УДК 519.682.5

О ПРОБЛЕМЕ ВВЕДЕНИЯ СРЕДСТВ РАСПРЕДЕЛЁННОГО МНОГОАГЕНТНОГО ПРОГРАММИРОВАНИЯ В ЛОГИЧЕСКИЙ ЯЗЫК СО СТРОГОЙ ТИПИЗАЦИЕЙ

А. А. Морозов, О. С. Сушкова, А. Ф. Полупанов

Институт радиотехники и электроники им В.А. Котельникова РАН

Статья поступила в редакцию 5 июля 2016 г.

Аннотация. Рассмотрена проблема введения средств многоагентного программирования в логический язык со строгой (сильной) типизацией. Для обеспечения корректного взаимодействия агентов предложен подход на основе комбинированной системы типов, идея которого заключается в том, что принцип статической проверки типов в логическом языке смягчается, а именно, проверка корректности использования объекта, полученного из другой программы (агента), откладывается до тех пор, пока он не понадобится для удалённого вызова предикатов. Во всех остальных случаях осуществляется статическая проверка типов. На основе предложенного подхода реализовано расширение объектно-ориентированного логического языка Акторный Пролог, поддерживающее распределённое программирование. Конечной целью создания этого расширения языка является разработка средств распределённого логического программирования для многоагентной обработки видеоинформации и распределённого интеллектуального видеонаблюдения.

Ключевые слова: распределённое логическое программирование, агентное логическое программирование, Акторный Пролог, строгая типизация, комбинированная система типов, номинативная система типов, структурная система типов, статическая проверка типов, динамическая проверка типов, удалённый вызов предиката, объектно-ориентированное логическое программирование, децентрализованное логическое программирование, параллельное логическое программирование, логический агент, трансляция Пролога в Джаву, интеллектуальное видеонаблюдение.

Abstract. The paper addresses the problem of the development of agent logic programming means. The Actor Prolog object-oriented logic language extension that supports distributed logic programming and remote predicate calls is described. This is intended for the multi-agent visual surveillance system implementation, that is, for the development of logic programs (agents) that acquire, analyze the video stream semantics in real time, and communicate with each other to facilitate the analysis and share obtained information / conclusions. The Actor Prolog language has a strong type system that is an important feature of the language and it is necessary for the fast and reliable executable code generation. Thus, the contradiction between the language strong type system and the idea of the software agents' independency was a problem to be resolved in the course of adapting the Actor Prolog language to the multi-agent paradigm. The problem of incorporation of distributed multi-agent programming means into the strongly typed logic language is considered. The approach to the multi-agent interaction based on the dynamic and static typing fusion is proposed. The language distinguishes instances of classes (the own worlds) created in a logic program and class instances created in other agents (the foreign worlds). The static type-checking and standard features of a nominative type system are implemented for all the own worlds like in the conventional Actor Prolog. At the same time, the dynamic type-checking and elements of a structural type system are implemented for all the foreign worlds. The program separates clearly the own and foreign worlds including the own worlds that were transmitted to other agents and then returned home. This mechanism combines the advantages of the static type-checking for the high-performance code generation with the flexibility of the dynamic type-checking that is necessary for the multi-agent systems programming.

Keywords: distributed logic programming, agent logic programming, Actor Prolog, strong typing, combined type system, nominative type system, structural type system, static type-checking, dynamic type-checking, remote predicate call, object-oriented logic programming, decentralized logic programming, concurrent logic programming, logical agent, Prolog to Java translation, intelligent visual surveillance.

Введение

Средства распределённого объектно-ориентированного логического программирования, описанные в этой статье, созданы для экспериментов с многоагентным интеллектуальным видеонаблюдением. Идея многоагентной обработки видеоинформации пришла из области искусственного интеллекта [4,5,6,7,8] и состоит в том, что система сбора и анализа видеоданных разбивается на множество независимых программ (агентов), обладающих автономностью (агент работает без прямого вмешательства со стороны пользователя и других агентов), социальными связями (агент взаимодействует с другими агентами посредством некоторых заранее предусмотренных механизмов), реактивностью (агент реагирует на внешние события) и проактивностью (агент планирует свои действия для достижения некоторых целей).

С теоретической точки зрения, использование многоагентного подхода к разработке систем интеллектуального видеонаблюдения способно обеспечить высокую гибкость, надёжность и расширяемость таких систем [9,10]. Например, если различные этапы низкоуровневой и высокоуровневой обработки видеоизображений выполняются отдельными агентами, новый вид анализа или распознавание нового вида аномального поведения людей могут быть добавлены в систему интеллектуального видеонаблюдения без изменения уже существующих агентов и даже без остановки системы. Для этого в систему просто добавляется новый агент, умеющий использовать результаты работы других агентов и способный передать им результаты своей работы.

В настоящее время декларативный подход к созданию многоагентных систем признан одним из наиболее перспективных направлений в этой области, разработано и реализовано большое количество декларативных языков и платформ многоагентного программирования [6]. Вместе с тем, агенты, работающие в составе системы интеллектуального видеонаблюдения, должны выполнять специфические операции с большими массивами бинарных данных, выходящие за рамки привычных операций обработки и обмена символьной

информацией, характерных для декларативных языков программирования. Поэтому является актуальной задача разработки декларативных платформ многоагентного программирования, ориентированных на распределённую обработку видеоинформации в реальном времени и децентрализованное интеллектуальное видеонаблюдение. Отличительными особенностями нашего подхода к решению задачи является использование параллельного объектно-ориентированного логического языка Акторный Пролог и трансляция логического языка в Джаву.

Использование средств объектно-ориентированного логического программирования для описания и анализа семантики динамических изображений ранее было предложено и исследовано в работах [11,12,13,14,15,16]. Было показано, в частности, что использование объектно-ориентированного логического языка и транслятора логического языка в Джаву обеспечивают возможность проводить логический вывод на логическом описании видеосцены в реальном времени и выявлять сложные сценарии аномального поведения людей. Разработанный ранее подход может быть легко адаптирован к программированию распределённых систем интеллектуального видеонаблюдения, поскольку использованный в работах [11,12,13,14,15,16] логический язык Акторный Пролог является одновременно объектно-ориентированным языком. Для этого не требуется даже расширение синтаксиса языка, достаточно просто обеспечить возможность удалённого вызова методов экземпляров классов Акторного Пролога из других логических программ (агентов).

В разделе 1 рассмотрена проблема введения средств многоагентного программирования в логический язык со строгой типизацией. Расширение системы типов языка Акторный Пролог и алгоритм проверки типов рассмотрены в разделе 2. В разделе 3 рассмотрен пример взаимодействия логических агентов с помощью удалённого вызова предикатов.

1. Проблема строгой типизации в многоагентных системах

Термин «удалённый вызов подпрограммы» обычно ассоциируется со стандартными механизмами взаимодействия распределённых объектов OMG CORBA, Java RMI, MS DCOM, используемыми в объектно-ориентированном программировании (ООП). Такой смысл термина имеет самое непосредственное отношение к обсуждаемой проблеме, потому что, в конечном итоге, взаимодействие агентов в Акторном Прологе реализуется с помощью механизма Java RMI. Вместе с тем, в контексте агентного логического программирования, этот термин имеет более широкое значение и связан с вопросами проектирования и реализации логического языка.

Одной из фундаментальных проблем в области объектно-ориентированного программирования является организация взаимодействия независимых программ (агентов), поддерживающих строгую типизацию данных [17]. Дело в том, что для статической проверки типов данных, передаваемых между распределёнными объектами (агентами), необходимо организовать обмен информацией об используемых этими программами типах данных (например, в форме интерфейсов классов) уже на этапе компиляции программ. Такой обмен информацией между агентами является крайне нежелательным потому, что означает появление связей между агентами ещё до начала их реального взаимодействия, а также усложнение жизненного цикла агентов. Строгая типизация является важным элементом логического языка Акторный Пролог, необходимым для получения быстрого и надёжного исполняемого кода [12,18], поэтому для введения в язык средств распределённого программирования, в первую очередь, необходимо разработать (выбрать) какой-то вариант решения проблемы строгой типизации в многоагентных системах.

Для обеспечения корректного взаимодействия распределённых объектов (агентов) в Акторном Прологе предложен подход на основе комбинированной системы типов, идея которого заключается в том, что принцип статической проверки типов в логическом языке смягчается, а именно, проверка

корректности использования объекта, полученного из другой программы (агента), откладывается до тех пор, пока он не понадобится для удалённого вызова предикатов. Во всех остальных случаях осуществляется статическая проверка типов. Ниже будут подробно рассмотрены расширение системы типов языка Акторный Пролог, а также правила проверки типов данных при взаимодействии агентов.

Сочетание в языке программирования объектно-ориентированного подхода со строгой типизацией данных является другой, хотя и тесно связанной с указанной выше, проблемой [19]. Известно, что строгая типизация очень полезна для описания интерфейсов классов в объектно-ориентированном программировании, но при этом «тип данных» и «класс» являются независимыми понятиями, и то, насколько гармонично сочетаются эти элементы в языке программирования, самым серьёзным образом влияет на удобство использования и эффективность реализации языка.

В Акторном Прологе используются оба понятия, «типы данных» (они же «домены») и «классы». Более того, понятия «элемент данных» и «экземпляр класса» были в языке разделены изначально [1,2]. Акторный Пролог поддерживает строгую типизацию различных видов простых и составных данных, таких как числа, символы, списки и пр. Одновременно с этим в языке поддерживаются классы и наследование на основе так называемого клаузального подхода к реализации логического ООП [20]. Экземпляры классов (они же «миры») могут обрабатываться в программе как обыкновенные термы, они могут передаваться в качестве аргументов в предикаты и входить в состав сложных структур данных. При этом для экземпляров классов используются специальные правила унификации, и, вообще говоря, именно для этой разновидности термов предназначены специальные механизмы обмена данными между логическими программами, обсуждаемые в этой статье.

В Акторном Прологе разные экземпляры классов всегда рассматриваются как разные сущности, то есть, унификация двух экземпляров классов является успешной, если и только если они являются одним и тем же экземпляром

одного и того же класса. Интерфейс класса содержит полную информацию о методах класса, включая имена, арность, типы аргументов и направление передачи аргументов (в Акторном Прологе поддерживается явная декларация направления передачи аргументов предиката – входной или выходной). Информация о детерминированности предикатов также содержится в интерфейсе. В языке используются три ключевых слова, обозначающие различные виды детерминированности предикатов: *nondeterm*, *determ* и *imperative*. Ключевое слово *nondeterm* информирует компилятор о том, что на поведение предиката не наложено никаких ограничений, то есть, он может вернуть один или несколько ответов (в случае отката программы), а также может сам вызвать откат программы (если его исполнение завершится неудачей). Ключевое слово *determ* сообщает компилятору о том, что исполнение предиката может завершиться успехом (он может вернуть только один ответ) или неудачей. Ключевое слово *imperative* накладывает на поведение предиката наиболее сильные ограничения: исполнение предиката должно обязательно завершиться успехом; фактически, это означает, что предикат должен выполняться как обычная подпрограмма в императивном языке программирования. Все указанные ограничения проверяются компилятором в ходе трансляции логической программы.

Использование механизма классов в Акторном Прологе усложняется тем, что язык поддерживает параллельные процессы, а также два разных вида вызовов методов класса: простой и асинхронный. Параллельные процессы являются специальной разновидностью экземпляров классов, они обозначаются с помощью двойных круглых скобок в конструкторах миров [3]. Простые вызовы методов – это обычные вызовы предикатов стандартного Пролога; предикат может быть вызван в заданном мире с помощью разделителя «?». Асинхронные вызовы предикатов обозначаются с помощью разделителей «<-» и «<<». Только этот вид вызовов методов применим к параллельным процессам, потому что, согласно семантике языка, любой простой вызов метода в параллельном процессе немедленно заканчивается неудачей. В языке введено

ключевое слово *internal*, с помощью которого можно сообщить компилятору, что некоторый слот экземпляра класса всегда содержит только простой мир (не процесс); это помогает компилятору анализировать детерминированность предикатов и оптимизировать программу. В данной статье основное внимание будет сосредоточено именно на асинхронных вызовах методов, потому что экземпляр класса, созданный и полученный из другой логической программы, является примером процесса, исполняемого параллельно относительно миров принимающей логической программы.

В общем случае, в распределённом Акторном Прологе используется статическая проверка типов, как в обычном Акторном Прологе. Динамическая проверка типов осуществляется лишь в том случае, если метод должен быть вызван в экземпляре класса, который был создан в другой логической программе (агенте) и каким-либо образом передан оттуда в рассматриваемую программу. Динамическая проверка корректности использования метода класса и типов его аргументов включает следующие операции:

1. Проверяется наличие метода с заданным именем и количеством аргументов в интерфейсе класса.
2. В случае необходимости, проверяется, может ли соответствующий предикат быть вызван в форме функции, возвращающей значение.
3. Если вызывается предикат с переменным количеством аргументов, проверяется, задекларирована ли такая возможность в интерфейсе класса.
4. Проверяется наличие заданного образца вызова, то есть, направления передачи аргументов предиката.
5. Осуществляется так называемое структурное сопоставление типов всех аргументов предиката.

Структурное сопоставление (проверка структурного соответствия) типов данных означает, что сравниваются не имена типов данных, а их структура. Далее процедура структурного сопоставления типов будет рассмотрена подробно для различных типов данных, реализованных в Акторном Прологе.

2. Система типов распределённого Акторного Пролога

Система типов Акторного Пролога поддерживает базовый набор простых и составных типов данных (доменов), на основе которых программист может определять свои собственные типы.

В языке реализованы следующие базовые простые типы данных: целые числа (INTEGER), вещественные числа (REAL), символы (SYMBOL) и текст (STRING). Отличие между целыми и вещественными числами заключается в том, что вещественные числа содержат точку. Отличие между символами и текстом заключается в том, что для внутреннего представления символьных данных в ходе выполнения программы используются номера строк в специальной таблице символов. На уровне синтаксиса языка, символьные данные обозначаются с помощью апострофов ('), а текстовые – с помощью кавычек ("). Ниже приведён пример использования встроенных простых типов данных для определения новых типов:

DOMAINS*:*

Год = *INTEGER.*

Высота = *REAL.*

Название = *SYMBOL.*

Сообщение = *STRING.*

Алгоритм проверки структурного соответствия требует точного совпадения базовых простых типов, в частности, целые и вещественные числа рассматриваются как разные типы данных.

В Акторном Прологе поддерживаются так называемые диапазоны и перечисления. Тип данных «диапазон» может быть определён с помощью целых или вещественных чисел, например:

Час = *[0 .. 24].*

Угол = *[0.0 .. 360.0].*

Алгоритм структурного сопоставления требует точного соответствия границ диапазонов и числовых типов (целые, вещественные). При этом, однако,

сравнение границ вещественных диапазонов осуществляется приблизительно, с точностью, заданной в опциях компилятора.

Тип данных «перечисление» может быть задан в виде набора литералов (констант) простых типов, а также специальных литералов # («неизвестная величина»), [ ] (пустой список), { } (пустое недоопределённое множество), например:

Час = *0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12.*

ДеньНедели = *'понедельник'; 'вторник'; 'среда'; 'четверг'; 'пятница'; 'суббота'; 'воскресенье'.*

В ходе проверки структурного соответствия перечислений проверяется, что они содержат одинаковый набор элементов.

Следует отметить, что в Акторном Прологе определение типа данных может включать имена других типов данных; в этом заключается принципиальное отличие системы типов Акторного Пролога от аналогичных систем в языках Turbo / Visual Prolog [21]. Например, переменная типа *ЧисловойTun*, определённого ниже, может содержать как целое, так и вещественное число:

ЧисловойTun = *INTEGER; REAL.*

В общем случае, определение типа в Акторном Прологе может быть рекурсивным и ссылаться на определения других типов данных. Алгоритм структурного сопоставления типов «разматывает» рекурсивные определения типов до получения набора базовых простых и составных типов, литералов и диапазонов, а затем устанавливает парное соответствие между элементами сравниваемых типов данных.

В Акторном Прологе используются три вида базовых составных типов данных: структуры, списки и недоопределённые множества. Все эти три разновидности составных типов несопоставимы друг с другом, в частности, в

отличие от стандартного Пролога, списки не являются разновидностью структур.

Определение составного типа «структура» состоит из имени (функтора) и последовательности имён типов аргументов, заключённых в круглые скобки, например:

Дата = date(Год,Месяц,Число).

Алгоритм проверки структурного соответствия типов данных «структура» проверяет совпадение функторов, количества аргументов, а также структурное соответствие типов всех аргументов. Равенство имён типов данных при этом, как было указано выше, не требуется.

Определение типа данных «список» включает тип элемента списка и символ «звёздочка», например:

ВажныеДаты = Дата.*

В ходе проверки структурного соответствия типов «список» проверяется структурное соответствие типов элементов списков.

Определение типа данных «недоопределённое множество» в Акторном Прологе представляет собой неупорядоченный набор пар «символьное имя: имя типа данных» [1,2], заключённый в фигурные скобки, например:

Дата = {год: Год, месяц: Месяц, день: Число}.

Алгоритм структурного сопоставления типов данных «недоопределённое множество» проверяет структурное соответствие типов данных, заданных в соответствующих парах. Проверяется, что типы данных содержат пары с одинаковыми символьными именами, порядок пар при этом не имеет значения.

В Акторном Прологе есть два экзотических типа данных – «анонимный тип данных» (простой тип данных, обозначается с помощью подчерка «_») и «любое недоопределённое множество» (составной тип данных, обозначается «{_}»). Первый используется для описания типов аргументов предикатов, обрабатывающих любые значения, например, процедур печати, а второй для описания типов аргументов предикатов, обрабатывающих любые значения, но

только в форме недоопределённых множеств (никаких ограничений на символьные имена пар при этом не накладывается), например, параметры HTTP-запроса:

Параметры_HTTP_запроса = { _ }.

Алгоритм проверки структурного соответствия позволяет поставить в соответствие «анонимному типу данных» только «анонимный тип данных» и типу «любое недоопределённое множество» только тип «любое недоопределённое множество».

Все правила структурного сопоставления типов, описанные выше, в равной степени пригодны как для статической, так и для динамической проверки типов данных. Отличия возникают при рассмотрении типов, содержащих имена классов. В Акторном Прологе в определении типа данных может быть указано имя класса в круглых скобках; это будет означать, что в состав структуры данных может быть включён экземпляр указанного класса. Такой тип данных мы будем называть далее тип данных «мир». Пример определения типа данных «мир»:

ОбработчикЗапросов = ('МойКласс').

Тип данных «мир» в Акторном Прологе рассматривается как простой тип. Значением типа может быть как простой экземпляр класса, так и параллельный процесс; в определении типа ничего не говорится о том, должен ли экземпляр класса являться процессом или нет. Компилятор простого (не распределённого) Акторного Пролога проверяет, что миры данного типа строго соответствуют указанному классу или принадлежат классу, являющемуся его потомком. В распределённом Акторном Прологе это правило ослаблено.

В распределённом Акторном Прологе компилятор гарантирует, что мир соответствует указанному классу в том и только в том случае, если проверяемый мир создан в той же логической программе (а значит, указанная проверка всегда возможна технически). В случае если проверяемый мир был создан в другой логической программе, проверка не осуществляется. Это

означает, например, что экземпляр класса, созданный в другом агенте, может быть присвоен любой переменной, принадлежащей некоторому типу данных, в определении которого указан какой-то произвольный класс.

Таким образом, алгоритм структурного сопоставления типов «мир» разрешает сопоставлять любые классы, если они определены в разных программах, имена классов при этом просто игнорируются. В распределённом Акторном Прологе экземпляр класса может быть передан в другую логическую программу и будет принят без проверки интерфейса класса, если принимающая программа ожидает получение экземпляра какого-то другого класса. Проверка корректности использования данного экземпляра класса будет осуществлена лишь в том случае, если принимающая программа попытается вызвать в полученном экземпляре класса какой-то метод. Для этого будет применена описанная выше процедура динамической проверки метода класса и типов его аргументов. Она подтвердит пригодность «внешнего» экземпляра класса для обработки соответствующего сообщения или вызовет ошибку времени исполнения.

Заметим, что описанная система типов является примером включения в номинативную систему типов элементов структурной системы типов. Кроме того, в язык со статической проверкой типов введены элементы динамической проверки типов. Далее мы будем называть систему типов распределённого Акторного Пролога комбинированной системой типов.

Очевидно, что для реализации описанного алгоритма проверки типов необходима информация о происхождении экземпляров классов. Акторный Пролог поддерживает внутреннюю таблицу экземпляров классов, созданных в ходе выполнения логической программы и переданных вовне. Другая внутренняя таблица содержит все экземпляры классов, полученные тем или иным способом из других логических программ. Эти таблицы позволяют программе различать собственные и «чужие» экземпляры классов и использовать эту информацию в алгоритме структурного сопоставления типов. В частности, логическая программа способна узнать «свой» экземпляр класса,

переданный в другую логическую программу и затем полученный обратно.

3. Пример удалённого вызова предиката

Рассмотрим пример удалённого вызова предиката. Предположим, что существуют два агента, назовём их условно «Распознаватель» и «Наблюдатель», которые должны совместными усилиями осуществлять поиск и распознавание людей в видеопотоке. При этом «Распознаватель» способен идентифицировать людей, находящихся в указанных координатах (возможно, он управляет своей собственной pan-tilt-zoom камерой), а «Наблюдатель» анализирует поведение людей на видео и вычисляет координаты людей, которых необходимо идентифицировать. По условию задачи, агенты являются разными логическими программами. Они должны после начала работы установить связь и обмениваться информацией по мере необходимости.

Начнём с описания агента «Распознаватель». Логическая программа, приведённая ниже, создаёт экземпляр класса и записывает его во внешний файл, так чтобы его могли «увидеть» другие логические программы. Когда какая-то внешняя программа найдёт этот экземпляр класса и передаст в него координаты человека (вызовет в нём метод *координаты_человека*), программа просто напечатает их на экране.

Согласно семантике Акторного Пролога, исполнение агента «Распознаватель» начинается с создания экземпляра класса *'Main'*. Класс *'Main'* является потомком предопределённого класса *'Console'*, реализующего управление текстовым окном.

```
class 'Main' (specialized 'Console'):
внешний_файл          = ('ОбменДанными');
[
PREDICATES:
координаты_человека(REAL,REAL)                    – (i,i);
MODEL:
?координаты_человека(X,Y).
```

Класс *'Main'* содержит один слот *внешний_файл*, значением которого является экземпляр класса *'ОбменДанными'*. С помощью класса *'ОбменДанными'* реализован обмен данными через текстовую базу данных – это, пожалуй, самый простой способ управления внешним файлом в Акторном Прологе. В разделе PREDICATES описан один-единственный предикат *координаты_человека*; у него два входных аргумента вещественного типа. В самом агенте «Распознаватель» этот предикат нигде не вызывается, поэтому в программе пришлось создать раздел MODEL и указать в нём, что предикат *координаты_человека*, возможно, будет вызван (с указанным количеством аргументов); в противном случае транслятор удалит этот предикат из текста программы в ходе оптимизации исполняемого кода.

```
CLAUSES:
goal:–!,
        внешний_файл ? insert(self),
        внешний_файл ? save("g:/РазделяемыеДанные.db"),
        writeln("Жду информацию...").
координаты_человека(X,Y):–
        writeln("X= ",X," Y= ",Y).
]
```

В разделе CLAUSES реализованы предикаты *goal* и *координаты_человека*. Предикат *goal* вызывается автоматически при создании экземпляра класса *'Main'* в начале работы программы. Он с помощью стандартного предиката *insert* и ключевого слова *self* записывает экземпляр класса *'Main'* в базу данных *внешний_файл*, затем с помощью стандартного предиката *save* записывает эту базу данных в файл «g:/РазделяемыеДанные.db» и выводит на экран надпись «Жду информацию...». Предикат *координаты_человека*, как уже было сказано, может быть вызван извне. При этом он напечатает на экране полученные координаты *X* и *Y*.

В тексте программы «Распознаватель» определён ещё один, вспомогательный, класс *'ОбменДанными'*. Этот класс является потомком

предопределённого класса *'Database'*, предназначенного для управления простыми базами данных. В разделе DOMAINS класса определён тип данных со стандартным именем *Target*. Это необходимо для того, чтобы информировать систему управления базой данных *'ОбменДанными'* о типе данных, которые будут в ней храниться. В данном случае объявлено, что тип данных *Target* включает экземпляры класса *'Main'*.

```
class 'ОбменДанными' (specialized 'Database'):
[
DOMAINS:
Target                  = ('Main').
]
```

Теперь опишем агент «Наблюдатель». Представим, что это логическая программа, которая считывает из внешнего файла экземпляр класса некоторого другого агента (логической программы), которая хотела бы получить координаты человека, отличающегося аномальным поведением, для анализа его изображения и идентификации. Агент «Наблюдатель» посредством удалённого вызова предиката должен послать ему некоторые координаты, после чего вывести на экран сообщение «Информация отправлена...».

Как и в предыдущем случае, исполнение агента «Наблюдатель» начинается с создания экземпляра класса *'Main'*. Класс *'Main'* в этой логической программе также является потомком предопределённого класса *'Console'*. Класс *'Main'* также содержит слот *внешний_файл*, значением которого является экземпляр некоторого класса *'ОбменДанными'*. В разделе PREDICATES описан предикат *послать_координаты*. У предиката *послать_координаты* один входной аргумент, значением которого должен быть экземпляр класса, наследующего интерфейс *'ПринимающийАгент'*, описанный далее.

```
class 'Main' (specialized 'Console'):
внешний_файл        = ('ОбменДанными');
[
```

```
PREDICATES:
послать_координаты('ПринимающийАгент')        – (i);
CLAUSES:
goal:–
        внешний_файл ? load("g:/РазделяемыеДанные.db"),
        внешний_файл ? find(Помощник),!,
        послать_координаты(Помощник).
послать_координаты(Помощник):–
        Помощник << координаты_человека(125.009,1107.144),
        writeln("Информация отправлена...").
]
```

В разделе CLAUSES реализованы предикаты *goal* и *послать_координаты*. Предикат *goal* вызывается автоматически при создании экземпляра класса *'Main'* в начале работы программы. Он с помощью стандартного предиката *load* загружает базу данных *внешний_файл*, а затем с помощью стандартного предиката *find* извлекает из неё экземпляр класса некоторой внешней программы. Полученный экземпляр класса передаётся в качестве аргумента в предикат *послать_координаты*, который осуществляет в нём удалённый вызов предиката *координаты_человека*, а потом выводит на экран надпись. Обратите внимание, что используется асинхронный вызов [3] предиката *координаты_человека* (с помощью префикса «<<»). Асинхронный вызов предиката необходим потому, что прочитанный из внешнего файла экземпляр класса не является внутренним по отношению к процессу «Наблюдатель» и, следовательно, не может принимать простые вызовы предикатов. Перед вызовом предиката будет осуществлена динамическая проверка метода класса и типов его аргументов.

В тексте логической программы определён также интерфейс *'ПринимающийАгент'*, описывающий методы, которые, как ожидает агент «Наблюдатель», должен поддерживать другой агент. Обратите внимание, что интерфейс *'ПринимающийАгент'* никак не связан с определениями предикатов

в логической программе «Распознаватель», и соответствие этих определений будет установлено динамически, во время работы агентов.

```
interface 'ПринимающийАгент':
[
PREDICATES:
координаты_человека(REAL,REAL)                – (i,i);
]
```

Вспомогательный класс *'ОбменДанными'* определён так же, как в программе «Распознаватель». Единственным отличием является то, что тип *Target* включает экземпляры классов, наследующих интерфейс *'ПринимающийАгент'*.

```
class 'ОбменДанными' (specialized 'Database'):
[
DOMAINS:
Target          = ('ПринимающийАгент').
]
```

Запустим на исполнение логическую программу «Распознаватель». Агент создаст на диске «*G:*» файл «РазделяемыеДанные.db» и выведет на экран надпись (см. рис. 1).

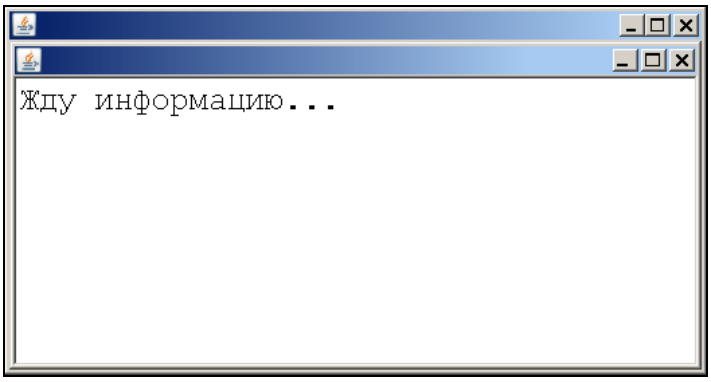

Рис. 1. Агент «Распознаватель» опубликовал экземпляр своего класса и готов принимать удалённые вызовы из других агентов.

Файл «РазделяемыеДанные.db» является текстовым, потому что предопределённый класс *'Database'* предназначен для хранения данных в виде, удобном для человека. Содержимое файла может выглядеть примерно так (содержимое текстовой строки сокращено):

('fe ... wt0ljs14r4x55uyf07tgbhvepgh05ibgefzoum5n41zjd9eh95zc');

Строка символов в апострофах и круглых скобках является обычным термом Акторного Пролога, а именно, текстовым представлением экземпляра класса. На уровне реализации языка, этот текст является закодированной заглушкой (Java RMI stub), поставленной в соответствие экземпляру класса. Заметим, что наличие компактного текстового представления экземпляра класса (точнее, ссылки на экземпляр класса) даёт принципиальную возможность общения агентов не только через файлы, но и по любым другим протоколам связи, включая электронную почту.

Теперь запустим агент «Наблюдатель». Эта программа прочитает из файла «РазделяемыеДанные.db» ссылку на экземпляр класса программы «Распознаватель» и осуществит удалённый вызов предиката (см. рис. 2).

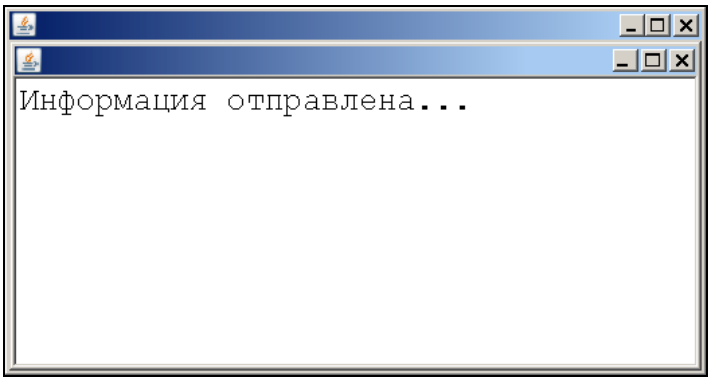

Рис. 2. Агент «Наблюдатель» прочитал экземпляр класса агента «Распознаватель», осуществил удалённый вызов предиката в этой логической программе и вывел сообщение на экран.

Агент «Распознаватель» примет удалённый вызов из агента «Наблюдатель» и выведет на экран полученные координаты человека (см. рис. 3).

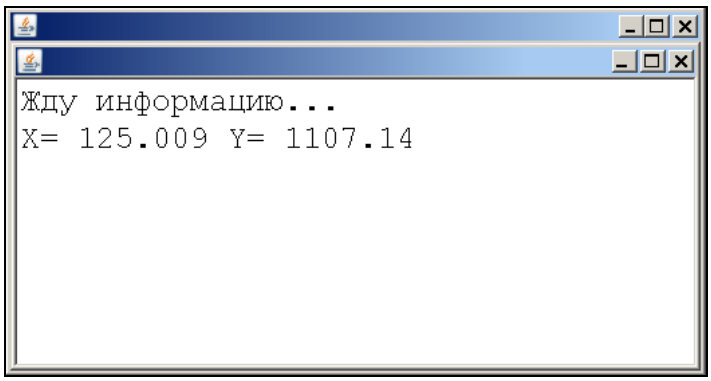

Рис. 3. Агент «Распознаватель» принял удалённый вызов предиката из агента «Наблюдатель» и вывел полученные данные на экран.

Рассмотренный пример иллюстрирует основные принципы реализации удалённых вызовов предикатов в распределённой версии Акторного Пролога, включая динамическую проверку типов и внешнее представление передаваемых ссылок на экземпляры классов.

Заключение

Разработано расширение логического языка Акторный Пролог, поддерживающее возможности распределённого логического программирования. В основе распределённой версии языка лежит новое решение проблемы строгой типизации в многоагентных системах на основе комбинированной системы типов. Распределённая версия Акторного Пролога совмещает преимущества языка со строгой типизацией, а именно, возможность получения быстрого и устойчивого исполняемого кода, с возможностями динамической проверки типов при взаимодействии агентов. Разработанные средства логического программирования являются основой для создания декларативной платформы многоагентного программирования, ориентированной на распределённую обработку видеоинформации в реальном времени и децентрализованное интеллектуальное видеонаблюдение. Разработанные средства распределённого логического программирования планируется использовать для многоагентного интеллектуального видеонаблюдения.

Работа поддержана РФФИ (грант 16-29-09626-офи_м).

Литература

1. Морозов А.А. Акторный Пролог // Программирование. – 1994. – № 5. – с. 66-78.
2. Morozov A.A. Actor Prolog: an Object-Oriented Language with the Classical Declarative Semantics // Proc. of IDL'99 workshop / Ed. by K. Sagonas and P. Tarau. – Paris, 1999. – http://www.cplire.ru/Lab144/paris.pdf .
3. Morozov A.A. Logic Object-Oriented Model of Asynchronous Concurrent Computations // Pattern Recognition and Image Analysis. – 2003. – Vol. 13, No. 4. – pp. 640-649.
4. Russell S., Norvig P. Artificial Intelligence. A Modern Approach. – London: Prentice-Hall, 1995.
5. Shen W., Hao Q., Yoon H.J., Norrie D.H. Applications of agent-based systems in intelligent manufacturing: An updated review // Advanced Engineering Informatics. – 2006. – Vol. 20. – pp. 415-431.
6. Baldoni M., Baroglio C., Mascardi V., Omicini A., Torroni P. Agents, Multi-Agent Systems and Declarative Programming: What, When, Where, Why, Who, How? // A 25-year Perspective on Logic Programming / Ed. by A. Dovier and E. Pontelli. – Berlin, Heidelberg: Springer-Verlag, 2010. – pp. 204-230.
7. Gascueña J.M., Fernández-Caballero A. On the use of agent technology in intelligent, multisensory and distributed surveillance // The Knowledge Engineering Review. – 2011. – Vol. 26:2. – pp. 191-208.
8. Kravari K., Bassiliades N. A Survey of Agent Platforms // Journal of Artificial Societies and Social Simulation. – 2015. – Vol. 18, No. 1. – Paper 11. – http://jasss.soc.surrey.ac.uk/18/1/11.html . – DOI: 10.18564/jasss.2661.
9. Vallejo D., Albusac J., Castro-Schez J.J., Glez-Morcillo C., Jiménez L. A multi-agent architecture for supporting distributed normality-based intelligent surveillance // Engineering Applications of Artificial Intelligence. – 2011. – Vol. 24. – pp. 325-340.
10. Ejaz N., Manzoor U., Nefti S., Baik S.W. A Collaborative Multi-Agent Framework for Abnormal Activity Detection in Crowded Areas // International

Journal of Innovative Computing, Information and Control. – 2012. – Vol. 8, No. 6 (June). – pp. 4219-4234.

11. Morozov A.A., Vaish A., Polupanov A.F., Antciperov V.E., Lychkov I.I., Alfimtsev A.N., Deviatkov V.V. Development of concurrent object-oriented logic programming platform for the intelligent monitoring of anomalous human activities // BIOSTEC 2014 / Ed. by G. Plantier, T. Schultz, A. Fred, H. Gamboa. – CCIS 511. – Springer International Publishing, 2015. – pp. 82-97.
12. Morozov A.A., Polupanov A.F. Intelligent visual surveillance logic programming: Implementation issues // CICLOPS-WLPE 2014 / Ed. by T. Ströder and T. Swift. – Aachener Informatik Berichte no. AIB-2014-09. – RWTH Aachen University, 2014. – pp. 31-45. – http://aib.informatik.rwth-aachen.de/2014/2014-09.pdf .
13. Morozov A.A., Polupanov A.F. Development of the logic programming approach to the intelligent monitoring of anomalous human behaviour // OGRW 2014 / Ed. by D. Paulus, C. Fuchs, D. Droege. – Koblenz: University of Koblenz-Landau, 2015. – No. 5. – pp. 82-85. – https://kola.opus.hbz-nrw.de/files/915/OGRW_2014_Proceedings.pdf .
14. Morozov A.A., Polupanov A.F., Sushkova O.S. An Approach to the Intelligent Monitoring of Anomalous Human Behaviour Based on the Actor Prolog Object-Oriented Logic Language // RuleML 2015 DC and Challenge. Proceedings of the 9th International Rule Challenge and the 5th RuleML Doctoral Consortium / Ed. by N. Bassiliades, P. Fodor, A. Giurca, G. Gottlob, T. Kliegr, G.J. Nalepa, M. Palmirani, A. Paschke, M. Proctor, D. Roman, F. Sadri, N. Stojanovic. – Berlin: CEUR, 2015. – https://www.csw.inf.fu-berlin.de/ruleml2015-ceur .
15. Morozov A.A. Development of a Method for Intelligent Video Monitoring of Abnormal Behavior of People Based on Parallel Object-Oriented Logic Programming // Pattern Recognition and Image Analysis. – 2015. – Vol. 25, No. 3. – pp. 481-492.
16. Morozov A.A., Sushkova O.S. The intelligent visual surveillance logic programming Web Site. – 2016. – http://www.fullvision.ru .

17. Odell J. Objects and agents compared // Journal of Object Technology. – 2002. – No. 1. – pp. 41-53.
18. Morozov A.A., Sushkova O.S., Polupanov A.F. A translator of Actor Prolog to Java // RuleML 2015 DC and Challenge. Proceedings of the 9th International Rule Challenge and the 5th RuleML Doctoral Consortium / Ed. by N. Bassiliades, P. Fodor, A. Giurca, G. Gottlob, T. Kliegr, G.J. Nalepa, M. Palmirani, A. Paschke, M. Proctor, D. Roman, F. Sadri, N. Stojanovic. – Berlin: CEUR, 2015. – https://www.csw.inf.fu-berlin.de/ruleml2015-ceur .
19. Nierstrasz O., Dami L. Component-oriented software technology // Object-Oriented Software Composition / Ed. by O. Nierstrasz and D. Tsichritzis. – Prentice Hall, 1995. – pp. 3-28.
20. Davison A. A survey of logic programming-based object oriented languages. – Technical Report 92/3. – Melbourne, Australia: Dep. of Computer Science, University of Melbourne, 1992.
21. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003.