ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

ВВЕДЕНИЕ В РАЗРАБОТКУ МНОГОАГЕНТНЫХ СИСТЕМ В СРЕДЕ JASON

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ AgentSpeak

Учебное пособие
для студентов, обучающихся
по специальности 230101

МОСКВА 2009

ББК 32.973.26_018.1
Я7
С50
УДК 004.432(075)
Рецензенты: А.В. Гиглавый
В.В. Костюк

С50 С.С. Смирнов, В.А. Смольянинова Введение в разработку многоагентных систем в среде Jason. Основы программирования на языке AgentSpeak: учебное пособие/ Государственное образовательное учреждение высшего профессионального образования «Московский государственный институт радиотехники, электроники и автоматики (Технический университет)» – М., 2009. – 140с.

В учебном пособии дается начальное представление об агентах и многоагентных системах (МАС), рассказывается о BDI-модели агентов. Излагается методика концептуального анализа поведения агентов, предлагаемая авторами. На базовом уровне рассказывается о возможностях среды Jason как инструментария для разработки и выполнения МАС. Основное внимание уделяется описанию языка программирования агентов AgentSpeak, используемого в среде Jason, и его освоению в объеме, достаточном для выполнения лабораторных работ.

Пособие предназначено для студентов, изучающих МАС в рамках дисциплины «Системы искусственного интеллекта».

Табл. 9. Ил. 36. Библиогр.: 9 назв.

Печатается по решению редакционно-издательского совета университета.

ISBN

© Смирнов С.С.,
Смольянинова В.А., 2009
© МИРЭА, 2009

Введение

Начиная с 90-х годов XX-го века, под влиянием ряда социально-технологических факторов в информатике формируется новый подход к решению научно-практических задач, связанный с разработкой, так называемых, агентов и многоагентных систем (МАС). Что же это за факторы [9]?

1. Возрастающее количество распределенных систем.

Современный микропроцессор можно охарактеризовать как достаточно дешевое и многофункциональное устройство, занимающее мало места в пространстве. В связи с этим появилась возможность встраивать его в большое количество бытовых устройств, чего невозможно было сделать раньше из-за неприемлемых размеров или нецелесообразной дороговизны. Реализация этой возможности привела к появлению у человека потребности организовать сетевое взаимодействие устройств различного назначения, что привело к большому количеству распределенных в пространстве вычислительных систем, которые могут объединяться для решения конкретных задач.

2. Увеличение степени интеграции вычислительных устройств.

Сегодня в рамках одного устройства может интегрироваться множество спецвычислителей, например, домашний компьютер с несколькими аудио и видео картами, многоядерным процессором уже ни для кого не редкость.

Из пунктов 1 и 2 возникает вопрос: как обеспечить эффективность работы программного обеспечения с распределенными, высоко интегрированными системами? Ответом на него является использование многоагентного подхода.

3. Увеличение степени социальной интеграции.

Современные технологии позволяют людям, никогда не встречающимся друг с другом лично, организовывать сообщества по интересам, выполнять общую работу – создавать новое и принимать коллективные решения, что потребовало пункта 4.

4. Увеличение степени интеллектуальности программного обеспечения самого разного назначения.

Программное обеспечение не только решает специфиче-

ские вычислительные задачи, но и становится посредником во взаимодействии между людьми, реализует функции персональных помощников в общении и принятии решений. Ему делегируется все больше полномочий, оно становится представителем интересов личности в информационной среде. В связи с этим появляется необходимость в адекватном этим интересам поведении программного обеспечения в меняющейся среде.

Возникают вопросы: как запрограммировать адекватную модель поведения? Как организовать целесообразное взаимодействие большого количества интеллектуальных программ? Как проверить, что программное обеспечение эффективно решает делегированные ему задачи?

5. Возрастающая ориентация на человека.

По мере развития вычислительной техники и наработки программного обеспечения происходит постепенный уход от машинно-ориентированного программирования к концепциям и метафорам, отражающим человеческое понимание мира. Так мы переходим от программирования путем переключения тумблеров, через ассемблеры, процедурные и объектно-ориентированные языки к языкам разработки агентов и многоагентных систем. Иначе говоря, на каждом последующем этапе развития разработчики пишут программы в терминах все более высокоуровневых абстракций, которые находятся все дальше от машинного кода и все ближе к человеческому языку.

Помимо перечисленных социально-технологических факторов, перед человечеством также стоит ряд практических задач, решение которых без привлечения теории агентов и многоагентных систем обойдется либо слишком дорого, либо попросту невозможно. Примерами подобных задач являются следующие.

1. Моделирование поведения социальных систем с целью проверки теорий, описывающих это поведение:

- для построения планов эвакуации;
- для координации действия спасателей;
- для управления движением;
- для управления различными предприятиями, организациями,

городами, странами;
- для проведения социологических исследований;
- для проведения экономического моделирования рынков;
- для проведения экологических и биологических исследований;
- и т.д.

2. Задачи логистики.
3. Задачи оптимизации поточных линий.
4. Качественный поиск информации в WEB.
5. Ведение военных действий с применением боевых роботов.
6. Автономное управление космическими аппаратами.

Все вышеперечисленное обусловило возникновение многоагентного подхода и обеспечило ему актуальность на многие годы вперед.

В настоящее время множество научно-исследовательских и образовательных учреждений, фирм и промышленных предприятий по всему миру занимается проблематикой МАС. Наиболее заметны в этой области следующие организации [1]:
- университет Карнеги Мэллон (Carnegi Mallon University);
- Массачусетский университет;
- университет г. Болоньи (Univrsita di Bologna);
- ряд университетов и колледжей Великобритании (Stanford University, Manchester Metropolitan University);
- IBM;
- Microsoft;
- DEC;
- Apple;
- Toshiba;
- Hewlett Packard и д.р.

У нас исследования по данной тематике проводятся в:
- Исследовательском центре искусственного интеллекта при Институте программных систем РАН (г. Переславль-Залесский);
- Институте проблем управления РАН;
- Санкт-Петербургском институте информатики и автоматизации РАН;
- Санкт-Петербургском государственном электротехническом

университете;
- Санкт-Петербургском техническом университете;
- Уфимском государственном авиационно-техническом университете;
- Таганрогском радиотехническом университете;
- Вологодском государственном техническом университете;
- МГТУ имени Баумана;
- Московском государственном институте радиотехники, электроники и автоматики.

Многими перечисленными и не перечисленными организациями было разработано большое количество инструментальных средств, сред моделирования и платформ, предназначенных для создания агентов, МАС и обеспечения их работы. Огромный перечень подобных систем можно найти в интернете, среди них такие системы как: Jade, Cougaar, RePast, MASSON, Jason, Aglobe, NetLogo и многие другие. В целом существующее положение с инструментарием разработки МАС можно охарактеризовать как начало стандартизации. Была создана специальная организация – FIPA (www.fipa.org), задачей которой является разработка и внедрение стандартов в области МАС. Пока среди множества уникальных систем в качестве промышленной платформы разработки МАС была признана JADE (Java Application Development Environment) [8] как наиболее соответствующая стандартам, однако мы будем изучать другую систему.

Задачами предлагаемого учебного пособия является знакомство с основными понятиями теории агентов и МАС, с одной из множества методик разработки МАС, а также с языком программирования агентов AgentSpeak и основанной на нем средой разработки и выполнения МАС – Jason.

Система Jason была выбрана из множества по ряду причин. Во-первых, AgentSpeak является уже классическим, но продолжающим свое развитие языком программирования агентов, получившим признание за рубежом. Во-вторых, он декларативный и очень сильно похож на язык Пролог, а это значит, что у него есть все достоинства декларативных языков (в от-

личие от Java, которая обязательна при работе с Jade) и у читателя имеется возможность изучения некоторых вопросов по литературе, посвященной Прологу. В-третьих, Jason в настоящее время считается лучшим интерпретатором AgentSpeak и на начальном уровне освоения не требует владения Java. И, наконец, имеется внушительная монография от разработчиков [2] и множество научных статей, посвященных AgentSpeak и Jason, что существенно облегчает их изучение.

Тем не менее, использование всех возможностей Jason и реализация на нем серьезных систем предполагает, помимо хорошего понимания теории агентов и знания AgentSpeak, также и уверенное владение следующими языками:

- языком коммуникации агентов KQML – для расширения встроенных в AgentSpeak возможностей коммуникации.

- Java – для программирования модели среды взаимодействия агентов, создания пользовательского интерфейса, тонкой настройки механизмов Jason.

В настоящем пособии авторы не ставят перед собой задачу осветить все вопросы, касающиеся разработки МАС в Jason. К сожалению, мы не можем опереться на массовое знание Java нашими студентами, и не будем затрагивать вопросы, относящиеся к программированию на нем, а также не будем требовать использования Java при выполнении лабораторных работ в Jason.

Язык KQML в нашем пособии затрагивается лишь постольку, поскольку он лежит в основе коммуникации между агентами, и его упрощенная версия встроена в AgentSpeak.

В связи с вышесказанным, основной упор делается на знакомство с возможностями языка AgentSpeak и на то, как с его помощью можно разрабатывать МАС, построенные исключительно на информационном обмене между агентами средствами встроенного KQML, т.е. без модели физической среды взаимодействия.

Пособие рассчитано на изучение в течение одного семестра и ориентировано на выполнение лабораторного практикума.

1. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ АГЕНТОВ

1.1. Понятие агента

К сожалению, до сих пор не выработано четкое определение понятия «агент», поскольку каждый исследователь склонен трактовать его немного по-своему. Наиболее общепризнанным определением считается следующее.

Агент – это система, находящаяся в некоторой среде, воспринимающая эту среду посредством рецепторов, и воздействующая на нее посредством эффекторов (см. рис. 1).

Многоагентная система – это система, в которой поставленная задача решается за счет совместной деятельности (кооперации, противоборства) многих агентов в некоторой среде.

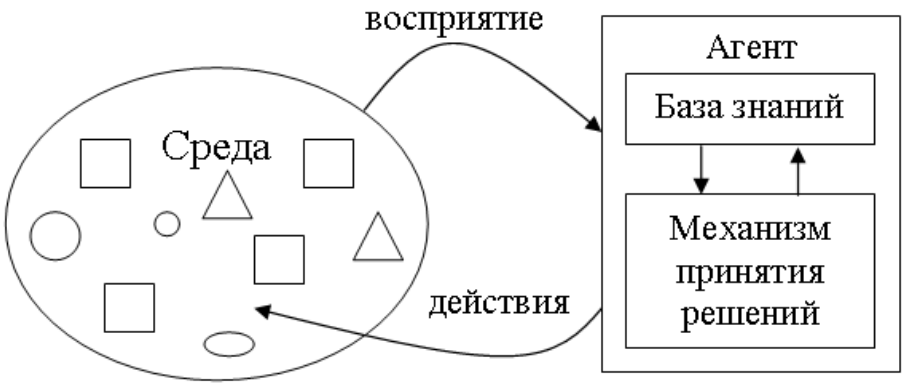

Рис. 1

Считается, что агент должен обладать, по крайней мере, некоторыми из следующих основных свойств [3]:

- Автономность. Агент должен работать без ежесекундного вмешательства и контроля человека.

- Обучаемость. Агент должен обучаться всему, что может освоить для компенсации неполных или неверных исходных знаний.

- Реактивность. Агент должен оперативно реагировать на изменения в среде.

- Проактивность. Помимо оперативных реакций агент должен самостоятельно выбирать себе текущие цели и искать способы их достижения. Для этого агент должен не только уметь распознавать сложившиеся ситуации, но и благоприятные возможности.

- Социальность. Агент должен взаимодействовать (возможно, сотрудничать) с другими агентами в рамках достижения целей при помощи какого-либо языка коммуникации.

- Рациональность. Агент должен быть способен находить рациональные решения задач в соответствии с поставленными критериями.

Также выделяется ряд дополнительных свойств, которыми агенты могут обладать:

- Мобильность. Это способность агента перемещаться по вычислительной сети (или в физической среде).

- Правдивость. Агент специально не сообщает неверную информацию.

- Доброжелательность. Если агенты не имеют конфликтующих целей, то каждый из них может выполнять все, о чем просят другие.

- Эмоциональность. Способность моделировать эмоции, для выражения своего текущего состояния.

Примерами агентов являются:

- компьютерные вирусы;
- интеллектуальные помощники;
- поисковые боты в сети интернет;
- некоторые персонажи компьютерных игр;
- роботы;
- и т.д.

В тоже время агентами не являются:

- объекты в классическом понимании объектно-ориентированного программирования (см. далее);
- отдельно взятые интеллектуальные технологии (например, системы обработки правил, нейронные сети и т.д.);
- мастера, планировщики и т.п.

Среда с точки зрения агента – это все, что вне его и при этом доступно для восприятия и изменения. Среда может состоять из других агентов и пассивных объектов (ресурсов). Среды можно классифицировать по следующим признакам.

1. По степени наблюдаемости: *полностью наблюдаемая* – если рецепторы агента предоставляют ему полную информацию о состоянии среды или *частично наблюдаемая* – в противном случае.

2. По степени определенности последствий принимаемых решений: *детерминированная* – если следующее состояние среды однозначно определяется предыдущим состоянием и реализуемым действием или *стохастическая* – иначе.

3. По связности принимаемых решений: *эпизодическая* – если цель агента правильно реагировать на каждый последовательно воспринимаемый эпизод и последующие решения не зависят от предыдущих – или *последовательная*, если это не так.

4. По динамике происходящих изменений среды подразделяются на: *статические* – если в среде не происходит изменений, пока агент принимает решения, и *динамические* – если изменения происходят.

5. По плавности изменения состояний среды бывают: *дискретные* – если изменения происходят скачкообразно, и *непрерывные* – если изменения происходят плавно.

6. По количеству находящихся в среде агентов - *одноагентные* и *многоагентные*.

Следует заметить, что агенты могут различаться по структуре, определяющей возможности восприятия среды. Поэтому одна и та же среда с точки зрения разных агентов может обладать разными свойствами. Например, мир для лягушки близок к дискретному, т.к. ее глаза приспособлены видеть только объекты, движущиеся со скоростью не меньше пороговой. Объекты, обладающие меньшей скоростью, для нее неподвижны.

1.2. Программные агенты и программные объекты

Те, кто знаком с объектно-ориентированным программированием (ООП), часто не видят никакой разницы между объектами и агентами [3].

Действительно, программные агенты это особый подкласс программных объектов, в которых человек склонен видеть самостоятельных деятелей и переносить на них человеческие качества (такие как наличие целей, обладание способностями и т.д.). Заманчиво думать, что объекты «решают», что им делать в конкретной ситуации и «просят» других сообщить им некоторую информацию.

Однако теория агентов это не подмена терминологии ООП,

приводящая к самообману разработчиков, как может показаться с первого взгляда (см. также п.1.4).

Между понятиями «объект» и «агент» есть как очевидные сходства, так и заметные различия.

Различие первое

Один из основных принципов ООП – инкапсуляция – говорит о том, что только сам объект имеет непосредственный доступ к своим переменным, описывающим его состояние. Иначе говоря, объект единолично владеет своим ***состоянием***, и только сам может его изменять, получив соответствующее сообщение. Однако объект не имеет полного контроля над своим ***поведением.*** Любой публичный метод объекта, если он вызван соответствующим сообщением, будет выполнен. Причем объекты ***обязаны*** иметь публичные методы, иначе из них невозможно будет построить систему, предназначенную для достижения общих для всех объектов целей.

В противовес объектам, агенты ***не обязаны*** иметь общих ***целей***, поскольку могут разрабатываться множеством разнообразных организаций и частных лиц для представления их собственных интересов в некоторой виртуальной среде. Поэтому агент «А» не должен выполнять свои методы только потому, что агент «Б» посылает ему соответствующие сообщения. Выполнение данных методов может и не лежать в области «интересов» агента «А». Иначе говоря, агент имеет полный контроль над своим ***поведением.*** Для организации взаимодействия между агентами, взамен жесткого выполнения методов в ответ на полученное сообщение, используется механизм ***запросов***. Если агенту «А» нужно, чтобы агент «Б» выполнил некоторое действие, то он посылает агенту «Б» запрос на выполнение данного действия, но агент «Б» на основе своего механизма принятия решений ***сам решает***, выполнять данное действие или нет. Для формулирования запросов существуют специальные языки, например KQML.

Естественно, никто не может запретить нам внутри каждого метода реализовать некий механизм принятия решений, который будет определять стоит ли выполнять данный метод или нет. Однако так мы не сможем реализовать вторую отличительную осо-

бенность агентов.

Различие второе

Гибкость автономного поведения агентов, обусловленная такими их свойствами как реактивность, проактивность, социальность и т.д. – является второй отличительной особенностью агентов.

Стандартная технология ООП ничего не может сказать о том, как разработать систему, обладающую столькими интегрированными свойствами. Эти вопросы лежат в области проектирования интеллектуальных систем, которые обладают централизованным или иерархически организованным механизмом логического вывода. И именно благодаря подобному механизму агенты должны решать, что им делать в настоящий момент.

Различие третье

В подлинно многоагентной системе каждый агент имеет собственный поток управления или выполняется на отдельной вычислительной машине. Иначе говоря, агенты работают параллельно, а не в режиме разделения времени. В то время как система объектов обычно имеет общий поток управления.

Конечно, ничто не мешает нам имитировать параллельность работы посредством режима разделения времени, но тогда мы вынуждены будем согласиться и с имитацией автономии. Подобные имитации возникают, когда пытаются смоделировать работу многоагентной системы (и самих агентов и среду) в пределах одного компьютера. Данный подход не требует большого количества аппаратуры, однако существенно возрастают требования к ее качеству, т.е. вычислительной мощи. Например, увеличение количества работающих агентов, обладающих чуть более сложным, чем рефлекторное, поведением, в таких средах как NetLogo Breve быстро сделает модель неповоротливой и сложной для наблюдения.

1.3. Агенты и экспертные системы

Между агентами и экспертными системами можно найти следующие различия [3].

Во-первых, классические экспертные системы, типа MYCIN, взаимодействуют ***только*** с пользователем и получают от него всю необходимую для принятия решений информацию. Иначе говоря, у них нет датчиков, посредством которых они автоматически воспринимают среду функционирования, т.е. среды для них фактически не существует. Это приводит к следующим двум отличиям.

Во-вторых, классические ЭС не способны к реактивному и проактивному поведению.

В-третьих, классические ЭС не способны к самостоятельному взаимодействию с другими ЭС, т.е. кооперации, координации, ведению переговоров и т.д.

В свою очередь, ЭС реального времени во многом очень похожи на агентов и чаще всего отличаются от них отсутствием социального элемента поведения.

1.4. BDI-модель агента

В первую очередь агенты рассматриваются как посредники, делегаты, искусственные представители интересов людей в некоторой виртуальной (или реальной) среде, и только потом это понятие распространяется на других субъектов, например, животных.

Поэтому, когда говорят о поведении агентов и характеризуют их состояние, то употребляют такие термины как:

- убеждения (**B**eliefs);
- желания (**D**esires);
- намерения (**I**ntention).

Иногда к этому списку добавляют «надежды», «опасения» и другие чисто человеческие характеристики. Модель агента, построенная с использованием перечисленных понятий, называется BDI-моделью.

Может показаться, что перенесение подобных терминов на искусственные сущности надуманно. Однако этому есть свое рациональное объяснение.

Во-первых, когда мы описываем поведение людей, мы естественным образом используем эти термины, а поскольку агенты должны заменять людей в некоторой деятельности, следовательно, они должны иметь модели наших убеждений, желаний и про-

чего, закладываемые в них разработчиками. Тогда зачем, спрашивается, между естественно-языковым описанием поведения и его программной моделью вносить лишний терминологический барьер вроде переменных, процедур, методов, модулей и т.д.? Этот барьер заставляет разработчика терять время и силы на то, чтобы решить каким образом лучше всего отразить моделируемое понятие в терминах используемого языка программирования. Поэтому многие агентные языки программирования ориентированы именно на построение программы в терминах BDI-модели.

Во-вторых, устранение терминологического барьера – это не самая главная задача BDI-модели. Главное заключается в том, что программирование в BDI-понятиях обязывает среду разработки обрабатывать их в соответствии с тем содержанием, которое в них вкладывается человеком. Иначе говоря, в среде должен быть механизм логического вывода, который моделирует принятие решений человеком на основе его убеждений, желаний и т.д. Дадим определение основным понятиям BDI-модели.

Убеждения – это декларативные знания, которые считаются истинными с точки зрения данного агента. В связи с этим разные агенты могут иметь разные убеждения относительно одних и тех же вещей. Кроме того, могут быть убеждения относительно убеждений и прочих BDI-элементов. Убеждения, оказавшиеся ложными, должны быть отвергнуты. Таким образом, агент должен действовать исходя из текущих убеждений, но допускать, что они могут быть ложны. Следующие фразы содержат примеры убеждений:

- «Она думала, что на улице холодно» - убеждение о состоянии объекта;
- «Она была убеждена, что он сделает ей предложение» - убеждение о намерениях другого;
- «Она была уверена, что он знает ее подругу» - убеждение об убеждении другого;
- «Он думал, что она хочет в Париж» - убеждение о желаниях другого.

Желания – это цели существования агента, т.е. его целевые состояния. Желания можно понимать как убеждения, которые

нужно сформировать. Деятельность агента обычно обусловлена достижением его собственных целей (удовлетворением желаний), что, однако, не исключает участие агента в достижении целей других агентов.

Намерения – это планы по достижению целей, принятые к исполнению. Они могут и не согласовываться с собственными желаниями агента, а строиться на основе чужих желаний и объективных необходимостей. У агента в настоящий момент может быть множество намерений, среди которых он должен постоянно выбирать, чем заняться в настоящий момент.

Система Jason, предлагаемая читателю для ознакомления, работает с классическим определением BDI-модели, основные понятия которой это убеждения, цели и намерения. Подробнее о реализации BDI-модели в Jason см. раздел 4.4

2. МЕТОДИКА АНАЛИЗА КОНЦЕПТУАЛЬНЫХ СТРУКТУР ДЕЙСТВИЙ В ПРИМЕНЕНИИ К ОПИСАНИЮ ПОВЕДЕНИЯ АГЕНТОВ

В настоящее время существует множество методик разработки МАС, но в большинстве своем они еще не вышли на промышленный уровень и используются в академической среде. Также ни одна из них пока не претендует на право называться окончательным и универсальным подходом. Все они обладают своими достоинствами, которые позволяют им хорошо описывать одни моменты, и недостатками, благодаря которым упускаются из виду другие.

Все известные методики можно классифицировать следующим образом [1]:

- базирующиеся на традиционном объектно-ориентированном проектировании и расширяющие его в сторону агентов;
- базирующиеся на методах инженерии знаний;
- базирующиеся на организационно-ориентированных представлениях;
- сочетающие в себе элементы трех выше названных классов.

В данном пособии мы не будем делать обзор всех этих методик, поскольку это займет много места и будет трудно сделать лучше, чем это уже сделано в работах [1,3].

Авторами пособия предлагается методика концептуального анализа, построенная на основе метода ситуационного анализа, описанного в работах Л.С. Болотовой (Загадской), например, в [4]. Данная методика относится ко второму классу, рассмотрим ее подробнее.

Для начала возьмем за основу определение концептуальной модели предметной области (КМПрО), данное в [5], и немного адаптируем его применительно к агентам.

$$КМПрО = \{A, X, C, V(C), R, G(A), Ru\}, \text{ где:}$$

$A = \{a_1, a_2, \dots a_n\}$ – множество имен субъектов (т.е. агентов), действующих в предметной области (ПрО);

$X = \{x_1, x_2, \dots, x_n\}$ - множество имен объектов (пассивных ресурсов), существующих в ПрО (в среде функционирования агентов), и необходимых агентам для их деятельности , $n \geq 0$;

$C = \{c_1, c_2, \dots c_n\}$ – общее множество имен свойств агентов и объектов, но каждый агент или объект описывается своим набором свойств. Например, для x_i имеем множество имен свойств $c_i = \{c_{i1}, c_{i2}, \dots, c_{im}\}$, $m \geq 0$;

V(C) – множество областей значения свойств:

$$c_{ij} = \{ c_{ij}^1, c_{ij}^2, \dots, c_{ij}^k\},$$

при этом, $k>0$ (в описании объекта может не быть свойств, но если есть хотя бы одно свойство, то у него обязательно есть хотя бы одно значение);

$R = \{r_1, r_2, \dots r_l\}$ - множество имен отношений, в которые могут вступать агенты и объекты (в разных сочетаниях) моделируемой ПрО, $l >= 0$;

$G(A) = \{g_1, g_2, \dots g_v\}$ - множество имен действий (процессов, принимаемых решений), которые агенты могут совершать над объектами и друг другом, $v>0$ (хотя бы одно действие в задачах моделирования всегда есть);

Ru – множество законов среды функционирования агентов (например, модели физических законов).

Объективное состояние ПрО $S_{ПрО}$ представляет собой совокупность всех *фактов*. Под *фактами* понимаются означенные индивидуальные свойства и отношения существующих в данный момент агентов и объектов, отражающие объективное (т.е. не за-

висящее от представлений агентов) положение вещей. Факты представляют собой минимальные фрагменты знаний, из которых может образовываться множество убеждений агента. Также во множество убеждений могут входить субъективные представления агента о состоянии ПрО, которые тоже выражаются как означенные свойства и отношения, но не являются фактами.

Объективное состояние ПрО изменяется под влиянием действий из множества $G(A)$ и определяется в фиксированный момент t_i времени следующим образом:

$$S_{ПрО}\ (t_i) = \{A(t_i),\ X(t_i),\ C(t_i),\ R(t_i)\} \qquad (1)$$

или проще, с использованием понятия «факт»: $S_{ПрО}\ (t_i) = \{f_i\}$.

Субъективное представление агента a_i относительно состояния ПрО ($S_{ПрО}\ (a_i,t_i)$) может частично пересекаться, являться частью или полностью совпадать с объективным состоянием ПрО. Пересечение говорит о том, что помимо истинных представлений о состоянии ПрО у агента также могут быть и ложные представления. Если $S_{ПрО}(a_i,t_i) \subset S_{ПрО}(t_i)$, значит у агента неполное представление о среде. Первые два варианта возможны у агентов, которые функционируют в частично наблюдаемых средах. Полное совпадение характерно только для полностью наблюдаемых сред. В общем случае, чтобы агент мог хоть как-то работать в среде, его представления о ней хоть в чем-то должны соответствовать действительности, поэтому:

$$S_{ПрО}\ (a_i,t_i) \cap S_{ПрО}\ (t_i) \neq 0,$$

а в идеале $S_{ПрО}\ (a_i,t_i) = S_{ПрО}\ (t_i)$.

В независимости от того, насколько $S_{ПрО}(a_i,t_i)$ пересекается с $S_{ПрО}(t_i)$, агенты вынуждены действовать исходя из $S_{ПрО}(a_i,t_i)$, что оказывает свое влияние на различие ожидаемых и получаемых результатов.

Каждому действию из множества $G(A)$ соответствует *состояние-предусловие*, т.е. состояние $S_{ПрО}(a_i,t_i)$, в соответствии с которым данное действие может быть применено. В результате выполнения этого действия изменится $S_{ПрО}(t_i)$, если реализация действия не противоречит законам среды функциониро-

вания агентов. Иначе говоря, реализация действия формирует новые факты, которые в дальнейшем могут стать убеждениями агента. Получившееся новое состояние ПрО является *состоянием-постусловием* данного действия. Однако для недетерминированных сред очередное состояние формируется как сумма воздействий всех агентов, а также независимых от них процессов среды. В этом случае состояние-постусловие с точки зрения конкретного агента – это состояние, возможное с некоторой долей уверенности.

Таким образом, каждое действие представляет собой функциональное отображение одного (или нескольких) состояний в другое (другие). В зависимости от предусловия $S_{ПрО}(a_i,t_i)$ на момент $t_i \in [t_{нач}, t_{кон}]$ агентом выбирается то действие, которое с его точки зрения необходимо для решения текущей задачи. Множество $S_{ПрО}(a_i)=\{S_{ПрО}(a_i,t_j)\}_{j \in 1...n}$ является субъективным представлением о пространстве состояний ПрО с точки зрения агента a_i. Обозначим через F взаимоотображение субъективного пространства состояний агента a_i и множества его действий:

$$F: S_{ПрО}(a_i) \leftrightarrow G(a_i)$$

Задача моделирования состоит в том, чтобы перевести ПрО из объективного начального состояния – $S_н \in S_{ПрО}$ в объективное целевое – $S_ц \in S_{ПрО}$. Процесс решения задачи, таким образом, заключается в том, чтобы определить цепочку действий разных агентов, последовательное (или местами одновременное) применение которых к начальному состоянию ПрО переведет ее в целевое состояние. Схема решения выражается формулой:

$$Z = (S_н \xrightarrow{G(A)} S_ц)$$

А целевое состояние выражением:

$$S_ц = g_i\ (g_n(g_m(\ldots\ldots g_k(\ S_н\ ))))$$

Последовательность действий разных агентов ($g_k,\ldots,g_m,g_n,g_i$) представляет собой алгоритм решения задачи, стоящей перед МАС, и определяет ее целевую линию поведения, реализуемую в дальнейшем планами агентов.

Для определения подхода к разработке баз знаний агентов особенно важно двустороннее отношение ***F*** между пространством действий и пространством состояний (ситуаций). Суть его в том, что ситуации «вызывают», активизируют к исполнению действия разных агентов и сами изменяются при их реализации. То, что должно измениться при переходе от старой ситуации к новой, и есть **содержание действия** агента, описываемого соответствующим понятием из множества *G*.

Анализируя содержание действия, мы можем получить и структуры связанных с ним ситуаций, а, следовательно, выйти на других агентов и объекты с их свойствами и отношениями, участвующими в реализации действия.

Таким образом, действия оказываются основной отправной точкой анализа задачи, что вполне естественно, т.к. конечная цель разработки любого программного продукта – это реализация или моделирование каких-либо процессов.

В соответствии со сказанным, предлагаемая методика концептуального анализа предметной области задачи с целью разработки базы знаний агентов и определения структуры МАС, называется **методикой анализа концептуальных структур действий** [6]. Суть ее в заключается в следующем.

Для каждого действия, которое должны выполнять агенты разных типов или которое должно выполняться средой, строится его **концептуальная структура** (КСД, рис.2). Она включает в себя следующие элементы:

- указание на **субъекты** действия, т.е. агенты, которые должны или могут выполнять данное действие;
- указание на объекты действия, т.е. агенты или объекты, изменение состояние которых является целью данного действия: оно может изменять какие-то свойства и/или отношения агента (объекта);
- указание на компоненты действия, т.е. другие агенты или объекты ПрО, необходимые для описания условий и реализации данного действия;
- указание на свойства и/или отношения субъектов, объектов и компонентов действия, определяющие посредством

своих значений условия его совершения и его результаты, т.е. описывающие ситуации предусловия и постусловия данного действия.

Для изображения КСД применяются следующие графические обозначения.

1. В овалах изображаются объекты, играющие в рамках действия разные роли – субъекты, объекты и компоненты действия. Субъект располагается в вершине КСД, а объект – в основании. При этом субъект и объект действия образуют прямую линию, проходящую через имя действия, которая называется осью КСД, а компоненты располагаются как бы вокруг этой оси. Соблюдение этой прямой обязательно, т.к. благодаря ей КСД можно прочитать: сразу видно, кто делает, что делает, над чем и посредством чего. В противном случае в прочтении КСД возникнет путаница.

2. Имя действия – глагол в неопределенной форме или словосочетание на его основе – изображается в шестиугольнике.

3. Свойства изображаются при помощи прямоугольников.

4. Непоименованные сплошные связи – это ролевые отношения между понятиями, увязывающие их в единую структуру, такие как: «быть объектом действия», «быть субъектом действия», «быть компонентом действия», «быть свойством объекта».

5. Отношения между объектами, раскрывающие семантику действия, (элементы множества R) изображены сплошными дугами или прямыми со стрелками, определяющими направление отношения (стрелка исходит из ***субъекта отношения*** и входит в ***объект отношения***), и точками для задания имени отношения. При наличии между двумя объектами более одного однонаправленного отношения на соответствующей связи ставится необходимое количество точек с названиями отношений.

6. Пунктирные линии изображают отношения между свойствами и/или отношениями, от которых зависят значения других свойств и/или отношений объектов.

7. Для описания ситуаций условий и результатов принято следующее соглашение:

- выражение $a\ \Psi\ b$, где $\Psi \in \{=, \neq, \geq, \leq, >, <\}$, «*a*» – фактор, «*b*» – фактор или константа – описывает некоторое условие;

- выражение $a =/ b$ – описывает результат действия, т.е. фактор «*a*» станет равен значению фактора «*b*» или константе;

- выражение $a\ \Psi\ b\ /\ c$ – говорит о том, что фактор «*a*» принимает участие и в описании условия и в описании результата действия;

- выражение $a\ \Psi\ b\ //\ c$ – описывает ***последействие*** данного действия, т.е. указывает, что значение фактора *«a»* должно измениться на *«c»* после совершения действия. Изменение этого фактора не является основной целью данного действия, но обязательно должно произойти, чтобы действие могло считаться завершенным. Пример: действие «достать еду из холодильника» имеет поддействия «открыть холодильник», «взять еду» и «закрыть холодильник»; последнее выполняется, когда основная цель уже достигнута, *В(Еда, Холодильник) = ложь*, но закрыть холодильник после этого совершенно необходимо!

8. Кроме того, на КСД также могут изображаться свойства и отношения, не изменяющиеся в процессе моделирования данной задачи, а просто фиксирующие дополнительные условия. Подобные факторы помечаются пустыми кружочками и двойными рамочками.

Далее на основании анализа имеющейся КСД происходит переход к описанию действий, которые необходимо выполнить для формирования условий рассматриваемого действия. Эти действия называются ***поддействиями*** данного действия, или его действиями нижнего уровня. Для них строятся свои КСД, раскрывающие их содержание (действия следующего уровня), и так далее до получения действий, которые могут считаться элементарными для данной задачи.

Таким образом, мы имеем строгий алгоритм декомпозиции процессов задачи, согласно которому действия каждого нового уровня обоснованно «вытекают» из действий верхнего уровня. В результате получается связанное пространство действий и решений агентов, описывающее процессы, подлежащие моделированию. Множества агентов, объектов, их свойств

и отношений также выделяются строго в соответствии с содержанием действий.

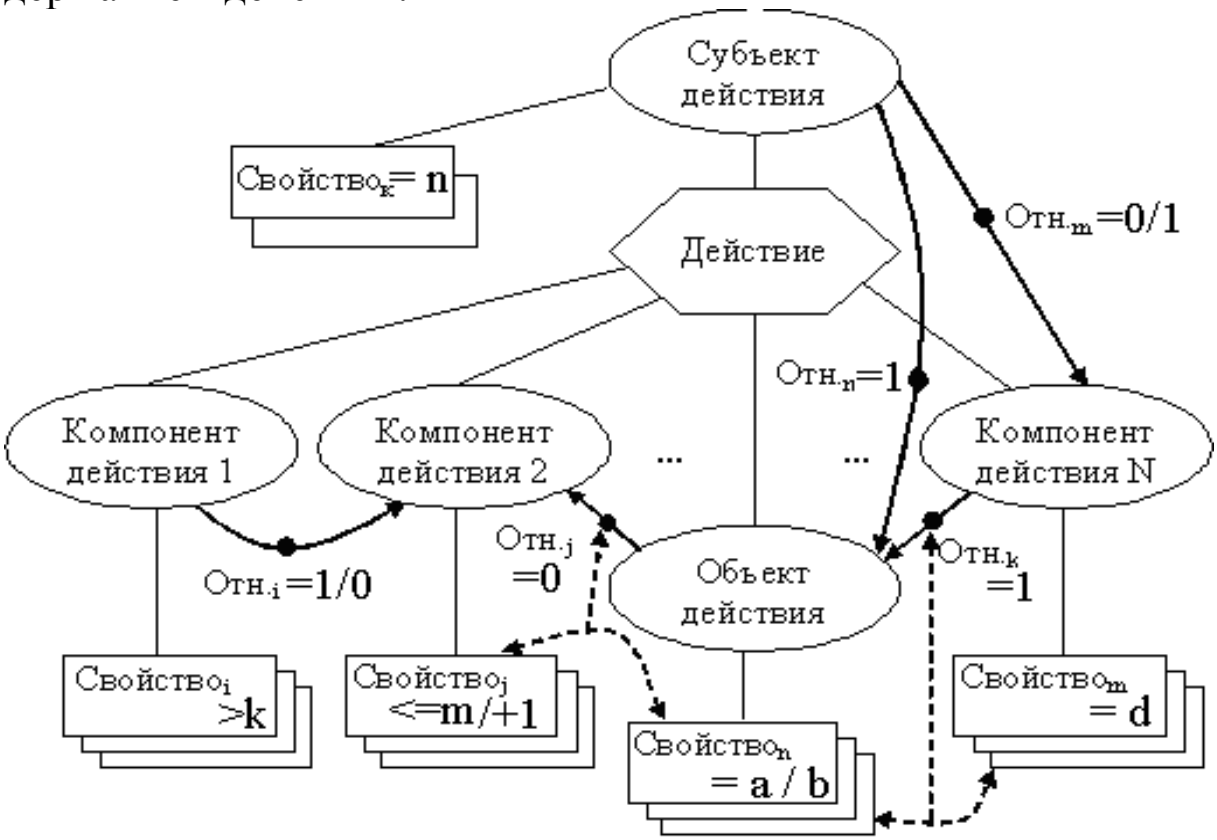

Рис.2.

Из этого следует, что мы получим необходимую начальную информацию об агентах и среде их взаимодействия, позволяющую построить структурную схему МАС и приступить к ее программной реализации. Причем эта информация не избыточна, т.е. не имеет фрагментов, не относящихся к моделируемым процессам.

Методика анализа концептуальных структур действий подразумевает выполнение следующих шагов:

1. Определение действия, соответствующего постановке задачи (в общем случае таких действий может быть несколько и принадлежать они могут разным агентам, тогда они анализируются последовательно).

2. Построение КСД, соответствующей этому действию. Она является корневой схемой, т.е. самой общей (абстрактной), и считается схемой «нулевого» уровня. Построение этой схемы является очень важным моментом методики, так как эта схема показывает, с

чего начать, дает отправную точку для проведения концептуального анализа. На данном шаге начинается формирование множеств агентов и объектов, их свойств и отношений.

3. Анализ условий выполнения данного действия и определение на основе этого его возможных поддействий. На этом шаге начинается выявление и разделение функций и обязанностей агентов, формирование для каждого типа агентов множества возможных подцелей, определение новых процессов, подлежащих моделированию.

4. Последовательный анализ выявляемых поддействий с построением их КСД.

Процесс анализа заканчивается, когда не осталось непроанализированных условий и по каждому условию получен четкий ответ о причинах его формирования, которые могут быть следующими:

- выполнение поддействия;
- на основе данных, поступающих из среды;
- на основе данных, поступающих от пользователя.

Пример применения данной методики разобран в разделе, посвященном выполнению лабораторных работ.

3.ПРЕДСТАВЛЕНИЕ О ЯЗЫКАХ КОММУНИКАЦИИ АГЕНТОВ

Для общения агентов друг с другом с целью обмена знаниями или для организации совместных действий при распределенном решении сложных задач нужны специальные языки.

С 1990 года по инициативе DARPA ведутся разработки стандартных протоколов общения между агентами.

В этом разделе мы кратко познакомимся с двумя из них, практически без изменений вошедшими в современные протоколы [3].

1.Knowledge Query and Manipulation Language (KQML).

Это язык запросов и манипулирования знаниями. KQML обычно позиционируется как, так называемый, «внешний» язык для обмена сообщениями между агентами. Посредством него, как в конверт, «заворачивается» основное содержание сообщения, которое обычно написано на другом языке. Иначе говоря, он

определяет «формат конверта» и ни в коей мере не определяет формат, на котором должно быть написано содержание сообщения, хотя содержание сообщения также может быть написано на KQML.

2.Knowledge Interchange Format (KIF) – это формат обмена знаниями.

KIF представляет собой формат для описания содержательной части сообщения, которое «упаковывается» в «KQML-конверт». Он также может быть использован и как основной язык представления знаний внутри агента.

Рассмотрим чуть подробнее названные языки.

3.1. Язык KIF

Этот язык очень похож на язык логики первого порядка в LISP-нотации. Также он имеет много общего с языком разработки экспертных систем среды CLIPS [7].

Используя KIF, агент может описывать свойства объектов предметной области и отношения между объектами, а также может использовать в этом описании обобщения, относящиеся ко всем объектам. Для этого в словаре KIF определены базовые логические связки и кванторы, а также цифры, символы и зарезервированные слова. Лиспообразная нотация KIF также позволяет обрабатывать списки объектов.

Приведем примеры.

(= (temperature ml) (scalar 83 Celsius))

В данном выражении утверждается, что температура некоего объекта ml составляет 83 градуса Цельсия. Знак равенства – это отношение между двумя понятиями предметной области: свойством и значением свойства. При этом в KIF temperature рассматривается как функция с одним аргументом - ml, а scalar – как функция с двумя аргументами - 83 Celsius. Знак равенства – это стандартный символ KIF, а имена функций temperature и scalar – это уже пользовательское расширение языка, которое должно быть описано специальным образом.

Следующий пример показывает, как может быть введено новое понятие предметной области в терминах уже определен-

ных понятий. В нем утверждается, что объект является холостяком (bachelor), если этот объект мужчина и он не женат.

```
(defrelation bachelor (?x) :=
        (and (man ?x)
            (not (married ?x)
            )
        )
)
```

Как можно заметить, данный язык обладает довольно сложным синтаксисом благодаря большому количеству скобок. Здесь «?x» – это имя переменной, обозначающей некий объект. Символ «:=» - символ определения. Man и married – соответственно классификация и свойство объекта «?x», записанные в предикатной форме.

Следующий пример показывает, как в KIF может быть описано отношение «подкласс-класс».

```
(defrelation person (?x) :=> (mammal ?x))
```

В примере говорится, что всякий объект, который является человеком, является и млекопитающим.

3.2. Язык KQML

Сообщение на языке KQML можно понимать как программный объект, в котором определен императив (т.е. глагол, описывающий совершаемую посредством данного сообщения операцию) и ряд его параметров (пар вида атрибут/значение) [3].

Вот пример KQML-сообщения:

```
(ask-one
    :content (PRICE IBM ?price)
    :receiver stock-server
    :language KIF
    :ontology NYSE-TICKS
)
```

Смысл этого сообщения следующий: некий агент-отправитель спрашивает агента-получателя (stock-server) о цене на какой-то товар от IBM.

Разберем структуру данного сообщения.

«ask-one» - это императив (зарезервированное слово), кото-

рый используется агентом, чтобы задать вопрос другому агенту, на который требуется только один из возможных ответов.

Другие поля сообщения, начинающиеся через «:» - это атрибуты сообщения.

Поле **«:content»** - описывает содержание сообщение, представленное в данном случае в формате KIF.

«:receiver» - имя получателя сообщения.

«:language» - название языка, на котором написано сообщение (получатель должен его понимать и вообще-то это необязательно должен быть KIF).

«:ontology» - определяет имя словаря терминов, используемых в сообщении (например, онтологическое расширение языка KIF).

Также в сообщении могут содержаться и другие атрибуты:

«:force» - определяет, возможен ли в будущем случай, что отправитель сообщения будет отрицать содержание этого сообщения.

«:reply-with» - определяет метку сообщения, которая нужна, если отправитель предполагает дожидаться ответа. Метка нужна, чтобы отправитель смог в будущем определить, на какой его запрос был получен ответ. Это необходимо, когда отправитель ведет параллельно много переговоров с одним или многими абонентами на разные темы.

«:in-reply-to» - ссылка на метку сообщения, которая показывает, на какое сообщение отвечает отправитель.

«:sender» - определяет отправителя сообщения.

Что касается множества существующих на сегодняшний день императивов, то их более 40 и в рамках данного пособия не имеет смысла перечислять их все, упомянем в ознакомительных целях лишь некоторые из них.

ask-about – отправитель хочет получить из базы знаний получателя всю имеющуюся информацию о чем-либо и ожидает, что ответ будет представлен одним сообщением.

stream-about – то же самое, что и ask-about, только ответных сообщений от получателя может быть много (поток), но все они должны касаться одной темы.

eos – императив, обозначающий конец ответного потока сообщений на полученный ранее запрос stream-about

ask-all – отправитель хочет получить всю возможную информацию из базы знаний получателя, касающуюся темы запроса.

ask-if – отправитель хочет знать, есть ли в базе знаний получателя ответ на посылаемое сообщение.

advertise – отправитель информирует получателя, что может выполнить императив, содержащийся в поле content.

break – отправитель хочет прекратить диалог с получателем.

broadcast – отправитель просит получателя отправить данное сообщение по всем имеющимся у получателя каналам.

broker-one – отправитель просит получателя помочь ему обработать другой императив, содержащийся в поле content.

delete-all – отправитель просит получателя удалить из базы знаний получателя всю информацию, совпадающую с заданным образцом.

recommend-all – отправитель хочет узнать у получателя имена всех агентов, которые могут ответить на заданное сообщение.

subscribe – отправитель просит получателя подписать его на рассылку сообщений заданного содержания.

sorry – отправитель не может предоставить получателю больше информации, чем уже сообщил.

tell – отправитель сообщает получателю фрагмент своей базы знаний, касающейся некоего запроса.

untell – отправитель сообщает получателю, что рассматриваемый фрагмент не содержится в базе знаний отправителя.

И т.д.

Перечисленное множество императивов дает представление о выразительной мощности языка KQML.

Приведем пример диалога.

```
Dialogue (c)
(advertise
   :sender A
   :language KQML
   :ontology K10
   :content
      (subscribe
        :language KQML
```

```
        :ontology K10
        :content
            (stream-about
                :language KIF
                :ontology motors
                :content ml)
      )
)
(subscribe
    :sender B
    :receiver A
    :reply-with si
    :content
      (stream-about
          :language KIF
          :ontology motors
          :content ml
      )
)
(tell
    :sender A
    :receiver B
    :in-reply-to si
    :content (= (torque ml) (scalar 12 kgf))
)
(tell
    :sender A
    :receiver B
    :in-reply-to si
    :content (= (status ml) normal)
)
(untell
   :sender A
   :receiver B
   :in-reply-to si
   :content (= (torque ml) (scalar 12 kgf))
```

```
)
(tell
   :sender A
   :receiver B
   :in-reply-to si
   :content (= (torque ml) (scalar 15 kgf))
)
(eos
  :sender A
  :receiver B
  :in-reply-to si
)
```

Диалог начинается, с того, что агент А императивом **advertise** информирует агента В, что может подписать его на собственную рассылку (поток сообщений), касающуюся мотора m1. Проще говоря, агент А предлагает агенту В в будущем получать от себя сообщения по данной тематике. Далее агент В императивом **subscribe** соглашается с предложением агента А и просит подписать себя на эту рассылку. Далее следует 2 сообщения с императивом **tell** от агента А относительно описания мотора m1. Потом описание мотора m1 изменилось по каким-то внешним причинам, и информация, касающаяся крутящего момента, стала неверна. Агент А сообщает об этом агенту В посредством императива **untell**, после чего императивом **tell** сообщает новую информацию о крутящем моменте. На этом текущий сеанс рассылки завершен, о чем свидетельствует следующий далее императив **eos.**

Из приведенного здесь примера можно почерпнуть представление о диалоге агентов при помощи протоколов KIF и KQML. Как будет видно в дальнейшем, в Jason реализована надстройка над этими протоколами, облегчающая работу с наиболее часто используемыми императивами, однако не запрещающая использование KQML в его естественной форме.

4. ЗНАКОМСТВО СО СРЕДОЙ JASON

4.1. Что такое Jason?

Jason [2] – это перспективное инструментальное средство для создания МАС на основе комбинации двух языков – расширенного AgentSpeak и Java, разработанное сотрудниками Австралийского технологического университета.

Первая версия Jason вышла в 2003г и к настоящему времени претерпела ряд улучшений. Аббревиатура «Jason» расшифровывается как «**J**ava-based **A**gentSpeak interpreter used with **S**ACI for multi-agent distribution **o**ver the **n**et». Как следует из этого названия, первоначально Jason рассматривался только как средство разработки МАС на основе архитектуры SACI, позволяющего распределенным агентам взаимодействовать по сети. Но к настоящему времени он уже способен моделировать работу МАС в пределах одного компьютера, а также взаимодействовать с другими МАС, построенными на базе промышленной платформы JADE [8].

Сам Jason также написан на Java и распространяется бесплатно под лицензией LGPL. Скачать Jason можно с сайта разработчиков по адресу http://jason.sourceforge.net/. Для своей работы Jason требует JDK версии не ниже 5.

AgentSpeak – это язык описания логики работы агентов и взаимодействий между ними, ориентированный на программирование BDI-модели. В целом, AgentSpeak представляет собой агентно-ориентированную версию языков логического программирования и достаточно похож на Prolog. Для обеспечения коммуникации между агентами в AgentSpeak встроена поддержка KQML.

Java в Jason используется для программирования моделей специфических сред функционирования агентов, разработки пользовательских интерфейсов МАС и модификации стандартной работы самого Jason.

Фактически Jason – это интерпретатор AgentSpeak, обладающий возможностью расширения функциональности за счет открытости и использования Java.

Система Jason поставляется как надстройка над текстовым редактором Jedit или расширение для среды разработки Eclipse. В комплект поставки, ориентированный на Jedit входят:

- собственно редактор Jedit, с уже подключенным к нему Jason-ом в качестве надстройки;

- исходные программные коды Jason;
- библиотека примеров;
- документация разработчика.

JDK в поставку не входит, но свободно может быть скачен с сайта производителя http://java.sun.com/.

Несмотря на свою привязку к Java, Jason – это в первую очередь интерпретатор AgentSpeak, поэтому для начала работы с Jason знание Java не обязательно. Однако, следует помнить, что уверенное знание Java откроет перед разработчиком МАС в Jason широкие горизонты возможностей. Тем не менее, для определенного класса задач и уровня требований к представлению результатов моделирования, вполне хватает возможностей одного AgentSpeak.

Программа МАС на языке AgentSpeak в среде Jason состоит из проектного файла и файлов с программами агентов. Запущенная на выполнение МАС работает циклично, до тех пор, пока не будет остановлена пользователем вручную, самими агентами или интерпретатором по причине ошибки.

К сожалению, Jason пока не умеет работать с русским языком и не позволяет писать по-русски даже комментарии.

4.2. Установка и настройка Jason

Приведенная в данном разделе информация взята из программной документации (файл Jason-1.3\doc\mini-tutorial\getting-started\index.html) и дополнена комментариями от авторов пособия.

Инсталляция Jason не требует никаких особых навыков, достаточно просто распаковать среду в предназначенную для нее папку. Далее, в зависимости от используемой Вами операционной системы, Jason запускается одним из следующих способов:

- для Windows - файл Jason.exe;
- для MacOS - файл Jason.app;
- для Linux – файл bin/jason.sh.

После запуска открывается окно, показанное на рис. 3. Как Вы понимаете, реальный вид окна будет зависеть от используемой операционной системы, точнее от компонентов,

отвечающих за изображение пользовательского интерфейса. Поэтому окно, которое Вы увидите, может быть и другим.

Рис. 3

Большую часть окна занимают элементы управления редактора Jedit - это панель быстрого доступа и верхняя текстовая область для кодов программ агентов. Элементы, относящиеся к Jason, расположены в нижней половине. Справа от центра – панель с кнопками управления проектом (слева направо): запуск МАС, запуск МАС в режиме отладки, остановка работы МАС, открытие проекта, создание нового проекта, добавление новых агентов в проект, создание среды функционирования агентов, создание нового внутренней функции, очистка информационной консоли. В правом нижнем углу находится списковая рамка, куда будут помещены имена созданных агентов.

При первом запуске, скорее всего, потребуется сделать элементарные настройки, а именно указать путь к местонахождению JDK. Для этого следует вызвать команду меню «Plugins -> Plugins

Options -> Jason». В появившемся окне (рис.4) установить правильный путь к JDK в разделе «Java HOME».

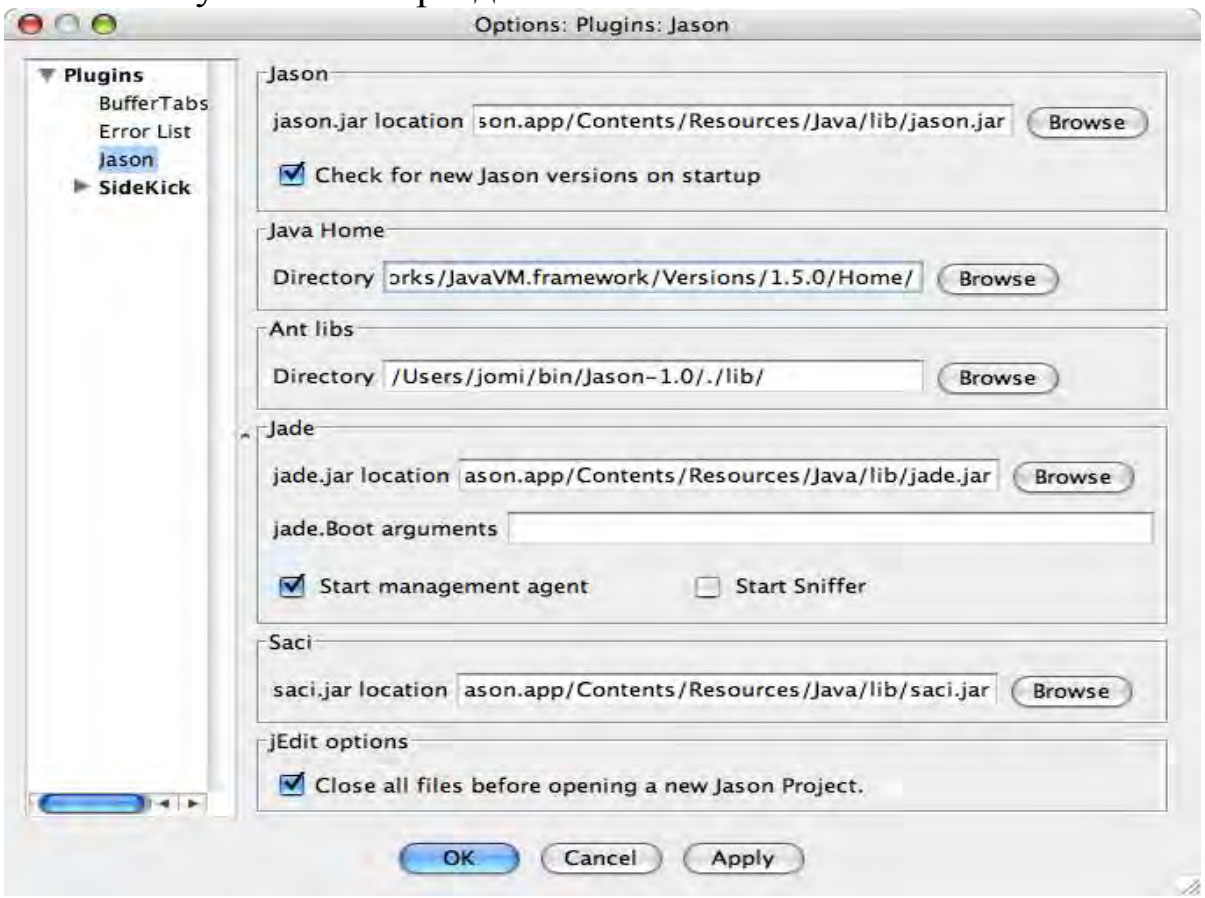

Рис. 4

Теперь Jason готов к работе.

4.3 Запуск демонстрационных примеров

Jason поставляется с библиотекой примеров, которые условно разделены на 2 категории:

- примеры, демонстрирующие отдельные возможности и приемы программирования, расположены в каталоге «demos»;
- примеры, представляющие собой простые МАС, расположены в каталоге «examples».

Чтобы открыть пример, нужно найти в соответствующем каталоге файл проекта, который имеет расширение «.mas2j».

Для запуска проекта необходимо нажать привычную многим кнопку со значком «Play».

Мы не будем заниматься разбором демонстрационных приме-

ров, поскольку есть раздел «Выполнение лабораторных работ», в котором очень подробно разобрана достаточно большая программа.

4.4 Основы программирования на языке AgentSpeak

Проект МАС в Jason состоит из файла проекта, имеющего расширение «.mas2j», файлов с программами агентов (минимальное количество агентов – один), которые должны иметь расширение «.asl», файлов модели среды и пользовательского интерфейса (не обязательно), написанных на Java. Организация файла проекта упрощенно будет рассмотрена в разделе 5.2.5.6.

Программа каждого агента пишется на языке AgentSpeak и структурно состоит из трех частей, относительное расположение которых фиксировано:

- изначальные убеждения и правила;
- изначальные цели;
- планы достижения целей.

Рассмотрим синтаксические конструкции языка, позволяющие описать все эти составляющие программы агента.

4.4.1 Убеждения

Убеждения в Jason формализуются в виде высказываний, предикатов или их строгих отрицаний, а также в виде правил, устанавливающих функциональную зависимость одних выражений от других. Убеждения хранятся в базе убеждений агента, описывают декларативную составляющую его знаний, и могут меняться в процессе его работы.

Предикатные символы и высказывания ***всегда*** обозначаются, начиная ***с маленькой буквы***, а термы – в зависимости от их типа:

- переменные – обозначаются с заглавной буквы;
- числовые константы – естественно, цифрами;
- индивидные константы, функциональные переменные и функциональные константы – обозначаются, начиная с маленькой буквы;
- строки – в двойных кавычках;
- списки – в квадратных скобках через запятую.

Как и в Прологе, для обозначения неопределенной переменной

(т.е. переменной, конкретизация которой несущественна в некотором случае), используется ***символ «_»*** (нижнее подчеркивание).

Изначальные убеждения агента записываются в самом начале его программы и ***разделяются точкой***. При этом ***сначала*** перечисляются высказывания и предикаты, а ***потом*** приводятся правила на их основе.

В синтаксисе правил, помимо привычных символов, применяемых для записи предикатов и математических операций, используется ***символ «:-»***, отделяющий посылку от заключения. При этом посылка расположена справа от этого символа, а заключение слева.

Приведем примеры простых убеждений.

a) started. – убеждение в форме высказывания.

b) factorial(0,1). – убеждение-предикат, утверждающее, что факториал нуля равен единице.

c) money(25). – предикат с числовой константой, описывающий некую наличность.

d) inLove(bob, lucy). – предикат, утверждающий, что Боб влюблен в Люси. При этом bob и lucy – индивидные константы.

e) inLove(bob, X). – предикат, утверждающий, что Боб в кого-то влюблен. X – предметная переменная.

f) names(“Bob”, “Lucy”). – предикат со строковыми термами.

g) p(q(15)). – предикат с термом функциональной константой.

h) k(m(N,10)). – предикат с функциональной переменной в качестве терма.

i) inLove(_, _). – предикат с неопределенными переменными.

j) list([a,b,2,“this is a string”]). – предикат с термом-списком, содержащим разнородные элементы.

Приведем примеры правил.

a) p(X,Y):-q(X,Z)&d(Z,Y). – предикат p(X,Y) станет убеждением, если имеются 2 убеждения q(X,Z) и d(Z,Y) с общим термом Z.

b) clear(X) :- not(on(Y,X)). – предикат clear(X) станет убеждением, если отсутствует убеждение on(Y,X).

c) price(X) :- .random(R) & X = (10*R)+100. – переменная X в предикате price(X) вычисляется случайным образом. Здесь «.random(R)» вызов внутренней функции получения случайного

числа (см. справочник). После ее срабатывания происходит вычисление выражения «X = (10*R)+100» и конкретизация X в price(X).

Заключения правил вычисляются не сразу при запуске программы агента, а ***только*** при обращении к ним.

4.4.2 Типы данных

При рассмотрении убеждений в предыдущем разделе мы для большей ясности использовали привычную для нас терминологию, освоенную в рамках исчисления предикатов. Однако, терминология авторов Jason немного отличается от нашей, что нашло отражение в названиях некоторых функций языка. Покажем эти различия на рис. 5.

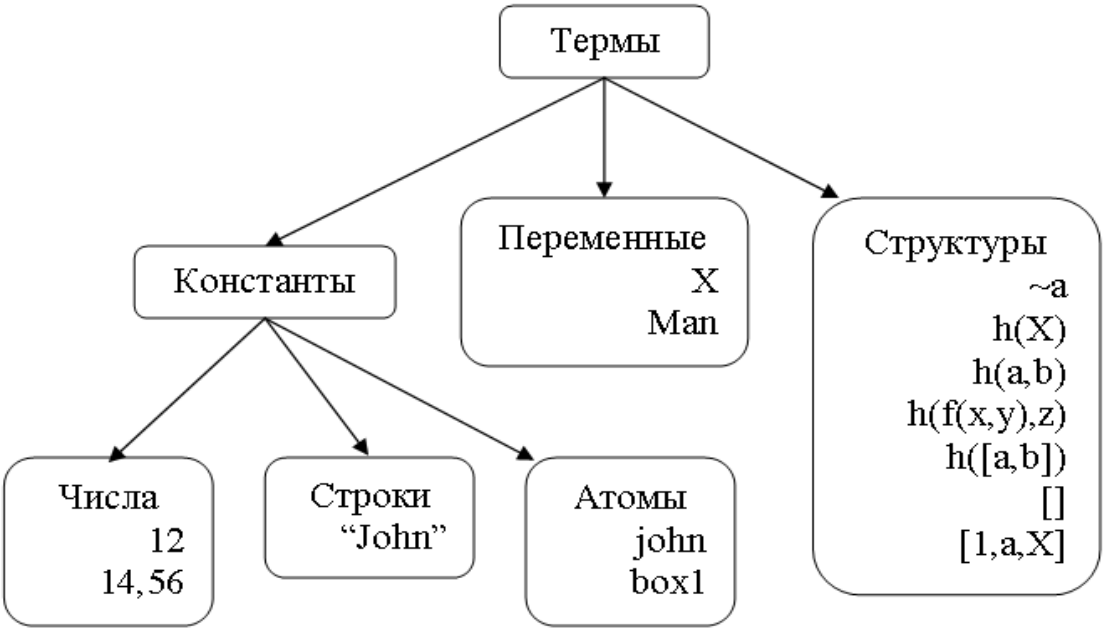

Рис. 5

Как видно из рисунка, индивидные константы названы атомами, а функциональные переменные, функциональные константы и списки подпадают под структурный тип данных.

4.4.3 Сильное и слабое отрицание

В программах на AgentSpeak можно использовать логическое отрицание двух видов:

- сильное отрицание (обозначается символом тильда «~»);

- слабое отрицание (обозначается как **«not»**).

Понятие сильного отрицания подразумевает абсолютную убежденность в ложности некоторой информации, т.е. в базе убеждений агента содержится отрицание факта, например:

~a.

~on(X,Y).

~inLove(bob, helen).

Понятие слабого отрицания допускает истинность информации, просто нам об этом наверняка неизвестно. Иначе говоря, в базе убеждений агента отсутствует эта информация, например:

not a – в базе убеждений отсутствует утверждение «a»,

not p(X,Y) - в базе убеждений отсутствует p(X,Y).

Таким образом, комбинация операторов «~» и «not» дает нам следующие четыре вида выражений (см. табл. 1).

Таблица 1

Обозначение	Расшифровка
F	F содержится в базе убеждений агента (т.е. агент убежден, что F истина)
~F	~F содержится в базе убеждений агента (т.е. агент убежден, что F ложно)
not F	F не содержится в базе убеждений агента (т.е. агент ничего не знает про F)
not ~F	~ F не содержится в базе убеждений агента (т.е. агент ничего не знает про F)

4.4.4 Цели

Цели подразделяются на ***достигаемые*** и ***цели-проверки.***

Достигаемые цели – это желаемые убеждения, для формирования которых в обязательном порядке должен существовать план.

Цели-проверки – это цели, заключающиеся в определении значений термов заданных убеждений для дальнейших операций над ними в теле плана.

Достигаемые цели образуются от имен убеждений, путем добавления знака «**!**», который ставится спереди.

Например.

!ended. – цель – формирование убеждения ended.

!print_fact(5). – цель – формирование убеждения print_fact(5).

Достигаемые цели могут присутствовать в базе убеждений изначально, а могут ставиться в планах в качестве подцелей (подзадач). При вызове подзадачи в теле плана у разработчика есть возможность повлиять на расположение операций плана в стеках намерений (см. п.). По-умолчанию операции подзадачи располагаются в стеке намерений основной задачи. Однако иногда полезно (особенно в рекурсивных планах) расположить операции подзадачи в отдельном стеке. Для этого вместо префикса «!» при вызове подзадачи следует использовать префикс «!!».

Если агент при запуске программы должен проявлять какую-то активность, а не ждать внешних событий, то в его базе убеждений должна присутствовать, хотя бы, одна изначальная цель.

Цели-проверки также образуются от имен убеждений, но с использованием знака «**?**».

Например.

?money(Y). – в базе убеждений будет найден «наиболее свежий» предикат money(Y) и произойдет конкретизация переменной Y.

?inLove(bob, X). – аналогично, в базе убеждений будет найден предикат inLove(bob, X) и произойдет конкретизация переменной X.

?p(X,_,Z). - в базе убеждений будет найден предикат p(X,_,Z) и произойдет конкретизация первого и третьего термов.

?p(_,_). – цель не достижима, т.к. нет свободных переменных для конкретизации. Обработка цели приведет к ошибке.

?p(a,b). – цель не достижима, т.к. нет свободных переменных для конкретизации. Обработка цели приведет к ошибке.

4.4.5 Планы

Планы описывают процедурную составляющую знаний агента. План состоит из четырех частей:

- метки плана (необязательна);
- события активации плана;
- контекстных ограничений (могут отсутствовать);

- содержания плана (может отсутствовать).

Эти части разделяются специальными символами-маркерами, как будет показано ниже.

Метка плана позволяет задать плану уникальный идентификатор. При помощи метки можно выполнять над планом различные операции (см. справочник в прил.1), а также определять дополнительные свойства планов (см. п. 4.4.7).

Событие активации обычно располагается на следующей строке после метки или отделяется от нее пробелом.

Имя метки начинается с символа «@», после которого должна следовать маленькая буква, а далее – любые латинские буквы, цифры. Приведем примеры меток.

@p1 – правильная метка.

@P1 – неправильная метка.

@mm12CV – правильная метка.

@12pm – неправильная метка.

Событием активации плана может быть любое изменение, происходящее с убеждениями и целями. Например, события порождаются, когда:

- происходит добавление или удаление убеждений;
- происходит возникновение новых или отказ от прежних целей;
- запрашивается несуществующая в базе убеждений информация;
- нарушается процесс выполнения плана.

Имена событий начинаются со знаков «+» и «-» и образуются от имен убеждений и целей как показано в табл. 2.

Таблица 2

Обозначение события	Пояснение
+k	Событие добавления убеждения k
-k	Событие удаления убеждения k
+!k	Событие возникновения цели !k
-!k	Событие отказа от достижения цели !k (по причине ошибки в плане или штатным образом)
+?k	Событие возникновения цели-проверки

-?k	Событие отказа от проверки ?k (по причине ошибки в плане или штатным образом)

Рассмотрим подробнее три последних события, указанных в таблице. Допустим, у нас имеется план, обслуживающий достигаемую цель «!p», в теле которого присутствует цель-проверка «?f(X)». Когда интерпретатор Jason переходит к обработке цели-проверки, то вначале он пытается достичь ее путем считывания заданных термов из базы убеждений.

Если в базе убеждений в явном виде нет необходимого для этого убеждения, то он пытается вывести его из имеющихся правил.

Если подходящие правила не будут найдены, то генерируется событие «+?f(X)» в надежде на то, что найдется план, вычисляющий значение терма X.

Если такой план будет найден, но должным образом не завершит свою работу, например, по причине ошибки, то генерируется событие отказа от цели-проверки «-?f(X)».

Если плана для обработки «+?f(X)» не найдено, то происходит отказ от достижения цели «!p» и генерируется событие «-!p».

События «-?f(X)» и «-!p» также могут быть перехвачены и обработаны соответствующими планами. Посредством подобных планов обычно реализуется обработка исключительных ситуаций.

Контекстные ограничения позволяют описывать дополнительные условия активации планов. Таким образом, при возникновении одного и того же события в ***разных условиях*** будут активироваться ***разные*** планы. Контекстные ограничения представляют собой логические выражения, составленные из оценок убеждений и логических функций Jason. Вместе с событием контекстные ограничения образуют ***заголовок плана.***

Термы-переменные, конкретизированные в контекстных ограничениях, сохраняют свое значение и в теле плана.

Приведем примеры контекстных ограничений в отрыве от остальных частей плана:

«p(X)&q(Y)&X>Y» – ограничения вычисляются слева направо, поэтому сначала будет конкретизирована переменная X, потом Y, и, наконец, они будут сравнены.

«onBox & not haveBanana» - ограничение, составленное из двух убеждений в форме высказываний. Используется слабое отрицание для убеждения «haveBanana».

«.count(inLove(bob,_), N)&N>3» - в ограничении использована внутренняя функция подсчета количества убеждений заданной структуры.

Тело плана – это последовательность операций, реализующих предназначение плана:
- математических операций;
- вызовов внутренних функций Jason;
- действий, направленных на объекты среды функционирования МАС;
- коммуникационных актов.

Отдельные выражения в теле плана отделяются ***точкой с запятой***, а в конце плана ставится ***точка.***

Обобщенная синтаксическая конструкция плана выглядит следующим образом.

«Метка

Имя события : Контекстные ограничения <– Тело плана.»

Обратите внимание на то, что между именем события и контекстными ограничениями обязательно ***ставится двоеточие***, а между условием активации и содержанием ***ставится стрелка влево «<– »***.

Возможны также следующие варианты структуры плана.

1. Без контекстных ограничений.

«Имя события <– Тело плана.»

2. Без контекстных ограничений и тела плана.

«Имя события.» или ***«Имя события<-true***.»

3. Без тела плана, но с контекстными ограничениями.

Имя события : Контекстные ограничения

В первом случае план может быть принят к исполнению как только зафиксировано отслеживаемое событие, без всяких дополнительных условий.

Во втором и третьем случае событие связано с целью, которая считается достигнутой автоматически и никаких конкретных операций выполнять не нужно. Во втором случае – как только

произошло событие выдвижения цели, а в третьем – при соблюдении дополнительных условий.

Рассмотрим синтаксис описания планов на примерах.

Пример №1

Напишем классическую программу «Hello World».

```
started.
+started <- .print("Hello World!"). // My first program
```

Здесь «started.» - начальное убеждение. При запуске программы по всем начальным убеждениям и целям генерируются события добавления, приводящие к активации планов.
«+started» - отслеживаемое нашим единственным планом событие.

Контекстные ограничения в данном примере отсутствуют.

Тело плана состоит из одной строки – это «.print("Hello World!")» - вызов внутренней функции среды Jason, предназначенной для печати текстовых сообщений в специальном окне.

Обратите внимание, что вызов внутренних функций всегда ***начинается с точки***, стоящей перед именем функции (перечень внутренних функций смотри в прил.1).

В конце плана, после символов «//» даны комментарии. Если комментарии превышают по объему одну строчку, то соответствующих фрагмент можно ограничить двумя парами символов «/*» и «*/»

Пример №2

Допустим, требуется вычислить факториал числа 5.

```
fact(0,1).
+fact(X,Y): X < 5
        <- +fact(X+1, (X+1)*Y).
+fact(X,Y): X == 5
        <- .print("fact 5 == ", Y).
```

Теперь у нас два разных плана, которые активируются при возникновении одного и того же убеждения fact(X,Y), но в разных контекстных условиях - «X < 5» и «X == 5». В данном примере ***следует обратить внимание*** на операцию ***добавления нового убеждения***, использованную в теле первого плана.

Эта операция обозначается символом «+» перед именем добавляемого убеждения. Ее выполнение на следующем цикле работы агента приведет к возникновению одноименного события, которое активирует один из планов.

Также при написании планов могут быть использованы операции удаления убеждений (обозначается символом «-»), и замены убеждений (обозначается комбинацией «-+»).

В примере показано рекурсивное выполнение планов по событиям добавления новых убеждений пока соблюдаются контекстные ограничения.

Первый план можно перевести на русский язык следующим образом: *«Если возникло убеждение, что факториал X равен Y и выполняется ограничение, что X<5, то следует добавить в базу убеждений тот факт, что факториал X+1 равен (X+1)*Y».*

Обратите внимание, что событие «+fact(X,Y)» является как бы шаблоном для описания множества событий, возникающих, если переменным X и Y приписать конкретные значения.

Программа будет работать следующим образом. Благодаря наличию начального убеждения «fact(0,1).» и первому плану, у агента будут последовательно формироваться следующие убеждения:

fact(1,1).
fact(2,2).
fact(3,6).
fact(4,24).
fact(5,120).

При возникновении последнего убеждения первый план уже не активируется, зато активируется второй план, который выведет результат вычисления факториала.

Данный пример не может считаться программой, написанной в хорошем стиле, поскольку в процессе ее работы происходит порождение множества промежуточных убеждений, занимающих память. Этого на самом деле можно было бы избежать. Как это сделать рассмотрим на следующем примере, в котором дается представление о работе с целями.

Пример №3

```
!print_fact(5).                                  //  выражение №1
+!print_fact(N)
                <- !fact(N,F);                   //  выражение №2
                   .print("Factorial of ", N, " is ", F).
+!fact(N,1): N == 0.                            // выражение №3
+!fact(N,F): N > 0
      <- !fact(N-1,F1);                         // выражение №4
         F = F1 * N.
```

Изначальная цель данного агента напечатать факториал 5, что формализовано как выражение №1. Изначальная цель активирует первый план, в котором вызывается достижение подцели, помеченной как выражение №2. Эта подцель говорит о том, что прежде чем выводить на печать значение факториала его нужно еще рассчитать. Выражение №3 представляет собой план с пустым телом. Он говорит о том, что когда возникает цель подсчитать факториал 0, то следует считать его равным единице и больше ничего делать не нужно.

Третий план срабатывает, когда требуется подсчитать факториал числа N>0 и говорит о том, что сначала надо достичь подцели, рассчитав факториал N-1 и поместив его значение в переменную F1, а далее следует рассчитать факториал N умножением F1 на N.

Выражение №4 выполняется рекурсивно.

Итак, мы видим, что изначальная цель активирует первый план, который запускает рекурсивные вычисления. Построение стека рекурсивно вычисляемых выражений продолжается в третьем плане, который постепенно уменьшает N, доводя его до нуля. Как только N становится равно 0, срабатывает второй план, начинается выборка элементов стека и накопление значения факториала. Последним выражением в стеке будет печать результата из первого плана.

Эффективность в данном примере достигается за счет работы со стеком целей, что значительно быстрее поиска в базе убеждений, которая представляет собой список.

Пример №4(а)

Приведем примеры на обработку других событий. Напишем

программу, которая будет пытаться подсчитать сумму.

```
a(10).
b(15).
!start.
+!start <- ?a(X);
           ?b(Y);
           ?c(Z);
           .print("Summa = ",X+Y+Z).
```

Мы задали 2 начальных убеждения a(10) и b(15) и установили цель «!start». Рассмотрим тело плана, реализующего достижение этой цели. Первые 3 строчки в теле плана – это цели-проверки. Однако для достижения цели «?c(Z)» в базе убеждений нет информации. Нет также никакого другого способа определения значения Z, нет и планов, обрабатывающих подобные ошибки. Поэтому если мы запустим нашу программу, то получим системное сообщение об ошибке.

Пример №4(b)

Модифицируем предыдущий пример и добавим в него правило для определения «c(Z)».

```
a(10).
b(15).
c(Z):-a(X)&b(Y)&Z=X+Y.
!start.
+!start <- ?a(X);
           ?b(Y);
           ?c(Z);
           .print("Summa = ",X+Y+Z).
```

Добавленное правило утверждает, что Z определяется как сумма X и Y. Теперь наша программа заработает и определит сумму, равную 50.

Пример №4(c)

Результат, полученный в предыдущем примере, можно было получить и другим способом. Удалим правило, но добавим план

для обработки события «+?c(Z)».

```
a(10).
b(15).
!start.
+!start <- ?a(X);
           ?b(Y);
           ?c(Z);
           .print("Summa = ",X+Y+Z).
+?c(Z) <-  ?a(X);
            ?b(Y);
            Z=X+Y.
```

Результат работы программы не изменится.

Пример №4(d)

А теперь спровоцируем ошибку во втором плане предыдущего примера, чтобы этот план не смог выполниться. Поступим уже показанным способом – добавим в план попытку считывания несуществующего убеждения «d(F)».

```
a(10).
b(15).
!start.
+!start <- ?a(X);
           ?b(Y);
           ?c(Z);
           .print("Summa = ",X+Y+Z).
+?c(Z) <-  ?a(X);
            ?b(Y);
            ?d(F);
             Z=X+Y+F.
-?c(Z) <-.print("Error computing Z per d"); Z=X+Y.
```

Теперь программа не сможет подсчитать сумму через обработку события «+?c(Z)». Однако это еще не повод отчаиваться, поскольку генерируется событие отказа от достижения цели-проверки «-?c(Z)», и мы все еще имеем возможность обработать это исключение в качестве последней попытки повлиять на значение Z. Что и было сделано в третьем плане

примера. В результате программа опять пришла к верному результату, но информировала нас, что не смогла получить его через убеждение «d(F)».

Пример №4(е)

Вернемся еще раз к примеру №4(а). Предположим, что нас не интересуют способы вычисления «c(Z)», если его нет в базе убеждений. Тогда единственное, что мы можем еще обработать,- это событие отказа от изначальной достигаемой цели «!start».

```
a(10).
b(15).
!start.
+!start <- ?a(X);
           ?b(Y);
           ?c(Z);
           .print("Summa = ",X+Y+Z).
-!start <-.print("Forbidden plan!").
```

Не найдя всех выше перечисленных способов определить «c(Z)», интерпретатор отказывается от достижения цели «!start» и мы получаем сообщение "Forbidden plan!".

4.4.6 Операторы

В табл. 3 приведены элементарные операторы, употребляемые в планах для вычисления выражений.

Таблица 3

оператор	Назначение
+	Сложение (не путать событием добавления)
-	Вычитание (не путать событием удаления)
*	Умножение
/	Деление
mod	Получение остатка от деления
div	Целочисленное деление
|	Списковый оператор
=	Присвоение
==	Равенство

\==	Не равенство
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
=..	Разложение литерала на составляющие

Последний из перечисленных операторов достаточно специфичен, хотя и имеет аналог в Прологе. Он позволяет разобрать предикатную запись на предикатный символ, список термов и список аннотаций. Случаи его применения не рассматриваются в данном пособии. Интересующиеся могут обратиться к первоисточнику [2, стр. 55].

4.4.7 Аннотации

Одна из важных особенностей системы Jason – это работа с аннотациями**.** С помощью аннотаций можно описывать свойства убеждений и планов как объектов манипуляций. Аннотации делают программу более изящной и позволяют более гибко организовать работу интерпретатора. Однако вся мощь их использования раскрывается в сочетании с Java, поэтому желающим узнать по этому вопросу больше мы рекомендуем обратиться к первоисточнику [2, 34]. В данном пособии мы рассмотрим лишь некоторые варианты использования предопределенных аннотаций.

Аннотация записывается в квадратных скобках непосредственно после убеждения или метки плана. Она может состоять из нескольких выражений, отделяющихся запятой.

Выражение аннотации состоит из оператора и его параметра.

Аннотации для убеждений

1. Аннотация [source(x)]

Утверждает, что источником аннотированного убеждения является «x». Используется наиболее часто. Кроме того, данной аннотацией автоматически снабжаются все убеждения. Аргумент «x» может принимать следующие значения:

- «percept» – если источник убеждения собственное воспри-

ятие среды посредством датчиков;

- «self» – если агент формирует данное убеждение в результате логического вывода;

- «имя агента» – если источник убеждения другой агент.

Аннотация «source» полезна для организации интерфейса между агентами. Каждому начальному убеждению можно приписать такую аннотацию и это будет означать, что убеждение может меняться только при получении сообщений от агентов, указанных в аннотации. Если этого не сделать, то убеждение будет меняться при получении соответствующих сообщений от любых источников. Приведем пример.

likes(john,music) [source(john)]

Агент Джон сказал, что любит музыку. Если мы изначально знаем, что Джон любит музыку, и предварительно внесли это убеждение в базу, то только сам Джон сможет его изменить.

Проверку аннотаций довольно часто приходится включать в условие активации планов, например, так:

+likes(john,music) [source(A)]:A== john <- !presentAGuitar(A).

Что означает, если мы получили убеждение, что Джон любит музыку, и сказал нам об этом сам Джон, то подарить ему гитару. Если у этого убеждения будет другой источник, то план не сработает.

2. Аннотация [degOfCert(x)].

Задает степень уверенности в истинности аннотируемого убеждения. По-умолчанию данная аннотация автоматически не влияет на работу механизма логического вывода. Однако, во-первых, она открывает путь к модернизации этого механизма средствами Java, во-вторых, ее все-таки можно использовать, если организовать ее обработку в планах вручную.

Аннотации для планов

1. Аннотация [atomic]

Данная аннотация запрещает агенту отвлекаться от обработки плана на другие события.

2. Аннотация [all_unifs]

По-умолчанию, если контекстные ограничения плана могут быть удовлетворены, то они удовлетворяются единственным из

возможных способов (на основе наиболее свежей информации), не смотря на то, что в базе убеждений можно найти другие подходящие варианты конкретизации переменных. Используя данную аннотацию, можно попросить план использовать все варианты конкретизации. Например.

a(10).

a(20).

!g.

@p[all_unifs] +!g : a(X) <- print(X).

В результате программа должна вывести два числа.

3. Аннотация [breakpoint]

При вызове плана с этой аннотацией программа должна приостановить свою работу и перейти в режим отладки.

4.4.8 Работа со списками

Работа со списками в AgentSpeak происходит также как и в Прологе. Списки представляются в памяти как бинарные деревья (см. рис. 6)

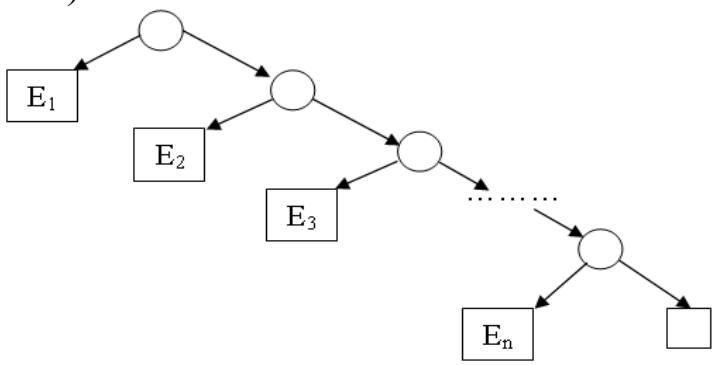

рис. 6

В списке выделяются так называемые голова и хвост. Голова списка – это его первый элемент (допустим E_1), который посредством специального оператора объединения (обозначенного кружочком) соединяется с его хвостом, представляющим собой остальные элементы списка, увязанные операторами объединения. Теперь рассмотрим хвост (начиная с элемента E_2). Он тоже является списком, значит, в нем тоже есть своя голова (E_2) и свой хвост (начиная с E_3). И так далее. В конце списка последний его элемент E_n объединяется с пустым эле-

ментом, формально завершающим список.

Такая организация списка позволяет рекурсивно его обрабатывать, постепенно отделяя головы от хвостов.

В AgentSpeak списки как константы обозначаются в квадратных скобках, а их элементы следуют через запятую, например.

list([a, b, c, d]). – убеждение list содержит в качестве терма список.

Списки могут содержать разнородные и повторяющиеся элементы, в том числе могут быть вложенными, например.

list([a, a, 12, “string”, f(a), g(X), [a, b, c]]).

Для отделения головы списка от его хвоста применяется специальный оператор «|» - вертикальная черта. Например, следующая программа рекурсивно выведет на экран все элементы списка, задаваемого в качестве аргумента в начальной цели.

```
!start([a, b, c, d]).
+!start([H|T]):H\==[] <-.print(H);
                        !start(T).
+!start(L)<-true.
```

Обратите внимание, что первый план рекурсивно выполняется до тех пор, пока голова списка не равна пустому списку. Как только перебор доходит до пустого элемента срабатывает второй план и программа завершает свою работу.

Многие внутренние функции требуют в качестве аргументов списки или возвращают список результатов (см. прил. №1).

В последние версии Jason (начиная с 1.1.2) в экспериментальном порядке введена поддержка условного и циклических операторов в теле плана. С их помощью тоже можно организовать обработку списков (см. прил.1, функция «.member»). Однако, по мнению авторов пособия, реализация данных операторов пока не выглядит до конца отлаженной, поэтому в пособии эти возможности не рассматриваются. Более подробную информацию можно почерпнуть из файла «release-notes.txt».

4.4.9 Общие рекомендации по составлению и расположению планов

Во-первых, если Вы разрабатываете не тривиальную те-

стовую программу, а более-менее серьезную модель, то прежде, чем браться за программирование следует провести концептуальный анализ, результатами которого следует руководствоваться при программировании.

Во-вторых, необходимо помнить, что если Вы разрабатываете план для обслуживания события возникновения цели при каких-то интересующих Вас ограничениях, то Вы должны также задуматься над тем, возможно ли данное событие в отсутствии этих ограничений. Потому, что если появится цель, а подходящего по ограничениям плана для нее не существует, то возникнет ошибка. Чтобы ее избежать можно, например, использовать пустые планы-заглушки.

```
!p(6).
+!p(X):X<5 <- .print(X).
+!p(X)<-true.
```

Приведенная программа ничего не напечатает, т.к. первый план не сработает, а сработает второй план-заглушка.

В-третьих, при программировании поведения агентов довольно часто бывает так, что контекстные ограничения разных планов описывают пересекающиеся классы ситуаций, например:

```
p(10).
q(8).
+!start:p(X)&X>5 <- .print("X>5").
+!start:p(X)&q(Y)&X>5&Y>3 <- .print("X>5 and Y>3").
```

Следовательно, в некоторых ситуациях между планами возможен конфликт. По-умолчанию процедура выбора плана по контекстным ограничениям выбирает первый подходящий план. Первый – значит расположенный в тексте программы раньше других. Достаточно велика вероятность того, что при таком выборе будет выбран наименее подходящий в данной ситуации план. В частности, в приведенном примере будет выбран первый план, хотя условия второго более специфичны и тоже выполняются. Для корректировки логики работы программы в подобных случаях могут быть использованы следующие подходы.

1) Устранить пересечение ситуаций, описываемых контекстными ограничениями. Для этого необходимо дополнить наиболее общие ограничения новыми условиями, например, так:

+!start:p(X)&X>5& not q(_) <- .print(“X>5”).

+!start:p(X)&q(Y)&X>5&Y>3 <- .print(“X>5 and Y>3”).

Теперь первый план сработает, только если в базе убеждений нет никаких предикатов q(_).

2) Расположить планы в базе планов агента в порядке расширения классов ситуаций, описываемых контекстными ограничениями. Иначе говоря, сначала нужно расположить планы с наиболее «длинными» ограничениями, потом – с наиболее короткими. В приведенном примере первый и второй планы нужно поменять местами.

+!start:p(X)&q(Y)&X>5&Y>3 <- .print(“X>5 and Y>3”).

+!start:p(X)&X>5 <- .print(“X>5”).

3) Переопределить процедуру выбора планов по ограничениям (способ подходит для уверенно владеющих Java и неизбежен при разработке сложных приложений). Этот способ позволяет Вам реализовать любую необходимую эвристику, что приведет к более гибкому поведению агентов, упростит процесс программирования планов, но замедлит процесс логического вывода соразмерно сложности новой процедуры.

Имея в виду эти рекомендации можно приступать к выполнению лабораторных работ.

4.5 Работа агента в среде Jason

Для понимания работы МАС в Jason требуется знать, как устроен и как работает используемый в нем механизм логического вывода агентов. Рассмотрим рис. 7. На нем жирными контурами обозначены основные информационные модули базы знаний агента:

- база убеждений;
- множество событий;
- база планов;
- множество стеков-намерений.

Кроме того введены следующие обозначения:

- S_M – модуль выбора сообщений;

- F_{SA} – фильтр социально-значимых сообщений;
- ОУ – модуль обновления убеждений;
- ПУ – модуль пересмотра убеждений;
- $ОС_1$ – модуль сопоставления заголовков планов с произошедшим событием;
- $ОС_2$ - модуль сопоставления контекстных ограничений планов с текущими убеждениями;
- S_E – модуль выбора события;
- S_O – модуль окончательного выбора плана;
- S_I – модуль выбора намерения;
- ВН – модуль выполнения намерений;

Также некоторые модули дополнительно обозначены цифрами для удобства ссылок.

Функционирование агента в пределах одного цикла подчиняется следующему обобщенному алгоритму.

1. Восприятие среды.

Работа агента на очередном цикле принятия решений начинается с восприятия среды. Состояние среды определяется состоянием объектов находящихся в ней, т.е. значением их свойств и отношений в данный момент времени, которые агент может «наблюдать» своими программными датчиками. Информация о состоянии объектов передается агенту в символьной форме – в виде списка литералов.

2. Обновление базы убеждений.

Как только произошло восприятие среды необходимо обновить базу убеждений, чтобы отразить в ней произошедшие в среде изменения. Обновление происходит по следующему простому алгоритму (допустим, P – воспринятое множество литералов, B – база убеждений, L – литерал):

- Если $L \in P$, но $L \notin B$, то $B = B \cup L$.
- Если $L \notin P$, но $L \in B$, то $B = B / L$, т.е. литерал исключается из базы убеждений.

Каждое изменение в базе убеждений сопровождается генерацией соответствующего события. События бывают двух типов – *внешние* и *внутренние*. Внешние события генерируются в результате восприятия среды. К внутренним событиям приводит

реализация некоторых намерений.

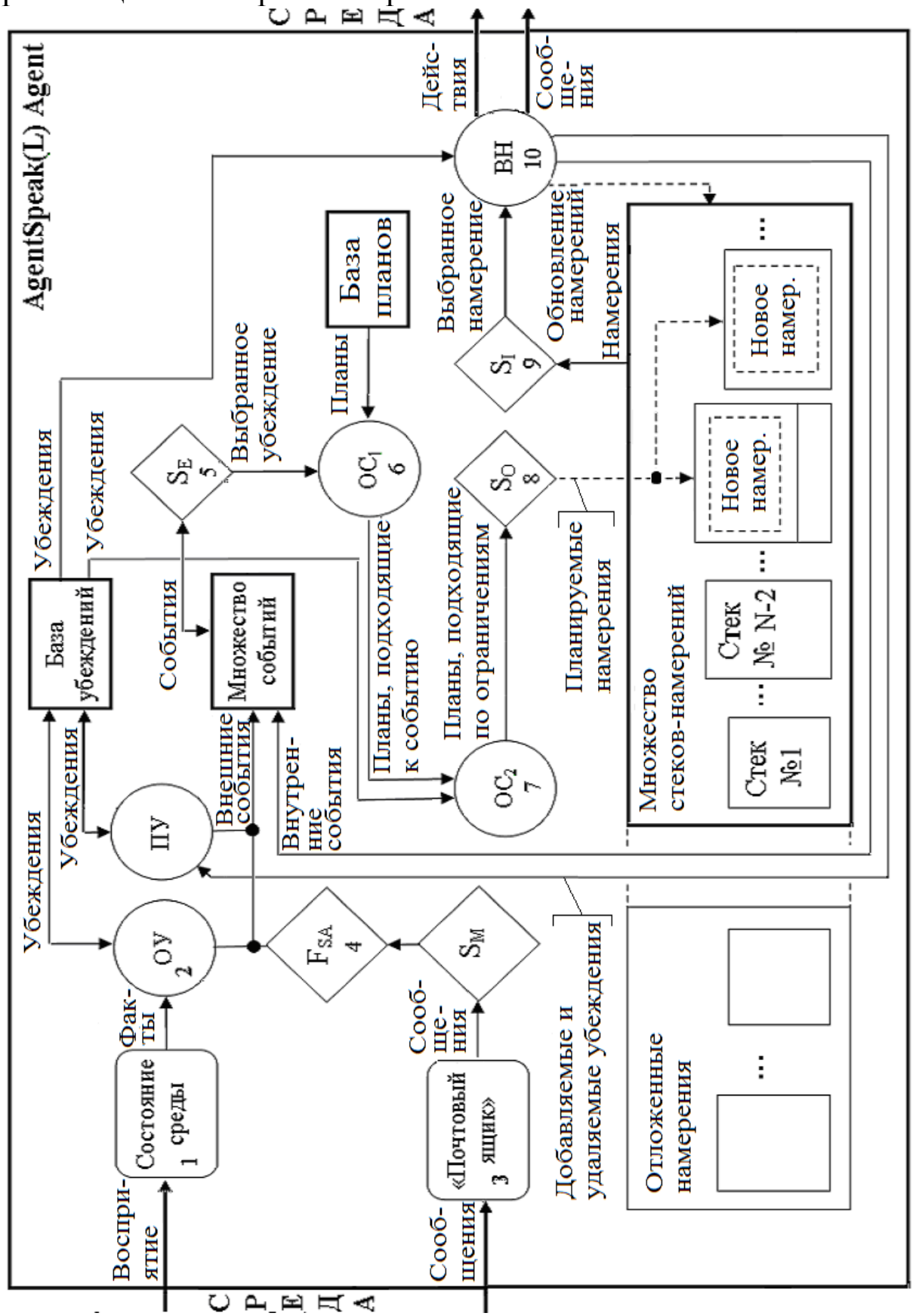

Рис. 7

3. Получение сообщений от других агентов.

Помимо внешней среды агенты также могут получать информацию от других агентов. Эта информация пересылается в виде KQML сообщений. На каждом цикле агент может получить множество сообщений, которые по мере получения пополняют специальный список (почтовый ящик). Однако только одно из ранее полученных сообщений будет обработано агентом на этом цикле. Какое это будет сообщение, определяет специальная функция выбора (S_M). По-умолчанию эта функция выбирает последнее (т.е. наиболее свежее) сообщение в списке. При необходимости, можно модифицировать соответствующий Java-метод и реализовать собственную функцию выбора.

4. Фильтр «социально значимых» сообщений (F_{SA}).

Агента можно настроить так, чтобы в его базу убеждений попадала информация не из всех сообщений (как по умолчанию), а только из значимых для него. Для этого тоже нужно переопределить соответствующий Java-метод, реализующий фильтрацию. Данный фильтр позволяет заранее избавиться от всех «спамерских» сообщений, а все прочее сообщения, пройдя фильтр, будут добавлены к множеству произошедших событий.

Обратите внимание, что информация из сообщения, прошедшего все предыдущие этапы, ***не сразу*** попадает в базу убеждений. Сначала сообщение существует в виде события, ожидающего обработки. И только потом, когда это событие будет выбрано для обработки (модуль 5), когда, возможно, будут обработаны планы, связанные с возникновением данного события-сообщения (модули 6,7,8), когда, наконец, через стеки намерений сообщение достигнет механизма исполнения намерений, только тогда информация будет извлечена из сообщения и передана на модуль модификации базы убеждений.

5. Выбор события для обработки.

В процессе работы агента происходит множество событий, как в среде, так и во внутреннем состоянии агента. Однако в рамках одного цикла он не может обработать их все. Поэтому он вынужден выбрать одно событие для обработки в данном цикле.

Это делается операцией, обозначенной на схеме как S_E. Данная операция также может быть переопределена. По умолчанию база событий работает как стек FIFO. Если база событий пуста, то переходим на пункт 9.

6.Считывание подходящих планов.

Как только выбрано событие для текущей обработки, происходит его сопоставление с отслеживаемыми событиями в условиях активации всех планов агента. Это делается путем сопоставления предикатного символа события из заголовка плана с предикатным символом выбранного события с учетом возможных унификаций термов и аннотаций планов, в которых может проверяться источник события. В результате получаем подмножество планов, которые могут быть применены для обработки данного события.

7. Определение планов, подходящих по контексту.

Далее среди планов, подходящих для обработки выбранного события, выбирается подмножество, у которых удовлетворяются контекстные ограничения с учетом аннотаций проверяемых в них фактов.

8.Определение применяемого плана.

Теперь среди подходящих по контекстным ограничениям планов требуется выбрать один. Это делается посредством операции S_O. Данная операция также может быть переопределена, если в этом есть необходимость и если Вы владеете Java. По умолчанию выбор плана осуществляется в зависимости от его места в базе планов агента. Иначе говоря, у первого плана в тексте программы самый высокий приоритет, у последнего – самый низкий. Данный подход позаимствован из среды Prolog и может быть полезен при составлении планов с рекурсией.

Далее операции выбранного для применения плана добавляются во множество намерений, принимаемых к осуществлению. Операции одного плана образуют стек и понимаются как одно намерение. Намерения по достижению подцелей плана образуют один стек вместе с намерениями плана. Таким образом, каждое новое событие порождает новое намерение. Однако иногда бывает полезно вмешаться в распределение операций одного плана по намерениям. Например, в некоторых рекурсивных алго-

ритмах стек чрезмерно разрастается, что может привести к его переполнению. Поэтому введен специальный префикс «!!», который будучи применен в теле плана к вызову подцели, помещает операции, относящиеся к достижению подцели, в новый стек, т.е. образует новое намерение.

9.Выбор намерения для осуществления.

Любые процессы для своего осуществления требуют определенного времени. И множество операций в теле плана тоже требует времени для выполнения. Это время измеряется в циклах работы агента. Как мы знаем, мир не стоит на месте, и пока мы что-то делаем, вокруг происходят разные события, которые, возможно, требуют немедленной реакции. Поэтому за один цикл агент выполняет не весь план, а осуществляет только одну операцию из плана, при этом отслеживая события и модернизируя множество намерений. Следовательно, необходима функция выбора намерения для обработки на текущем цикле. Эта функция обозначена как S_I. Как и прочие функции, эта тоже может быть переопределена средствами Java. По-умолчанию данная функция реализует алгоритм циклического обслуживания, который работает следующим образом. Множество намерений реализовано в виде списка, поэтому на текущем цикле из этого списка берется первое намерение, а из него первая операция, которая передается на выполнение. Выполненная операция удаляется из намерения, а само намерение перемещается в конец списка. Таким образом, постепенно будут обработаны все намерения. Такой подход гарантирует равномерность распределения внимания агента между всеми намерениями. Непрерывное выполнение намерения обеспечивается аннотацией «atomic».

10.Выполнение операции из выбранного намерения.

Выполняемой операцией может быть любая из дозволенных в Jason, а именно.

- Выполнение внешних операций (над объектами среды функционирования агентов). При этом выполнение намерения, вызвавшего данную операцию, приостанавливается (поскольку ожидается отклик среды), но выполнение других намерений будет продолжено на следующем цикле.

- Постановка новых подцелей. Некоторые задачи для своего решения требуют решения подзадач, или достижения подцелей. Постановка подцели является внутренним событием и приводит к активации соответствующих подпланов. При этом выполнение основного (т.е. вызвавшего данную подцель) плана приостанавливается до тех пор, пока не будут достигнуты все его подцели.

- Выполнение операции конкретизации. Используются в теле плана для считывания информации из базы убеждений посредством конкретизации переменных соответствующего предиката.

- Добавление, удаление, обновление отдельных фактов в базе убеждений. Реализуется через внутренние события «+», «-», «-+».

- Вызов внутренних функций из библиотек, поставляемых вместе со средой Jason. К внутренним функциям, в частности, относится операция посылки сообщений другим агентам (перечень внутренних функций смотри пр. №1).

- Вычисление выражений.

11. Выполнение обслуживающих операций и завершение цикла работы агента.

В конце цикла работы агента требуется выполнить несколько вспомогательных операций. Во-первых, необходимо проверить статут приостановленных намерений. Если операции, ожидающие завершения (вызовы внешних операций, ожидание ответов от других агентов), завершены успешно, то возобновить выполнение соответствующих намерений. Во-вторых, это «сборка мусора» - удаление успешно отработавших намерений. В-третьих, это генерация уведомляющих событий для операций, завершенных по ошибке.

4.6 Инструменты отладки МАС в Jason

В Jason существует несколько способов отладки. Самые простые из них следующие.

1. Вывод в текстовую консоль подробных отчетов о деятельности агентов и их изучение. Для этого следует использовать внутреннюю функцию «.print» во всех интересующих местах тела плана.

2. Использование стандартного инспектора внутреннего состояния агентов. Инспектор вызывается при запуске про-

граммы в режиме отладки (кнопка «debug»). Его окно (рис.8) состоит из трех областей: слева список работающих в настоящий момент агентов, из которого можно выбирать; основную площадь окна занимает информация о внутреннем состоянии выбранного агента; внизу находятся элементы управления выполнением МАС в режиме отладки, которые позволяют запускать работу МАС (или выбранного агента) на заданное число циклов, просматривать историю изменения состояния агентов, выбирать способ отображения информации.

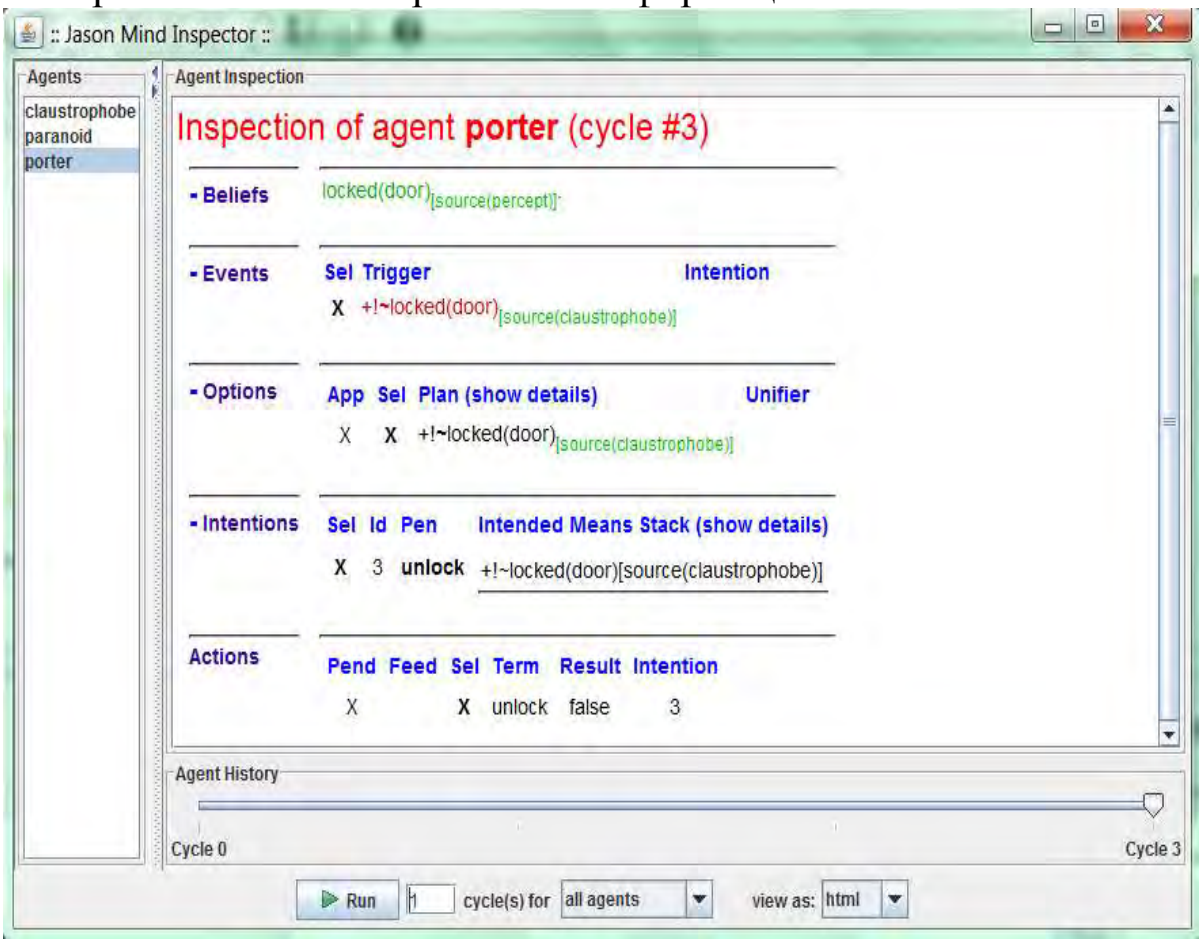

Рис.8

Параллельно с работой инспектора в текстовую консоль выводится развернутый технический отчет об изменениях состояний всех агентов (рис. 9).

3. Использование сниффера сообщений, предназначенного для разработки МАС под платформу Jade (рис.11). Отличительной особенностью данного инструмента является то, что

он позволяет в наглядной графической форме по циклам отобразить процесс обмена сообщениями между различными агентами среды (пользовательскими и служебными). Нажимая на отдельные сообщения, изображенные стрелками, можно получить о них более содержательную информацию. Чтобы сниффер стал доступен, необходимо, во-первых, в окне установок Jason, касающихся Jade (рис.10), установить флажок «start sniffer», а во-вторых, в файле проекта изменить тип используемой архитектуры МАС с «Centralized» на «Jade», например:

```
MAS MyMAS {
        Infrastructure: Jade;
        Agents: ……
        …….
}
```

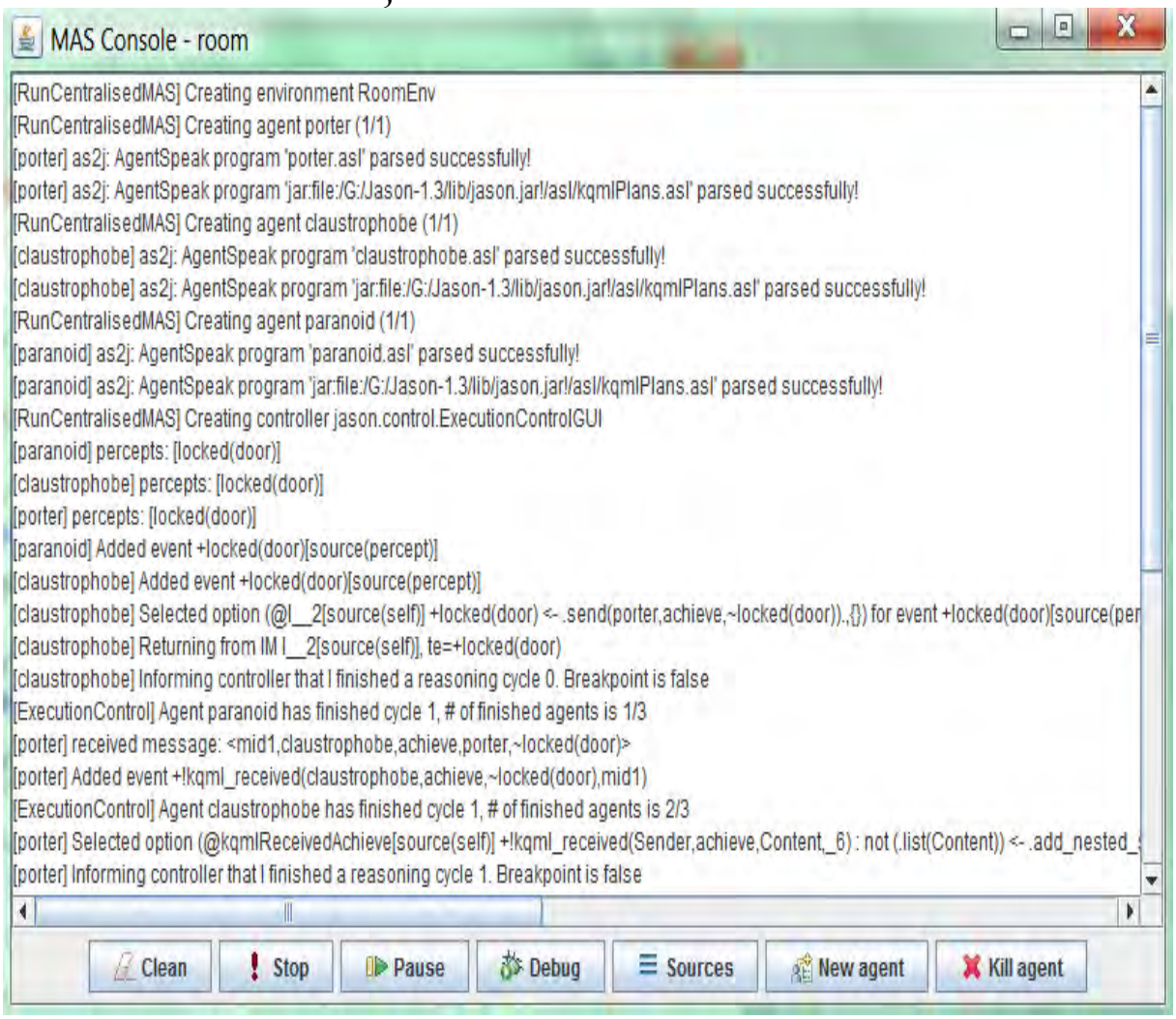

Рис. 9

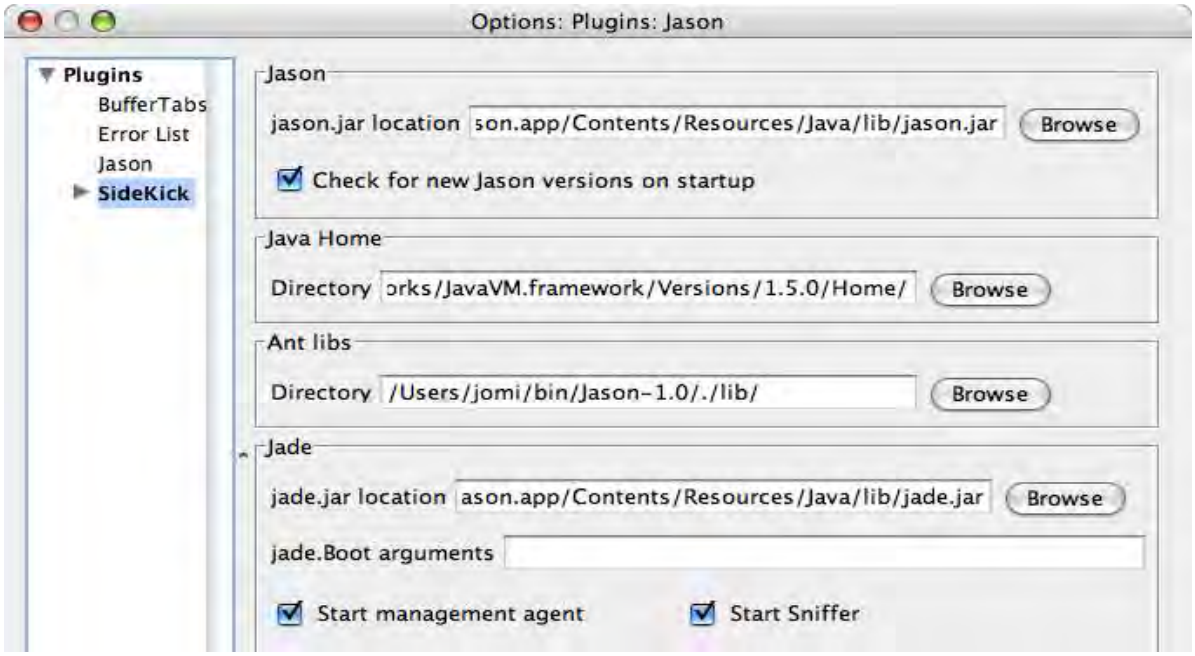
Options: Plugins: Jason
Plugins
BufferTabs
Error List
Jason
SideKick
Jason
jason.jar location
son.app/Contents/Resources/Java/lib/jason.jar
Browse
Check for new Jason versions on startup
Java Home
Directory
orks/JavaVM.framework/Versions/1.5.0/Home/
Browse
Ant libs
Directory
/Users/jomi/bin/Jason-1.0/../lib/
Browse
Jade
jade.jar location
ason.app/Contents/Resources/Java/lib/jade.jar
Browse
jade.Boot arguments
Start management agent
Start Sniffer

Рис.10

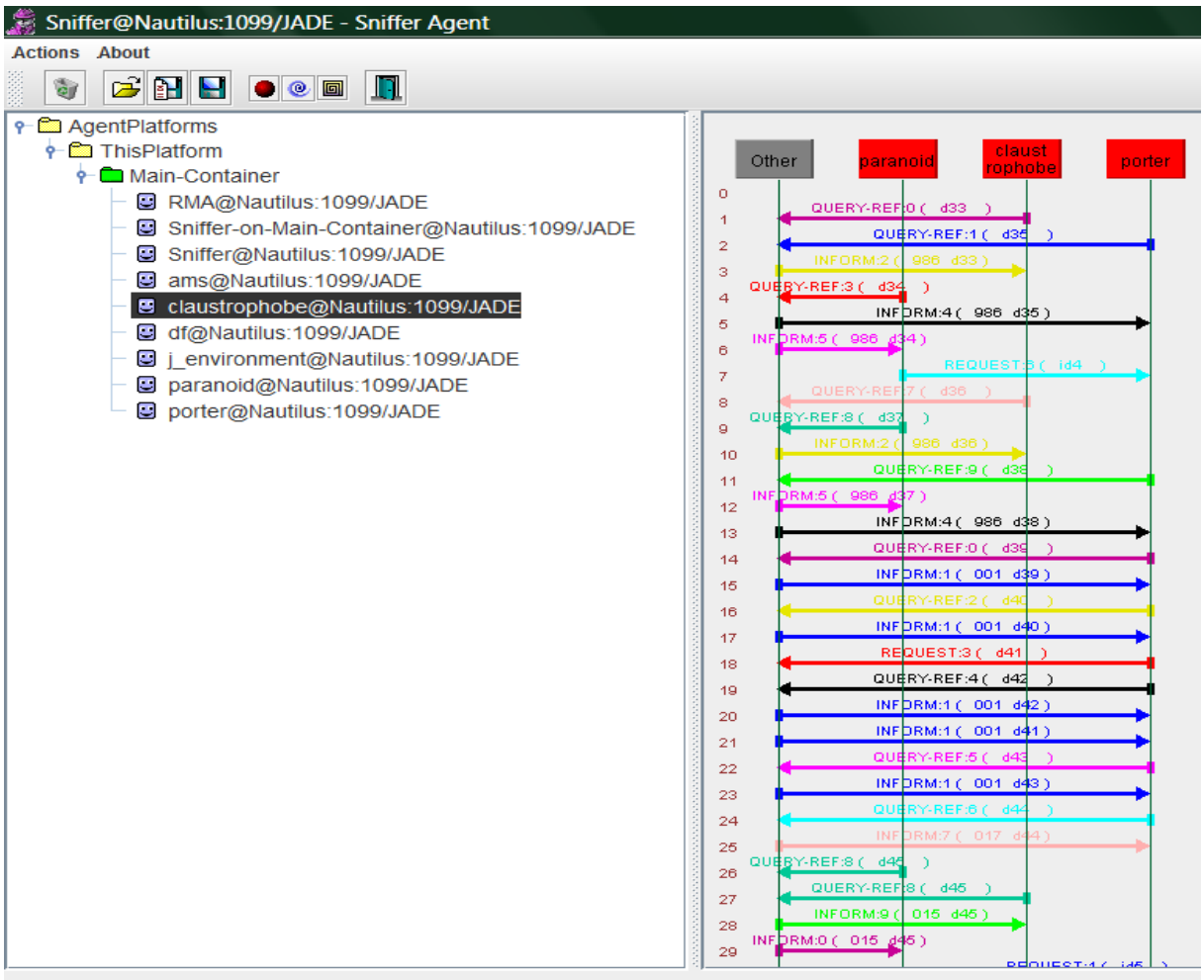
Sniffer@Nautilus:1099/JADE - Sniffer Agent
Actions
About
AgentPlatforms
ThisPlatform
Main-Container
RMA@Nautilus:1099/JADE
Sniffer-on-Main-Container@Nautilus:1099/JADE
Sniffer@Nautilus:1099/JADE
ams@Nautilus:1099/JADE
claustrophobe@Nautilus:1099/JADE
df@Nautilus:1099/JADE
j_environment@Nautilus:1099/JADE
paranoid@Nautilus:1099/JADE
porter@Nautilus:1099/JADE
Other
paranoid
claust rophobe
porter

Рис.11

5. ВЫПОЛНЕНИЕ ЛАБОРАТОРНЫХ РАБОТ

5.1 Описание процесса

Выполнение лабораторных работ должно происходить по следующему сценарию.

1.Организационная работа.

- Самостоятельный выбор предметной области и постановка задачи в ней (предполагается, что студент придумает себе такую задачу, в решении которой он хорошо разбирается – т.е. считает себя экспертом).
- Согласование результатов выбора с преподавателем (задание должно быть уникальным, подходить под критерии многоагентности и согласовываться с нравственностью).
- Окончательная формулировка задания и фиксация его на бланках, подписываемых студентом и преподавателем (после этого задание изменено быть не может).

2. Проектирование.

- Концептуальный анализ деятельности агентов, подразумевающий:
- построение концептуальных структур действий по указанной методике;
- построение навигационной карты концептуального анализа;
- выписывание концептуальной модели.
- Разработка обобщенной структурной схемы МАС.
- Разработка вспомогательных алгоритмов и построение блок-схем для них. К таким алгоритмам относятся:

• алгоритмы работы агентов, выполняющих сервисные функции для всей МАС (например, порождение и удаление агентов, сбор статистики);

• алгоритмы, описывающие законы среды функционирования (если предполагается разрабатывать среду);

• алгоритмы работы пользовательского интерфейса.

3. Программирование агентов (и среды их функционирования, если предусмотрено) в системе Jason.

4. Тестирование, отладка и доработка модели в процессе экспериментирования.

5. Подготовка отчета, отражающего пункты 1-4.

5.2 Пример выполнения лабораторных работ

5.2.1 Постановка задачи

Приведем относительно небольшой, но содержательный пример разработки МАС в рамках выполнения лабораторных работ на языке AgentSpeak в среде Jason. Этот пример не является частью библиотеки примеров, поставляемых вместе со средой, и был разработан авторами пособия.

Зададимся задачей разработать весьма абстрактную модель работы гипотетического фастфуда, в котором работают следующие сотрудники:

- повар;
- работник склада готовых продуктов;
- несколько продавцов.

Задачей повара является приготовление продуктов и обеспечение склада некоторым минимальным их количеством.

Работник склада распределяет готовые продукты между продавцами по их запросу.

Продавцы обслуживают клиентов, которые заходят в фастфуд через неравные интервалы времени.

Целью моделирования является наблюдение за процессом организации покупателей в очереди.

5.2.2 Концептуальный анализ поведения агентов

Деятельность каждого агента анализируется последовательно, в соответствии с их внешними целями, по методике, описанной в [6] и интерпретированной в данном пособии с точки зрения особенностей многоагентных систем.

Анализ следует начинать со структуры, отражающей основной моделируемый процесс. Так как в нашей системе предполагается более, чем один агент, таких структур тоже может быть несколько. Наиболее вероятно, что все они приведут к, практически, одинаковым концептуальным моделям, поэтому нет большой разницы, с какой мы начнем, но, так или иначе, это должно быть описание задачи моделирования на самом «верхнем», т.е. абстрактном, уровне.

Что такое фастфуд? Это пищевое предприятие, куда приходят люди, чтобы купить какое-то количество порций разного вида уже готовой еды. С этой точки зрения, в качестве первой корневой структуры можно рассмотреть такую (рис. 12):

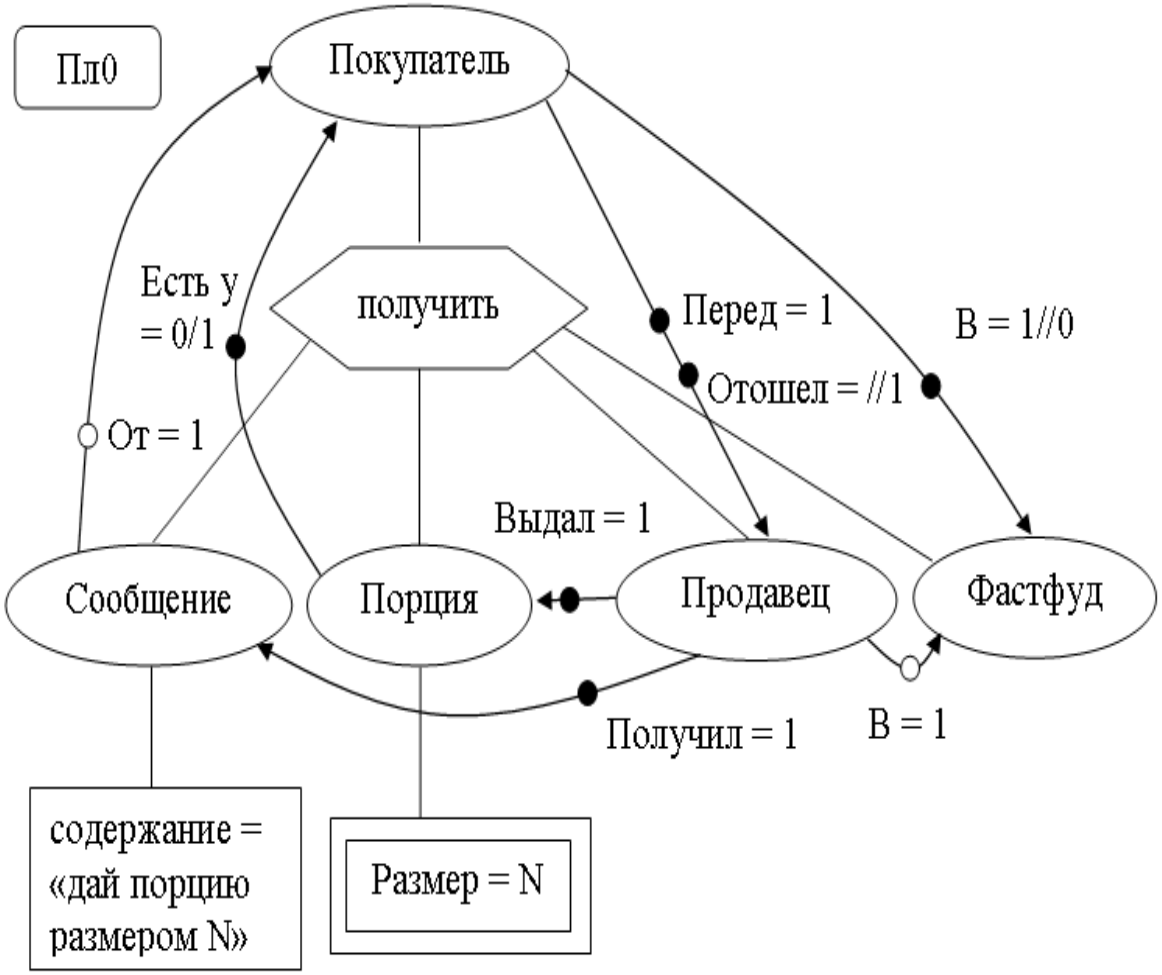

Рис. 12

Далее, следует помнить, что вся первичная информация о работе агентов выявляется в процессе гипотетического диалога с экспертами (или сотрудниками фастфуда) по названной методике и формализуется в виде концептуальных структур.

Согласно методике, у данной схемы выявляются **6 поддействий**: покупатель должен оказаться в фастфуде, затем перед каким-либо продавцом, сообщить ему свой заказ, продавец должен ему его выдать, после чего покупатель должен отойти от него и покинуть фастфуд.

Таким образом, следующей будет схема на рис. 13:

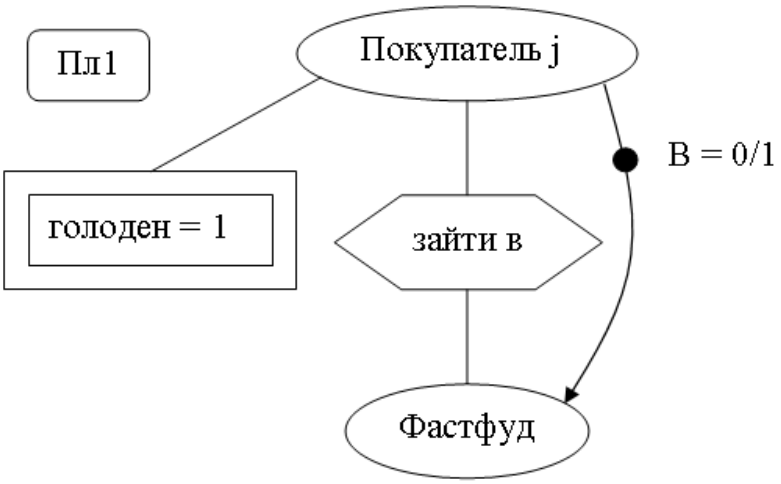

Рис. 13

Когда покупатель зайдет в фастфуд? Когда он голоден. А как он окажется перед каким-либо продавцом? Зайдя, он должен подойти к нему. Соответствующая схема могла бы быть обозначена как Пл2. Однако, данное действие может быть выполнено двумя путями в зависимости от того, есть ли на текущий момент времени продавец, свободный от обслуживания покупателей, или ко всем продавцам стоят очереди. Рассмотрим сперва последний вариант, обозначенный как Пл2.1 (рис. 14):

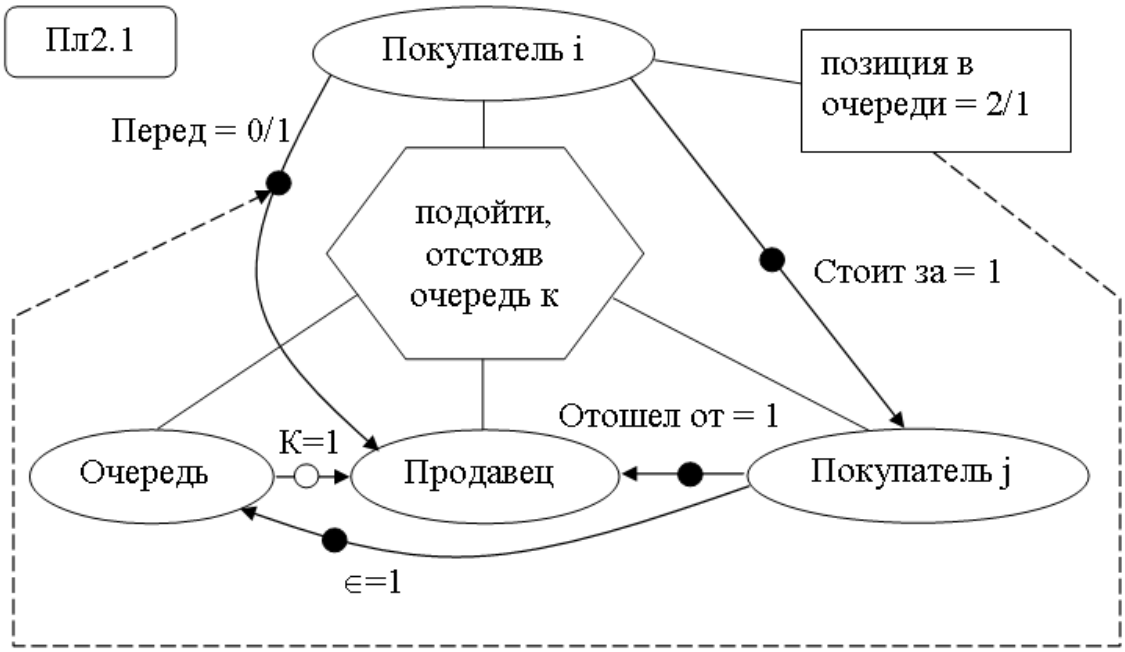

Рис.14

Когда покупатель (например, i-й), отстояв очередь, приблизится к продавцу? Когда другой покупатель (например, j-й), за которым стоит наш i-й, отойдет от продавца, и позиция i-го сдвинется с 2 (он был уже вторым в очереди) на 1 (а стал первым в очереди).

А как покупатель окажется стоящим в очереди за кем-то? Он займет очередь (рис. 15):

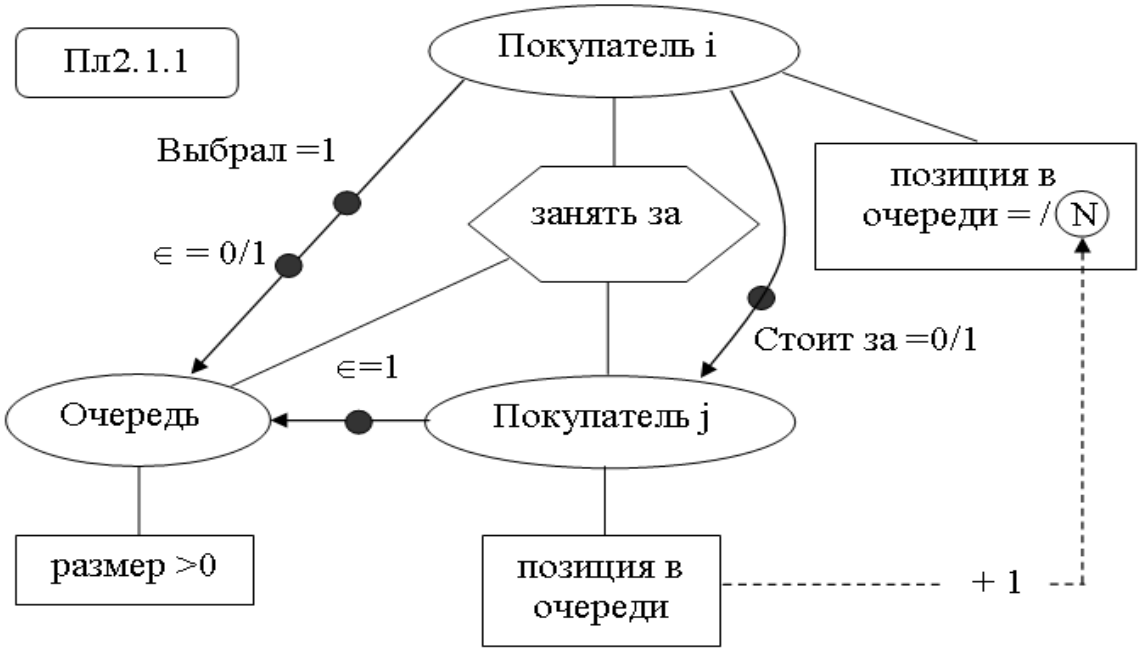

Рис. 15

Но в какую очередь встанет покупатель? В ту, которую он выберет по какому-либо критерию, например, по длине (рис. 16):

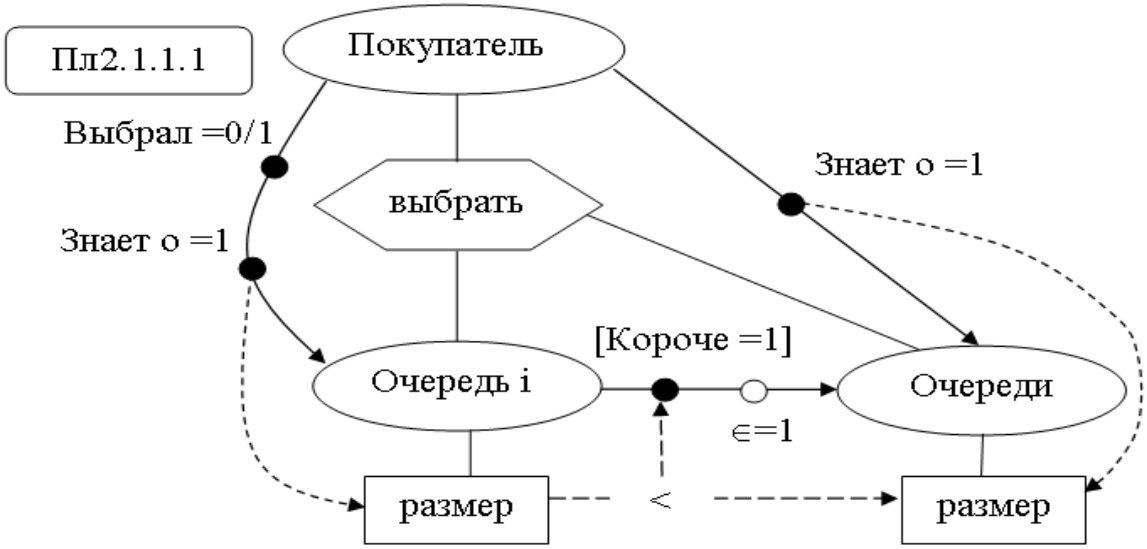

Рис. 16

А каким образом покупатель узнает о размерах очередей? Ему сообщают другие покупатели, уже стоящие в очередях, когда тот у них спрашивает (рис. 17):

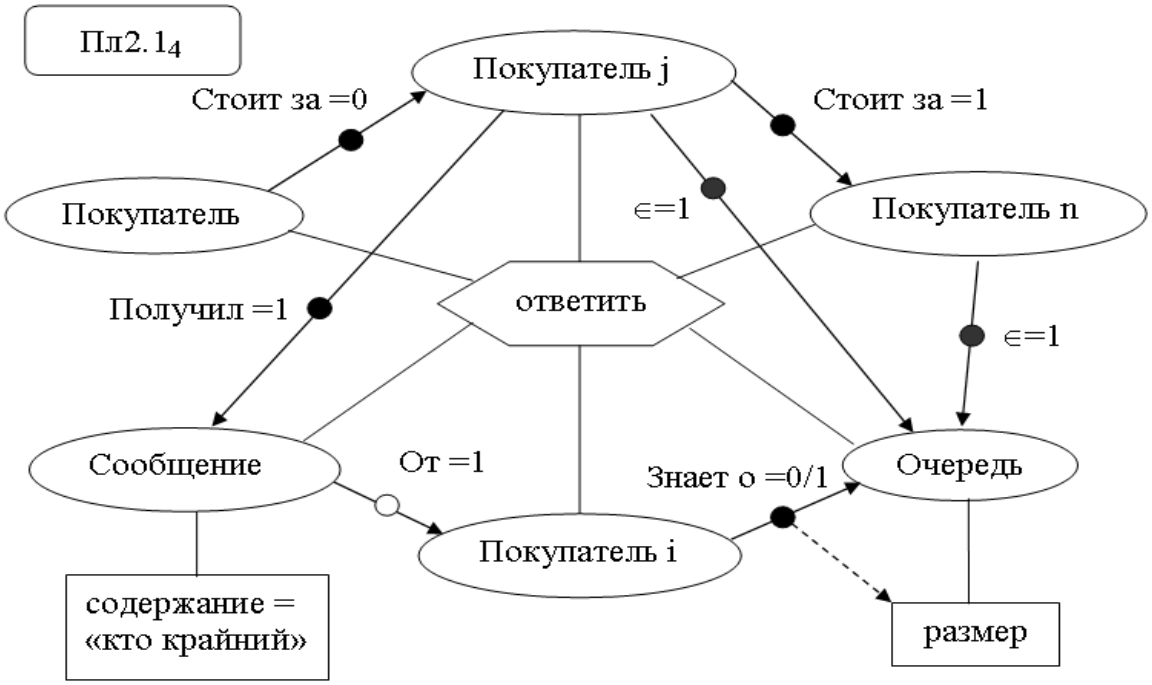

Рис. 17

А когда покупатель спрашивает «кто крайний»? Когда он вошел в фастфуд и еще не сориентировался в очередях (рис .18):

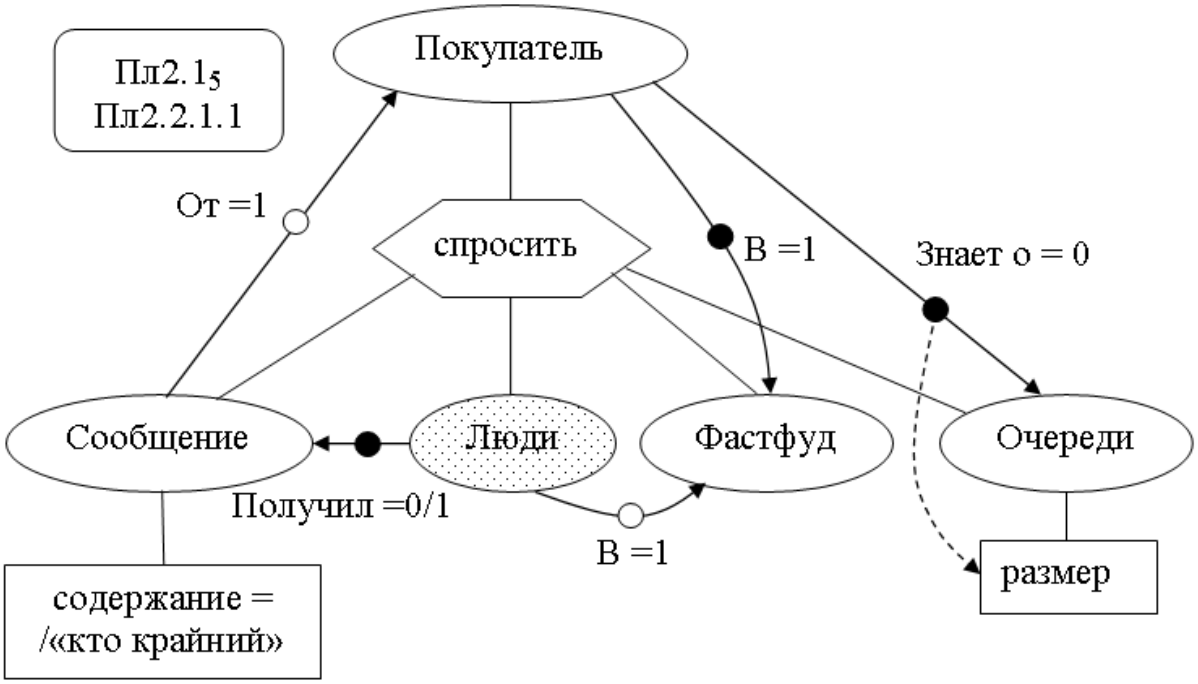

Рис. 18

Покупатель адресует свой вопрос всем, кто может его услышать. Это отражено в схеме с помощью абстрактного понятия «Люди» (показанного штриховкой). Соответствующие иерархические отношения между выявленными понятиями-объектами изображены на рис. 19:

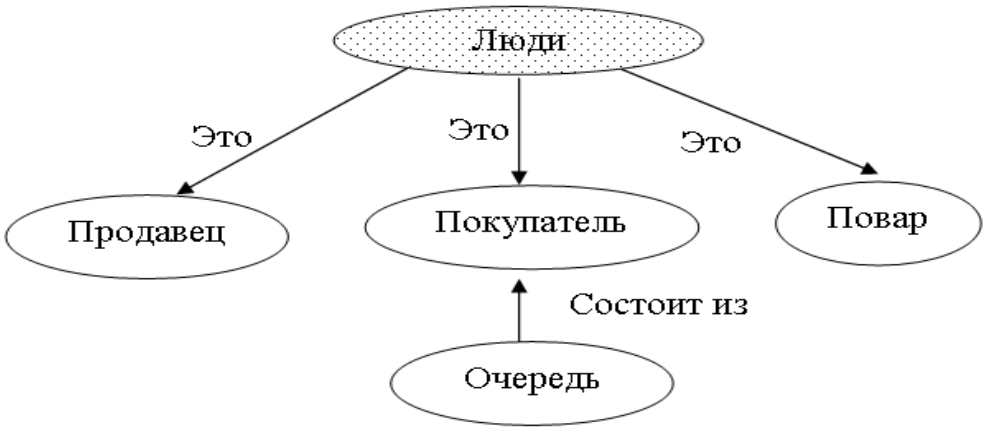

Рис. 19

Сообщение «кто крайний» могут получить и продавец, и повар, и другие покупатели. Но как они должны реагировать на него? Повар – никак, т.к. он не имеет к очереди непосредственного отношения. Но продавец должен ответить в случае, когда очередь отсутствует и пришедший покупатель может подойти к нему без очереди (рис. 20):

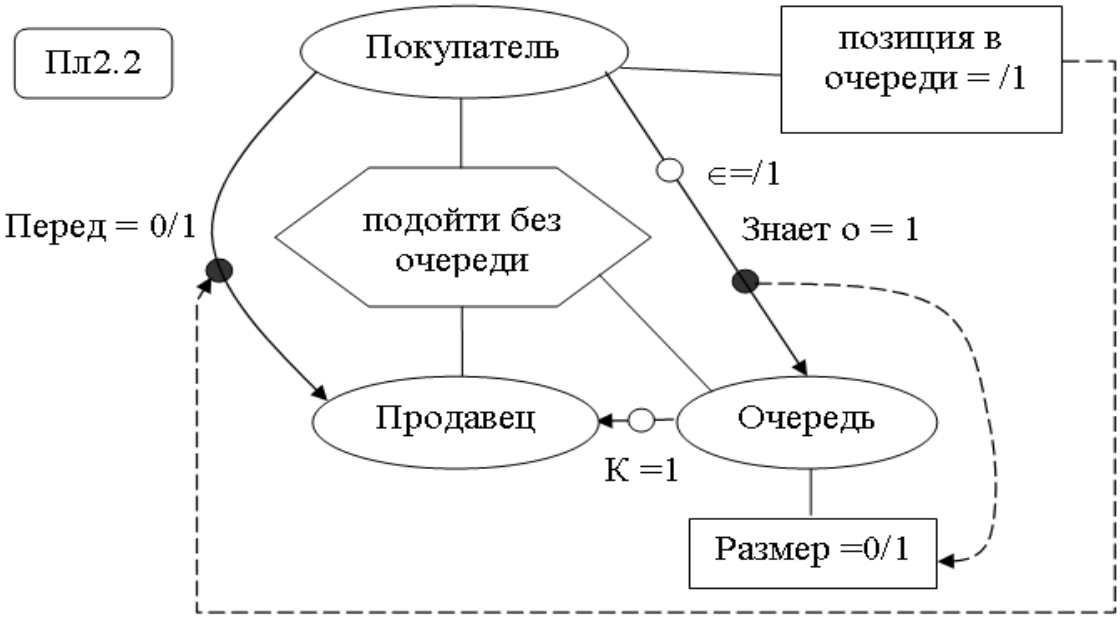

Рис. 20

Покупатель в этом случае сам начинает очередь (становится в ней первым). Узнает об отсутствии ее он из ответа продавца (рис. 21):

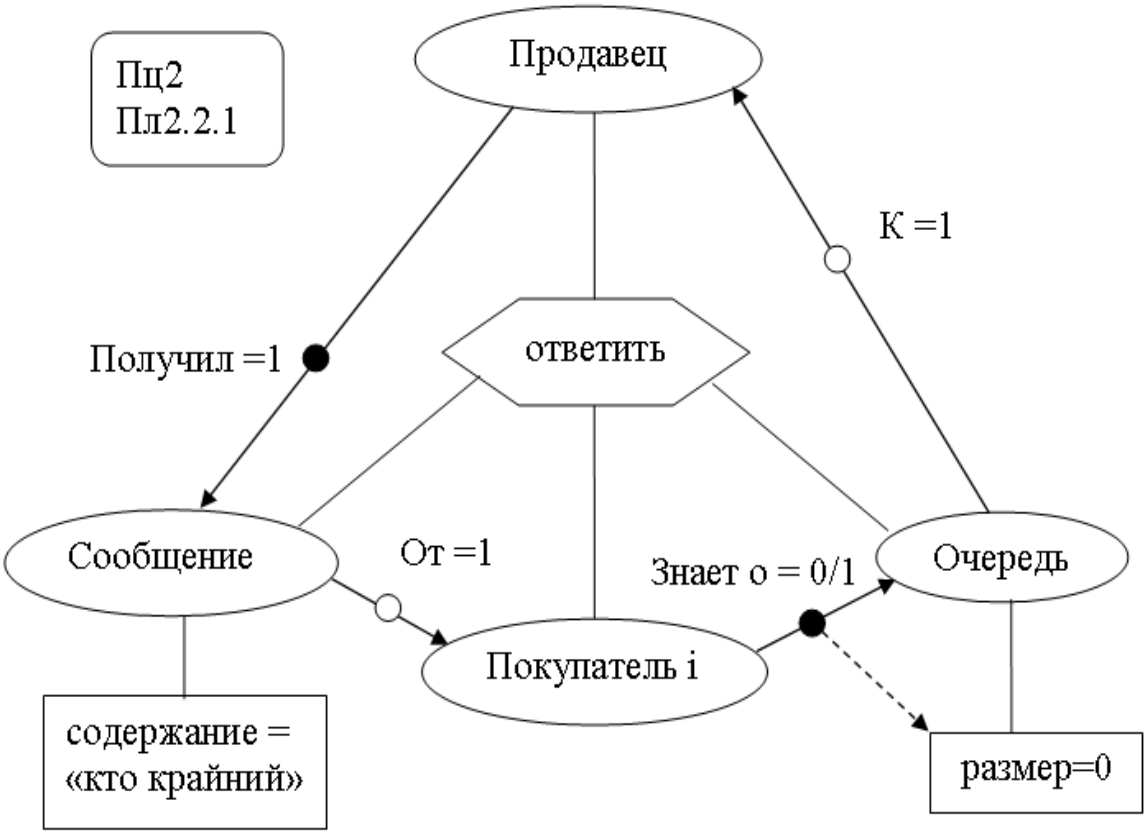

Рис. 21

Эта схема имеет два номера: верхний указывает на то, что это действие выполняет уже другой агент – продавец, отвечающий на запрос покупателя, а нижний показывает место данной схемы в раскрытии содержания действия «Покупатель подойти к продавцу».

Поддействием ее является, как и выше в схеме «Покупатель j ответить покупателю i», схема «Покупатель спросить людей», у которой в связи с этим также два номера (рис. 18).

А что нужно, чтобы, стоя в очереди, продвигаться к ее началу, меняя текущую позицию? Для этого необходимо, чтобы впереди стоящие покупатели все-таки покидали очередь. А почему покупатель отходит от продавца? Потому, что он получил свой заказ (рис. 22). Отходя от продавца, обслуженный покупатель инициирует продвижение очереди.

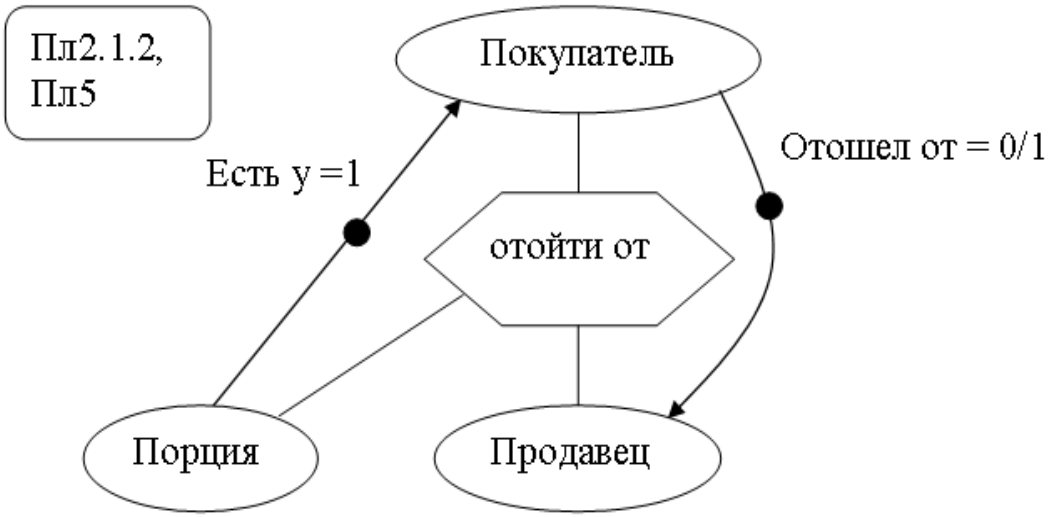

Рис. 22

А как покупатели продвигаются в очереди? Они меняют свою позицию, когда тот, кто стоит впереди, продвигается вперед или уходит, а на его прежнем месте образуется пустая позиция, которая постепенно перемещается от начала очереди к ее концу по мере продвижения покупателей (рис. 23):

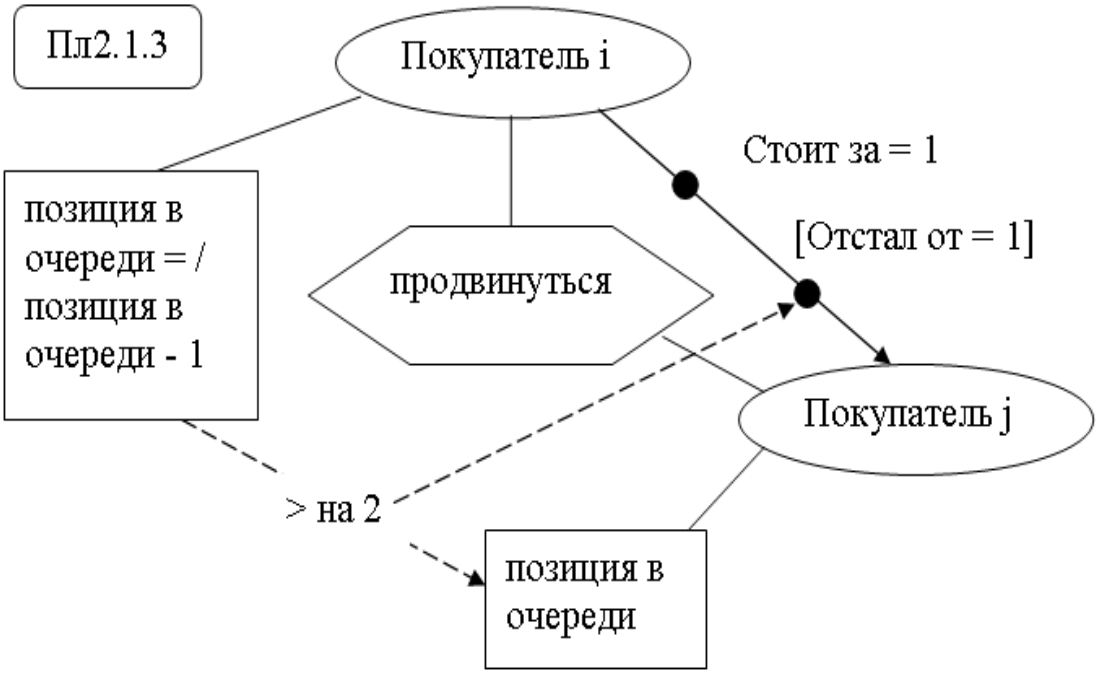

Рис. 23

Что же делает покупатель, когда наконец окажется перед продавцом? Он обращается к нему со своим заказом (рис. 24):

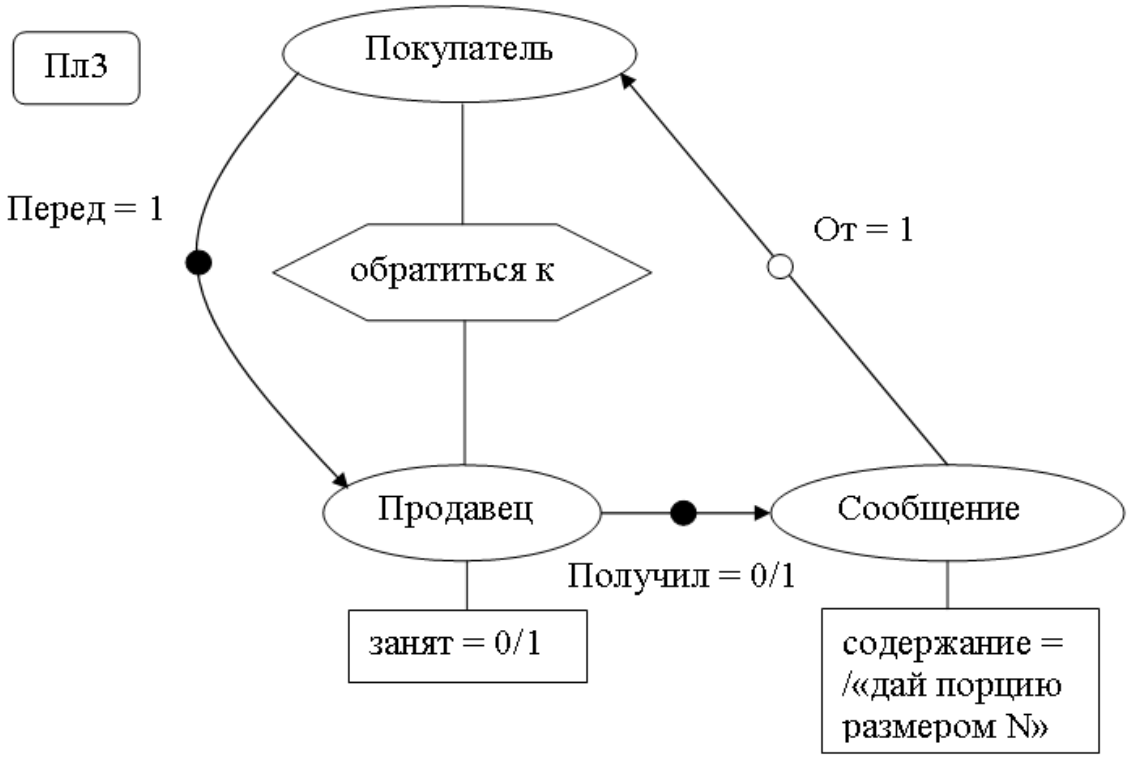

Рис. 24

И что будет делать продавец? Он должен выдать покупателю требуемую порцию. Для действия покупателя «получить порцию» это является поддействием с номером 4, но для продавца это уже корневая структура (рис. 25):

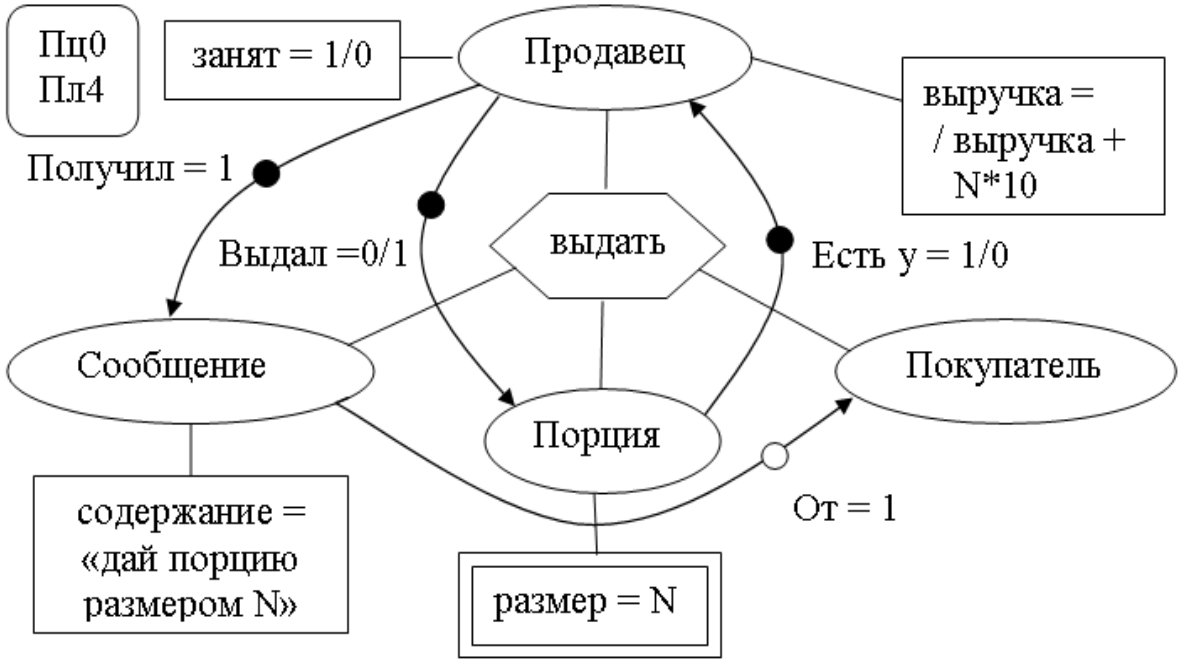

Рис. 25

Поддействием этого действия является структура на рис. 26, отвечающая на вопрос, как порция появляется у продавца:

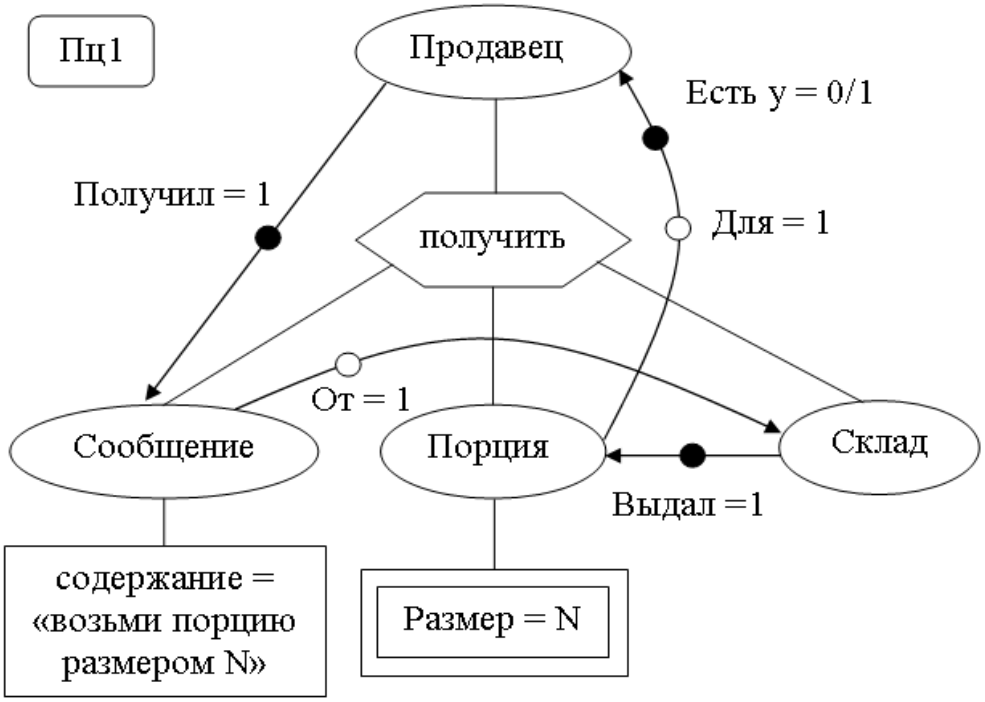

Рис. 26

Порцию продавцу выдает склад. Но почему он это делает? Потому, что получил запрос от продавца (рис. 27):

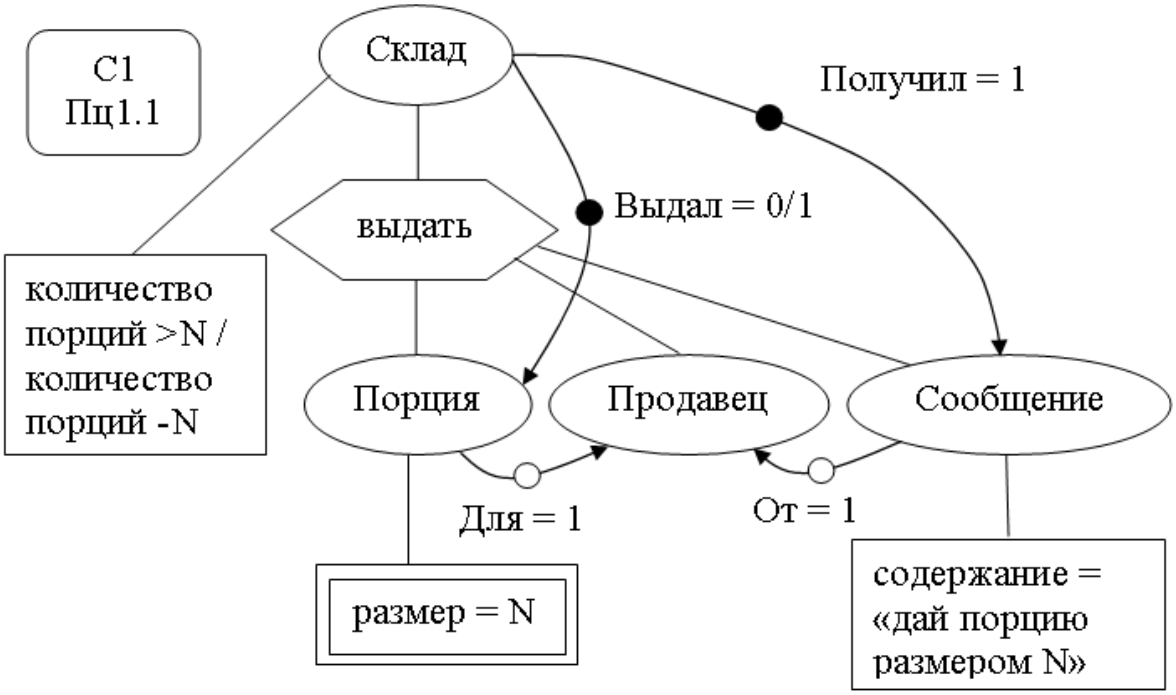

Рис. 27

А когда продавец обращается к складу с таким запросом? Когда он получил заказ от покупателя, а порции у него нет (рис. 28):

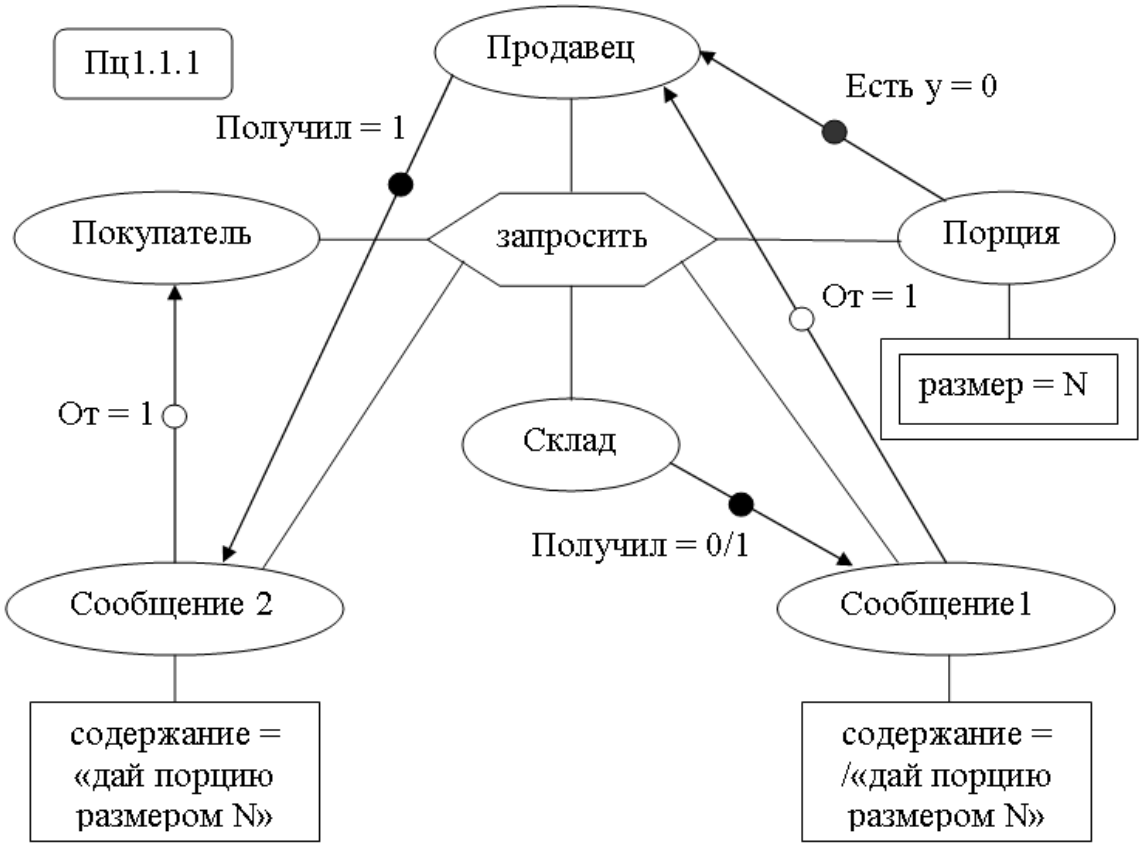

Рис. 28

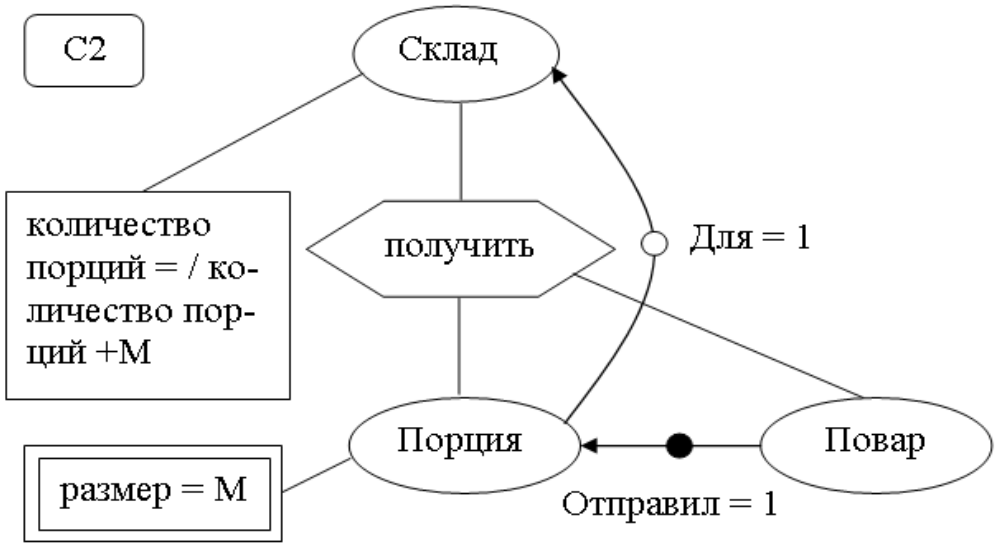

Рис. 29

Продавец не накапливает у себя порции еды, пока их никто не

заказывает, поэтому «Продавец запросить склад» является обязательным поддействием для «Склад выдать порцию», а эта структура, в свою очередь, - поддействие для «Продавец получить порцию».

А откуда порция берется на складе? Склад получает ее от Повара (см. рис. 29). У склада в данной задаче всего два действия: «выдать порцию» и «получить порцию», описывающие его назначение. Это действия одного уровня и могут рассматриваться как две корневые структуры.

Для повара действие «отправить порцию» является корневой схемой. Повар может отправить порцию еды на склад, если он ее приготовил (рис. 30):

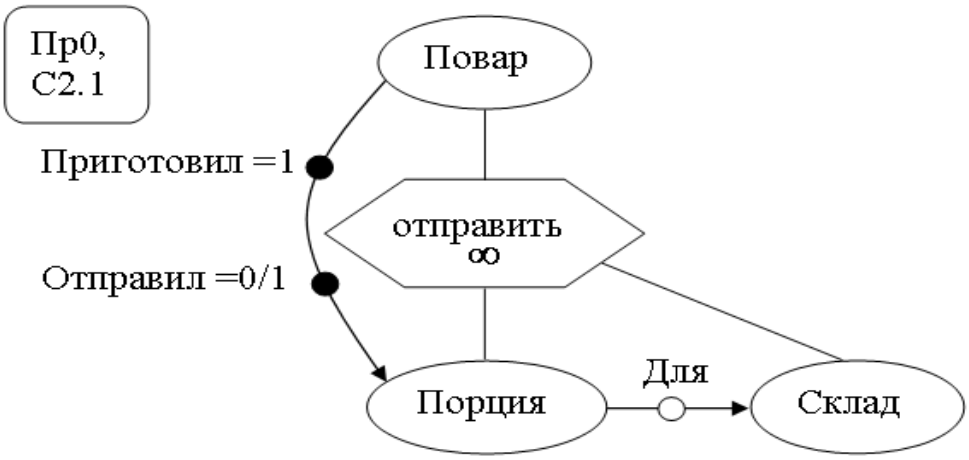

Рис. 30

Что нужно знать повару, для того, чтобы преступить к готовке? Если абстрагироваться от прочих ресурсов, то это будет скорость работы (рис. 31):

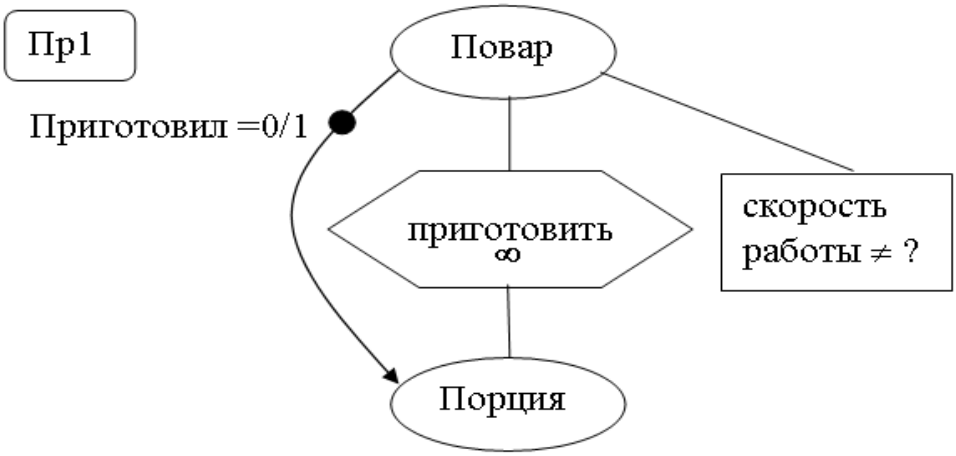

Рис. 31

Каким образом повар может определить, с какой скоростью ему готовить? Он должен посмотреть, сколько порций имеется на складе. Допустим, если осталось от 5 и меньше, то надо готовить быстро, а если нет, то можно готовить не спеша (рис. 32, 33):

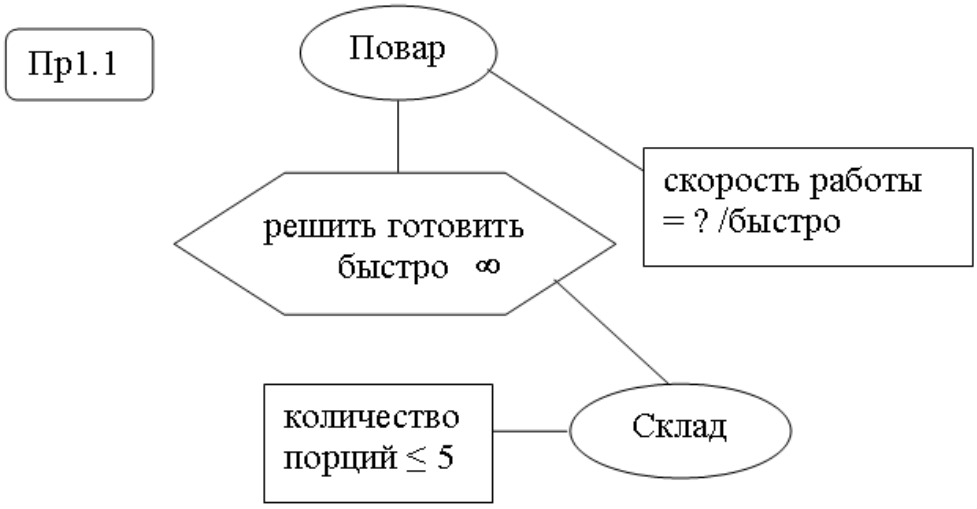

Рис. 32

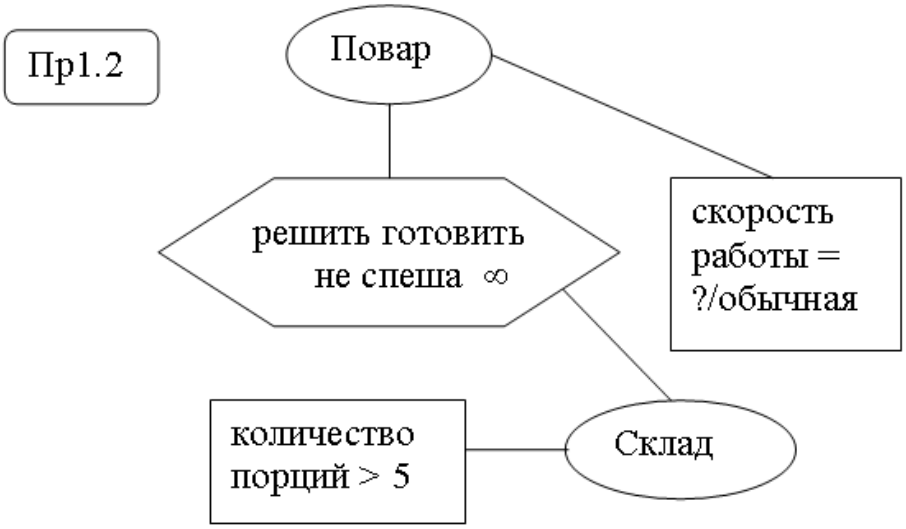

Рис. 33

Мы рассмотрели действия, относящиеся непосредственно к процессу выдачи покупателю желаемой порции. Но получив ее, покупатель должен еще отойти от продавца и покинуть фастфуд. Этому соответствуют два последействия схемы «Покупатель получить порцию». Действие «отойти от продавца» уже встречалось на рис. 18, где второй номер указывает на его участие в по-

лучении порции. Схема для действия «Покупатель покинуть фастфуд» показана на рис. 34:

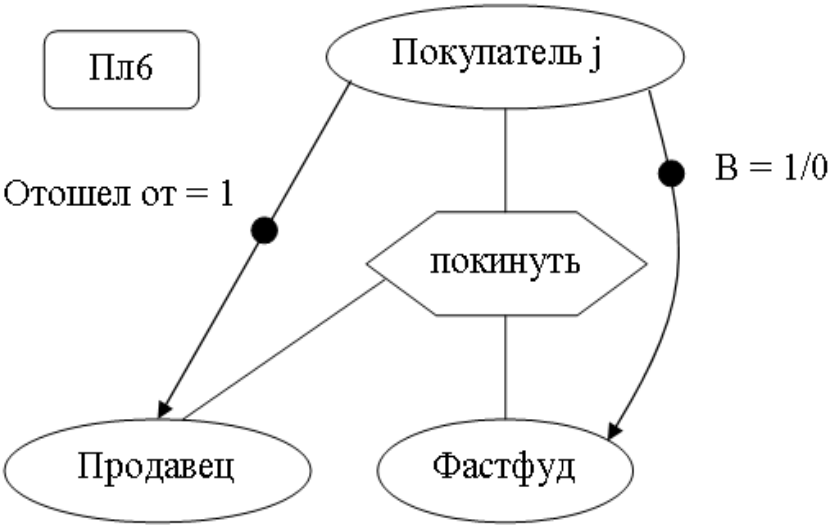

Рис. 34

Проследив по КСД за информационным обменом между агентами, можно построить структурную схему МАС. Условная структурная схема моделируемого предприятия показана на рис. 35:

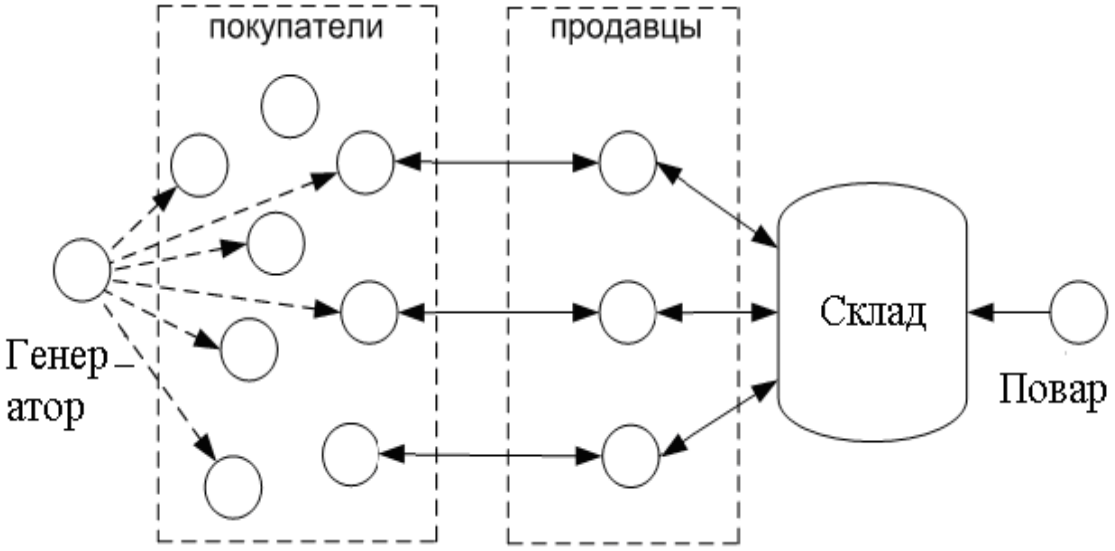

Рис. 35

Концептуальный анализ завершен. Теперь мы можем построить навигационную карту, помогающую проследить ход нашего концептуального анализа (см. рис. 36), и выписать концептуальную модель.

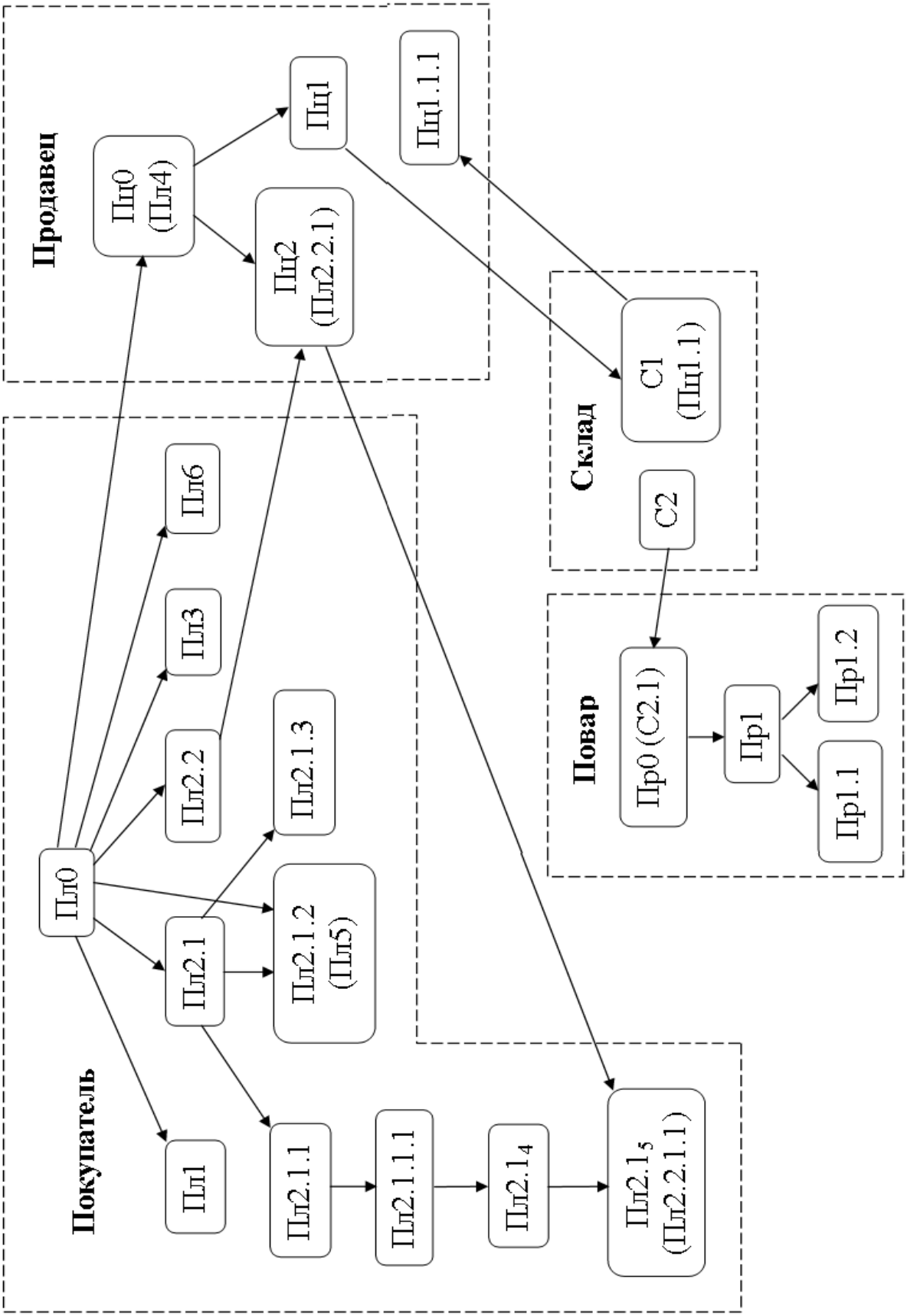
Продавец
Пц0
(Пл4)
Пц1
Пц2
(Пл2.2.1)
Пц1.1.1
Покупатель
Пл0
Пл1
Пл2.1
Пл2.2
Пл3
Пл6
Пл2.1.1
Пл2.1.2
(Пл5)
Пл2.1.3
Пл2.1.1.1
Пл2.1.4
Пл2.1.5
(Пл2.2.1.1)
Склад
С1
(Пц1.1)
С2
Повар
Пр0 (С2.1)
Пр1
Пр1.1
Пр1.2

Рис. 36

5.2.3 Концептуальная модель задачи «Фастфуд»

A = {Повар, Склад, Продавец, Покупатель}.
Классы: {Повар, Продавец, Покупатель} ∈ Люди;
X = {Порция, Очередь, Сообщение, Фастфуд}.
R = {В(Покупатель, Фастфуд),
Приготовил(Повар, Порция),
ОтправилДля(Повар, Порция, Склад);
ВыдалДля(Склад, Порция, Продавец);
ПолучилОт(Склад, Сообщение, Продавец);
ПолучилОт(Продавец, Сообщение, Покупатель);
ПолучилОт(Покупатель, Сообщение, Покупатель);
ПолучилОт(Люди, Сообщение, Покупатель);
ЕстьУ(Порция, Продавец);
ЕстьУ(Порция, Покупатель);
Перед(Покупатель, Продавец);
СтоитЗа($Покупатель_i$, $Покупатель_j$);
ОтошелОт(Покупатель, Продавец);
Выбрал(Покупатель, Очередь);
ЗнаетО(Покупатель, Очередь[размер])
};
C = { Повар: (скорость работы [строковый тип: «быстро», «обычная»]);
Склад: (количество порций [числовой тип]);
Сообщение: (содержание [строковый тип: «дай порцию размером N», «кто крайний»]);
Порция: (размер [числовой тип]);
Продавец: (занят [логический тип]; выручка [числовой тип]);
Покупатель: (позиция в очереди [числовой тип]);
Очередь: (размер [числовой])
};

G = {Отправить, Приготовить, Решить готовить быстро, Решить готовить не спеша, Получить, Выдать, Запросить, Обратиться к, Подойти, отстояв очередь к, Подойти без очереди к, Продвинуться, Занять за, Отойти, Выбрать, Ответить, Спросить, Войти в, Покинуть}.

5.2.4 Описание вспомогательных алгоритмов

Как было сказано ранее, для работы модели могут потребоваться агенты, выполняющие сервисные функции по обеспечению работы основной части МАС.

Согласно результатам концептуального анализа клиенты должны появляться в фастфуде и уходить из него, однако исследование того, почему и когда они приходят, не лежит в рамках нашей задачи. Поэтому в нашем случае целесообразно завести агента «генератор клиентов», который должен отвечать за порождение случайно распределенного потока клиентов, задание им изначальных параметров, завершение их работы, вывод сообщения о конце моделирования.

Алгоритм его работы тривиален, поэтому мы не будем рисовать для него блок схему. Желающие могут восстановить ее по текстовому описанию самостоятельно (в отчете по лабораторным необходимые блок-схемы должны быть).

Генерация клиентов происходит в цикле со счетчиком через случайные (в пределах заданного диапазона) промежутки времени. Для каждого агента предварительно конструируется его имя путем присоединения к слову «customer» текущего значения счетчика. Далее создается агент-покупатель и ему посылается сообщение, в котором передается величина заказа, который тому предстоит сделать.

Завершение работы покупателей происходит по их запросу, и при этом подсчитывается их количество. Если количество сгенерированных покупателей равно количеству завершивших работу, то выводится сообщение о конце моделирования со статистикой выручки у продавцов.

5.2.5 Программная реализация

Далее, руководствуясь полученными концептуальными структурами, можно переходить к программированию планов. При этом следует иметь в виду, что полученные результаты концептуального анализа хоть и являются главным ориентиром, но по ним нельзя сказать какое точно количество планов у

нас получится. Это объясняется тем, то некоторые концептуальные структуры могут распадаться на несколько планов, а некоторые могут объединяться в один. Также возможно, что некоторой группе КСД будет соответствовать группа планов.

Например, действия повара Пр0(С2.1) и Пр1 последовательны, безальтернативны и Пр0(С2.1) зависит только от результатов Пр1. Поэтому они могут быть объединены в один общий план. Объединение в данном случае позволит достичь следующего:

- немного сэкономить память, убрав из базы убеждений агента «повар» отношение «приготовил»;
- немного ускорить работу модели, поскольку действия повара цикличны, а мы, таким образом, избавляем интерпретатор Jason от необходимости обработки одного промежуточного события, связанного с обслуживаем отношения «приготовил».

Указанные выгоды пренебрежимо малы, если речь едет о такой небольшой модели, как наша. Однако могут стать существенным осложняющим фактором в больших моделях.

С другой стороны, условия действия Пр1 формируются альтернативными действиями Пр1.1 и Пр1.2, поэтому Пр1.1 и Пр1.2 порождают отдельные планы.

Кроме того, мы проводили анализ, предполагая, что все агенты действуют во времени последовательно. Однако это в большинстве случаев не так, и агенты действуют параллельно. Поэтому среди них возможны конфликты за ресурсы. Подобные конфликты очень трудно предсказать на этапе концептуального анализа, поэтому планы для обработки подобных случаев приходится разрабатывать в процессе тестирования и отладки программы.

Скоро мы рассмотрим тексты программ для всех агентов, а пока в таблице 4 показана взаимосвязь концептуальных структур действий с полученными в результате реализации планами. Каждый план снабжен меткой, образованной от имени агента и порядкового номера плана. Это сделано не только для присвоения планам аннотаций, но и для возможности ссылаться на них в тексте.

Таблица 4

Метка структуры	Метка плана	Комментарий
Пр0(С2.1) Пр1	@cc1	Пр0(С2.1) и Пр1 последовательны, безальтернативны и Пр0(С2.1) зависит только от результатов Пр1.
Пр1.1	@cc2	
Пр1.2	@cc3	
---	@cc4	Блокировка ненужных сообщений
С1(Пц1.1)	@st1 @st2	С1(Пц1.1) обусловлено достаточностью еды на складе. Однако по логике задачи (при наличии собственного повара) мы не можем не обслужить клиента даже, если на складе кончилась еда. Поэтому @st2 откладывает обработку запроса продавца до тех пор, пока повар не приготовит достаточное количество еды.
С2	@st3	
----	@st4	Блокировка ненужных сообщений
Пц0(Пл4) Пц1	@s5	Отношение «Есть у» не стало убеждением, поскольку еда не задерживается у продавца и при получении со склада сразу передается покупателю (смотри также @cc1).
Пц1.1.1	@s3	
Пц2 (Пл2.2.1)	@s1	
---	@s4	План предназначен для разрешения конфликта, когда несколько покупателей обращаются к одному продавцу с запросом на обслуживание. Не следует непосредственно из концептуального анализа
----	@s2	Блокировка ненужных сообщений
Пл0	@s5 @c15 @c16	Процесс передачи порции от продавца к покупателю начинается в плане @s5, а заканчивается в планах @c15, @c16, где покупатель отходит и покидает фастфуд.

Пл1	@g1	
Пл2.1 Пл2.1.3	@c13 @c14 @c17	Покупатель оказывается перед продавцом в результате продвижения в очереди. Продвижение реализуется по-разному, в зависимости от того, стоит покупатель в конце очереди, в середине, или вторым.
Пл2.1.1	@c7 @c12	Покупатель занимает очередь за другим покупателем. @C12 просто выводит информационное сообщение, подтверждающее занятие очереди. При реализации потребовалось также использовать и обратную форму отношения «стоит за», т.е. «стоит перед».
Пл2.1.1.1	@c4 @c5	Прежде чем выбирать, нужно собрать информацию обо всех очередях.
Пл2.1$_4$	@c2 @c3	Если покупатель не крайний в своей очереди, то он молчит при помощи плана @c3
Пл2.1$_5$ (Пл2.2.1.1)	@c1	
Пл2.1.2 (Пл5)	@c15 @c16	Реализация различается в зависимости от того, последний это покупатель в очереди или нет.
Пл2.2	@c6	
Пл3	@c10	
Пл6	@c15 @c16 @g3	Процесс завершения работы агента-покупателя инициируется в планах @c15 и @c16, а реализуется в плане @g3
----	@c8 @c9 @c11	Разрешение конфликта при попытке занять очередь одновременно с другим покупателем. Не следуют непосредственно из концептуального анализа.

В нижеследующих программных кодах каждый план оформлен в виде таблицы, в левой части которой дается нумерация строк плана, чтобы можно было на них ссылаться в комментариях, в нижней части сами комментарии, и остальное место занимает содержание плана. Это сделано исключительно в целях изложения материа-

ла в данном пособии и не означает, что в виде таблиц планы оформляются и в средах разработки. Начальные убеждения и цели в связи с их простой структурой оформлены также как и в средах разработки. Комментарии к планам даются поначалу подробные, далее только относительно важных или новых моментов.

За дополнительной информацией о работе отдельных функций можно обращаться в справочник, расположенный в конце пособия, а также к первоисточникам [2].

5.2.5.1 Программа агента «Генератор клиентов»

(файл «generator.asl»)

1. База убеждений

maxDelay(500). // максимальная величина паузы между
//порождением 2х клиентов.
maxAgentCounter(10). // максимальное количество
//порождаемых клиентов.
agentCounter(0). // счетчик созданных клиентов.
completeCounter(0). // счетчик клиентов, завершивших свою
//работу.
sellers([seller1, seller2, seller3]). //список продавцов
!start. // начальная цель, инициализирующая работу генератора.

2. Планы по достижению целей (табл. 5)

Таблица 5

1	**@g1** **+!start: agentCounter(AC) & maxAgentCounter(MAC) & AC<MAC <-**
2	-+agentCounter(AC+1);
3	?maxDelay(MD);
4	?agentCounter(C);
5	.wait(math.round(math.random(MD)));
6	.concat("customer(",C,")",Name);
7	.create_agent(Name, "customer.asl");
8	V=math.round(math.random(3))+1;
9	.send(Name, tell, orderValue(V));

10	!!start.

Если еще не создано заданное количество клиентов, то данный план рекурсивно порождает клиентов и определяет им параметр «желаемое количество порций» (orderValue), выдерживая случайную паузу. Строка №1 определяет условия активации плана. План настроен на перехват события возникновения цели !start, после чего проверяются контекстные ограничения. В них сначала происходит конкретизация переменных АС и МАС посредством считывания из базы убеждений указанных предикатов, потом их сравнение. В строке 2 увеличивается счетчик созданных клиентов. В строках 3 и 4 происходит считывание из базы убеждений значений максимально возможной паузы и текущего значения счетчика клиентов. В строке 5 выдерживается случайная пауза, величина которой вычисляется через обращение к функциям генерации случайных чисел и округления, определенных в модуле math. В строке 6 из отдельных фрагментов путем конкатенации собирается имя для порождаемого клиента. Это имя помещается в переменную Name. В строке 7 происходит непосредственное создание нового клиента с заданным именем, который работает по заданной программе, хранящейся в указанном файле. В строке 8 происходит вычисление очередной случайной величины, которая далее рассматривается как объем заказа клиента. В строке 9 генератор сообщает только что созданному клиенту убеждение, в котором указан объем заказа. Строка 10 рекурсивно вызывает этот же самый план, причем в базе намерений создается для него новый стек (т.к. используется оператор «!!»).

1	**@g2 +!start** <-true.

План «заглушка», который срабатывает, когда перестают выполняться контекстные ограничения предыдущего плана.

1	**@g3[atomic]**
2	**+!finishMe[source(Agent)] <-**
3	?completeCounter(C);
4	-+completeCounter(C+1);
5	.kill_agent(Agent).

План завершает работу программного агента (клиента) по его запросу. Строка №1 задает плану метку «g3» и приписывает к

нему аннотацию «atomic», говорящую о том, что выполнение данного плана не должно прерываться обработкой других намерений. Строка №2 определяет условие активации плана. Здесь это событие возникновение цели «!finishMe», которая поступает от агента «Agent», о чем и говориться в аннотации «source». В строках 3 и 4 ведется подсчет количества завершивших работу клиентов. В строке 5 вызывается функция завершения работы агента.

1	**@g4[atomic]**
2	**+completeCounter(C): maxAgentCounter(X)& X==C <-**
3	.print("SIMULATION COMPLETE");
4	?sellers(S);
5	for (.member(Y,S))
6	{ .send(Y, askOne, salesProceeds(Z), salesProceeds(Z));
7	.print("The sales proceeds of ",Y," is ",Z) }.

План срабатывает в конце процесса работы МАС. Он обрабатывает событие изменения счетчика завершенных агентов и выполняется, когда количество завершенных агентов становится равно количеству созданных. В строке 4 из базы убеждений считывается список продавцов. Строка 5 определяет заголовок цикла. Цикл выполняется, пока функция «.member» возвращает true. Эта функция, применяемая в цикле подобным образом, последовательно перебирает элементы списка и помещает их в переменную Y. Она возвращает true до тех пор, пока список не кончится. Строки 6 и 7 описывают тело цикла. В строке 6 генератор запрашивает у продавца Y содержание убеждения «salesProceeds(Z)» и ожидает ответа. После получения ответа в строке 7 выводится информация о выручке данного продавца.

1	**@g5** **+!who_is_last<-**true.

План «заглушка», обеспечивающий нейтральную реакцию генератора при получении цели «who_is_last», поступающей всем агентам от покупателя. Иначе говоря, при возникновении данной цели она считается автоматически достигнутой без выполнения каких-либо действий.

5.2.5.2. Программа агента «Клиент»

(файл «customer.asl»)

1. База начальных убеждений

sellers([seller1, seller2, seller3]). //список продавцов.
!toPutOnQueue. // начальная цель – встать в очередь.

2. Планы по достижению целей (табл. 6)

Таблица 6

1	**@c1 +!toPutOnQueue <-**
2	.wait(200);
3	.print("Hello! Who is last?");
4	.broadcast(achieve, who_is_last).
Данный план, в соответствии с изначально заданной целью, срабатывает первым, сразу после создания клиента. В плане выводится приветственное сообщение и всем агентам МАС (в строке №4) рассылается запрос «кто крайний?». Конечно, с рациональной точки зрения совсем не нужно рассылать этот запрос всем агентам, поскольку он важен только для клиентов в очередях. Но, с одной стороны, в жизни, когда вы в фастфуде громко спрашиваете «кто крайний?», Вас вполне может услышать повар. Просто он не реагируют на Ваш запрос, поскольку это не в его обязанностях. А с другой стороны, без модели физической среды нет иного способа получить эту информацию, поскольку только что созданный агент-клиент вообще не имеет никакого представления о наличии и расположении других клиентов В строке №2 выдерживается небольшая пауза, необходимая для того, чтобы после порождения клиента, прежде, чем он начнет действовать, до него успело дойти сообщение от генератора с количеством покупаемой пищи. Если эту строку убрать из плана, то при запуске модели, время от времени, будет возникать ошибка невозможности реализовать намерения.	
1	**@c2[atomic]**
2	**+!who_is_last[source(Customer)]:** **.all_names(A) & .substring(Customer, A) & inFront(_)** **& not behind(_) <-**

3	?numberInQueue(N);
4	.my_name(MyName);
5	.print("I last, ",Customer);
6	.send(Customer,tell,queue(MyName,N)).

Если данный клиент получил от другого клиента запрос «кто крайний?», и при этом в его базе убеждений присутствует убеждение, что он за кем-то занимал (inFront), и отсутствует убеждение, что за ним кто-то занимал (behind), то он сообщает запросившему клиенту убеждение «queue», в котором передает свое имя и собственный номер в очереди (строка 6).

Обратите внимание, что в строке 2 путем комбинации функций «.all_names(A) & .substring(Customer, A)» проверяется дополнительное условие существования отправителя запроса «кто крайний?» в системе. Это необходимо потому, что неизвестно точно когда будет обработан полученный запрос. Возможны случаи, когда за прошедшее до момента обработки время клиент-отправитель успеет решить свою задачу и будет завершен.

1	**@c3 +!who_is_last <-** true.

Во всех остальных случаях, т.е. когда данный клиент не последний в очереди, при получении запроса «кто крайний?» клиент не реагирует на данный запрос.

1	**@c4[atomic]**
2	**+queue(Agent,N): .count(queue(_,_),3) <-**
3	!selectQueue(LastInQueue);
4	!setInQueue(LastInQueue);
5	.abolish(queue(_,_)).

Если данный клиент получил в свою базу убеждений 3 предиката формата «queue(_,_)», т.е. собрал информацию о 3-х возможных очередях в соответствии с количеством продавцов, то он должен выбрать подходящую очередь и занять в ней место (о чем ставятся соответствующие подцели в строках 3 и 4). Передаваемая при вызове подцелей переменная «LastInQueue» изначально является свободной переменной. Она необходима для передачи информации между двумя подпланами. После занятия места в очереди первичную информацию о размерах очередей можно забыть (строка 5).

1	**@c5[atomic]**
2	**+!selectQueue(LastInQueue) <-**
3	.findall(N,queue(_,N),List);
4	.min(List,MinQueueSize);
5	?queue(LastInQueue,MinQueueSize).

В плане осуществляется выбор наиболее короткой очереди. В строке 3 вызывается функция поиска всех возможных вариантов означиваний переменной N в убеждениях со структурой «queue(_,N)». Найденные варианты помещаются в список в переменную List. Далее в этом списке находится минимальный элемент (строка 4) и помещается в переменную «MinQueueSize», после чего по значению этого элемента определяется имя агента, сообщившего минимальную длину очереди и помещается в «LastInQueue» (строка 5).

Обратите внимание, что переменная «LastInQueue» передается в план как свободная и только в строке 5 принимает значение, которое и возвращается в родительский план. После чего из родительского плана это значение передается в следующий подплан.

1	**@c6[atomic]**
2	**+!setInQueue(LastInQueue):sellers(S)& .member(LastInQueue, S) <-**
3	+inFront(LastInQueue);
4	.print("I make a queue to ",LastInQueue,".");
5	-+numberInQueue(1).

В плане осуществляется постановка данного клиента в очередь в случае, если клиент в этой очереди оказался первым. При этом имя продавца уже определено на этапе выбора очереди и хранится в переменной «LastInQueue». Клиент запоминает, что перед ним стоит выбранный ранее продавец (строка 3) и запоминает, что он первый в созданной очереди к этому продавцу (строка 5).

1	**@c7[atomic]**
2	**+!setInQueue(LastInQueue):.all_names(A)& .substring(LastInQueue,A)&sellers(S)& not .member(LastInQueue, S) <-**
3	.my_name(MyName);
4	?queue(LastInQueue,N);

5	.print("I select a queue behind ",LastInQueue,". The queue size is ",N);
6	.send(LastInQueue,tell,behind(MyName));
7	+inFront(LastInQueue);
8	-+numberInQueue(N+1).

В плане происходит постановка клиента в очередь, если данный клиент вынужден занимать ее за другим клиентом. При этом дополнительно проверяется существование этого другого клиента, т.к. возможен случай, что тот ответил и завершил работу. Имя другого клиента уже определено на этапе выбора очереди и хранится в переменной «LastInQueue». Данный клиент сообщает другому клиенту, что он за ним занимает (строка 6), сам запоминает за кем он стоит (строка 7), а также запоминает свой номер в очереди (строка 8).

1	**@c8[atomic]**
2	**+!setInQueue<-**
3	!toPutOnQueueOnceMore.

Данный план срабатывает в случаях, когда не срабатывают предыдущие два плана. Очевидно, что попытка занять очередь не удалась по причине завершения работы крайнего в этой очереди. Следовательно нужно попытаться занять очередь еще раз.

1	**@c9[atomic]**
2	**+!toPutOnQueueOnceMore<-**
3	.drop_all_desires;
4	.print("OK, I ask once again: WHO IS LAST ???");
5	.broadcast(achieve, who_is_last).

План реализует еще одну попытку занять очередь, однако теперь надо предварительно почистить память от намерений и убеждений, оставшихся с предыдущей попытки (строка 3).

1	**@c10[atomic]**
2	**+inFront(Agent): sellers(S)& .member(Agent, S) <-**
3	?orderValue(X);
4	.send(Agent,achieve,serveMe(X));
5	.print("Now I must be served by ",Agent, "! My order is ",X).

План обрабатывает событие возникновения убеждения, что перед данным клиентом кто-то стоит. Проверяется, является ли этот кто-то продавцом. Если да, то план срабатывает, клиент просит продавца выдать ему порцию еды размером «X» (строка 4).	
1	**@c11[atomic]**
2	**+behind(X): .count(behind(_), 2) <-**
3	.print("No, ",X," behind me occupied");
4	.send(X,achieve,toPutOnQueueOnceMore);
5	.abolish(behind(X)).
План предназначен для обработки конфликтной ситуации, когда у данного агента формируются убеждения, что за ним пытаются занять очередь двое. Тогда агент просит второго занимавшего попытаться занять очередь еще раз и забывает о его существовании (строки 4, 5).	
1	**@c12 +behind(X)<-**
2	.print("OK, ",X," you are behind me...").
Если не срабатывает предыдущий план, то данный агент может позволить агенту X занять за ним очередь. Иначе говоря, убеждение «behind(X)» можно оставить в базе убеждений данного агента. Обратите внимание, что этот план нельзя менять местами с предыдущим, т.к. благодаря особенностям работы процедуры выбора конфликтующих планов, произойдет нарушение в логике работы модели и очереди превратятся в деревья.	
1	**@c13[atomic]**
2	**+!next:behind(X) <-**
3	?numberInQueue(N);
4	-+numberInQueue(N-1);
5	.send(X,achieve,next).
План моделирует продвижение данного клиента в очереди, при условии, что за ним еще кто-то стоит. Клиент уменьшает значение своего счетчика очереди и посылает сзади стоящему клиенту уведомление о продвижении.	
1	**@c14[atomic]**
2	**+!next:not behind(_) <-**
3	?numberInQueue(N);

4	-+numberInQueue(N-1).
План моделирует продвижение данного клиента в очереди, при условии, что за ним никто не стоит. Клиент просто уменьшает значение своего счетчика очереди.	
1	**@c15[atomic]**
2	**+meal(X)[source(Seller)]: behind(Y) <-**
3	.print("I got meal from ", Seller,". Who is next !? (",Y,")");
4	.send(Y,achieve,seller(Seller));
5	.send(Y,achieve,next);
6	.send(generator,achieve,finishMe).
План моделирует получение клиентом заказа от продавца при условии, что за ним кто-то стоит. Обслуженный клиент уходит (просит генератора клиентов завершить его работу, строка 6), при этом сообщает имя продавца, обслуживающего данную очередь, следующему клиенту (строка 4), и инициирует продвижение очереди (строка 5).	
1	**@c16[atomic]**
2	**+meal(X)[source(Seller)]: not behind(Y) <-**
3	.print(" I got meal from ", Seller,". Our queue is empty !!!");
4	.send(generator,achieve,finishMe).
План аналогичен предыдущему, только при условии, что за клиентом никто не стоит. Обслуженный клиент просто уходит.	
1	**@c17[atomic]**
2	**+!seller(Seller) <-**
3	-+inFront(Seller).
План подменяет убеждение о впередистоящем клиенте на убеждение о впередистоящем продавце. Срабатывает, если данный клиент является вторым в очереди, когда обслуженный перед ним клиент уходит и сообщает ему имя продавца.	

5.2.5.3 Программа агента «Продавец»

(файл «seller.asl»)

1. База начальных убеждений

queue(0). //Флаг отсутствия очереди к данному продавцу
salesProceeds(0). // Выручка продавца.

maxServeDelay(2000). //Время обслуживания одного клиента

2. Планы по достижению целей (табл. 7)

Таблица 7

1	**@s1** **+!who_is_last[source(Customer)]: queue(0) <-**
2	.my_name(MyName);
3	.print("I HAVE NO QUEUE, ",Customer);
4	.send(Customer, tell, queue(MyName,0)).
Если к продавцу нет очереди, то при получении запроса клиента «кто крайний», продавец называет себя и говорит, что свободен (строка 4).	
1	**@s2** **+!who_is_last<-true.**
Если к продавцу есть очередь, то при получении запроса клиента «кто крайний» он молчит.	
1	**@s3** **+!serveMe(X)[source(Customer)]: not capturedBy(_) <-**
2	+capturedBy(Customer);
3	.print(" I'm captured by ", Customer);
4	.abolish(queue(0));
5	?maxServeDelay(Delay);
6	.wait(Delay);
7	.send(store, achieve, moreMeal(X)).
Если продавец не занят, то при получении от клиента запроса на обслуживание, он его обрабатывает. При этом продавец устанавливает флаг занятости клиентом (строка 2), сбрасывает флаг отсутствия очереди, выполняет задержку, имитирующую время облуживания (строки 5, 6), и посылает запрос в хранилище (строка 7).	
1	**@s4** **+!serveMe(X)[source(Customer)]: capturedBy(_) <-**
2	.print("I am busy, ",Customer," go to queue...");
3	.send(Customer, achieve, toPutOnQueueOnceMore).

Если продавец занят, и получает еще один запрос на обслуживание, то это означает возникновение конфликта. Поэтому продавец сообщает отправителю запроса, что необходимо занять очередь.	
1	**@s5** **+!takeMeal(X) <-**
2	?capturedBy(Customer);
3	?salesProceeds(Y);
4	K=Y+X*10;
5	-+salesProceeds(K);
6	.send(Customer, tell, meal(X));
7	.abolish(capturedBy(_)).
План моделирует акт продажи порции, объемом X (строка 6), и получения за нее выручки, размером 10*X. В процессе работы модели выручка накапливается (строки 3, 4, 5). После обслуживания данного клиента продавец освобождается для следующего (строка 7).	

5.2.5.4 Программа агента «Хранилище»

(файл «store.asl»)

1. База начальных убеждений

mealsInStore(6). // Начальное количество порций в хранилище.

2. Планы по достижению целей (табл. 8)

Таблица 8

1	**@st1** **+!moreMeal(X)[source (Seller)]:** **mealsInStore(MIS) & MIS>=X<-**
2	-+mealsInStore(MIS-X);
3	.print(" There are ", MIS-X, " meals in storage");
4	.wait(1000);
5	.send(Seller, achieve, takeMeal(X)).

План моделирует выдачу продавцу со склада порции еды размером X, если получен соответствующий запрос и на складе есть еда в достаточном количестве.	
1	**@st2** **+!moreMeal(X)[source (Seller)]:** **mealsInStore(MIS) & MIS<X<-**
2	.wait(1000);
3	!moreMeal(X)[source (Seller)].
Если в настоящее время на складе нет еды, то обработка запроса откладывается на некоторое время, в течение которого, скорее всего, еда на складе появится. План рекурсивный. Обратите внимание, что при рекурсивном вызове в аннотации сохраняется имя продавца (строка 3), приславшего изначальный запрос. Если аннотацию убрать, то на очередной итерации имя продавца потеряется и источником цели «!moreMeal(X)» будет сам склад, что приведет к ошибкам.	
1	**@st3** **+!takeNewMeal <-**
2	?mealsInStore(MIS);
3	-+mealsInStore(MIS+1).
План моделирует получение порции еды от повара.	
1	**@st4** **+!who_is_last<-**true.
План «заглушка», обеспечивающий нейтральную реакцию склада при получении цели «who_is_last», поступающей всем агентам при создании очередного клиента.	

5.2.5.5 Программа агента «Повар»

(файл «cook.asl»)

1. База начальных убеждений

timeOfProduce(3000). // Время производства одной порции.
!produceMeals. // Цель – производство еды.

2. Планы по достижению целей (табл. 9)

Таблица 9

1	**@cc1** **+!produceMeals <-**
2	?timeOfProduce(TOP);
3	.wait(TOP);
4	.send(store, achieve, takeNewMeal);
5	.send(store, askOne, mealsInStore(Rest), mealsInStore(Rest));
6	!setProduceSpeed(Rest);
7	!!produceMeals.
Рекурсивный план, моделирующий производство одной порции еды за условное время. Произведенная порция отправляется на склад. Тут же отслеживается текущее количество еды на складе и вызывается подзадача корректировки скорости производства (строки 5, 6). Выполнение операции в строке 5 приостанавливает обработку плана до получения ответа от хранилища. В строке 7 выполняется рекурсивный вызов с выделением нового стека намерений.	
1	**@cc2** **+!setProduceSpeed(Rest): Rest >5 <-**
2	-+timeOfProduce(3000).
План выбирает одно из возможных значений скорости производства еды. Если количество порций на складе больше 5, то устанавливается нормальная скорость производства.	
1	**@cc3** **+!setProduceSpeed(Rest): Rest<=5 <-**
2	-+timeOfProduce(300).
План выбирает одно из возможных значений скорости производства еды. Если количество порций на складе от 5 и меньше, то устанавливается высокая скорость производства. Условно считается, что порция готовится за 300 мс модельного времени.	
1	**@cc4** **+!who_is_last<-** true.

План «заглушка», обеспечивающий нейтральную реакцию повара при получении цели «who_is_last», поступающей всем агентам при создании очередного клиента.

5.2.5.6 Файл проекта МАС «fastfood.mas2j»

Для запуска модели в среде Jason не достаточно только программ агентов. Еще необходимо создать файл проекта, в котором будет, как минимум, описана архитектура МАС и указаны работающие в ней агенты. Организация проектного файла подробно рассмотрена в первоисточниках [,раздел 4.3]. В данном начальном пособии мы не будем подробно касаться этого вопроса, а рассмотрим только параметры, использованные в нашем примере.

Итак, в нашем проектном файле находится следующий код.

```
MAS fastfood {
                infrastructure: Centralised
                agents: customer_generator; seller#3; store; cook;
            }
```

После зарезервированного слова «MAS» следует название многоагентной системы «fastfood», файлы которой хранятся на диске в одноименной папке, и далее, в фигурных скобках, следует описание конфигурации МАС.

Поле «infrastructure» предназначено для указания типа разрабатываемой МАС. Значения данного поля могут быть следующие:

- «Centralised»,- означает, что моделирование работы МАС (всех ее агентов) будет происходить на одном компьютере;
- «Saci», - означает, что агенты распределены между различными вычислительными машинами;
- «Jade» - означает, что разрабатываемая МАС интегрируется с платформой JADE.

В поле «agents» через «;» указываются агенты, автоматически создаваемые интерпретатором Jason при запуске моделирования. Эти имена совпадают с именами файлов, в которых хранятся программы агентов. По умолчанию создается по одному экземпляру каждого агента.

Если необходимо завести несколько экземпляров однотипных агентов, то сразу после имени агента ставится символ «#», после которого указывается необходимое число агентов (см. агент «seller»).

Если в процессе моделирования агенты должны появляться и исчезать динамически (как, например, клиенты в рассматриваемом примере), то имена таких агентов ***не указываются*** в поле «agents», а их создание и удаление производится в планах при помощи специальных команд интерпретатора «.create_agent» и «.kill_agent».

Заготовки для файла проекта и файлов агентов создаются автоматически при работе с диалогами среды Jason в процессе создания нового проекта. Для простых задач файл проекта практически не нуждается в ручном корректировании.

5.2.6 Исследование работоспособности МАС

Проведем несколько испытаний построенной модели. Поскольку модель является демонстрационной, то и цель исследования у нас очень простая: установить, смогут ли наши агенты правильно скоординировать свое взаимодействие при разной интенсивности потока заказов. На результаты моделирования будет оказывать влияние случайная задержка до появления в фастфуде очередного клиента, время обслуживания клиентов, а также работа механизма доставки сообщений среды Jason.

Ниже представлены несколько примеров отчетов о моделировании. Каждый отчет это протокол диалога агентов. В начале каждого предложения в квадратных скобках стоит его автор.

Эксперимент №1

Ничего не будем менять в исходных убеждениях агентов. Получаем следующий отчет о моделировании:

[customer(1)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(1)
[seller3] I HAVE NO QUEUE, customer(1)
[seller2] I HAVE NO QUEUE, customer(1)

[customer(1)] Now I must be served by seller2! My order is 1
[customer(1)] I make a queue to seller2.
[seller2] I'm captured by customer(1)
[customer(2)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(2)
[seller3] I HAVE NO QUEUE, customer(2)
[customer(1)] I last, customer(2)
[customer(2)] Now I must be served by seller3! My order is 3
[customer(2)] I make a queue to seller3.
[seller3] I'm captured by customer(2)
[customer(3)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(3)
[customer(1)] I last, customer(3)
[customer(2)] I last, customer(3)
[customer(3)] Now I must be served by seller1! My order is 1
[customer(3)] I make a queue to seller1.
[seller1] I'm captured by customer(3)
[customer(4)] Hello! Who is last?
[customer(2)] I last, customer(4)
[customer(3)] I last, customer(4)
[customer(1)] I last, customer(4)
[customer(4)] I select a queue behind customer(1). The queue size is 1
[customer(1)] OK, customer(4) you are behind me...
[customer(5)] Hello! Who is last?
[customer(2)] I last, customer(5)
[customer(3)] I last, customer(5)
[customer(4)] I last, customer(5)
[customer(5)] I select a queue behind customer(3). The queue size is 1
[customer(3)] OK, customer(5) you are behind me...
[customer(6)] Hello! Who is last?
[customer(2)] I last, customer(6)
[customer(5)] I last, customer(6)
[customer(4)] I last, customer(6)
[customer(6)] I select a queue behind customer(2). The queue size is 1
[customer(2)] OK, customer(6) you are behind me...
[customer(7)] Hello! Who is last?

[customer(5)] I last, customer(7)
[customer(4)] I last, customer(7)
[customer(6)] I last, customer(7)
[customer(7)] I select a queue behind customer(6). The queue size is 2
[customer(6)] OK, customer(7) you are behind me...
[customer(8)] Hello! Who is last?
[customer(5)] I last, customer(8)
[customer(4)] I last, customer(8)
[customer(7)] I last, customer(8)
[customer(8)] I select a queue behind customer(4). The queue size is 2
[customer(4)] OK, customer(8) you are behind me...
[store] There are 5 meals in storage
[store] There are 3 meals in storage
[customer(9)] Hello! Who is last?
[customer(8)] I last, customer(9)
[customer(5)] I last, customer(9)
[customer(7)] I last, customer(9)
[customer(9)] I select a queue behind customer(5). The queue size is 2
[customer(10)] Hello! Who is last?
[customer(5)] OK, customer(9) you are behind me...
[customer(8)] I last, customer(10)
[customer(9)] I last, customer(10)
[customer(7)] I last, customer(10)
[customer(10)] I select a queue behind customer(7).The queue size is3
[customer(7)] OK, customer(10) you are behind me...
[store] There are 3 meals in storage
[customer(1)] I got meal from seller2. Who is next !? (customer(4))
[customer(4)] Now I must be served by seller2! My order is 3
[seller2] I'm captured by customer(4)
[customer(1)] I finished!
[customer(2)] I got meal from seller3. Who is next !? (customer(6))
[customer(6)] Now I must be served by seller3! My order is 2
[seller3] I'm captured by customer(6)
[customer(2)] I finished!
[customer(3)] I got meal from seller1. Who is next !? (customer(5))
[customer(5)] Now I must be served by seller1! My order is 2

[seller1] I'm captured by customer(5)
[customer(3)] I finished!
[store] There are 4 meals in storage
[store] There are 2 meals in storage
[store] There are 0 meals in storage
[customer(4)] I got meal from seller2. Who is next !? (customer(8))
[customer(8)] Now I must be served by seller2! My order is 4
[seller2] I'm captured by customer(8)
[customer(4)] I finished!
[customer(6)] I got meal from seller3. Who is next !? (customer(7))
[customer(7)] Now I must be served by seller3! My order is 2
[customer(6)] I finished!
[seller3] I'm captured by customer(7)
[customer(5)] I got meal from seller1. Who is next !? (customer(9))
[customer(9)] Now I must be served by seller1! My order is 4
[seller1] I'm captured by customer(9)
[customer(5)] I finished!
[store] There are 1 meals in storage
[customer(8)] I got meal from seller2. Our queue is empty !!!
[customer(8)] I finished!
[store] There are 3 meals in storage
[store] There are 0 meals in storage
[customer(7)] I got meal from seller3. Who is next !? (customer(10))
[customer(10)] Now I must be served by seller3! My order is 3
[seller3] I'm captured by customer(10)
[customer(7)] I finished!
[customer(9)] I got meal from seller1. Our queue is empty !!!
[customer(9)] I finished!
[store] There are 4 meals in storage
[customer(10)] I got meal from seller3. Our queue is empty !!!
[generator] SIMULATION COMPLETE
[generator] The sales proceeds of seller1 is 70
[generator] The sales proceeds of seller2 is 80
[generator] The sales proceeds of seller3 is 100
[customer(10)] I finished!

Из отчета видно, что агенты правильно выбирают очередь, размеренно продвигаются в ней и обслуживаются. Когда на складе кончаются продукты, то все ожидают, когда повар приготовит новые. Максимальная очередь образовалась к третьему продавцу, ее размер составил 4.

При повторном моделировании результат, возможно, будет другим. Давайте это проверим.

Эксперимент №2

Еще раз оставим исходные данные без изменений. В консоль выводится отчет о моделировании:

```
[customer(1)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(1)
[seller2] I HAVE NO QUEUE, customer(1)
[seller3] I HAVE NO QUEUE, customer(1)
[customer(1)] Now I must be served by seller3! My order is 1
[seller3]   I'm captured by customer(1)
[customer(1)] I make a queue to seller3.
[customer(2)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(2)
[seller2] I HAVE NO QUEUE, customer(2)
[customer(1)] I last, customer(2)
[customer(2)] Now I must be served by seller2! My order is 2
[customer(2)] I make a queue to seller2.
[seller2]   I'm captured by customer(2)
[customer(3)] Hello! Who is last?
[customer(2)] I last, customer(3)
[customer(1)] I last, customer(3)
[seller1] I HAVE NO QUEUE, customer(3)
[customer(3)] Now I must be served by seller1! My order is 3
[customer(3)] I make a queue to seller1.
[seller1]   I'm captured by customer(3)
[customer(4)] Hello! Who is last?
[customer(2)] I last, customer(4)
[customer(1)] I last, customer(4)
[customer(3)] I last, customer(4)
```

[customer(4)] I select a queue behind customer(3). The queue size is 1
[customer(3)] OK, customer(4) you are behind me...
[customer(5)] Hello! Who is last?
[customer(2)] I last, customer(5)
[customer(4)] I last, customer(5)
[customer(1)] I last, customer(5)
[customer(5)] I select a queue behind customer(1). The queue size is 1
[customer(1)] OK, customer(5) you are behind me...
[customer(6)] Hello! Who is last?
[customer(2)] I last, customer(6)
[customer(5)] I last, customer(6)
[customer(4)] I last, customer(6)
[customer(6)] I select a queue behind customer(2). The queue size is 1
[customer(2)] OK, customer(6) you are behind me...
[customer(7)] Hello! Who is last?
[customer(5)] I last, customer(7)
[customer(6)] I last, customer(7)
[customer(4)] I last, customer(7)
[customer(8)] Hello! Who is last?
[customer(7)] I select a queue behind customer(4). The queue size is 2
[customer(7)] I last, customer(8)
[customer(5)] I last, customer(8)
[customer(4)] I last, customer(8)
[customer(4)] OK, customer(7) you are behind me...
[customer(8)] I select a queue behind customer(4). The queue size is 2
[customer(4)] No, customer(8) behind me occupied
[customer(6)] I last, customer(8)
[customer(8)] OK, I ask once again: WHO IS LAST ???
[customer(7)] I last, customer(8)
[customer(5)] I last, customer(8)
[customer(8)] I select a queue behind customer(5). The queue size is 2
[customer(6)] I last, customer(8)
[customer(5)] OK, customer(8) you are behind me...
[customer(9)] Hello! Who is last?
[customer(8)] I last, customer(9)
[customer(7)] I last, customer(9)

[customer(6)] I last, customer(9)
[customer(9)] I select a queue behind customer(6). The queue size is 2
[customer(6)] OK, customer(9) you are behind me...
[store] There are 5 meals in storage
[store] There are 3 meals in storage
[customer(10)] Hello! Who is last?
[customer(8)] I last, customer(10)
[customer(9)] I last, customer(10)
[customer(7)] I last, customer(10)
[customer(10)] I select a queue behind customer(7).The queue size is3
[customer(7)] OK, customer(10) you are behind me...
[store] There are 0 meals in storage
[customer(1)] I got meal from seller3. Who is next !? (customer(5))
[customer(5)] Now I must be served by seller3! My order is 1
[customer(1)] I finished!
[seller3] I'm captured by customer(5)
[customer(2)] I got meal from seller2. Who is next !? (customer(6))
[customer(6)] Now I must be served by seller2! My order is 3
[seller2] I'm captured by customer(6)
[customer(2)] I finished!
[customer(3)] I got meal from seller1. Who is next !? (customer(4))
[customer(4)] Now I must be served by seller1! My order is 2
[seller1] I'm captured by customer(4)
[customer(3)] I finished!
[store] There are 6 meals in storage
[store] There are 3 meals in storage
[store] There are 1 meals in storage
[customer(5)] I got meal from seller3. Who is next !? (customer(8))
[customer(8)] Now I must be served by seller3! My order is 3
[seller3] I'm captured by customer(8)
[customer(5)] I finished!
[customer(6)] I got meal from seller2. Who is next !? (customer(9))
[customer(9)] Now I must be served by seller2! My order is 2
[seller2] I'm captured by customer(9)
[customer(6)] I finished!
[customer(4)] I got meal from seller1. Who is next !? (customer(7))

```
[customer(7)] Now I must be served by seller1! My order is 2
[seller1]   I'm captured by customer(7)
[customer(4)] I finished!
[store]   There are 1 meals in storage
[store]   There are 0 meals in storage
[customer(8)]   I got meal from seller3. Our queue is empty !!!
[customer(8)] I finished!
[store]   There are 1 meals in storage
[customer(7)] I got meal from seller1. Who is next !? (customer(10))
[customer(10)] Now I must be served by seller1! My order is 4
[seller1]   I'm captured by customer(10)
[customer(7)] I finished!
[customer(9)]   I got meal from seller2. Our queue is empty !!!
[customer(9)] I finished!
[store]   There are 3 meals in storage
[customer(10)]   I got meal from seller1. Our queue is empty !!!
[generator] SIMULATION COMPLETE
[generator] The  sales proceeds of seller1 is 110
[generator] The  sales proceeds of seller2 is 70
[generator] The  sales proceeds of seller3 is 50
[customer(10)] I finished!
```

В данном случае мы можем видеть, что покупатели №7 и №8 пришли практически одновременно и оба попытались занять очередь за покупателем №4. Конфликт разрешает покупатель №4 и выбирает того, кто, на его взгляд, пришел раньше. Это оказался покупатель №7. А покупателю №8 придется еще раз пытаться занять очередь. В результате мы видим, что покупатель №8 занимает за покупателем №5, и, в конце концов, обслуживается у продавца №3.

Эксперимент №3

Уменьшим максимально возможную задержку между приходами клиентов. Для этого заменим в генераторе клиентов убеждение maxDelay(500) на maxDelay(250).

Получаем следующий отчет:

[customer(1)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(1)
[seller2] I HAVE NO QUEUE, customer(1)
[customer(2)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(2)
[seller2] I HAVE NO QUEUE, customer(2)
[seller3] I HAVE NO QUEUE, customer(1)
[seller3] I HAVE NO QUEUE, customer(2)
[customer(1)] Now I must be served by seller3! My order is 2
[customer(1)] I make a queue to seller3.
[seller3] I'm captured by customer(1)
[customer(2)] Now I must be served by seller3! My order is 3
[customer(2)] I make a queue to seller3.
[seller3] I am busy, customer(2) go to queue...
[customer(2)] OK, I ask once again: WHO IS LAST ???
[seller1] I HAVE NO QUEUE, customer(2)
[seller2] I HAVE NO QUEUE, customer(2)
[customer(3)] Hello! Who is last?
[seller1] I HAVE NO QUEUE, customer(3)
[seller2] I HAVE NO QUEUE, customer(3)
[customer(1)] I am last, customer(3)
[customer(3)] Now I must be served by seller2! My order is 2
[customer(3)] I make a queue to seller2.
[seller2] I'm captured by customer(3)
[customer(1)] I am last, customer(2)
[customer(2)] I am last, customer(3)
[customer(2)] Now I must be served by seller2! My order is 3
[customer(2)] I make a queue to seller2.
[seller2] I am busy, customer(2) go to queue...
[customer(2)] OK, I ask once again: WHO IS LAST ???
[customer(3)] I am last, customer(2)
[seller1] I HAVE NO QUEUE, customer(2)
[customer(1)] I am last, customer(2)
[seller1] I'm captured by customer(2)
[customer(2)] Now I must be served by seller1! My order is 3
[customer(2)] I make a queue to seller1.

[customer(4)] Hello! Who is last?
[customer(2)] I am last, customer(4)
[customer(1)] I am last, customer(4)
[customer(3)] I am last, customer(4)
[customer(4)] I select a queue behind customer(3). The queue size is 1
[customer(3)] OK, customer(4) you are behind me...
[customer(5)] Hello! Who is last?
[customer(2)] I am last, customer(5)
[customer(4)] I am last, customer(5)
[customer(1)] I am last, customer(5)
[customer(5)] I select a queue behind customer(1). The queue size is 1
[customer(1)] OK, customer(5) you are behind me...
[customer(6)] Hello! Who is last?
[customer(2)] I am last, customer(6)
[customer(5)] I am last, customer(6)
[customer(4)] I am last, customer(6)
[customer(6)] I select a queue behind customer(2). The queue size is 1
[customer(2)] OK, customer(6) you are behind me...
[customer(7)] Hello! Who is last?
[customer(5)] I am last, customer(7)
[customer(4)] I am last, customer(7)
[customer(6)] I am last, customer(7)
[customer(7)] I select a queue behind customer(6). The queue size is 2
[customer(6)] OK, customer(7) you are behind me...
[customer(8)] Hello! Who is last?
[customer(5)] I am last, customer(8)
[customer(4)] I am last, customer(8)
[customer(7)] I am last, customer(8)
[customer(8)] I select a queue behind customer(4). The queue size is 2
[customer(4)] OK, customer(8) you are behind me...
[customer(9)] Hello! Who is last?
[customer(8)] I am last, customer(9)
[customer(5)] I am last, customer(9)
[customer(7)] I am last, customer(9)
[customer(9)] I select a queue behind customer(5). The queue size is 2
[customer(5)] OK, customer(9) you are behind me...

[customer(10)] Hello! Who is last?
[customer(8)] I am last, customer(10)
[customer(9)] I am last, customer(10)
[customer(7)] I am last, customer(10)
[customer(10)] I select a queue behind customer(7).The queue size is3
[customer(7)] OK, customer(10) you are behind me...
[store] There are 4 meals in storage
[store] There are 2 meals in storage
[customer(1)] I got meal from seller3. Who is next !? (customer(5))
[customer(5)] Now I must be served by seller3! My order is 4
[seller3] I'm captured by customer(5)
[customer(1)] I finished!
[customer(3)] I got meal from seller2. Who is next !? (customer(4))
[customer(4)] Now I must be served by seller2! My order is 2
[seller2] I'm captured by customer(4)
[customer(3)] I finished!
[store] There are 2 meals in storage
[customer(2)] I got meal from seller1. Who is next !? (customer(6))
[customer(6)] Now I must be served by seller1! My order is 3
[seller1] I'm captured by customer(6)
[customer(2)] I finished!
[store] There are 3 meals in storage
[store] There are 1 meals in storage
[customer(5)] I got meal from seller3. Who is next !? (customer(9))
[customer(9)] Now I must be served by seller3! My order is 1
[customer(5)] I finished!
[seller3] I'm captured by customer(9)
[customer(4)] I got meal from seller2. Who is next !? (customer(8))
[customer(8)] Now I must be served by seller2! My order is 2
[seller2] I'm captured by customer(8)
[customer(4)] I finished!
[store] There are 3 meals in storage
[store] There are 1 meals in storage
[customer(9)] I got meal from seller3. Our queue is empty !!!
[customer(9)] I finished!
[customer(8)] I got meal from seller2. Our queue is empty !!!

[customer(8)] I finished!
[store] There are 1 meals in storage
[customer(6)] I got meal from seller1. Who is next !? (customer(7))
[customer(7)] Now I must be served by seller1! My order is 4
[seller1] I'm captured by customer(7)
[customer(6)] I finished!
[store] There are 3 meals in storage
[customer(7)] I got meal from seller1. Who is next !? (customer(10))
[customer(10)] Now I must be served by seller1! My order is 1
[seller1] I'm captured by customer(10)
[customer(7)] I finished!
[store] There are 6 meals in storage
[customer(10)] I got meal from seller1. Our queue is empty !!!
[generator] SIMULATION COMPLETE
[generator] The sales proceeds of seller1 is 110
[generator] The sales proceeds of seller2 is 60
[generator] The sales proceeds of seller3 is 70
[customer(10)] I finished!

Теперь мы видим, что не повезло покупателю №2. Он дважды пытался обратиться к продавцам – сначала к третьему, потом ко второму, но в обоих случая находились те, кто считал, что они пришли раньше. В обоих случаях продавцы разрешали конфликт и неудачливому покупателю приходилось заново искать себе место. И только с третьей попытки ему удалось пробиться к свободному в тот момент продавцу №1.

Проведенные эксперименты иллюстрируют 3 класса наиболее вероятных сценариев развития сюжета и подтверждают, по крайней мере, частичную работоспособность МАС. Но процесс тестирования МАС обычно проходит долго и очень трудно, поскольку еще не достаточно развит инструментарий отладки и поведение самой МАС трудно предсказуемо от запуска к запуску. Поэтому даже в таком простом примере в дальнейшем можно обнаружить непредвиденные результаты (например, нарушение логики при определенных конфликтах). Однако оставим задачу дальнейших исследований данного

примера уважаемому читателю. Это послужит хорошей практикой наблюдения и отладки системы. Исходные коды демонстрационного примера можно получить у авторов пособия или на лабораторных занятиях. О всех замеченных случаях неправильной работы данной модели также просьба по возможности сообщать авторам.

5.2.7 Дополнительные задания

Приведенные здесь задачи ***не являются*** заданиями на лабораторные работы, а предлагаются в целях самостоятельного изучения среды Jason и языка AgentSpeak. Вам предлагается самостоятельно модифицировать разобранный пример и исследовать работу получившихся МАС. В качестве направлений модернизации могут быть следующие.

1. Первое задание очень простое. Измените программу работы повара так, чтобы он сообщал о выбранной им скорости приготовления пищи.

2. Добавьте покупателям свойство «время ожидания», по истечении которого покупатели будут терять терпение и уходить из очереди.

3. Измените способ пополнения хранилища готовыми блюдами. Пусть теперь в фастфуде работают несколько поваров, изначально обладающих разной, но фиксированной скоростью приготовления, которые будут динамически подключаться к приготовлению по мере необходимости.

4. Расширьте ассортимент блюд. Пусть повар может готовить разные блюда, а покупатели, соответственно, выбирать что им нравится на имеющиеся у них средства.

5. Возьмите в качестве исходной модели, модель, описанную в пункте 3, и добавьте в нее учет расходов на оплату труда сотрудникам фастфуда. Исследуйте зависимость получаемой прибыли от количества обслуживающего персонала.

6. Добавьте поварам свойство «уровень мастерства», от которого будет зависеть качество приготавливаемых блюд. Добавьте клиентам какое-либо поведение в зависимости от качества полученной ими еды.

7. Попробуйте спровоцировать конфликт за место в очереди среди более чем 2х покупателей. Пронаблюдайте, правильно ли он будет разрешен, и если нет, внесите исправления в программу.

8. Для простоты в нашем примере количество продавцов фиксировано, и если его изменить, то модель работать не будет. Модернизируйте программу так, чтобы она могла работать с произвольным числом продавцов, которое задается изначально.

9. Возьмите за основу результаты, полученные в предыдущем задании. Сделайте так, чтобы продавцы подключались к работе динамически, например, чтобы при достижении очередью определенного размера, на рабочее место приходил очередной продавец.

10. Измените работу генератора клиентов. По-умолчанию генератор реализует равномерное распределение вероятности прихода клиентов в течении всего времени моделирования. Теперь, предположим, что период моделирования условно соответствует суткам. Известно, что в разное время суток интенсивность прихода клиентов различается. Смоделируйте это изменение интенсивности.

11. Задание повышенной сложности (для владеющих Java).

Посмотрите, как на самом деле уныла наша МАС без модели среды! Мы же хотели смоделировать работу реальной организации, которая находится в физическом пространстве, где перемещаются и принимают решения реальные люди! А что получилось? Без среды наши агенты «слепы» и «неподвижны»! Все что они могут - это «кричать» и надеяться, что им ответят, такие же неподвижные слепцы! Вся их совместная работа это обмен фантазиями о том, как все могло бы происходить в среде и то, если эта среда не оказывает собственного влияния на состояние предметной области.

Изучите соответствующие разделы документации, и запрограммируйте модель среды, в которой у агентов будет «зрение», посредством которого они смогут «видеть» положение других агентов и ресурсов в пространстве. Это позволит сделать информационный обмен между агентами более целенаправленным, а поведение более гибким и подконтрольным законам среды. Реа-

лизуйте логику выбора и продвижения в очереди на основе анализа «зрительной» информации, поступающей из среды.

12. Задание повышенной сложности (для владеющих Java).

Разработайте оконный пользовательский интерфейс для любого варианта модели фастфуда. Интерфейс, как минимум, должен позволять изменять начальные убеждения агентов, а также визуализировать сам процесс моделирования.

Заключение

В данном пособии была сделана первая попытка познакомить отечественного студента с языком AgentSpeak и системой Jason [2], которые достаточно популярны на западе, особенно в образовательной и научно-исследовательской среде. Пособие не претендует на полноту охвата затронутых вопросов и фактически содержит только материал необходимый и достаточный для выполнения лабораторных работ студентами, не владеющими Java. Аналогичных публикаций на русском языке авторами обнаружено не было.

В качестве заключения, хотелось бы привести ряд напутственных предостережений о некоторых подводных камнях, возникающих на пути разработчика МАС [3, стр. 251]. Эти предостережения адресованы тем, кто решил заняться разработкой МАС на более серьезном уровне.

1. Переоценка возможностей многоагентного подхода, или непонимание того, где агенты действительно могут быть полезны.

Технология МАС в настоящее время все еще очень молода и является объектом пристального внимания. Как водится в таких случаях, было сделано много громких заявлений о потенциале этой технологии, однако большинство неудач ее использования было связано именно с переоценкой этого потенциала и с непониманием существующих ограничений. При разработке МАС важно помнить, что агенты, в конечном счете, всего лишь программы, пусть даже программы «повышенной интеллектуальности», поэтому все их возможности лежат в

рамках интеллектуальных способностей своих разработчиков.

2. Многоагентный фанатизм.

Хотя агенты и могут использоваться в широком диапазоне приложений, – они не универсальное решение. Есть много задач, для решения которых больше подходят традиционные средства, вроде ассемблера и классического ООП. Поэтому, учитывая относительную незрелость многоагентной технологии и небольшое число законченных приложений, прежде чем пытаться применить многоагентный подход для конкретной задачи нужно четко представлять себе его преимущества по сравнению с другими способами решения.

3. Стремление найти универсальное решение для уникальной задачи.

Данная проблема характерна для разработки программ самого разного назначение, но особенно заметна при разработке агентов. Обычно проявляется в виде стремления разработать некую универсальную архитектуру, более мощную систему отладки или другой кажущийся очень важным инструмент там, где это не нужно. Во-первых, существующие архитектуры агентов и МАС подходят для большинства задач, поэтому прежде чем браться за разработку принципиально новой архитектуры необходимо изучить возможности существующих. Во-вторых, весьма возможно, что решаемая задача является достаточно уникальной и на ближайшем горизонте подобных задач не предвидится. Поэтому в данном случае частное решение является наиболее рациональным выбором.

4. Вы забываете, что разрабатываете программное обеспечение.

Процесс написания кода любой, даже тривиальной, МАС – это во многом процесс экспериментирования. Эксперименты затягивают разработчиков и они забывают, что вообще-то разрабатывают программное обеспечение. В результате проект застопоривается, не укладывается в сроки, теряет финансирование. И это не из-за каких-то специфических проблем, связанных с агентным подходом, а просто потому, что остановился сам процесс написания нового кода.

5. Архитектура МАС плохо распараллелена.

Самая заметная отличительная особенность плохой МАС – малое количество агентов, которые работают параллельно. Если таких агентов мало, то следует серьезно задуматься о целесообразности использования МАС для решения задачи.

6. Агенты слишком интеллектуальны.

В процессе разработки МАС существует большое искушение сделать «очень умных» агентов. В результате в одном агенте часто интегрируются различные технологии ИИ (естественно-языковые интерфейсы, планировщики, программы доказательства теорем и т.д.), которые целесообразней было бы распределить по отдельным, более простым агентам. В результате слишком умных агентов становится трудно отлаживать и интегрировать вместе в рамках единой МАС.

7. Все рассматривается как агенты.

После знакомства с технологией МАС многие начинают все задачи безальтернативно рассматривать через призму агентов, видя в таком подходе некую концептуальную чистоту. Однако очень быстро все заканчивается тем, что появляются агенты для элементарных операций типа сложения и вычитания. Нетрудно заметить, что подобный подход не эффективен, т.к. издержки на управление отдельными агентами и организацию коммуникации между ними быстро перевесят выгоды.

8. Выделяется слишком мало агентов.

Если одним начинающим разработчикам агенты мерещатся везде, то другие наоборот, затрудняются в разбиении общей задачи на подзадачи для отдельных агентов. В результате получается система, которая не может использовать достоинства многоагентного подхода, т.к. вся функциональность реализуется двумя-тремя агентами. Полученная система, скорее всего, даже будет работать, но результат, что называется, не симпатичный.

9. Агенты слишком свободны в организации взаимодействий.

Динамика, наблюдаемая в МАС, обычно очень сложна и даже хаотична. Слишком много свободы во взаимодействии большого числа агентов приводит к тому, что становится слишком трудно

понять, что же происходит в системе. Единственный способ – продолжительные наблюдения за многократными запусками системы. Эти наблюдения, конечно, интересны, но снижают эффективность отладки, поэтому организацию взаимодействия агентов стараются подчинить четким и понятным правилам.

Успехов Вам в разработке новых и интересных приложений! Авторы надеются, что приведенная в пособии информация будет для Вас полезной.

Приложение №1

СПРАВОЧНИК ПО ФУНКЦИЯМ СРЕДЫ JASON

Внутренние функции

Напоминаем, что вызов любой внутренней функции начинается с точки.

Функции коммуникации

1) Функция «.broadcast»

Посылает сообщение всем агентам. Первый аргумент – императив, второй – посылаемый литерал.

По-умолчанию команда использует сокращенную версию KQML, что, тем не менее, не исключает подключение новых императивов. Используются следующие императивы:

tell – сообщить получателю литерал;

untell – сообщить получателю, что передаваемый литерал уже не истинен;

achieve – просьба к получателю достичь указанной цели;

unachieved – просьба прекратить достижение цели;

askOne – реализует запрос информации одним агентом из базы убеждений другого. Запрашиваемое убеждение передается в виде предиката-шаблона, в котором требуется конкретизировать заданные переменные. При этом в качестве ответа ожидается только один вариант конкретизации. Запросивший агент ожидает ответа, приостанавливая свою работу, но время ожидания может быть и ограниченно.

askAll – запрос у получателя информации по заданному

предикату-шаблону (ожидаются все возможные ответы);

tellHow – сообщить получателю план;

untellHow – сообщить получателю, что план не верен (обычно ожидается удаление плана из базы знаний получателя);

askHow – запрос всех планов для обработки события.

Следует помнить, что вся передаваемая между агентами информация автоматически всегда сопровождается аннотацией [source(A)], где A – имя агента отправителя, что позволяет получателю проверить источник сообщения.

Примеры использования

a) .broadcast(tell,value(10)). – Всем агентам сообщается предикат value(10).

b) .broadcast(achieve, who_is_last). – Всех агентов просят достичь цели «who_is_last».

2) Функция «.send»

Посылает сообщение заданному агенту. Может иметь переменное число параметров в зависимости от используемого императива.

Первый параметр – имя агента-адресата сообщения, – может быть представлен индивидной константой, параметрическим предикатом, строкой или списком, в котором перечислены имена адресатов в одной из упомянутых форм.

Второй параметр – используемый императив (см. операцию «.broadcast»).

Третий – обычно содержание сообщения.

Примеры использования

a) .send(store, askOne, mealsInStore(Rest), mealsInStore(Rest)).

Здесь третий параметр это шаблон запрашиваемого убеждения, в котором необходимо конкретизировать переменную «Rest»; четвертый параметр – это место для записи полученного ответа – в данном случае это тот же самый предикат, хотя могла быть и просто локальная переменная плана. При наличии четвертого параметра агент будет ожидать ответа в этом же цикле моделирования. Если четвертый параметр опустить, то это будет означать, что получение незамедлительного ответа не принципиально для отправителя запроса, отправитель продолжит свою ра-

боту и будет обрабатывать полученный в будущем ответ стандартным способом перехвата события изменения базы убеждений (см. следующий пример).

Далее предположим, что агент отправитель это «jomi», а агент получатель – «rafael».

b) .send(rafael,askOne,value(beer,X)): «jomi» посылает запрос «value(beer,X)» агенту «rafael». При этом дальнейшее выполнение намерений агента «jomi» не приостанавливается. Когда «rafael» ответит на запрос, то «jomi» получит событие «+value(beer,10)[source(rafael)]».

.send(rafael,askOne,value(beer,X),A,2000): аналогично примеру «a)» с поправкой на «jomi» и «rafael-я» и той разницей, что «jomi» ждет ответа 2 секунды. Если по истечении этого времени ответ не получен, то в переменную «A» помещается «timeout».

c) .send(rafael,tell,value(10)): «jomi» посылает value(10) агенту «rafael». Литерал «value(10)[source(jomi)]» будет добавлен в базу убеждений «rafael-я», а также будет сгенерировано соответствующее событие.

d) .send(rafael,achieve,go(10,30): «jomi» посылает параметризованную цель «go(10,30)» «rafael-ю». Когда «rafael» получит данное сообщение, то в его очередь событий добавится «+!go(10,30)[source(jomi)]».

3) Функция «.my_name»

В переменную аргумент помещается уникальное имя агента, присвоенное ему при создании автоматически или разработчиком.

Например, **.my_name(I).**

После выполнения команды значением переменной можно манипулировать в теле плана или его контекстных ограничениях.

Функции над строками и списками

1) Функция «.concat»

Объединяет (конкатенирует) аргументы в строки и списки. Имеет N аргументов.

Первые N-1 аргументы определяю конкатенируемые величины. Это могут быть любые термы, но термы, не являющиеся строками и списками автоматически преобразуются в строки.

Последний («N»-ый) параметр это переменная или список, куда будет помещен результат.

Примеры использования

a) .concat("a","b",X): В переменную X помещается строка "ab".

b) .concat("a b",1,a,X): В переменную X помещается строка "a b1a".

c) .concat([a,b,c],[d,e],[f,g],X): В переменную X помещается список [a,b,c,d,e,f,g].

2) Функция «.length»

Определяет или проверяет длину строки или списка. Первый аргумент – проверяемая величина, второй – переменная для хранения результата или константа для сравнения.

Примеры использования

a) .length("abc",X): в X помещается 3.

b) .length([a,b],X): в X помещается 2.

c) .length("a",2): возвращает false – размер строки не равен 2.

3) Функция «.min»

Ищет минимум среди списка заданных (возможно разнородных) величин. Может использоваться для проверки истинности суждений о минимуме. Работает с двумя параметрами, назначение которых понятно из следующих примеров.

Примеры использования

a) .min([c,a,b],X): в результате элемент «a» будет помещен в переменную X, поскольку в алфавите он расположен раньше других букв.

b) .min([b,c,10,g,f(10),5,f(4)],X): здесь X станет равным 5, поскольку сравнивается как бы простота элементов списка. Самые сложные элементы – функции с аргументами, проще – величины, обозначаемые буквами, самые простые – числа. Внутри названных категорий элементов тоже происходит сравнение. Среди чисел - ищется минимальное по величине; среди буквенных последовательностей – самые короткие и состоящие из букв, расположенных ближе к началу алфавита; среди функций – с самым маленьким числом аргументов, каждый из которых тоже должен быть максимально «простым».

c) .min([3,2,5],2): результат «true» - 2 это минимальный элемент в списке.
d) .min([3,2,5],5): результат «false».
e) .min([],X): результат «false» - нечего помещать в переменную X.

4) Функция «.max»

Обратная операция по отношению к «.min». Назначение аргументов то же самое.

Примеры использования

a) .max([c,a,b],X): в X помещается «c».
b) .max([b,c,10,g,f(10),5,f(4)],X): в X помещается «f(10)».
c) .max([3,2,5],2): возвращает false.
d) .max([3,2,5],5): возвращает true.
e) .max([],X): возвращает false.

5) Функция «.member»

Проверяет, содержится ли некоторый терм в списке. Если терм свободная переменная, то последовательно (если используется в цикле) унифицирует ее с элементами списка.

Примеры использования

a) .member(c,[a,b,c]): возвращает true.
b) .member(3,[a,b,c]): возвращает false.
c) .member(X,[a,b,c]): в X помещается «a».
d) +!start <- for (.member(X,[a,b,c]))
{ .print(X)
}. – в переменную X последовательно помещаются элементы списка, «[a,b,c]» с целью выполнения над этими элементами операций, предусмотренных в теле цикла (в данном случае это просто печать в консоль вывода).

6) Функция «.nth»

Обращается к n-ному элементу списка. Может использоваться как для считывания элемента списка, так и для проверки его значения. Обладает 3-мя параметрами. Первый –индекс элемента, к которому производится обращение; второй – список элементов; третий – либо свободная переменная для считывания значения, либо константа для сравнения. Индексация элементов списка начинается с 0.

Примеры использования

a) .nth(0,[a,b,c],X): в переменную X помещается «a».
b) .nth(2,[a,b,c],X): в переменную X помещается «c».
c) .nth(0,[a,b,c],d): возвращает false – нулевой элемент списка не равен «d».
d) .nth(0,[a,b,c],a): возвращает true.
e) .nth(5,[a,b,c],X): возвращает error – в списке нет 5-го элемента.

7) Функция «.sort»

Выполняет сортировку списка элементов по возрастанию. Может применяться как для получения отсортированного списка, так и для проверки того, является ли один список отсортированным вариантом второго. Среди разных типов элементов установлен следующий порядок сортировки: сначала упорядочиваются числа, потом атомы, потом выражения со структурой, потом списки (см. также пример «b», Функции «.min»).

Примеры использования

a) .sort([c,a,b],X): в переменную X помещается «[a,b,c]».
b) .sort([b,c,10,g,casa,f(10),[3,4],5,[3,10],f(4)],X): в переменную X помещается [5,10,b,c,casa,g,f(4),f(10),[3,4],[3,10]].
c) .sort([3,2,5],[2,3,5]): возвращает true.
d) .sort([3,2,5],[a,b,c]): возвращает false.

8) Функция «.substring»

Функция проверяет, принадлежит ли элемент множеству. В частности можно проверять, является ли одна строка подстрокой другой строки, принадлежит ли элемент списку. Может использоваться для определения индекса очередного вхождения элемента. Обладает 2-мя или 3-мя параметрами. Первый – искомый элемент, второй – множество, в котором производится поиск, третий (может отсутствовать) – переменная, в которую помещается индекс вхождения.

Примеры использования

a) .substring("b","aaa"): возвращает false – «b» не входит в «a».
b) .substring("b","aaa",X): возвращает false – аналогично.
c) .substring("a","bbacc"): возвращает true.
d) .substring("a","abbacca",X): возвращает true и в переменную X

помещается 0.

e) +!start <-

for (.substring("a","abbacca",X))

{

.print(X)

}. - в переменную X последовательно помещаются все индексы вхождения подстроки в строку (0,3,6).

f) .substring("a","bbacc",0): возвращает false – подстрока «a» не расположена в позиции 0.

g) .substring(a(10),b(t1,a(10)),X): возвращает true и в переменную X помещается 5, поскольку первый и второй аргументы предварительно преобразуются в строки.

h) .substring(a(10),b("t1,a(10)"),X): возвращает true и в переменную X помещается 6.

i) .substring (asd,[dsd, sde, asd, frd]) : возвращает true

Функции над библиотекой планов

1) Функция «.add_plan»

Добавляет новые планы в библиотеку планов агента в процессе его работы. Обладает двумя параметрами. Первый – строка или список строк, содержащих планы на языке AgentSpeak. Второй – необязательный параметр, указывающий источник получения данного плана. Если второй параметр опущен, то считается, что источником плана является сам агент, выполняющий данную функцию.

Примеры использования

a) .add_plan("+b : true <- .print(b)."): добавляет план «+b : true <- .print(b).» в библиотеку планов текущего агента с аннотацией [source(self)].

b) .add_plan("+b : true <- .print(b)", rafa): аналогично предыдущему примеру, только источник плана агент «rafa», поэтому аннотация будет [source(rafa)].

c) .add_plan(["+b : true <- .print(b)", "+b : bel <- .print(bel)"], rafa): добавляет 2 плана от агента rafa, представленных в виде списка.

2) Функция «.plan_label»

Получает код плана данного агента по его метке. Имеет 2

параметра. Первый – переменная, в которую будет считан код плана, второй – метка плана в виде атома.

Примеры использования

a) .plan_label(P,p1): в переменную P будет считан план с меткой «p1», включая саму метку.

3) Функция «.relevant_plans»

Получает список планов, относящихся к обработке заданного события. Используется, например, для обработки запроса «askHow». Имеет 2 параметра. Первый – имя отслеживаемого события, второй – имя переменной, в которую будет помещен список с кодами подходящих планов.

Примеры использования

a).relevant_plans("+!go(X,Y)",L): в переменную L будет помещен список с кодами планов для обработки события «+!go(X,Y)».

4) Функция «.remove_plan»

Удаляет план или несколько планов из библиотеки планов агента. Обладает 2мя параметрами. Первый – метка или список меток планов, которые необходимо удалить. Второй – источник планов. Если второй параметр опущен, то предполагается, что источник удаляемых планов сам агент. Команда полезна для удаления планов, полученных в результате запроса «tellHow».

Примеры использования

a) .remove_plan(l1): удаляет план, обладающий меткой «l1» с аннотацией «[source(self)]».

b) .remove_plan(l1,bob): удаляет план, обладающий меткой «l1» с аннотацией «[source(bob)]».

c) .remove_plan([l1,l2,l3]): удаляет 3 плана «l1[source(self)]», «l2[source(self)]», и «l3[source(self)]».

d) .remove_plan([l1,l2,l3],bob): удаляет 3 плана, полученных от агента bob
«l1[source(bob)]», «l2[source(bob)]», и «l3[source(bob)]».

BDI-Функции

1) Функция «.current_intention»

Возвращает описание текущих намерений агента. Полезно для случаев, когда агент должен проконтролировать собственные

намерения.

Описание намерений имеет следующую форму.

intention(<id>,<стек намерений>), а каждое из намерений описывается так:

im(<метка плана>,<список операций плана>).

Например:

```
intention(1,
[
        im("l__6",[".current_intention(I)"]),
        im("l__5",[".fail"]),
        im("l__4",["!g5(X)",".print(endg4)"]),
...
]).
```

Пример использования

а) .current_intention(X): в переменную X будет помещено описание текущих намерений в соответствии в указанной выше структурой.

2) Функция «.desire»

Проверяет, является ли аргумент желанием агента. Аргумент является желанием, если он отслеживается как событие по достижению цели «+!<desire >».

Пример использования

а) .desire(go(1,3)): возвращает true, если отслеживается событие «+!go(1,3)».

3) Функция «.drop_all_desires»

Сбрасывает все желание агента, включая те, которые перешли в стадию намерений (т.е. приняты на реализацию). Иначе говоря, очищается список зарегистрированных, но еще не обработанных событий, а также очищаются все стеки намерений.

Пример использования

а) .drop_all_desires.

4) Функция «.drop_all_events»

Обнуляет только все желания агента, не затрагивая намерения, т.е. необработанные события сбрасываются, а операции, принятые на выполнение, продолжают реализовываться.

Пример использования
а) .drop_all_events

5) Функция «.drop_all_intentions»

Обнуляет намерения агента, включая отложенные по причине ожидания, а также внутренние события, т.к. они были порождены обрабатываемыми намерениями.

Пример использования
а) .drop_all_intentions.

6) Функция «.drop_desire»

Сбрасывает желание агента, заданное аргументом, даже если оно уже перешло в стадию намерений. Аргумент представляет собой имя события.

Пример использования
а) .drop_desire(go(X,3)): удаляет все желания, подходящие под шаблон, например, «+!go(1,3)» вне зависимости от их источника, а также удаляет соответствующие стеки намерений.

7) Функция «.drop_event»

Удаляет заданное желание агента только из базы событий.

Пример использования
а) .drop_event(go(X,Y)): удаляет все желания, подходящие под шаблон, например, «+!go(1,3)» вне зависимости от их источника, не затрагивая тех, что перешли в стадию намерений.

8) Функция «.drop_intention»

Удаляет заданное намерение из базы намерений (включая отложенные намерения).

Пример использования
а) .drop_intention(go(1,3)).

9) Функция «.fail_goal»

Прерывает достижение цели, заданной цели аргументом, как если бы соответствующий план был прерван по причине ошибки. В результате генерируется событие отказа от цели.

Пример использования
а) .fail_goal(go(X,3)): прерывает достижение всех целей, подходящих под шаблон, например, «!go(1,3)». Генерируется событие

отказа от цели, например, «-!get_gold(1,3)», которое в дальнейшем может быть обработано специальным планом.

10) Функция «.intend»

Проверяет, является ли аргумент намерением. Аргумент является намерением, если план с меткой, определяемой этим аргументом, находится на исполнении во множестве намерений.

Пример использования

a) .intend(go(1,3)): возвращает true, если план с отслеживаемым событием «+!go(1,3)» находится в множестве намерений агента.

11) Функция «.succeed_goal»

Функция обратная функции «.fail_goal». Прекращает намерения по достижению цели, заданной аргументом, как если бы эта цель была уже достигнута.

Пример использования

a) .succeed_goal(go(X,3)): прерывает достижение целей «!go(X,3)», которые начинают считаться достигнутыми.

Функции определение типа термов

1) Функция «.atom»– проверяет, является ли аргумент атомом (высказыванием без отрицания), например «c». Числа, строки и свободные переменные не являются атомами.

Примеры использования

a) .atom(b(10)): возвращает false.
b) .atom(b): возвращает true.
c) .atom(~b): возвращает false.
d) .atom(10): возвращает false.
e) .atom("home page"): возвращает false.
f) .atom(X): возвращает true только если X конкретизировано атомом.
g) .atom(a(X)): возвращает false.
h) .atom(a[X]): возвращает false.
i) .atom([a,b,c]): возвращает false.
j) .atom([a,b,c(X)]): возвращает false.

2) Функция «.ground»

Проверяет, все ли переменные аргумента конкретизированы.

Возвращает истину, если у аргумента нет свободных переменных. Для чисел, строк и атомов функция возвращает истину.

Примеры использования

a) ground(b(10)): возвращает true.
b).ground(10): возвращает true.
c) .ground(X): возвращает false если X свободная переменная или обозначает терм со свободной переменной.
d) .ground(a(X)): возвращает false если X свободная переменная или обозначает терм со свободной переменной.
e) .ground([a,b,c]): возвращает true.
f) .ground([a,b,c(X)]): возвращает false если X свободная переменная или обозначает терм со свободной переменной.

3) Функция «.literal»

Проверяет, является ли аргумент литералом. Если да, то возвращает истину. Литералом является любая формула. Числа и строки литералами не являются.

Примеры использования

a) .literal(b(10)): возвращает true.
b) .literal(b): возвращает true.
c) .literal(10): возвращает false.
d) .literal("Jason"): возвращает false.
e) .literal(X): возвращает true, если X обозначает литерал, false - во всех других случаях.
f) .literal(a(X)): возвращает true.
g) .literal([a,b,c]): возвращает true.
h) .literal([a,b,c(X)]): возвращает true.

4) Функция «.list»

Проверяет, является ли аргумент списком. Если да, возвращает истину.

Примеры использования

a) .list([a,b,c]): возвращает true.
b) .list([a,b,c(X)]): возвращает true.
c).list(b(10)): возвращает false.
d) .list(10): возвращает false.
e) .list("home page"): возвращает false.

f) .list(X): возвращает true, если X обозначает список, false - во всех других случаях.
g) .list(a(X)): возвращает false.

5) Функция «.number»

Проверяет, является ли аргумент числом. Возвращает истину, если это так.

Примеры использования

a) .number(10): возвращает true.
b) .number(10.34): возвращает true.
c) .number(b(10)): возвращает false.
d) .number("home page"): возвращает false.
e) .number(X): возвращает true, если X обозначает число, false - во всех других случаях.

6) Функция «.string»

Проверяет, является ли аргумент строкой. Возвращает истину, если это так.

Примеры использования

a) .string("home page"): возвращает true.
b) .string(b(10)): возвращает false.
c) .string(b): возвращает false.
d) .string(X): возвращает true, если X обозначает строку, false - во всех других случаях.

7) Функция «.structure»

Проверяет, обладает ли аргумент структурой (функция, обратная atom).

a) .structure(b(10)): возвращает true.
b) .structure(~b): возвращает true.
c) .structure(b): возвращает false.
d) .structure(10): возвращает false.
e) .structure("home page"): возвращает false.
f) .structure(X): возвращает true, если X обозначает некую структуру, false - во всех других случаях.
g) .structure(a(X)): возвращает true.
h) .structure([a,b,c]): возвращает true.
i) .structure([a,b,c(X)]): возвращает true.

Другие внутренние функции

1) Функция «.abolish» – удаляет все убеждения, совпадающие по структуре с аргументом.

Удаляет из базы убеждений данного агента все убеждения заданной структуры. Шаблон удаляемых убеждений задается в качестве единственного параметра команды. При удалении каждого убеждения генерируется сопутствующее событие.

Примеры использования

a) .abolish(customer(_,_)) – удалить из базы убеждений все предикаты со структурой «customer(_,_)».

b) .abolish(c(_,8)[source(ag1)]): удалить из базы убеждений все предикаты со структурой «c(_,8)», полученные от агента «ag1».

2) Функция «.add_annot»

Добавляет аннотацию к литералу. Работает с 3-мя параметрами. Первый – литерал или список литералов, к которым требуется добавить аннотацию. Второй – добавляемая аннотация. Третий – переменная, в которую будет помещен результат работы функции.

Примеры использования

a) .add_annot(a,source(jomi),B): в переменную B будет помещено «a[source(jomi)]».

b) .add_annot(a,source(jomi),b[jomi]): приведет к ошибке, поскольку третий аргумент не является переменной.

c) .add_annot([a1,a2], source(jomi), B): в переменную B будет помещен список
«[a1[source(jomi)], a2[source(jomi)]]».

3) Функция «.at»

Генерирует событие через какое-то время в будущим. Идея взята из планировщиков операционных систем.

Функция оперирует 2-мя аргументами. Первый – строка, кодирующая время события относительно текущего времени. Второй – строка, содержащая имя события, которое будет сгенерировано. Обратите внимание: в записи первого аргумента важно соблюдать структуру – сначала слово «now», потом пробел, потом «+n», где n – величина паузы, затем пробел, и, наконец слово, обозначающее размерность времени, – «ms», «s», «m», «h», -

миллисекунды, секунды, минуты, часы.

Примеры использования

a) .at("now +1 m", "+!g"): генерируется событие «+!g» через 1 минуту от текущего момента.

b) .at("now +2 h", "+!g"): генерируется событие «+!g» через 2 часа.

4) Функция «.count»

Подсчитывает количество вхождений убеждений, заданных шаблоном, в базу убеждений агента. Может использоваться как для определения этого количества, так и для его проверки.

Примеры использования

a).count(a(2,_),N) – подсчитывает количество убеждений со структурой «a(2,_)» и помещает это количество в переменную N.

b) .count(a(2,_),5) – проверяет, имеется ли в базе убеждений ровно 5 убеждений со структурой «a(2,_)».

5) Функция «.create_agent»

Функция предназначена для создания другого агента. Функция требует 3-х аргументов. Первый – имя агента, которое может быть параметрическим предикатом, индивидной константой или строкой. Второй – это путь к файлу с программой работы агента. Третий – список необязательных параметров. Работа со списком необязательных параметров требует углубленного понимания среды Jason и языка Java. Полное описание данной функции приведено в [,стр.252], а в данном вводном пособии мы ограничимся лишь основным синтаксисом.

Примеры использования

a) .create_agent(bob,"x.asl"): создает агента с именем «bob» и программой поведения, определенной в файле "x.asl".

b) .create_agent(customer(N),"x.asl"): создает агента, чье имя определяется значением переменной N в предикате «customer(N)» и программой поведения, определенной в файле "x.asl".

c) .create_agent(S,"x.asl"): создает агента, чье имя определяется переменной S.

6) Функция «.kill_agent»

Уничтожает агента, чье имя задано параметром. Параметр может быть индивидной константой, предметной переменной. Данная

команда рассматривается авторами среды как подлежащая доработке, поскольку ее текущая реализация не учитывает параметров безопасности. Поэтому любой агент может уничтожить любого другого агента и ответственность за это лежит разработчиках прикладной МАС.

Примеры использования

a) .kill_agent(Agent), где Agent имя переменной

b) .kill_agent(customer), где customer константа.

7) Функция «.date»

Возвращает текущую дату в формате (год, месяц, день). Может использоваться как для получения текущей даты, так и для ее проверки.

Примеры использования

a) .date(Y,M,D): помещает в Y текущий год, в M текущий месяц, в D текущее число.

b) .date(2006,12,30): возвращает истину, если выполняется 30 декабря 2006, иначе - ложь.

8) Функция «.time»

Аналогично предыдущей функции, только работает со временем. Возвращает текущее время в формате (часы, минуты, секунды). Может использоваться как для получения текущего времени, так и для его проверки.

Примеры использования

a) .time(H,M,S): помечает в H текущий час, в M текущее значение минут, в S текущее количество секунд.

b) .time(15,_,_): возвращает истину, если сейчас от 15.00 до 16.00, иначе возвращает ложь.

9) Функция «.fail»

Будучи помещена в тело плана, функция приводит к его преднамеренному ошибочному завершению с генерацией события отказа от цели «-!g», где «g» - цель, на которую был ориентирован данный план. Это событие может быть, в свою очередь обнаружено и обработано специальными планами. Смотри также функцию «.fail_goal».

10) Функция «.findall»

Возвращает список возможных конкретизаций для заданно-

го предиката. Работает с тремя аргументами. Первый – терм, содержащий переменные, означивания которых должны привести к построению списка. Второй – предикат, в котором участвуют эти переменные, и который является шаблоном для поиска в базе убеждений. Третий – переменная, в которую будет помещен результирующий список.

Примеры использования

Предположим, что база убеждений содержит следующие предикаты:

{a(30),a(20),b(1,2),b(3,4),b(5,6)}, тогда

a) .findall(X,a(X),L): список L будет содержать варианты означиваний переменной X, которые определяются предикатом «a(X)», и будет равен «[30,20]».

b) .findall(c(Y,X),b(X,Y),L): список L будет содержать варианты означиваний предиката «c(Y,X)», которые определяются предикатом «b(X,Y)», и будет равен [c(2,1),c(4,3),c(6,5)].

11) Функция «.perceive»

Заставляет агента выполнить процесс восприятия среды немедленно. Функция полезна, если частота восприятия среды была изменена разработчиком, и агент стал воспринимать среду не на каждом цикле.

12) Функция «.random»

Возвращает случайное число от 0 до 1 и помещает его в переменную аргумент. Имеется также аналогичная функция, обладающая дополнительными возможностями и определенная в математическом модуле «math» (см. далее).

Пример использования

a) .random(X): в переменную X будет помещено случайное число от 0 до 1.

13) Функция «.stopMAS»

Останавливает работу всех агентов и среды. Используется для программного завершения работы МАС, в отличие от остановки МАС пользователем.

14) Функция «.wait»

Приостанавливает выполнение данного намерения на задан-

ный промежуток времени или до наступления определенного события. При этом осуществление остальных намерений агента продолжается. Работает с 1м или 2мя параметрами, задающими либо время ожидания в миллисекундах, либо имя ожидаемого события, либо и то и другое.

Примеры использования

a) .wait(1000): приостанавливает намерение на 1 секунду.

b) .wait("+b(1)"): приостанавливает намерение до тех пор, пока в базе убеждений не появится убеждение «b(1)».

c) .wait("+!g", 2000): приостанавливает намерение до тех пор, пока не будет получена цель «g» или пока не пройдет 2 секунды, смотря что случиться первым.

15) Функция «.all_names»

Возвращает в переменную-аргумент список всех агентов, работающих в системе в настоящий момент. Может быть полезна для проверки существования агента перед отправкой ему сообщения.

Пример использования

a) .all_names(A): в переменную A будет помещен список существующих агентов.

Функции внешнего модуля «math»

Вызов внешних функций начинается с имени модуля (в данном случае math), в котором они описаны, далее, через точку, следует имя функции и в скобках ее аргументы, если они есть.

1) Функция «math.random»

Возвращает случайное число. Если используется без аргументов, то возвращает число в диапазоне от 0 до 0,999(9). Если используется с аргументом N, то возвращает число в диапазоне от 0 до N-0.000(0)01. Для получения случайных целых чисел рекомендуется использовать совместно с функцией «math.round».

Примеры использования

a) math.random: возвращает случайное число в диапазоне [0,..,1).

b) math.random(10): возвращает случайное число от 0 до 9.999(9).

2) Функция «math.round»

Округляет числовой аргумент до ближайшего целого.

Примеры использования

a) math.round(3.345): возвращает 3.

b) math.round(3.645): возвращает 4.

c) math.round(3.5): возвращает 4.

3) Функция «math.abs»

Возвращает модуль (абсолютную величину) аргумента.

Пример использования

a) math.abs(-3): возвращает 3.

4) Функция «math.average»

Возвращает среднее арифметическое от списка чисел. Если список пустой, то в качестве признака ошибки функция возвращает специальную константу «NaN».

Примеры использования

a) math.average([3,4,6,4.6,8]): возвращает 5.12.

b) math.average(X): возвращает 5.12, если X = [3,4,6,4.6,8].

c) math.average([]): NaN.

5) Функция «math.ceil»

Возвращает ближайшее большее целое.

Пример использования

a) math.ceil(1.2): возвращает 2.

6) Функция «math.e»

Возвращает число e = 2.718281828459045.

7) Функция «math.floor»

Возвращает ближайшее меньшее целое.

Пример использования

a) math.floor(1.2): возвращает 1.

8) Функция «math.log»

Возвращает натуральный логарифм аргумента.

Пример использования

a) math.floor(math.e): возвращает 1.

9) Функция «math.pi»

Возвращает число π с точностью до 15 знака.

10) Функция «math.sqrt»

Возвращает квадратный корень аргумента.

Пример использования

a) math.sqrt(9): возвращает 3.

11) Функция «math.sum»

Возвращает сумму всех числовых элементов списка.

Пример использования

a) math.sum([1,3]): возвращает 4.

b) math.sum([3,a,"s",5]): возвращает 8.

c) math.sum([]): возвращает 0.

Библиографический список

1. Швецов А.Н. Агентно-ориентированные системы: от формальных моделей к промышленным приложениям.// Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению «Информационно- текоммуникационные системы», 2008. – 101 с.
2. Rafael H. Bordini, Jomi Fred Hubner, Michael Wooldridge Programming Multi-Agent Systems in AgentSpeak using Jason: WILEY, 2007. – 294 с.
3. Wooldridge Michael An Introduction to MultiAgent Systems: WILEY, 2002. – 365 с.
4. Болотова Л.С., Иноземцев С.П., Ященко Н.Ю. Модели и методы представления знаний в системах искусственного интеллекта. – СПб.: ВИКИ им. А.Ф.Можайского, 1991.- 116 с.
5. Болотова Л.С., Смирнов Н.А., Смольянинов А.А. Системы искусственного интеллекта. Теоретические основы СИИ и формальные модели представления знаний: Учебное пособие. – М.: МИРЭА, 2003. – 124 с
6. Болотова Л.С., Смирнов С.С., Смольянинова В.А. Системы искусственного интеллекта: Методические указания для выполнения лабораторных работ. – М.: МИРЭА, 2007.- 40 с.
7. Частиков А.П., Гаврилова Т.А., Белов Д.Л. Разработка экспертных систем. Среда CLIPS: Учебное пособие. – СПб.: БХВ-Петербург, 2003. – 600 с.
8. Fabio Bellifemine, Giovanni Caire, Dominic Greenwood Developing Multi-Agent Systems with JADE: WILEY, 2007. – 203 с.
9. Материалы сайта http://aivanoff.blogspot.com

Содержание

Введение 3
1. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ АГЕНТОВ 8
1.1. Понятие агента 8
1.2. Программные агенты и программные объекты 10
1.3. Агенты и экспертные системы 12
1.4. BDI-модель агента 13
2. МЕТОДИКА АНАЛИЗА КОНЦЕПТУАЛЬНЫХ СТРУКТУР ДЕЙСТВИЙ В ПРИМЕНЕНИИ К ОПИСАНИЮ ПОВЕДЕНИЯ АГЕНТОВ 15
3.ПРЕДСТАВЛЕНИЕ О ЯЗЫКАХ КОММУНИКАЦИИ АГЕНТОВ 23
3.1. Язык KIF 24
3.2. Язык KQML 25
4. ЗНАКОМСТВО СО СРЕДОЙ JASON 29
4.1. Что такое Jason? 29
4.2. Установка и настройка Jason 31
4.3 Запуск демонстрационных примеров 33
4.4 Основы программирования на языке AgentSpeak 34
4.4.1 Убеждения 34
4.4.2 Типы данных 36
4.4.3 Сильное и слабое отрицание 36
4.4.4 Цели 37
4.4.5 Планы 38
4.4.6 Операторы 47
4.4.7 Аннотации 48
4.4.8 Работа со списками 50
4.4.9 Общие рекомендации по составлению и расположению планов 51
4.5 Работа агента в среде Jason 53
4.6 Инструменты отладки МАС в Jason 59
5. ВЫПОЛНЕНИЕ ЛАБОРАТОРНЫХ РАБОТ 63
5.1 Описание процесса 63
5.2 Пример выполнения лабораторных работ 64
5.2.1 Постановка задачи 64
5.2.2 Концептуальный анализ поведения агентов 64

5.2.3 Концептуальная модель задачи «Фастфуд» 79
5.2.4 Описание вспомогательных алгоритмов.....80
5.2.5 Программная реализация.....80
5.2.5.1 Программа агента «Генератор клиентов».....84
5.2.5.2. Программа агента «Клиент».....87
5.2.5.3 Программа агента «Продавец» 92
5.2.5.4 Программа агента «Хранилище» 94
5.2.5.5 Программа агента «Повар» 95
5.2.5.6 Файл проекта МАС «fastfood.mas2j».....97
5.2.6 Исследование работоспособности МАС.....98
5.2.7 Дополнительные задания.....110
Заключение.....112
СПРАВОЧНИК ПО ФУНКЦИЯМ СРЕДЫ JASON 115
Библиографический список.....134

Сергей Сергеевич Смирнов
Валерия Аполлоновна Смольянинова

ВВЕДЕНИЕ В РАЗРАБОТКУ МНОГОАГЕНТНЫХ СИСТЕМ В СРЕДЕ JASON

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ AgentSpeak

Учебное пособие

Редактор Л.С. Болотова
Учебное пособие напечатано в авторской редакции

Подписано в печать 00.00.2007. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л.00,00 Усл. кр.-отт. 00,00. Уч.-изд. л. 00,00
Тираж 000 экз. С 00

Государственное образовательное учреждение
высшего профессионального образования
“Московский государственный институт радиотехники,
электроники и автоматики (технический университет)”
119454, Москва, пр. Вернадского, 78