ответ на них выполняет полезные действия.

Написание любых программных продуктов обычно сопровождается различными подходами по определению корректности приложения. Наиболее распространенным методом проверки корректности приложения является тестирование. Однако, несмотря на относительную простоту написания тестовых сценариев и проверки дефектов программы на этих сценариев, разработчик никак не сможет доказательно удостовериться в корректности выполнения программы с помощью этого подхода.

Другими подходами являются методы спецификации контрактов и их проверка в статическом либо динамическом режиме, а также верификация программной модели.

Верификация программной модели позволяет доказательно проверить корректное поведение на всех возможных входных воздействиях. Однако, так как верификация на модели требует построения модели определенной структуры (модель должна быть изоморфна модели Крипке), то в общем случае методы верификации неприменимы.

Поэтому представляет интерес исследование определенных классов программных моделей и разработка специальных методов верификации для них. Например, проводятся исследования методов верификации различных программных систем, состоящих из конечных автоматов, так как показано [1], что автоматные модели наилучшим образом подходят для верификации.

Для программных систем, построенных на основе акторной модели многопоточности, можно создать гораздо более мощные и выразительные

методы верификации. Ввиду того, что при таком подходе нет общего изменяемого состояния, и все взаимодействия подсистем осуществляется при помощи асинхронной передачи сообщений, то можно успешно создавать модели взаимодействия акторов, очень похожие на программные модели для автоматов. Например, можно создать программную модель для системы из взаимодействующих акторов, каждый из которых представляет собой конечный автомат, и эту модель можно будет верифицировать с помощью распространенных средств верификации на модели – *NuSMV, SPIN, Bogor* и т. д.

В настоящей работе сделана попытка создать метод верификации многопоточных систем, состоящих из конечных автоматов, взаимодействующих на основе акторной модели. В частности, получены следующие результаты:

- предложен подход к автоматическому построению моделей по исходному коду программной системы;
- предложен способ интеграции в процесс разработки программного обеспечения действий по обеспечению соответствия реализованной системы, построенной на основе акторной модели, спецификации.

Также в работе показано описана интеграция данного метода верификации в процесс разработки при использовании мультиязыковой среды *MPS*.

Работа имеет следующую структуру: в первой главе рассматриваются существующие методы проверки корректности программных систем, их достоинства и недостатки. Во второй главе описаны ключевые принципы акторной модели многопоточности и приведены существующие реализации этого подхода. В третьей главе описывается метод адаптации моделей систем из акторов, представляющих собой конечные автоматы, для успешного проведения верификации существующими верификаторами. Также в третьей главе дан краткий обзор инструмента, позволяющего во время разработки

автоматически вести построение модели по существующей программе и проводить верификацию во время разработки.

Глава 1. Обзор существующих методов контроля качества автоматных программ. Введение в акторную модель многопоточности

К автоматным программам могут быть успешно применены следующие методы анализа корректности:

1. Тестирование.
2. Верификация:
   - Доказательная верификация;
   - Проверка на модели (model checking).

1.1. Тестирование

Тестирование – это процесс выявления ошибок в программном обеспечении. Запуск программы на различных входных данных, а также проверка различных сценариев выполнения позволяют достаточно быстро (по сравнению с другими методами поиска ошибок) убедиться в корректности обработки данных сценариев. При этом строить модель самой программы не требуется.

Тестирование, применяемое после окончательного написания программы, не способно найти все ошибки. Как заметил Э. Дейкстра, тестирование помогает найти ошибки, но совершенно не гарантирует их отсутствия. Различают два основных вида тестирования:

- Тестирование белого ящика (*white-box testing).*
- Тестирование черного ящика (*black-box testing).*

При тестирования белого ящика, разработчик теста имеет доступ к исходному коду программы и может писать тестовые сценарии, которые связаны с библиотеками тестируемого ПО.

В случае с тестированием черного ящика, специалист по тестированию имеет доступ к ПО только через строго определенный программный интерфейс (*Application Programming Interface* – API). Как правило, тестирование ведется с использованием спецификаций или иных документов, описывающих требования к системе.

В последнее время большую популярность получило модульное тестирование (*unit-testing*) – процесс тестирования, ориентированный на проверку отдельных модулей исходного кода программы. Такое тестирование обеспечивает до определенной степени работоспособность и устойчивость компонент.

Важно отметить, что исправление одной ошибки может привести к появлению другой. В рамках данного подхода приходится писать тесты для каждого нетривиального метода или функции. Это позволяет достаточно быстро проверить, не привело ли очередное изменение к *регрессии* – к появлению ошибок в уже написанных и оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

1.2. Верификация

Само понятие «верификация» связано с термином *артефакт* [4]. *Артефактом* жизненного цикла ПО называются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО. Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемых в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам.

1.2.1. Формальная верификация

Формальная верификация представляет собой процесс доказательства с помощью формальных методов корректности или некорректности алгоритмов, программ и систем в соответствии с заданным описанием их свойств. Она требует высококвалифицированных специалистов в области формальных доказательств и логического вывода.

В общем случае, задача, решаемая в рамках данного подхода, является алгоритмически неразрешимой. При этом весь процесс формального

доказательства связан с огромной ручной работой, что делает его малоприменимым на практике [5].

1.2.2. Верификация на модели

Под верификацией на модели понимают метод формальной верификации, позволяющий проверить, удовлетворяет ли заданная модель системы спецификациям, написанным на формальном языке. Применение данного подхода позволяет для заданной модели поведения системы с конечным (возможно, очень большим) числом состояний проверить выполнимость некоторого логического требования (спецификации), обычно формулируемого в терминах языка *темпоральной логики* (*LTL*, *CTL* и т. д.). Таким образом, можно проверить не только условия на мгновенное состояние системы, но и историю его развития со временем.

Метод верификации на модели хорошо подходит для проверки логики «автоматных» программ. Это связано с тем, что при верификации модели обычно по программе строится модель Крипке [2], которая, фактически, является специальным видом автомата, что упрощает ее построение по автоматной программе [1]. Поэтому возможно создание формальных методов построения модели Крипке по автоматной программе и доказательство корректности этих методов. Таким образом, для программ этого класса можно исключить ошибки при построении модели по программе.

Отдельно стоит отметить простоту описания формальных требований к модели по спецификации автоматных программ. Например, такие требования, как «После состояния «открыто» всегда следует состояние «закрыто»» или «Выходное воздействие *z1* не должно появляться после входного воздействия *x1*» могут быть записаны в терминах темпоральной логики для состояний и переходов автоматов, а не модели программы [6].

Ввиду того, что при автоматном подходе модель эквивалентна верифицируемой программе, обработка контрпримеров также упрощается. При этом путь, приводящий к ошибке в модели Крипке, может быть легко

отображен в путь в автоматной программе [7]. Это позволяет наглядно продемонстрировать нарушение спецификации.

Таким образом, верификация автоматных программ является существенно более простой задачей, чем верификация программ общего вида. При этом многие этапы могут быть автоматизированы, что позволит более широко применять ее в реальных проектах и существенно снизить процент ошибок при ручном построении модели.

1.3. Темпоральные логики

Одним из языков, на котором можно специфицировать свойства систем, является *темпоральная логика* [8]. Свойства систем описываются в темпоральной логике при помощи темпоральных формул.

Примеры свойств, которые могут описываться в темпоральной логике:

- система в любом варианте своего функционирования не будет находиться ни в одном из состояний из заданного класса;
- система при некотором функционировании когда-нибудь попадет в некоторое состояние из заданного класса.

Как правило, при проведении рассуждений о темпоральных формулах рассматриваются не всевозможные формулы, а только формулы из некоторого ограниченного класса. Классы темпоральных формул принято называть *темпоральными логиками.*

Наиболее известны основные темпоральные логики:

- *логика линейного времени* (*LTL*);
- *логика вычислительных деревьев* (*CTL*).

Во всех темпоральных формулах основными структурными элементами являются *утверждения*. Утверждения имеют тот же смысл, что и в системах переходов – для каждого состояния q каждой системы переходов и каждого утверждения p определено значение $q(p) \in \{0,1\}$. Совокупность всех утверждений обозначается символом $\mathcal{P}$.

Каждая темпоральная логика Ф должна удовлетворять следующим условиям:

1. $\mathcal{P} \subseteq \Phi$
2. Символы **1** и **0** принадлежат Ф.
3. Если $\psi, \eta \in \Phi$, то знакосочетания $\overline{\psi}$, $\psi \wedge \eta$, $\psi \vee \eta$ тоже принадлежат логике Ф.

   Формулы из п. 3 называются булевыми комбинациями формул ψ и η.

1.4. Язык темпоральных логик *CTL*

Темпоральная логика *CTL* определяется следующими дополнительными правилами:

- если $\psi \in \mathrm{CTL}$, то *CTL* также содержит следующие шесть формул:

$$\mathbf{AX}\,\psi \quad \mathbf{EX}\,\psi$$
$$\mathbf{AF}\,\psi \quad \mathbf{EF}\,\psi$$
$$\mathbf{AG}\,\psi \quad \mathbf{EG}\,\psi$$

- если $\psi, \eta \in \mathrm{CTL}$, то CTL также содержит следующие две формулы:

$$\mathbf{AU}(\psi, \eta) \quad \mathbf{EU}(\psi, \eta).$$

Символы, помеченные полужирным шрифтом (**AX** и т. д.) в данных формулах называются ***CTL*-операторами**.

Символы **A** и **E** означают соответственно «все» и «существует» («any» и «exists»).

Символ **X** (ne**X**t) означает, что в следующем состоянии будет выполняться формула ψ.

Символ **F** (**F**uture) означает, что когда-нибудь после текущего состояния выполнится формула ψ.

Символ **G** (**G**lobally) означает, что после текущего состояния всегда будет выполняться формула ψ.

Также символ **U** означает, что формула ψ будет выполняться до выполнения формулы η.

1.5. Язык темпоральной логики *LTL*

Темпоральная логика LTL определяется следующими дополнительными правилами:

- если $\psi \in \mathrm{LTL}$, то *LTL* также содержит следующие три формулы:

$$\mathbf{X}\,\psi, \mathbf{F}\,\psi, \mathbf{G}\,\psi$$

- если $\psi, \eta \in \mathrm{CTL}$, то *CTL* также содержит следующую формулу:

$$\mathbf{U}(\psi, \eta).$$

В отличие от логики *CTL*, все вышеописанные формулы в логике *LTL* имеют смысл «для всех следующих» состояний.

1.6. Акторная модель многопоточности

1.6.1. Текущее положение многопоточного программирования

Не так давно ведущие производители микропроцессоров, такие как *Intel* и *AMD*, объявили о том, что, в связи с тем, что они практически подошли к теоретическому пределу скорости передачи электрического сигнала, они больше не могут увеличивать производительность способами, основанными на традиционном подходе. Поэтому были выпущены первые процессоры с многоядерной архитектурой.

Если верить закону Мура об увеличении числа транзисторов в два раза каждые полтора – два года, то нас ожидает дальнейшее увеличение числа ядер и в новых поколениях процессоров.

Если посмотреть на это со стороны программиста, то эта ситуация свидетельствует о будущей революции в технологиях разработки программного обеспечения.

Как говорит Херб Саттер, бесплатный обед закончился [9]. Раньше, если большинство приложений, написанных в последовательном стиле, просто

увеличивали скорость своей работы за счет увеличения тактовой частоты процессора, то сейчас это уже перестало быть правдой. Для того, чтобы максимально использовать архитектуры новых поколений процессоров, приложения должны быть построены на многопоточных – параллельных, вычислениях.

1.6.2. Традиционный подход к написанию многопоточных приложений

Традиционным подходом к написанию многопоточных приложений является архитектура многопоточности с общим изменяемым состоянием. Потоки параллельно выполняют свои секции кода, но для того, чтобы общие данные, используемые потоками, оставались в консистентном состоянии, нужна синхронизация. Для этого используются различные виды блокировок: мьютексы, Read-Writer-Lock'и, семафоры и т. д. Однако в достаточно больших проектах программисту сложно уследить за возможным поведением программы, и возникают различные проблемы: взаимоблокировки, соревнования за ресурс и т. д.

Если рассматривать некоторые подходы к проверке правильности программ, такие как тестирование, то эти подходы не могут точно дать уверенность в том, что программа действительно работает корректно. Это связано с тем, что, как сказал Дейкстра, «тестирование помогает найти ошибки, но совершенно не гарантирует их отсутствия» [10]. В отличие от последовательных приложений, многопоточное приложение каждый раз ведет себя по-разному: какие-то потоки выполняются раньше, какие-то позже. Перебрать всевозможные варианты запуска потоков невозможно уже в даже небольших приложениях. Для того чтобы как-то справится с этой задачей, необходимо либо повсеместно увеличивать квалификацию программистов, либо повышать уровень абстракции при построении и поддержке многопоточных приложений.

1.6.3. Акторная модель

В 1973 г. Карл Хьюитт с указанными выше соавторами предложили решение этой проблемы в виде новой архитектуры – архитектуры многопоточности с посылкой сообщений, также называемой акторной моделью (*Actor Model Concurrency*) [11].

Ключевые принципы акторной модели:

- все является актором (в объектно-ориентированном программировании: все является объектом);
- нет никаких общих данных;
- асинхронная посылка сообщений между акторами для их взаимосвязи;
- у каждого актора существует очередь для буферизации входящих сообщений.

В этой модели акторы обладают следующими свойствами и возможными действиями:

- в качестве реакции на поступающее сообщение они могут:
  - изменять только свое внутреннее состояние;
  - создавать других акторы;
  - посылать сообщения другим акторам.
- акторы никогда не используют общее состояние и, поэтому им нет необходимости заботиться о блокировках и других видах синхронизации;
- акторы не блокируются, ожидая ответов на свои запросы.

1.6.4. Существующие реализации акторной модели

Для поддержки подхода на основе акторной модели были созданы различные языки программирования и библиотеки для языков общего назначения.

Акторы в языке *Erlang*

Erlang – язык программирования для написания многопоточных программ, разработанный в компании *Ericsson*. Он был задуман как отказоустойчивый язык для использования в приложениях реального времени.

Erlang – функциональный язык, имеющий конструкции для работы с распараллеливанием и многопоточностью. У него динамическая система типов, где типы проверяются на этапе исполнения программы. Однако, в этом языке можно полностью опустить систему типов – программная среда позволяет запускать программы с неправильными спецификациями типов. Это позволяет программе продолжать работать параллельно, несмотря на ошибки типов.

Erlang частично унаследовал синтаксис языка *Prolog*.

В языке *Erlang*, который был разработан для распараллеливания и масштабируемости, акторная модель является частью языка, так как язык разрабатывался для телекоммуникационных приложений, где выполнение огромного числа параллельных процессов обычное дело, просто невозможно себе представить *Erlang* без акторов. Акторы в *Erlang* называются *процессами*, которые запускаются встроенной функцией `spawn`.

Простое приложение, использующее акторную модель представлено ниже. Актор представляет собой обычный счетчик. Пошлем ему 100000 запросов на увеличение счетчика и посылаем запрос на печать его внутреннего значения:

```
-module(counter).
-export([run/0, counter/1]).

run() ->
    S = spawn(counter, counter, [0]),
    send_msgs(S, 100000),
    S.

counter(Sum) ->
    receive
        value -> io:fwrite("Value is ~w~n", [Sum]);
        {inc, Amount} -> counter(Sum+Amount)
    end.
```

```
send_msgs(_, 0) -> true;
send_msgs(S, Count) ->
    S ! {inc, 1},
    send_msgs(S, Count-1).
```

Erlang использует упреждающий планировщик процессов: он ограничивает квант времени для каждого актора. В случае превышения времени либо отсутствия сообщений в мэйлбоксе, планировщик приостанавливает актор и кладет его в очередь для дальнейшего запуска. Это позволяет создавать системы с очень большим числом акторов: при выполнении одних процессов другие не будут ожидать слишком долго.

Акторы в языке *Scala*

Акторы в языке *Scala* доступны из библиотеки `scala.actors`. Реализация библиотеки показывает выразительную мощь этого языка: вся функциональность, операторы и другие конструкции языка реализованы в самом языке, как библиотеки, без необходимости изменять сам язык.

То же приложение, что и в предыдущем разделе, представлено ниже:

```
import scala.actors.Actor
import scala.actors.Actor._

case class Inc(amount: Int)
case class Value


class Counter extends Actor {
    var counter: Int = 0;

    def act() = {
        while (true) {
            receive {
                case Inc(amount) =>
                    counter += amount
                case Value =>
                    println("Value is "+counter)
                    exit()
            }
        }
    }
}

object ActorTest extends Application {
    val counter = new Counter
    counter.start()
```

```
        for (i <- 0 until 100000) {
            counter ! Inc(1)
        }
        counter ! Value
        // Output: Value is 100000
    }
```

Акторы в этой библиотеке поддерживают часто используемый паттерн запрос/ответ:

Обычное использование	Использование конструкции reply

`receive {`
`  case Msg(sender, value) =>`
`  val r = process(value)`
`  sender ! Response(r)`
`}`
	`receive {`
	`  case Msg(value) =>`
	`  val r = process(value)`
	`  reply(Response(r))`
	`}`

Также данная библиотека поддерживает синхронную передачу сообщений (actor !? message). Однако, несмотря на наличие доступной языковой конструкции, этим необходимо пользоваться аккуратно, так как может возникнуть взаимоблокировка:

Актор A	Актор B

`actorB !? Msg1(value) match {`
`  case Response1(r) =>`
`  // ...`
`}`
`receive {`
`  case Msg2(value) =>`
`  reply(Response2(value))`
`}`
	`actorA !? Msg2(value) match {`
	`  case Response2(r) =>`
	`  // ...`
	`}`
	`receive {`
	`  case Msg1(value) =>`
	`  reply(Response1(value))`
	`}`

1.6.5. Акторы, основанные на потоках

Акторы, основанные на потоках операционной системы, представляют собой акторы, которые выполняются в своем собственном потоке *JVM* (*Java Virtual Machine* – виртуальная машина *Java*). Они выполняются планировщиком самой виртуальной машины, который использует упреждающий основанный на приоритетах алгоритм. Когда актор входит в блок receive, поток блокируется в ожидании входящих сообщений. Такое выполнение позволяет выполнять длительные операции, никак не влияя на выполнение других акторов.

Недостаток такого подхода состоит в том, что используются «тяжелые» потоки операционной системы. Это может привести к отсутствию памяти и уменьшению производительности системы из-за переключения контекстов между потоками.

1.6.6. Акторы, основанные на событиях

В ситуациях, когда этого нельзя допустить, можно применять акторы, основанные на событиях. В этом подходе, вместо собственного потока, акторы представляются замыканием, которое хранит в себе внутреннее состояние актора. Акторы, основанные на событиях, представляют более легковесную альтернативу, предоставляя возможность выполнения очень большого числа акторов параллельно.

1.7. Достоинства акторной модели для верификации по сравнению с традиционным подходом

Акторная модель позволяет более легко строить модели для верификации по сравнению с традиционным подходом. Если блокировки и синхронизации сложно представить в виде модели системы переходов, то представление актора с собственной очередью входящих сообщений довольно легко. Также, из-за гораздо большей независимости акторов (например, никакой другой объект не может изменять данные актора, кроме его самого), можно верифицировать некоторые акторы как черные ящики – только то, что они посылают какие-то сообщения недетерминированным образом. Благодаря этому, размер общей модели сокращается в несколько раз, что позволяет уменьшить время и память, требуемые для верификации. Другим способом упрощения модели является уменьшение размера очереди входящих сообщений.

Выводы по главе 1

В связи с тем, что ведущие производители микропроцессоров достигли теоретического предела в увеличении тактовой частоты кристаллов, они

перешли к производству новых поколений процессоров, основанных на многоядерных системах. Из этого следует, что программы больше не будут увеличивать свою производительность в традиционном стиле. Для того чтобы новые приложения полностью использовали новую архитектуру многоядерных приложений, они должны быть написаны в соответствующем, многопоточном, стиле.

Стандартные подходы для обеспечения многопоточности систем слишком громоздки и полны нетривиальных ошибок, поэтому были предложены новые модели для построения многопоточных систем. Одной из наиболее распространенных моделей является акторная модель.

Были представлены существующие реализации на языках *Erlang* и *Scala*. Если язык *Erlang* представляет собой более функциональный подход и может использоваться по большей мере исследователями и учеными, то *Scala* является языком общего применения, основанным на *Java*. Рассмотрены также два подхода к реализации модели выполнения акторов:

- модель, основанная на потоках;
- модель, основанная на событиях.

Если число акторов достаточно мало, то можно использовать модель, основанную на потоках, так как напрямую используются потоки операционной системы/процессора. В случае гораздо большего числа акторов, можно использовать модель, основанную на событиях. Тогда используются легковесные «представители» акторов в виде замыканий, которые выполняются в заданном числе потоков.

Акторная модель позволяет увеличить степень абстракции в создании параллельных систем. С помощью нее можно избегать различных ошибок, связанных с блокированием потоков. Так, например, используя только асинхронную систему сообщений, можно избавиться от проблемы взаимоблокировки. Так как в акторной модели не существует общего изменяемого состояния системы, то поэтому можно избавиться от большинства синхронизирующих и блокирующих конструкций.

Глава 2. Преобразование программной модели к форме, пригодной для автоматической проверки

Рассмотрим модель программы из автоматов, взаимодействующих на основе акторной модели. В данном подходе модель имеет следующие свойства:

- независимые акторы-автоматы;
- взаимодействие через асинхронную посылку сообщений;

Данные объекты являются реактивными объектами. Вычисление происходит путем посылки сообщения и вызова соответствующего метода актора. Каждое сообщение представляет единственный метод, который будет вызван при обработке сообщения. Каждый актор имеет бесконечную очередь входящих сообщений. Когда сообщение в начале очереди обрабатывается, соответствующий метод актора вызывается, а само сообщение удаляется из очереди.

Каждый актор-класс может быть представлен в виде класса и принадлежащего ему единственного потока вычислений. Модель, используемая при данном подходе, представляет собой набор акторов, как закрытая система. Она состоит из объектов, которые выполняются параллельно и взаимодействуют путем посылки сообщений. *Компонента* модели – подмножество акторов данной модели.

Использование очереди сообщений дает нам возможность предположить, что тело каждого метода выполняется атомарно. Действительно, выполнение тела метода не может быть прервано, так как в любое момент времени, в каждом акторе выполняется максимум один метод, соответствующий обрабатываемому сообщению. Заметим, что это никак не противоречит асинхронной природе взаимодействия акторов, так как нет конструкций ожидания прихода сообщения. Также этот факт позволяет уменьшить пространство состояний и делает модель более простой.

2.1. Композиционная верификация

В теории формальной верификации всегда пытаются доказать либо опровергнуть то, что модель соответствует набору спецификаций. Существует два основных подхода к верификации:

- проверка на модели;
- доказательные методы.

Проверка на модели представляет собой полную симуляцию модели на всех возможных входных воздействиях. В доказательных методах задача формулируется в виде доказательства теоремы в системе математических доказательств, и проектировщик модели пытается построить каркас доказательства, а программный доказатель теорем при его наличии дополняет до полного доказательства.

Одной из важнейших проблем в подходе, использующем проверку на модели, является проблема комбинаторного взрыва числа состояний. Для борьбы с этой проблемой используется метод *композиционной верификации.* Цель метода – проверить свойства компонент системы и вывести глобальные свойства системы из локальных. Главная сложность этого подхода это то, что локальные свойства необязательно остаются удовлетворенными на глобальном уровне.

В методе композиционной верификации спецификация системы декомпозируется на свойства компонент, которые в дальнейшем верифицируются по отдельности. Если удается доказать, что система удовлетворяет всем локальным свойствам, и показать, что объединение этих свойств приводит к исходной спецификации системы, то можно говорить о том, что система удовлетворяет данной спецификации.

Краткий обзор композиционной верификации

В самом простом виде, предположим, что система состоит из двух компонент P и Q, которые взаимодействуют друг с другом и с их окружением.

Данная система будет обозначаться как $P||Q$. Если ϕ_P – спецификация компоненты P ($P \vDash \phi_P$) и ϕ_Q – спецификация компоненты Q ($Q \vDash \phi_Q$), можно использовать свойство:

$$\frac{P \vDash \phi_P \;\&\; Q \vDash \phi_Q \;\&\; \phi_P \wedge \phi_Q \Rightarrow \phi}{P||Q \vDash \phi}$$

Как было сказано выше, локальное свойство ϕ_P не обязательно будет удовлетворено после того, как P будет объединено с Q. Для того чтобы использовать указанное выше свойство, объединенная система не должна влиять на удовлетворение локальных свойств. В нашем примере, это означает, что композиция P и Q не должна изменять свойства ϕ_P и ϕ_Qв целой системе.

При этом отметим, что компонента системы обычно проектируется для работы с некоторым окружением. Так, например, компонент P необязательно будет обладать необходимым свойством в произвольном окружении. Пространство достижимых состояний компоненты P в любом возможном окружении может быть фактически гораздо больше, чем пространство состояний композиции P и Q. Эта проблема называется *проблемой окружения*. Возможными подходами к решению этой проблемы являются *композиционная минимизация* и подход *предположений и гарантий*. В композиционной минимизации выбирается обрезанная версия компоненты Q (назовем ее Q'), которая наследует только поведение, видимое компоненте P. Q' называется *интерфейсным процессом* и моделирует обрезанное окружение. Свойство $P||Q$ может быть заменено на $P||Q'$.

2.2. Верификация модели системы, используемой в данном подходе

В качестве записи спецификаций для системы будем использовать языки темпоральной логики. Метод проверки на модели может быть применен только для конечных систем, поэтому будем использовать различные техники абстракции для того, чтобы сделать систему конечной. Неограниченные

очереди сообщений, неограниченные типы данных, а также неограниченное создание акторов не предусмотрено. Также, для уменьшения пространства состояний модели каждый вызов метода считается атомарной операцией. Далее в деталях показаны техники абстракции, используемые в данной работе.

- *Ограниченные очереди сообщений.* Методы проверки по модели не могут справиться с бесконечным числом состояний. Поэтому каждый актор-класс имеет максимальный размер очереди сообщений, указываемый пользователем при моделировании.
- *Атомарное выполнение тела метода.* Так как в рассматриваемом подходе не предусмотрено наличие операции ожидания сообщения, а также акторы никогда не используют общее изменяемое состояние, можно реализовать метод без потери общности атомарно. Более точно, все созданные сообщения будут посланы только после выполнения каждого метода, сохраняя порядок.

2.3. Операционная семантика создаваемой модели

Акторно-автоматная модель системы может быть представлена в виде системы переходов. Система переходов – это четверка $\langle S, L, T, s_0 \rangle$, где S – множество состояний, L – множество меток переходов, T – отношение переходов из состояния в состояние, а также s_0 – множество начальных состояний системы.

Каждый актор данной системы, a_i, имеющий уникальный идентификатор i, представлен в виде тройки $\langle V_i, M_i, K_i \rangle$, где V_i – множество состояний внутреннего конечного автомата, M_i – набор его методов-обработчиков, K_i – множество акторов, о которых «знает» данный актор a_i.

Для данной модели можно ввести универсальное множество I всех акторов, участвующих в данной модели, и универсальное множество $\mathcal{K}$ всех акторов, о которых «знают» элементы из множества I.

Каждое сообщение представлено в виде тройки $msg = \langle sendid, i, mtdid \rangle$, где *sendid* – идентификатор отправителя сообщения, *i* – идентификатор получателя сообщения, *mtdid* – метод получателя сообщения, вызываемого у актора a_i при получении сообщения. Для простоты, в данном определении будем игнорировать аргументы, передаваемые в метод.

Каждый актор имеет очередь сообщений, которая может быть представлена в виде конечной последовательности сообщений. Назовем множество всех конечных последовательностей заданного множества A как $seq(A)$. Очередь сообщений для компонента является мультиочередью, заключающей в себе все очереди сообщений акторов.

Основные определения, используемые в работе, приведены ниже:

- a_i – актор с уникальным номером *i*, представленный в виде $\langle V_i, M_i, K_i \rangle$.
- V_i – множество состояний конечного автомата актора a_i.
- M_i – множество методов-обработчиков актора a_i.
- K_i – множество акторов, о которых «знает» актор a_i.
- I – множество всех идентификаторов акторов.
- $\mathcal{K} = \bigcup_{i \in I} K_i$.
- $\mathcal{M} = ||_{i \in I} a_i$ – множество параллельно выполняющихся акторов, и также, $M_{\mathcal{M}} = \bigcup_{i \in I} M_i$, $K_{\mathcal{M}} = \bigcup_{i \in I} K_i$.
- $msg = \langle sendid, i, mtdid \rangle$ – сообщение, посланное акторов *sendid* для вызова метода *mtdid* актора *i*.
- $Q_{\mathcal{M}} = \prod_{i \in I} seq(I_{\mathcal{M}} \times M_i)$ – множество всех возможных состояний очереди сообщений модели $\mathcal{M}$.

Пространство состояний модели:

$$\prod_{i \in I} (S_i \times q_i),$$

где S_i – модель локального состояния актора a_i, а q_i – очередь сообщений данного актора.

Множество меток L – набор всех вызовов методов обработчиков сообщений.

Тройка $\langle s, l, s' \rangle \in S \times L \times S$ является элементом отношения переходов T тогда и только тогда, когда:

- в состоянии s существует такой i, что l – первое сообщение в очереди сообщений q_i, и e представлен в виде $\langle sendid, i, mtdid \rangle$.
- состояние s' получается из состояния s, выполняя два действия:
  - актор удаляет первое сообщение из очереди;
  - метод *mtdid* вызывается атомарно в состоянии s. Тело метода может добавлять сообщения в мейлбоксы акторов, а также изменять внутреннее состояние.

Начальным состоянием актора s_0 является начальное состояние внутреннего автомата и набор акторов, о которых «знает» данный актор.

Операционная семантика модели $\mathcal{M} = ||_{i \in M} a_i$ определяется как система переходов $\langle S, L, T, s_0 \rangle$:

- $S = \prod_{i=1}^{n} S_i \times q_i$, где $S_i = v_i \times K_i$. v_i – состояние внутреннего автомата актора, K_i – множество акторов и их методов, о которых «знает» актор a_i; q_i – множество всех возможных состояний очереди сообщений.
- $L = \bigcup_{i \in I} I \times M_i$ – множество меток, соответствующим всем возможным сообщениям таким, что $\forall (x, y, m) \in L$ найдется $m \in K_x$.
- $T \subseteq S \times L \times S$ – отношение переходов:

  $s_1 \rightarrow^l s_2 \in T \Leftrightarrow s_1, s_2 \in S$ и $l = (x, y, m) \in L$ – активный переход, что означает. $\exists i \in I \wedge q \in Q \mid l = head(s_1.q.i)$, т. е. l – сообщение в начале очереди, и s_2 отображается из s_1 и l следующим образом:

– Сообщение удаляется из $s1.q$, т.е. $s_2.q.y := tail(s1.q.y)$;

– Переход, вызванный сообщением $l = (x, y, m)$, выполняет метод m актора y атомарно:

- выполнение метода может изменять внутреннее состояние автомата;
- выполнение инструкции посылки сообщения изменяет состояние очереди сообщений принимающего актора.

- $s_0 = v_{init} \times q_0$ – начальное состояние модели. Каждый внутренний автомат переходит в начальное состояние, и q_0 определяется так, что у каждого актора a_i в очереди сообщений будет находиться одно сообщение $(i, i, init_i)$.

2.4. Композиционная верификация модели, используемая в данном подходе

В общем случае композиционная верификация более применима, когда модель системы естественно декомпозируема. В частности, модель, состоящая из независимых модулей, вполне подходит для композиционной верификации. Акторная модель, используемая в данном подходе, предоставляет такие независимые модули из-за наличия асинхронного механизма взаимодействия, который представляет собой только явную, не блокирующую операцию посылки сообщения.

- *Декомпозиция на компоненты.* В данном подходе, можно декомпозировать цельную модель на две сущности: компоненту как открытую систему, и все остальное в качестве окружения. Более точно, система разбивается на компоненту как набор модулей с внутренним состоянием и поведением, а также набор остальных модулей, которые моделируются как сущности, просто

посылающие сообщения. Будем называть первую и вторую группу модулей *внутренними* и *внешними* соответственно.

- *Моделирование окружения.* Так как акторы окружения в данном подходе для верификации никогда не исполняют собственные методы, то не требуется моделировать их очереди сообщений, состояния, а также их поведение.
- *Абстрагирование от аргументов и динамической топологии.* Для простоты будем абстрагироваться от наличия параметров в определениях методов. Это обусловлено тем, что методы с аргументами, ограничивающиеся конечным набором значений, могут быть промоделированы как несколько методов без параметров. Также, предполагая наличие верхней границы числа созданных объектов, можно промоделировать ограниченную динамическую топологию.
- *Динамическое создание акторов.* В композиционной верификации поведение внутренних акторов полностью моделируется без каких-либо техник абстракций. Таким образом, создание внутренних акторов также может быть легко промоделировано. При создании внешних акторов также можно их моделировать только посылаемыми ими сообщениями.

Ниже представлены основные определения для компонент:

- $C = ||_{i \in I_C} a_i$ – множество акторов $\{a_i | i \in I_C\}$, выполняющихся параллельно. Также $V_C = \bigcup_{i \in I_C} V_i, M_C = \bigcup_{i \in I_C} M_i, K_C = \bigcup_{i \in I_C} K_i$.
- $I_C \subseteq I$ – набор идентификаторов акторов, входящих в компоненту C.
- $C = ||_{i \in 1..n} C_i$ – параллельная композиция n компонент C_i. Также $I_C = \bigcup_{i \in 1..n} I_{C_i}, V_C = \bigcup_{i \in 1..n} V_{C_i}, M_C = \bigcup_{i \in 1..n} M_{C_i}, K_C = \bigcup_{i \in 1..n} K_{C_i}$.
- $Q_C = \prod_{i \in I_C} seq(I_C \times M_i)$ – множество возможных состояний очереди сообщений компоненты C, представленной как мульти-очередь.

Каждая очередь представлена как конечная последовательность сообщений, каждое из которых адресуется внутреннему актору.

Операционная семантика:

Операционная семантика компоненты $C = ||_{i \in I_C} a_i$ определяется как система переходов $\langle S_C, L_C, T_C, s_{C_0} \rangle$:

- $S_C = \prod_{i=1}^{n} S_i \times q_i$ – множество состояний компоненты, где $S_i = v_i \times K_i$. v_i – состояние внутренних автоматов акторов компоненты, K_i – множество акторов и их методов, о которых «знает» актор a_i; q_i – множество всех возможных состояний очереди сообщений.
- $L_C = \bigcup_{i \in I_C} I \times M_i$ – множество меток, соответствующим всем возможным сообщениям к компоненте C таким, что $\forall (x, y, m) \in L_C$ найдется $m \in K_x$.
- $T_C \subseteq S_C \times L_C \times S_C$ – отношение переходов:

  $s_1 \to^l s_2 \in T_C \Leftrightarrow s_1, s_2 \in S_C$ и $l = (x, y, m) \in L_C$ – активный переход, что означает $\exists i \in I_C \wedge q_C \in Q_C \mid l = head(s_1.q_C.i)$, т.е. l – внутреннее сообщение в начале очереди, или $l.x \notin I_C$, т.е. l – внешнее сообщение, и s_2 отображается из s_1 и l следующим образом:

  – Если $x \in I_C$, то сообщение удаляется из $s1.q_C$, т.е. $s_2.q_C.y := tail(s1.q_C.y)$, иначе $s_1.q_C$ не изменяется.

  – Переход, вызванный сообщением $l = (x, y, m)$, выполняет метод m актора y атомарно:

    - выполнение метода может изменять внутреннее состояние автоматов ($s_1.v_C$);

- посылка сообщения. Если получатель – внутренний актор, то изменяется очередь сообщений $s_1.q_C$. Иначе эта операция не изменяет внутреннего состояния компоненты.

- $s_{c_0} = v_{c_{init}} \times q_{c_0}$ – начальное состояние компоненты. Каждый внутренний автомат переходит в начальное состояние, и q_0 определяется так, что у каждого актора a_i в очереди сообщений будет находиться одно сообщение $(i, i, init_i)$.

2.5. Формальное обоснование предложенного подхода

Проблема комбинаторного взрыва состояний может быть преодолена использованием техник, заменяющих компоненты большого размера компонентами меньшего размера, которые удовлетворяют одни и те же свойства. При этом необходимо определить понятие эквивалентности данных структур, гарантирующее то, что две компоненты удовлетворяют один и тот же набор формул в заданной логике.

Моделирование состоит в создании компоненты, являющейся абстракцией исходной компоненты. Так как абстракция может скрывать некоторые детали исходной структуры, она может иметь меньшее по размеру множество состояний. Моделирование должно гарантировать, что любое поведение начальной структуры также будет проявляться и в абстракции. Однако, абстракция может иметь дополнительные поведения, которые невозможны в исходной компоненте.

Слабое моделирование и сохранение свойств

Объясним отношение слабого моделирования между компонентами нашей модели. Как было отмечено выше, внешние сообщения, посылаемые компоненте, присутствуют во всех состояниях. Действительно, в каждом состоянии системы переходов имеется множество текущих состояний

внутренних автоматов, мульти-очередь сообщений, а также множество внешних сообщений. Ввиду того что множество внешних сообщений константно во всех состояниях, его можно не учитывать.

Определим понятие *проекции* между двумя состояниями. Состояние $s_{C'}$ является проекцией состояния s_C (записывается как $s_{C'} \uparrow s_C$), если

- $I_{C'} \subseteq I_C$;
- общие акторы имеют одинаковые текущие состояния внутренних автоматов;
- мульти-очередь $s_{C'}.q_{C'}$ является проекцией $s_C.q_C$.

Мульти-очередь $q_{C'}$ является проекцией мульти-очереди q_C (записывается как $q_{C'} \uparrow q_C$), если

- $I_{C'} \subseteq I_C$;
- для каждого $i \in I_{C'}$ последовательность сообщений $\langle sendid, i, mtdid \rangle$ в очереди q_C, игнорируя сообщения с $sendid \in I_C - I_{C'}$, равна последовательности сообщений $q_{C'}$.

Теперь, используя данную терминологию, введем отношение слабого моделирования.

Определение 1 (Слабое моделирование). Даны две компоненты C' и C данной модели, представленные системами переходов $\langle S_{C'}, L_{C'}, T_{C'}, s_{C'_0} \rangle$ и $\langle S_C, L_C, T_C, s_{C_0} \rangle$ соответственно, таких что $I_{C'} \subseteq I_C$:

1. Отношение $H \subseteq S_C \times S_{C'}$ является отношением слабого моделирования между C' и C тогда и только тогда, когда для всех $s_C \in S_C, s_{C'} \in S_{C'}$, если $H(s_C, s_{C'})$, то выполняются следующие условия:
   a. $s_{C'} \uparrow s_C$;

b. Для любого состояния s_{C_1}и метки $l \in L_C$, таких что $(s_C, l, s_{C_1}) \in T_C$, существует состояние $s_{C'_1}$, такое что, $s_{C'_1} = s_{C_1}$ (если $l \notin L_{C'}$), или $(s_{C'}, l, s_{C'_1}) \in T_{C'}$(если $l \in L_{C'}$) и $H(s_{C_1}, s_{C'_1})$.

2. Компонента C' слабо моделирует компоненту C (записывается как $C \leq C'$), если существует отношение слабого моделирования H между C и C', такое что $H(s_{C_0}, s_{C'_0})$.

Теорема 1 (Отношение слабого моделирования между компонентой и ее композицией с произвольной компонентой). Для любых двух компонент C'и X заданной модели (заданные на одном и том же множестве акторов), C' слабо моделирует $C = C' || X$.

Доказательство. Рассмотрим $H = \{(s_C, s_{C'}) \in S_C \times S_{C'} | s_{C'} \uparrow s_C\}$. Требуется показать, что (1) H – отношение слабого моделирования, и что (2) $H(s_{C_0}, s_{C'_0})$.

1. H – отношение слабого моделирования:

(а) $s_{C'} \uparrow s_C$ - по построению H.

(б) Пусть $H(s_C, s_{C'})$ и $l \in L_C$ такое, что $(s_C, l, s_{C_1}) \in T_C$

i. Если $l \notin L_{C'}$, тогда $s_{C'}$ остается неизменным – $s_{C'_1} = s_{C'}$, и все еще имеем $H(s_{C_1}, s_{C'_1})$. Однако $l \notin L_{C'}$ означает, что l – сообщение к компоненте X, т.е. $l = (p, r, m), r \in I_X, r \notin I_{C'}$. В этом случае m будет выполнен и внутреннее состояния $C'(V_{C'})$ не изменится, а также сообщения, которые могут быть посланы m не будут положены в мультиочередь компоненты C'. Следовательно, $q_{C'}$ не изменится, и также будет существовать $H(s_{C_1}, s_{C'_1})$.

ii. Если $l \in L_{C'}$, то следовательно, $r \in I_{C'}$. В этом случае, необходимо показать, что l – активная метка в состоянии $s_{C'}$, а также что $s_{C'_1} \uparrow s_{C_1}$.

Сначала покажем, что l – активная метка в состоянии $s_{C'}$ во всех возможных ситуациях:

- l – внешняя метка и для C, и для C'. Известно, что $I_{C'} \subseteq I_C$, $I'_{C'} \subseteq I'_C$ и множество внешних сообщений к компоненте C является подмножеством сообщений к компоненте C'. Следовательно, l – активно в состоянии $s_{C'}$.
- l – внутренняя метка для компоненты C и внешняя для C'. Это означает, что l – сообщение, отправленное актором из компоненты X – $p \in X$. Любые внешние сообщения для C' являются активными, следовательно, l также активно в $s_{C'}$.
- l – внутренняя метка и для C, и для C'. Известно, что $H(s_C, s_{C'})$, и то, что $s_{C'} \uparrow s_C$, а также $q_{C'} \uparrow q_C$. Из определения проекции известно, что l – сообщение в вершине очереди в состоянии s_C. Поэтому оно также должно быть в вершине очереди в состоянии $s_{C'}$. Следовательно, l – активно в состоянии $s_{C'}$.

Теперь покажем, что $s_{C'_1} \uparrow s_{C_1}$ выполняется во всех трех ситуациях:

- выполнение метода m влечет за собой одни и те же изменения внутреннего состояния в обоих компонентах;
- метод может посылать сообщения акторам из компоненты C', одинаково изменяя очереди сообщений s_C и $s_{C'}$. Также он может посылать сообщения акторам из компоненты X. При этом очереди сообщений $s_{C'_1}.q_{C'}$ и $s_{C_1}.q_C$ оказываются разными, но все еще гарантируется $s_{C'_1} \uparrow s_{C_1}$ и $q_{C'} \uparrow q_C$.

2. Теперь покажем, что $s_{C'_0} \uparrow s_{C_0}$. Это следует из определения начального состояния в операционной семантике для компонент: $s_{C'_0}.V_{C'} \subseteq s_{C_0}.V_C$.

Также, $s_{C_0'}.q_{C'} \uparrow s_{C_0}.q_C$, так как только сообщения инициализации $(i, i, init_i)$ лежат в обеих очередях.

Теперь можно использовать модифицированную теорему Кларка [2], ограничиваясь свойствами безопасности (*safety property*).

Теорема 2 (Сохранение свойств). Если $C' \leq C$, тогда для любого свойства безопасности, записанной в виде формулы ϕ логики *LTL-X* (логика *LTL* без оператора *next*), из $C' \vDash \phi$ следует $C \vDash \phi$.

Используя теорему 2, можно вывести следующее следствие для композиционной верификации LTL-X свойств безопасности. Пусть $R = ||_{i=1}^{n} X_i$ – параллельная композиция n компонент X_i, $I_R = \bigcup_{i=1}^{n} I_i$.

Следствие (Композиционная верификация). Пусть $R = ||_{i=1}^{n} X_i$ и ϕ_{X_i} – свойство безопасности каждого X_i, заданного в логике *LTL-X*. Для того чтобы удостовериться, что ϕ_R – свойство системы R, достаточно доказать свойства методом проверки на модели для X_i ($\forall i = 1..n, X_i \vDash \phi_{X_i}$). После этого, если $\left(\wedge_{i=1}^{n} \phi_{X_i}\right) \Rightarrow \phi_R$, тогда ϕ_R – свойство системы R.

Глава 3. Реализация предложенного подхода

Изложенный метод преобразования программных моделей к форме, пригодной к их верификации, описанный в главе 2, предлагается интегрировать в процесс их разработки. При этом сам процесс создания программ, состоящих из автоматов, взаимодействующих по акторной модели, а также написания конфигурации для верификации создаваемой модели должен быть эффективным, позволяя сочетать возможности современной среды разработки (статические проверки, рефакторинг, авто-дополнение и т. д.) и семантическую проверку программного кода.

3.1. Мультиязыковая среда MPS

Для реализации предложенного подхода была выбрана мультиязыковая среда *MPS* (*http://www.jetbrains.com/mps*), которая позволяет как создавать новые языки, так и расширять уже существующие.

Заметим, что языки, разрабатываемые в среде *MPS* не являются текстовыми в традиционном понимании, так как пользователь пишет не текст программы, а вводит ее в виде *абстрактного синтаксического дерева* (АСД). Такой подход позволяет обойтись без создания лексических и синтаксических анализаторов при создании новых языков, а также настроить преобразования АСД в код на конкретном языке программирования, задать удобную среду для его редактирования. Кроме того, пользователь получает возможность после трансляции программы, написанной на языке, созданном в *MPS*, получить код, не зависящий от этой системы.

3.2. Архитектура реализации данного подхода.

В качестве исходной модели используется программная модель, написанная на разработанных в рамках данной работы языках *ActorsLanguage* и *StateMachineLanguage*. После написания программы, она будет генерироваться

в исходный код на одном из языков общего назначения, таких как *Java* или *C#*. Параллельно, исходная модель преобразуется в код, доступный для внешних систем, осуществляющих проверку на модели. Наиболее распространенными системами, осуществляющие проверку на модели, являются системы *NuSMV*, *SPIN*, а также *Bogor*.

3.3. Разработка языка для входа системы *NuSMV*

В качестве системы, осуществляющую проверку на модели была выбрана система *NuSMV*. Для того, чтобы правильно преобразовать входную модель программной системы в формат, воспринимаемый этой системой, потребовалось создать язык в системе *MPS*.

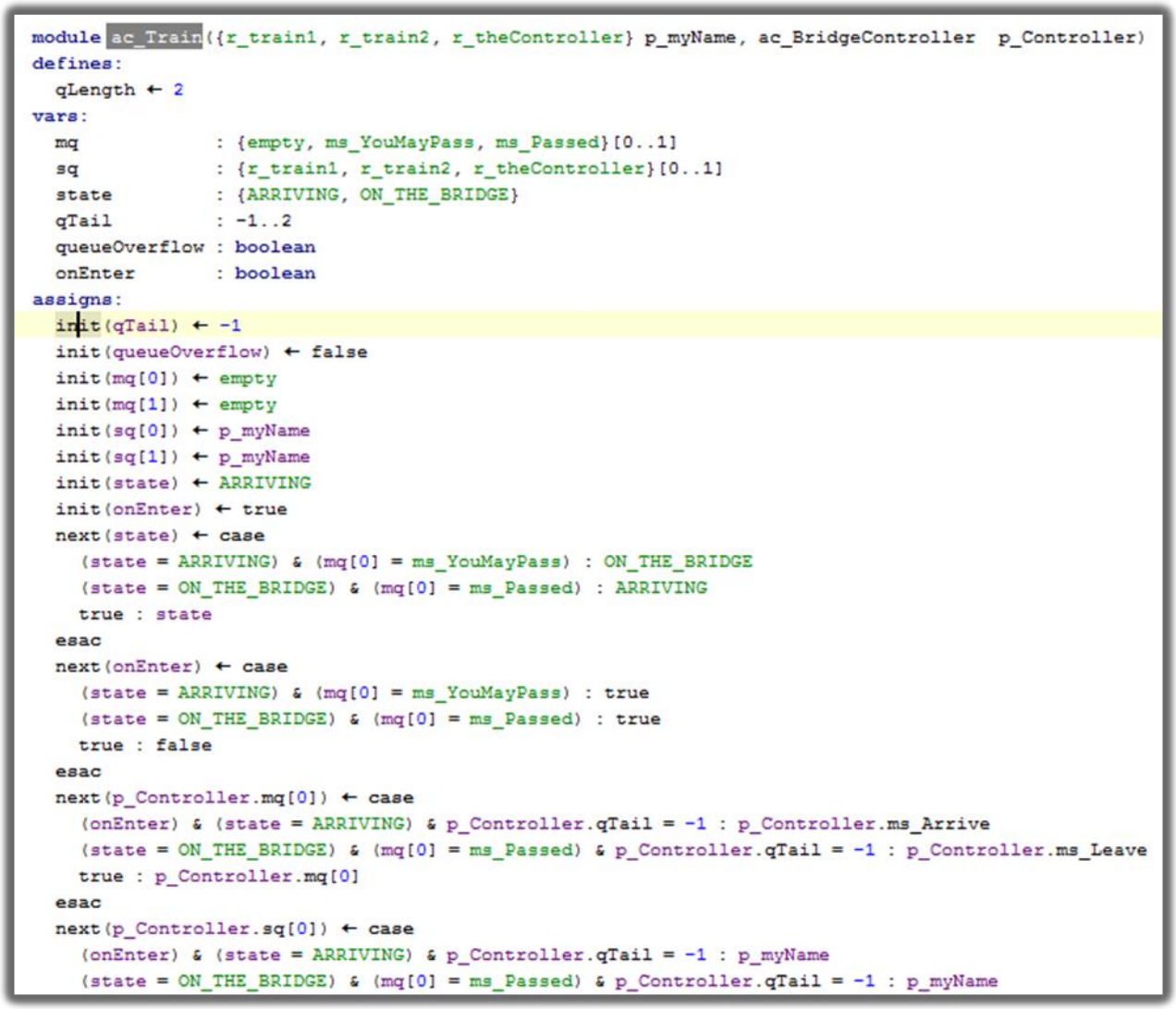
Для создания языка в системе *MPS* требуется разработать:

- Структуру абстрактного синтаксического дерева для разрабатываемого языка;
- Модель текстового редактора для каждого типа узла абстрактного синтаксического дерева;
- Модель ограничений на экземпляры абстрактного синтаксического дерева;
- Систему типов и правила преобразования элементов языка.

3.4. Описание необходимых сущностей языка *NuSMV*

Каждая программа, воспринимаемая системой *NuSMV* (рис. 1), представляет собой набор модулей, которые, в свою очередь, описывают свое внутреннее состояние, а также систему переходов в зависимости от входных воздействий.

В каждой программе существует модуль **main**, который указывает конфигурацию других модулей и является в некотором смысле стартовой точкой программы.



```
module ac_Train({r_train1, r_train2, r_theController} p_myName, ac_BridgeController  p_Controller)
defines:
  qLength ← 2
vars:
  mq            : {empty, ms_YouMayPass, ms_Passed}[0..1]
  sq            : {r_train1, r_train2, r_theController}[0..1]
  state         : {ARRIVING, ON_THE_BRIDGE}
  qTail         : -1..2
  queueOverflow : boolean
  onEnter       : boolean
assigns:
  init(qTail) ← -1
  init(queueOverflow) ← false
  init(mq[0]) ← empty
  init(mq[1]) ← empty
  init(sq[0]) ← p_myName
  init(sq[1]) ← p_myName
  init(state) ← ARRIVING
  init(onEnter) ← true
  next(state) ← case
    (state = ARRIVING) & (mq[0] = ms_YouMayPass) : ON_THE_BRIDGE
    (state = ON_THE_BRIDGE) & (mq[0] = ms_Passed) : ARRIVING
    true : state
  esac
  next(onEnter) ← case
    (state = ARRIVING) & (mq[0] = ms_YouMayPass) : true
    (state = ON_THE_BRIDGE) & (mq[0] = ms_Passed) : true
    true : false
  esac
  next(p_Controller.mq[0]) ← case
    (onEnter) & (state = ARRIVING) & p_Controller.qTail = -1 : p_Controller.ms_Arrive
    (state = ON_THE_BRIDGE) & (mq[0] = ms_Passed) & p_Controller.qTail = -1 : p_Controller.ms_Leave
    true : p_Controller.mq[0]
  esac
  next(p_Controller.sq[0]) ← case
    (onEnter) & (state = ARRIVING) & p_Controller.qTail = -1 : p_myName
    (state = ON_THE_BRIDGE) & (mq[0] = ms_Passed) & p_Controller.qTail = -1 : p_myName
```

Рис. 1. Пример синтаксиса программы на языке *NuSMV*

Каждый модуль имеет:

- аргументы, передаваемые ему при создании;
- набор внутренних переменных;
- блоки управления значением внутренних переменных.

Система позволяет использовать в качестве основных типов только примитивные целые (`byte`, `short`, `int`, `boolean`), массивы со статической длиной, а также типы перечисления.

Блоки управления значением внутренних переменных разделяются на две группы:

- блоки, отвечающие за инициализацию переменных (**init**);

- блоки, отвечающие за изменение переменных при различных условиях (**next).**

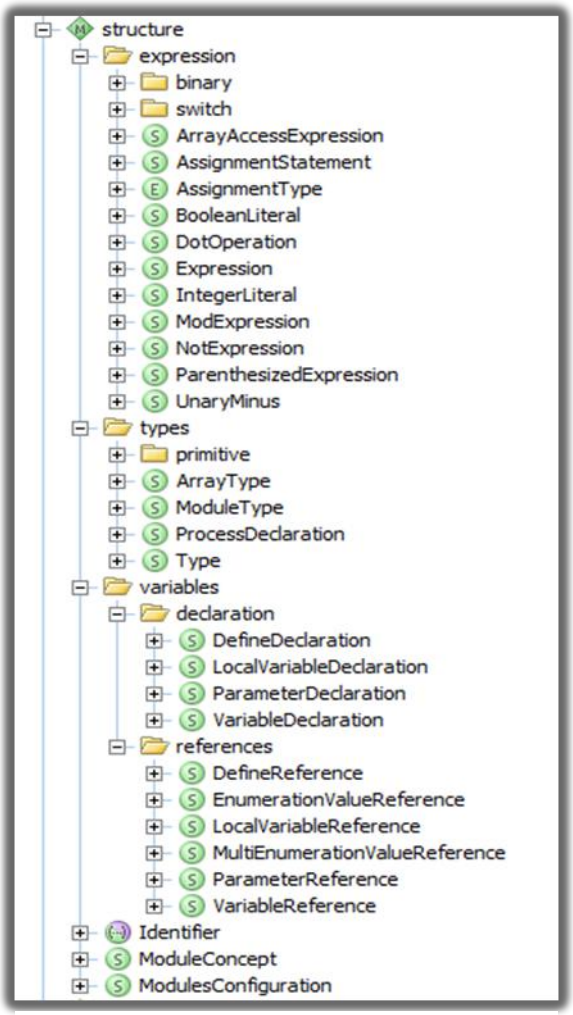


Рис. 2. Структура созданного языка для системы *NuSMV*

Условия задаются в виде булевой формулы, использующей обычные булевы операторы.

В блоках изменения внутренних переменных (**next)** для описания различных условия используется конструкция множественного условия (**case)**.

Также в процессе создания языка (рис. 2) была разработана система типов, которая позволяет вычислить ошибки типизации выражений во время написания/изменения сгенерированного кода.

3.5. Язык, осуществляющий генерацию программной модели в язык *NuSMV*

Основным элементом этого языка является конфигурация акторов, включающая размеры внутренних очередей сообщений, диапазонов значений внутренних типов данных, а также указания какие акторы будут моделироваться полностью (с поведением и внутренним состоянием), а какие будут моделироваться как черные ящики (только посылка сообщений).

Для лучшего понимания рассмотрим преобразование модели на основе примера. Система представляет собой работу блока управления развилки на

железной дороге для обеспечения передвижения поездов. Система состоит из двух типов сущностей:

- модуль управления развилкой (*BridgeController*);
- модуль, симулирующий движущийся поезд (*Train*).

В рамках данной работы каждая сущность представляет собой актор, внутреннее состояние которого моделируется с помощью конечного автомата (рис. 3, 4).

```
state machine for BridgeController {
  <listeners>

  initial state{OPEN} {
    on arrive(train) do {
      System.out.println(this.state + ".Train arrived");
      train.youMayPass();
    } transit to {TRAIN_RUNNING}
  }

  state{TRAIN_RUNNING} {
    on arrive(train) do {
      System.out.println(this.state + ".Train arrived");
    } transit to {ONE_TRAIN_RUNNING_ANOTHER_WANNA}

    initial state{ONE_TRAIN_RUNNING} {
      on leave(train) do {
        System.out.println(this.state + ".Train leaved");
      } transit to {OPEN}
    }

    state{ONE_TRAIN_RUNNING_ANOTHER_WANNA} {
      on leave(train) do {
        System.out.println(this.state + ".Train leaved");
        if (train == this.train1) {
          this.train1.youMayPass();
        } else {
          this.train2.youMayPass();
        }
      } transit to {ONE_TRAIN_RUNNING}
    }
  }
}
```

Рис. 3. Автомат для модуля управления развилкой

```
state machine for Train {
  <listeners>

  initial state{ARRIVING} {
    enter do {
      this.controller.arrive(this);
    }

    on youMayPass do {<no statements>}
        transit to {ON_THE_BRIDGE}
  }

  state{ON_THE_BRIDGE} {
    enter do {
      TimerJobs.passTheTrain(this);
    }

    on passed do {
      this.controller.leave(this);
    } transit to {ARRIVING}
  }
}
```

Рис. 4. Автомат для модуля, симулирующего поезд

Каждый модуль воздействует на другой посредством очереди сообщений. Благодаря созданному языку *actorsLanguage*, в коде программы это нигде не наблюдается, что позволяет улучшить читабельность кода.

Процесс преобразования в формат, воспринимаемый системой *NuSMV*:

- в каждом модуле создается переменная **state**, имеющая тип перечисления из всех возможных состояний автомата, а также соответствующие блоки управления этой переменной (рис. 5).

```
state         : {OPEN, TRAIN_RUNNING, ONE_TRAIN_RUNNING, ONE_TRAIN_RUNNING_ANOTHER_WANNA}
next(state) ← case
  (state = OPEN) & (mq[0] = ms_Arrive) : TRAIN_RUNNING
  (state = TRAIN_RUNNING) & (mq[0] = ms_Arrive) : ONE_TRAIN_RUNNING_ANOTHER_WANNA
  (state = TRAIN_RUNNING) & (mq[0] = ms_Leave) : OPEN
  (state = ONE_TRAIN_RUNNING) & (mq[0] = ms_Arrive) : ONE_TRAIN_RUNNING_ANOTHER_WANNA
  (state = ONE_TRAIN_RUNNING) & (mq[0] = ms_Leave) : OPEN
  (state = ONE_TRAIN_RUNNING_ANOTHER_WANNA) & (mq[0] = ms_Arrive) : ONE_TRAIN_RUNNING_ANOTHER_WANNA
  (state = ONE_TRAIN_RUNNING_ANOTHER_WANNA) & (mq[0] = ms_Leave) : ONE_TRAIN_RUNNING
esac
```

Рис. 5. Моделирование текущего состояния модуля управления развилкой

- создаются два массива (рис. 6), соответствующие очереди сообщений (**M**essage**Q**ueue для сообщений, **S**ender**Q**ueue для отправителей сообщений). Элементы массива **mq** имеют тип перечисления, построенный из набора обработчиков входных воздействий, указанных в интерфейсе актора.

```
mq            : {empty, ms_initial, ms_Arrive, ms_Leave}[0..3]
sq            : {r_train1, r_train2, r_theController}[0..3]
```

Рис. 6. Созданные массивы **mq** и **sq** для очереди сообщений при длине очереди 4

- для моделирования операции посылки сообщения создается набор блоков изменения переменных внешних акторов (рис. 7), отвечающих за операцию добавления сообщения в очередь.

```
next(p_t1.mq[0]) ← case
  (state = OPEN) & (mq[0] = ms_Arrive) & (sq[0] = r_train1) & (p_t1.qTail = -1) : p_t1.ms_YouMayPass
  (state = ONE_TRAIN_RUNNING_ANOTHER_WANNA) & (mq[0] = ms_Leave) & (sq[0] = r_train2) & (p_t1.qTail = -1) :
      p_t1.ms_YouMayPass
  true : p_t1.mq[0]
esac
next(p_t1.sq[0]) ← case
  (state = OPEN) & (mq[0] = ms_Arrive) & (sq[0] = r_train1) & (p_t1.qTail = -1) : p_myName
  (state = ONE_TRAIN_RUNNING_ANOTHER_WANNA) & (mq[0] = ms_Leave) & (sq[0] = r_train2) & (p_t1.qTail = -1) : p_myName
  true : p_t1.sq[0]
esac
```

Рис. 7. Блоки, отвечающие за операцию добавления в очереди внешних акторов

Так как в языке *NuSMV* нет встроенной инструкции вставки элемента в очередь, используются пара переменных **qTail** и **qLength**, которые предназначены для определения положения добавляемого элемента.

При моделировании акторов как черного ящика, в каждый модуль, который может принимать сообщения от данного актора, добавляется переменная **env_message**, отвечающая за недетерминированный приход сообщения (рис. 8).

```
env_message   : {empty, ms_Arrive_from_ext_r_train2, ms_Leave_from_ext_r_train2}
```

Рис. 8. Определение переменной **env_message** для моделирования приходящих сообщений

Выводы по главе 3

В данной главе рассмотрены детали реализации предлагаемого подхода к верификации систем из взаимодействующих по акторной модели конечных автоматов. Рассмотрена структура языков в мультиязыковой среде *MPS*:

- язык *actorsLanguage* для моделирования акторной модели;
- язык *stateMachine* для описания автоматного аспекта программного объекта;
- язык *NuSMV* для описания формата данных, воспринимаемого системой, осуществляющей проверку на модели.
- язык конфигурации модели для верификации;

Также рассмотрены на примере детали преобразования исходной модели в набор модулей на языке *NuSMV*:

- преобразование набора состояний и переходов автоматного аспекта программного объекта в объявление переменной *state* и объявления блоков управления этой переменной;
- моделирование очередей сообщений и их отправителей;
- моделирование отправки сообщений другим акторам;
- моделирование принятия сообщений от акторов, являющимися черными ящиками, с помощью недетерминированной переменной *env_message.*

Рассмотрен процесс интеграции данного подхода в процесс разработки программного обеспечения.

Глава 4. Результаты применения данного подхода для верификации различных систем

В рамках данной работы было рассмотрено два примера, на которых проводились замеры числа состояний и требуемого времени для осуществления верификации на модели программным средством *NuSMV*.

4.1. Задача развилки

Данная задача была рассмотрена в разд. **5** главы **3**. В качестве спецификации было рассмотрено свойство «два поезда никогда не окажутся на развилке одновременно», и были рассмотрены различные конфигурации получаемой модели (в скобках указаны объекты, моделируемые как черный ящик):

Конфигурация	Число достижимых состояний	Общее число состояний	Время	Использованная память (КБ)
2 Поезда/Развилка	203	5.08e+13	0.01 с	8956
1 Поезд/Развилка (1 Поезд)	231	2.38e+09	0.002 с	8612

4.2. Задача о философах

В данной задаче рассмотрен небольшая модель, состоящая из `N` философов и круглого стола, на котором разложено `N` вилок. Свойства объектов:

- каждый философ почти всегда думает, но иногда он заходит в столовую поесть;
- если нет места со свободной левой вилкой, то он уходит;
- философ садиться за свободное место со свободной левой вилкой:
    - он берет левую вилку и начинает ожидать освобождения правой;
    - после получения правой вилки, он какое-то время ест, а потом освобождает обе вилки и уходит думать.

В данной задаче было проверено свойство «никакие два соседних философа не используют одновременно одну вилку»:

Конфигурация	Число достижимых состояний	Общее число состояний	Время	Использованная память (КБ)
2 Фил/2 Вилки 3 Фил/3 Вилки 4 фил/4 Вилки	285 14671 390720	3.26e+22 8.79e+36 1.80e+52	0 с 12 с 6 мин. 28 с.	11136 19304 38700
2 Фил/1 Вилка (2 Вилки)	4132	1.16e+21	2 с	14076

ЗАКЛЮЧЕНИЕ

В последнее время все чаще возникает необходимость в написании многопоточных программ. Для этого существует традиционный подход – использование примитивов синхронизации и блокировок. Но такой подход имеет большой недостаток – сложность в проверки корректности. Поэтому все чаще используются различные методики абстрагирования от низкоуровневых примитивов потоков и синхронизации. Одной из таких абстракций является акторная модель многопоточности. Благодаря строгому набору правил, обеспечивающих совместную работу различных акторов, программные системы, построенные на основе акторной модели, обладают достаточной независимостью, что дает возможность строить более простые модели системы для верификации, а также с успехом проводить композиционную верификацию. Если при этом еще **реализовать внутреннее поведение акторов в виде конечного автомата**, то получаемая модель получается еще более простой и дает возможность проверять более сложные системы за меньшее время.

В рамках данной работы был разработан метод верификации программных моделей, построенных на основе взаимодействующих по акторной модели автоматов, и данный метод был внедрен в процесс разработки программных систем в среде *MPS*. Также были получены различные статистики времени и памяти, требуемых для осуществления верификации различных конфигураций модели.

В качестве направлений дальнейших исследований можно рассматривать расширение границ применимости данного подхода. Также в дальнейшем планируется использовать другие системы, осуществляющие верификацию на модели, такие как *SPIN* или *Bogor*.

СПИСОК ЛИТЕРАТУРЫ

1. *Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Верификация автоматных программ. Саратов: СГУ. 2007. http://is.ifmo.ru/verification/_KNIT-2007.pdf.

2. *Clarke E. M., Glumberg O., Peled D. A.* Model Checking, The MIT Press, Cambridge, Massachusetts, 1999.

3. *NuSMV User Manual*. http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html.

4. *Кулямин В. В.* Методы верификации программного обеспечения. Институт системного программирования РАН.

5. *Отчет по контракту о верификации автоматных программ.* Второй этап http://is.ifmo.ru/verification/_2007_02_report-verification.pdf.

6. *Курбацкий Е. А., Шалыто А. А.* Верификация программ, построенных при помощи автоматного подхода. СПбГ ПУ. 2008.

7. *Вельдер С. Э., Шалыто А. А.* Введение в верификацию автоматных программ на основе метода Model checking. http://is.ifmo.ru/download/modelchecking.pdf.

8. *Лившиц Ю.* Верификация программ. Курс лекций. http://yury.name/modern/

9. *Sutter H.* The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal. 2005.

10. *Э. Дейкстра*. http://en.wikipedia.org

11. *Actor Model*. http://en.wikipedia.org/wiki/Actor_model

12. *Жукова А. Р., Мазин М. А.* Акторное расширение языка Java в среде MPS.

13. *Sirjani M., Jaghoori M., Baier C., Arbab F.* Compositional semantics of an actor-based language using constraint automata /Proceedings of Coordination 2006. LNCS 4038, pp. 281– 297.

14. *Jaghoori M., Movaghar A., Sirjani M. Modere: The model-checking engine of Rebeca* /Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006), Dijon, France, pp. 1810 –1815.

15. *Sirjani M, de Boer F. S., Movaghar A.* Modular verification of a component-based actor language //J.UCS 11(10). 2005, pp. 1695 – 1717.

16. Jaghoori M., Sirjani M., Mousavi M., Movaghar A. Efficient symmetry reduction for an actor-based model /ICDCIT 2005. LNCS 3816, pp. 494 – 507.