АРХИТЕКТУРА ВЫСОКОПРОИЗВОДИТЕЛЬНОЙ СИСТЕМЫ МНОГОАГЕНТНОГО МОДЕЛИРОВАНИЯ

О.Р. Набиуллин,
аспирант Нижегородского филиала Государственного университета—Высшей школы экономики,
nabiulin@tecomgroup.ru
В.М. Норкин,
ассистент Нижегородского Государственного технического университета ,
mnmvik@mail.ru

В первом разделе данной работы описывается область применения, современные тенденции и проблемы в области многоагентного моделирования. Акцент делается на производительность и распределение вычислительной нагрузки. Во втором разделе описывается подход к агентному моделированию и определяются базовые термины, которые будут использованы в дальнейшем. В третьем разделе предлагается архитектура фреймворка и среды для распределённого многоагентного моделирования. Прототип, созданный на основе предложенной архитектуры, рассматривается в четвёртом разделе. Анализ архитектуры и сравнение с альтернативными продуктами приводится в пятом разделе. В шестом разделе приведены основные выводы и обсуждение дальнейших перспектив развития предложенного прототипа.

1. Введение

Актуальность задачи

Существует множество областей профессиональной деятельности, изученность которых не позволяет использовать строгие математические модели. В качестве примера таких областей можно назвать некоторые виды экономических взаимоотношений, социальные процессы или задачи с экспоненциальной сложностью. Невозможность использования строгих моделей не означает их отсутствие. Препятствием может стать вычислительная сложность модели, отсутствие модели как таковой, или сложность подбора параметров. Для некоторого класса таких областей деятельности в качестве альтернативы выступает имитационное моделирование [1], позволяющее использовать синергетический эффект взаимодействия большого количества относительно простых систем для изучения систем более высоких порядков. Использование фреймворков для агентного моделирования позволяет специалисту сосредоточиться на предмете исследования, не вдаваясь в подробности реализации среды для проведения экспериментов. В идеале среда должна предлагать механизмы для построения и запуска модели и механизмы для контроля и анализа результатов эксперимента.

Проблемы производительности фреймворков для агентного моделирования

Многие фреймворки для агентного моделирования используют языки, близкие к скриптовым: Java (Swarm, RePast J, Quicksilver, VSEdit, MASON, JADE), Python (RePast Py), Lisp и его варианты (HLSIM, SimAgent) [2]. Скриптовые языки обеспечивают бóльшую абстрагированность и простоту, за счёт производительности. Разработчики таких фреймворков пытаются максимально снизить «планку вхождения» и позиционируют свои продукты как инструмент для непрофессиональных программистов. При этом проблемы производительности рассматриваются как второстепенные [3].

Тем не менее, рассмотрим возможность увеличения производительности путём распределения вычислительной нагрузки.

При построении распределённой агентной системы встаёт несколько проблем. Для примера рассмотрим стандартную модель «heat-bugs» (описание модели приведено в четвёртом разделе). Требуется, чтобы все агенты-жуки обрабатывались последовательно. Этот пример – частный случай более общей проблемы: наличие во многих агентных моделях глобальных разделяемых ресурсов (поле), доступ к которым должен быть синхронизированным.

Исследование [4] показало: визуализация результатов моделирования в системах Swarm и RePast замедляет работу в несколько раз, поскольку работа по просчёту (рендерингу) графики осуществляется на каждом шаге моделирования. Решение этой проблемы – разделение моделирования и представления, например, переход от прямой реализации к реализации основанной на событиях [5].

Предположим, что влияние разделяемых ресурсов минимально (т.е. к ним обращаются редко). В этом случае ограничивающим фактором является факт, что агенты, составляющие систему, должны обмениваться сообщениями. Стоимость такого взаимодействия сильно варьируется в зависимости от того, находятся ли агенты физически на одной машине или на разных. Временная стоимость посылки сообщения через Ethernet на 3–4 порядка выше, чем стоимость посылки локального сообщения.

Многие стандартные модели используют концепцию дискретного времени. Это означает, что эволюция системы осуществляется итерациями (шагами). Если мы попытаемся равномерно распределить агентов по некоторому количеству компьютеров, это приведёт к тому, что более быстрые машины будут простаивать.

Вышеперечисленные причины приводят к следующим выводам:

- ✧ если задаться целью максимизировать производительность, то это нужно делать средствами модели;
- ✧ использование C++ в качестве языка описания модели позволит увеличить производительность системы, не теряя в гибкости;
- ✧ распределение агентной системы имеет смысл если выполнены следующие условия:
  - ✦ отдельный «шаг» работы системы достаточно ресурсоёмок с вычислительной точки зрения;
  - ✦ объём передаваемых данных не велик;
  - ✦ модель предусматривает некоторое количество почти изолированных кластеров агентов (которые могли бы работать в рамках одной машины)

Распределённые вычисления

Распределение вычислительной нагрузки между несколькими копьютерами не обязательно должно быть «эквивалентным». Разделение может осуществляться по принципу: исходные данные – вычисления – представление результатов (визуализация).

Создание виртуальной распределённой среды взаимодействия

Для прозрачного взаимодействия агентов в рамках распределённой системы надо создать виртуальную распределённую среду взаимодействия. Аналогом такой среды в Java можно считать JMS (Java Messaging Service); в случае Windows-окружения похожую функциональность представляет MSMQ – Microsoft Message Queuing. В табл. 1 представлено сравнение основных характеристик этих подходов.

Таблица 1

Сравнение распределённых сред обмена сообщениями

Параметр	ASF (SOP)	JMS	MSMQ
Язык	C++	Java	Любой (CAPI, COM API)
Работа в гетерогенной среде	Да	Да	Да
Работа в offline	Нет	Нет	Да
Платформа	Любая	Любая	Windows
Лицензия	Свободная	Коммерческая (часть J2EE)	Коммерческая (компонент Windows)
Поддержка транзакционности	Нет	Есть	Есть
Защита от взлома	SSL	SSL	Encryption + Windows ACL
Акцент	Производительность	Распределённость и транзакционность	Гарантированная доставка

2. Подход

Реализация агентного подхода взята из SObjectizer [2], поэтому периодически в тексте будет фигурировать именно SObjectizer, а не ASF.

Агенты

В строгом смысле слова агенты в ASF (и в SObjectizer), не являются агентами, по определению из ИИ.

SObjectizer определяет агентно-ориентированную модель, в рамках которой осуществляется проектирование реализации конкретных задач. В рамках этой модели любое приложение рассматривается как совокупность именованных объектов – агентов. Каждый агент имеет заранее определённые состояния. Взаимодействие между агентами осуществляется посредством обмена сообщениями. Агент выбирает сообщения, которые он желает обрабатывать – подписывается на сообщения. Возникновение и последующая обработка сообщения называется событием. Сообщение может породить несколько событий. Они будут обрабатываться в соответствии с назначенными им приоритетами т.н. приоритетная диспетчеризация событий. Агенты, имеющие одинаковое множество состояний, событий и сообщений, образуют класс агентов.

Сообщения

Сообщение – единственный механизм взаимодействия между агентами в SObjectizer. Имя сообщения должно быть уникальным в рамках агента. На C++ сообщение должно быть представлено структурой или классом. Для отсылки сообщения SObjectizer предоставляет несколько функций send_msg. Сообщения могут выполнять роль сигналов (сообщение без данных, важен сам факт отсылки сообщения) и роль обычных сообщений (содержащих данные).

Для получения сообщения агент должен подписаться на него. Сообщения в SObjectizer идентифицируются по паре имён: имени агента, владеющего сообщением, и имени самого сообщения. Такой способ идентификации позволяет различать сообщения от агентов одного типа (владеющих одинаковыми наборами сообщений). Понятие «владение агента сообщением» возникает из-за необходимости как-то структурировать множество всех сообщений. Например, если в противопожарной системе есть множество однотипных датчиков, а каждый из них отсылает сообщение о своём состоянии, нужно как-то различать, к какому именно датчику относится конкретное сообщение. В SObjectizer эта проблема решена привязкой имён сообщений к именам агентов – если агент зарегистрирован в SObjectizer, есть и его сообщения. Как только агент дерегистрируется, вместе с ним исчезают и сообщения, которыми он владел.

Сообщения могут рассылаться адресно (т.е. с указанием имени конкретного агента-получателя), и широковещательно (без указания имени получателя). В случае широковещательной рассылки сообщение получают все подписавшиеся на него агенты. Сообщения могут доставляться получателям сразу, в момент отсылки сообщения (обычные сообщения), могут быть отложенными (т.е. их доставка будет производится после истечения некоторого тайм-аута) и/или периодическими (т.е. их доставка будет осуществляться в цикле с некоторой паузой между циклами).

События

Событие – это реакция агента на сообщение. Каждое событие должно иметь уникальное в рамках агента имя. На C++ событие реализуется нестатическим методом C++-класса агента.

Во время подписки на сообщение агент сообщает SObjectizer, что появление некоторого сообщения (далее – инцидент) приводит к генерации такого-то события. Событие может быть подписано на ноль или более сообщений. Если событие подписано на несколько сообщений, то событие генерируется при возникновении любого из инцидентов. Обработка события заключается в том, что SObjectizer автоматически вызывает метод-обработчик события при возникновении сообщения.

События имеют приоритет, устанавливаемый в момент подписки. Сообщение может быть инцидентом сразу нескольких событий одного агента, если эти события имеют разные приоритеты. В этом случае при возникновении сообщения обработчики событий запускаются SObjectizer в порядке убывания их приоритетов.

Пробинг (опрос состояния)

Встроенные средства SObjectizer позволяют для любого агента узнать его состояние. Список состояний, поддерживаемых агентом, декларируется на этапе создания. Для полноценного анализа информации о состоянии недостаточно. Предположим, что полное состояние агента описывается «древовидным» набором свойств. Тогда поддержка пробинга сводится к описанию в терминах сообщений и событий интерфейса, поддерживаемого всеми агенты, создаваемой системы. В качестве решения этой проблемы можно создать специальный тип агента discoverable_agent_t, от которого должны наследоваться все «пользовательские агенты». Наследование агентами свойств своих предков в SObjectizer похоже на наследование классов в C++, множественное наследование допустимо с ограничением: имена сообщений и состояний агентов-предков не должны пересекаться. Discoverable_agent_t должен определять

набор сообщений, позволяющий узнать у агента его внутреннюю структуру, полное состояние и статистические параметры.

Наблюдатели (аггрегаторы)

Вводится специальный тип агентов – аггрегаторы. Аггрегатор – это агент, находящийся в системе, но не оказывающий на неё активного влияния. Задача аггрегатора – аккумуляция статистической и аналитической информации в соответствии с определёнными правилами и предоставление их в виде сообщений. Например, в случае модели heat bugs в системе присутствует два вида аггрегаторов: unhappiness_aggregator, собирающий информацию о среднем уровне «несчастливости» агентов-жуков, и hb_gui_server – агент, генерирующий некоторое кодированное представление текущей ситуации на поле.

Асинхронность

Все сообщения, передаваемые агентами, асинхронные. Сколько потоков будут выполнять работу по обработке сообщений, зависит от диспетчера, который задаётся при старте SObjectizer. При использовании некоторых типов диспетчеров приложение может не заботиться о синхронизации доступа к разделяемым данным, т.к. самих разделяемых данных не будет. На текущий момент все штатные диспетчеры SObjectizer гарантируют, что несколько обработчиков событий одного агента не будут запущены в разных контекстах. Это означает, что у агента нет проблемы доступа к своим данным из разных потоков. Поэтому и нет необходимости синхронизировать доступ к ним.

Кодогенерация

Среда для агентного моделирования должна предоставлять возможность декларативного (графического) описания аггрегаторов и введения этих новых сущностей в текущий эксперимент.

Проверка гипотез

Одним из ключевых элементов отличающих разрабатываемую среду от аналогов является интерактивная проверка гипотез. Физически проверка осуществляется построением новых аггрегаторов и включением их в текущий эксперимент. Среда должна предоставлять удобные средства контроля и мониторинга аггрегаторов.

Сценарий работы исследователя со средой ASF следующий. Запускается эксперимент, в процессе взаимодействия состояние среды меняется, и аггрегаторы выдают некоторую информацию о протекающем процессе. В случает модели heat bugs одним из видов аггрегатора становится агент, вычисляющий среднюю «несчастливость» жуков на каждом шаге работы системы. Этот агент посылает свои сообщения, отображаемые средой на компьютере пользователя. В общем случае исследователь контролирует эксперимент, проходящий на другом компьютере или наборе компьютеров. Начиная с определённого этапа, у него возникает гипотеза: средняя «несчастливость» жуков не поднимется выше 0.016 но и не опустится ниже 0.015. Исследователь описывает на специальном DSL (Domain Specific Language) агента, проверяющего эту гипотезу, с двумя сообщениями: msg_avg_ok, которое должно генерироваться каждые сто шагов, если условие гипотезы не было нарушено и msg_avg_fail (val), которое должно генерироваться, если одна из границ оказалась неверной. После этого по описаню агента на DSL генерируется C++ описание агента (динамическая библиотека, содержащая сервис, который при запуске зарегистрирует агента в системе и создаст один экземпляр). Созданная библиотека распространяется средствами ASF по всем хостам, участвующим в эксперименте, и запускается. Предположим, в некоторый момент времени пришло сообщение msg_avg_fail (0.0161). Исследователь делает вывод, что гипотеза не подтверждается, и уточняет параметры. Теперь генерируется агент, проверяющий рамки 0.002, 0.015. Предыдущий агент исключается из эксперимента, и его место занмает новый с уточнёнными параметрами. Графически этот процесс отображен на *рис. 1.*

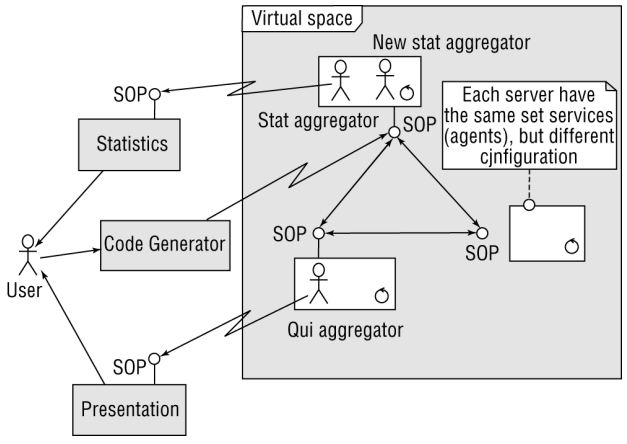

Рис. 1. Включение нового аггрегатора в эксперимент

Агентно-ориентированный подход – скорее дополнение к объектно-ориентированному подходу, нежели его замена. Фактически агентно-ориентированный подход является дополнением ООП в области динамического взаимодействия объектов, т.е.

в объектной модели разрабатываемой системы выделяются самостоятельные объекты, взаимодействующие друг с другом. Эти объекты объявляются агентами. Далее проектирование осуществляется практически так же, как и в ООП (описываются состояния, сообщения и события агентов), только с учётом того, что каждая из используемых сущностей (состояние, сообщение, событие) будет конкретным, заранее известным способом перенесена из проекта в программу.

3. Архитектура

После анализа работ [3, 7–10] выбрана архитектура, представляющая некоторое сужение SOA (Service Oriented Architecture).

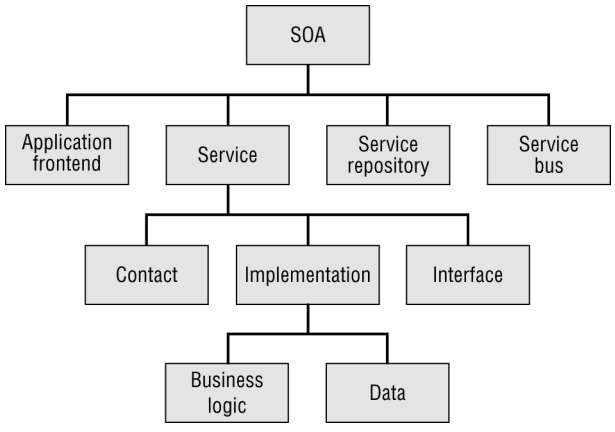

Рис. 2. Архитектура SOA

Один из ключевых аспектов проектируемой системы – его распределённость. Выбрана многозвенная (клиент-сервер) архитектура с одним важным дополнением: клиентская часть должна минимально отличаться от серверной. Этого удалось добиться в полной мере, выполняя различные задачи. Клиент и сервер содержат полный набор логики, что делает возможным смену ролей в runtime. При проектировании сформулированы следующие предпосылки:

- ✧ единая архитектурная концепция для разных приложений;
- ✧ модульность и расширяемость;
- ✧ быстродействие, близкое к Real-Time;
- ✧ принципиальная возможность перехода на другие программные платформы;
- ✧ самодиагностика;
- ✧ устойчивость к внутренним и внешним сбоям;
- ✧ защита информации и алгоритмов от несанкционированного доступа;
- ✧ конфигурирование «на лету».

Используемые библиотеки, технологии.

Standard C++

В качестве основного языка разработки выбран C++ как язык, обеспечивающий высокий уровень абстрагированности и позволяющий писать нативный эффективный код. [8, 11–14]. Обеспечение кросплатфоменности достигается использованием стандартного подмножества языка C++, а также инкапсуляцией платформенно-зависимых примитивов и сервисов. Большую часть инкапсуляции базовой функциональности операционной системы, такой как сокеты, потоки, динамические библиотеки, работа с переменными окружения, берут на себя библиотеки Boost и ACE.

Boost

Билиотека Boost – свободно распространяемый продукт с открытым исходным кодом (open-source) [15]. Исходники библиотеки и документация доступны по адресу www.boost.org [16]. Библиотека Boost – это собрание множества независимых библиотек, созданных независимыми разработчиками и тщательно проверенными на различных платформах. Можно считать, что Boost – это расширение стандартной библиотеки C++.

ACE

The ADAPTIVE Communication Environment (ACE) – одна из самых переносимых C++ библиотек, предназначенная для разработки сложных, многоплатформенных приложений, и широко используется во всем мире [17]. ACE представляет собой библиотеку, разделённую логически на несколько слоёв. На самом нижнем уровне находится кроссплатформенная реализация многих функций из стандартной библиотеки C и POSIX. Над ними – интерфейсные фасады, скрывающие детали реализации таких средств, как нити, процессы, примитивы синхронизации, различные механизмы межпроцессорного взаимодействия. Над ними каркасы.

Python

Python (па́йтон, пито́н) – мультипарадигменный язык программирования высокого уровня с динамической типизацией, автоматическим управлением памятью и удобными высокоуровневыми структурами данных, такими как словари (хеш-таблицы), списки, кортежи. Большая часть реализаций, в том числе и референсная CPython, – интерпретаторы. Поддерживает классы, модули (они могут быть объединены в пакеты), обработку исключений, а также многопоточные вычисления. Питон обладает простым и выразительным синтаксисом.

Язык поддерживает несколько парадигм программирования: структурное, объектно-ориентированное, функциональное и аспектно-ориентированное.

SObjectizer

SObjectizer – набор принципов, правил и ограничений на проектирование и реализацию программ в рамках агентного подхода, называемый агентной моделью; кросплатформенная OpenSource библиотека, http://sobjectizer.sourceforge.net, позволяющая реализовать приложение в терминах агентов, их состояний, сообщений и событий [18; 19].

4. Прототип

На основании проведённого исследования, а также изучения соответствующей аналитической литературы сформулированы следующие принципы проектирования:

- ✧ ядро – сервисы – Модель данных;
- ✧ настройка через внешние файлы;
- ✧ граф зависимостей между сервисами;
- ✧ несколько категорий сервисов;
- ✧ механизм Job Scheduler;
- ✧ использование интерфейсов и «перехватчиков» для вызова;
- ✧ шаблон Generic Attribute;
- ✧ единый сервис-контроллер для обработки запросов (конфигурируется через скрипт);
- ✧ транзакции;
- ✧ Service – Health – Monitoring в ядре;
- ✧ микрорестарт.

При создании прототипа выделено пять категорий сервисов:

- ✧ Boot-strap services – сервисы, в обязанность которых входит запуск и инициализация ядра;
- ✧ Kernel support services – сервисы ядра, сервисы реализующие основную часть функциональности framework (каркаса);
- ✧ Application support services – вспомогательные сервисы, аналог HAL – Hardware Abstraction Layer;
- ✧ Application services – сервисы специфичные для SensorMonitor;
- ✧ Service extensions – расширения сервера (plug-ins, script);

Состав каждой группы сервисов, для одного из проектов, созданных на ядре ASF, отражён на *рис. 3.*

Группы Application support services и Application services на *рис. 3* не содержат всех сервисов. В общем случае количество сервисов определяется приложением при запуске и в дальнейшем может изменяться.

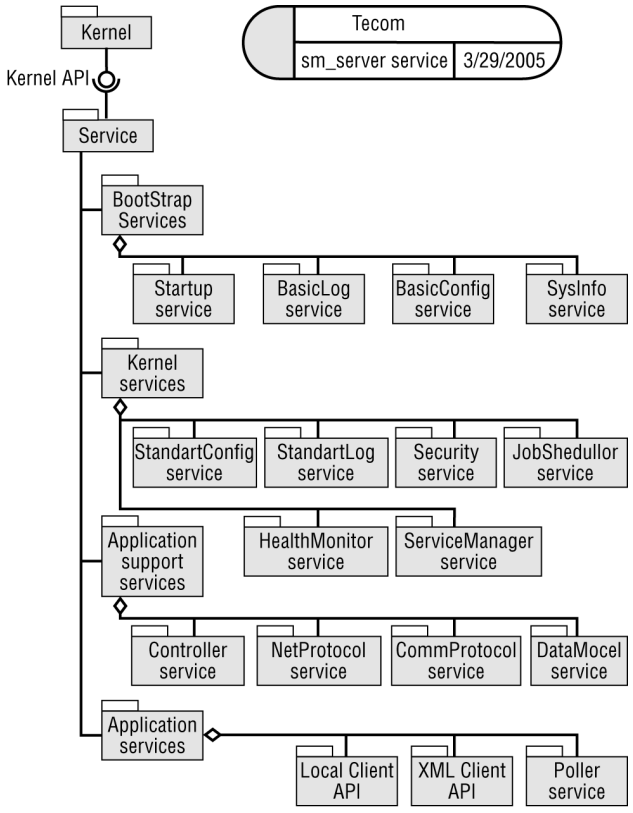

Рис. 3. Сервисы ASF

Ядро (Kernel) выступает в роли менеджера ресурсов (память, потоки, и т.д.) а также «черного ящика» для сервисов, через который они получают доступ к другим сервисам. Взаимодействие ядра с сервисами происходит асинхронно. Одна из ключевых особенностей – полная интроспекция: каждый сервис формально регистрирует свой внешний интерфейс в ядре. Таким образом, можно объединять сервисы, разработанные независимо друг от друга. В качестве средства оркестровки сервисов используется скрипт на языке Python.

Boot-strap services

Сервисы, входящие в группу Boot-strap отвечают за запуск ядра и базовую информацию о системе. Сюда входят:

- ✧ Startup service – сервис, которому передаётся управление при запуске сервера (экземпляр создается в функции main);
- ✧ BasicLog service – сервис, обеспечивающий журналирование (текстовый файл) на этапах запуска и остановки сервера;
- ✧ BasicConfig service – сервис, обеспечивающий доступ к конфигурационным параметрам.

- ✧ SysInfo service – сервис, через который ядро получает информацию о системе (количество доступной памяти, тип процессора и т.д.)

Сервисы этой группы тесно связаны с ядром, операционной системой и типом программы, играющей роль сервера в системе SensorMonitor (stand-alone, service, daemon), и любые изменения в их интерфейсах отражается на ядре.

Kernel services

Сервисы, входящие в группу Kernel реализуют основную функциональность framework (каркаса). Сюда относятся:

- ✧ StandartConfig service – сервис позволяющий другим частям системы получить доступ к своим конфигурационным параметрам (xml-файл);
- ✧ StandartLog service – сервис, обеспечивающий журналирование работы системы (текстовый файл, системные журналы, БД);
- ✧ Resource Manager – менеджер ресурсов обеспечивающий доступ к разделяемым ресурсам и их освобождение в случае сбоя сервиса, захватившего ресурс;
- ✧ JobScheduler – планировщик, обеспечивающий «одновременное» выполнение различных сервисов. Реализован как пул потоков (нитей);
- ✧ HealthMonitor – сервис, обеспечивающий мониторинг, завершение и перезапуск сервисов, в которых возникли ошибки (микрорестарт);
- ✧ ServiceManager – сервис, управляющий сервисами групп Application, и Application support, динамически подключаемыми на этапе загрузки.

Application support services

Сервисы, входящие в группу Application support, – прослойка между сервисами уровня приложения и аппратной платформой (операционной системой). Сюда относятся:

- ✧ Controller service – сервис, играющий роль драйвера Master-Controller;
- ✧ NetProtocol service – сервис, реализующий интерфейс к стандартныем сетевым протоколам;
- ✧ CommProtocol service – сервис, реализующий интерфейс в последовательному (RS-232) порту;
- ✧ DataModel service – сервис, представляющий собой модель предметной области.

Application services

Сервисы, входящие в группу Application services, реализуют фукциональность, специфическую для конкретного приложения. Сюда входят:

- ✧ SimulationManager – сервис обеспечивающий управление имитационным экспериментом;
- ✧ HeatBugsService – набор агентов для классической модели Heat-Bugs;
- ✧ HeatBugsGUI – визуализация модели HeatBugs (в т.ч. и распределенная);
- ✧ PythonScripting – управление приложением с помощью Python-скриптов.

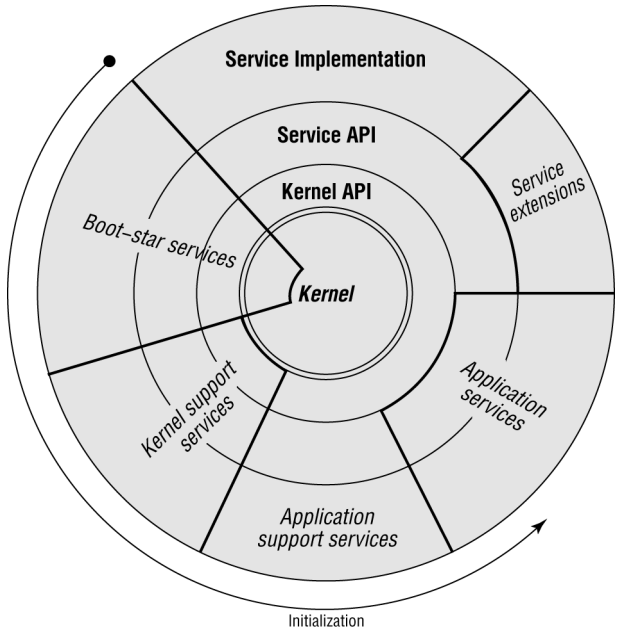

Рис. 4. Взаимодействие сервисов и ядра

SimulationManager

Сервис Simulation Manager, входящий в группу Application Services, обеспечивает поддержку проведения имитационных экспериментов. По сути этот сервис является интерфейсом между ядром ASF и SObjectizer. С точки зрения SObjectizer среда ASF представлена агентом system_agent, сообщения которого управляют ходом эксперимента. С другой стороны, для ASF эксперимент – это результат работы сервисов.

Важная особенность Simulation Manager – поддержка пробига. Класс discoverable_agent_t, базовый для всех агентов системы, реагирует на следующие сообщения:

- ✧ get_property_list_msg(string, string);
- ✧ get_property_msg(string, string);
- ✧ set_listener_msg(string, string);
- ✧ remove_listener_msg(string, string).

Первый аргумент каждого сообщения представляет собой «обратный адрес» отправителя. Строки аргументов, запросов и ответов содержат следующие специальные символы: «.,()\». Если строка

ответа должна содержать один из этих символов, они экранируются с помощью «\». Например: «ab.c» –> «ab\.c», «a\b» –> «a\\b». Все свойства строковые; бинарные данные передаются с помощью Base64-кодирования.

Сообщение get_property_list_msg(string, string) содержит два аргумента. Первый – «обратный адрес», второй – маска. Маска представляет собой строку вида «meters.temperature.m1» – спецификация полного имени свойства (ожидаемый ответ – пустая строка), или «meters.temperature.*» – спецификация частичного имени (ожидаемый ответ «meters.temperature.(m1, m2, m3)», если такие свойства существуют или «meters.temperature», если соответствующее свойство – лист дерева).

Сообщение get_property_msg(string, string) – запрос на получение значения свойства. Например, «meters.temperature.m1». Ожидаемый ответ в этом случае – «123». Или «meters.temperature.(m1, m2), meters.humidity.m1». Ожидаемый ответ – «(123,14),100».

Сообщения set_listener_msg(string, string) и remove_listener_msg(string, string) используются для подписки на изменение указанного в аргументе свойства агента. Структура сервиса Simulation Manager приведена на *рис. 5*.

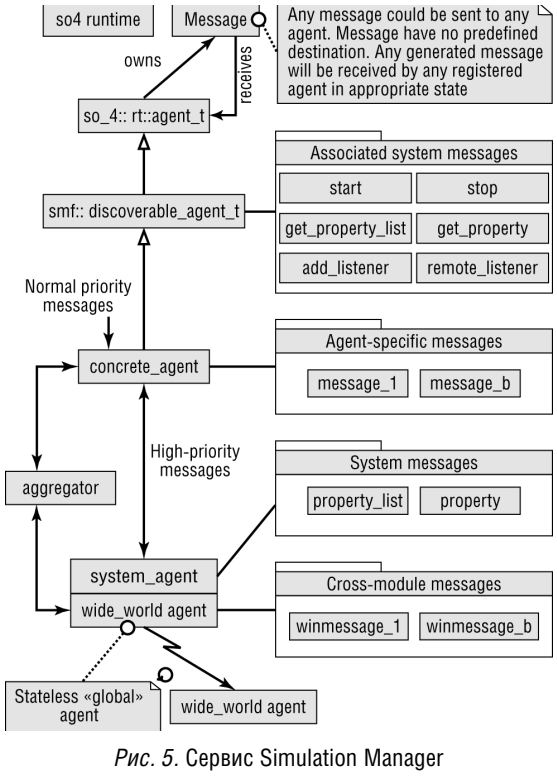

Рис. 5. Сервис Simulation Manager

С физической точки зрения все агенты представлены динамическими библиотеками, содержащими сервисы ASF. Будучи запущенным, такой сервис регистрирует в SObjectizer классы агентов. Создание экземпляров агентов осуществляется при вызове асинхронного метода сервиса. Такое двухфазное создание агентов призвано обеспечить гибкость при настройке системы и перезапуске эксперимента. Перезапуск происходит без перезагрузки (или даже остановки) сервисов. Соответствующая диаграмма развертывания приведена на *рис. 6*.

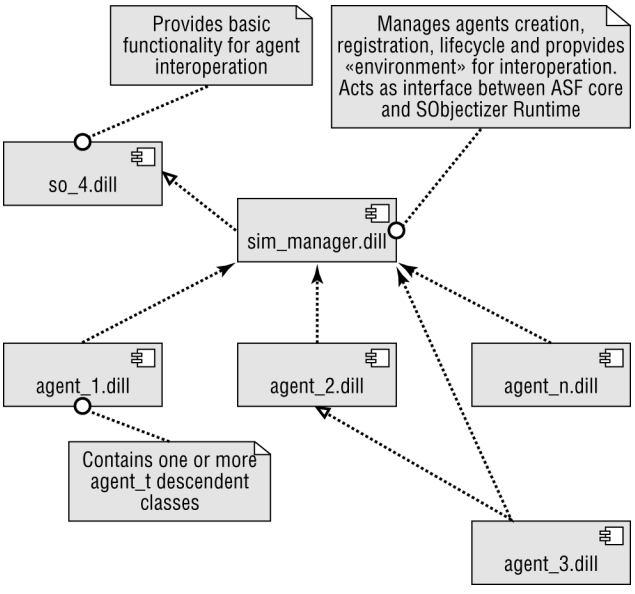

Рис. 6. Диаграмма развертывания для сервисов агентов

Принцип работы

Ключевой момент работы SObjectizer – диспетчеризация сообщений и событий. Механизм диспетчеризации события рассмотрен ниже.

Кто-то вызывает send_msg для отсылки сообщения. Функции send_msg указывается имя сообщения и, если необходимо, имя получателя. SObjectizer проверяет существование сообщения. Если такое сообщение действительно существует, создаёт экземпляр сообщения. В случае отложенного сообщения экземпляр сообщения отдаётся специальному объекту-таймеру для отсчёта тайм-аута. В противном случае начинается диспетчеризация сообщения. Для всех событий, подписанных на сообщение, формируется список заявок диспетчеру. Если сообщение отсылалось адресно, в список включаются только события агента-получателя. Заявки передаются диспетчеру для дальнейшей диспетчеризации. Диспетчер распределяет эти заявки по очередям заявок своих рабочих нитей. Например, диспетчер с одной рабочей нитью помещает все заявки в одну очередь. Диспетчер с активными объектами помещает заявки в очередь нити, обслуживающей конкретного агента.

Рабочая нить диспетчера извлекает из своей очереди очередную заявку. Она проверяет, может ли событие быть обработано в текущем состоянии агента.

Если может, то в контексте этой рабочей нити у объекта-агента вызывается метод-обработчик события. В противном случае событие игнорируется. После обработки очередной заявки сама заявка уничтожается. После уничтожения всех заявок, порождённых экземпляром сообщения, SObjectizer автоматически уничтожает экземпляр сообщения (если только сообщение не является периодическим).

Распределённое взаимодействие

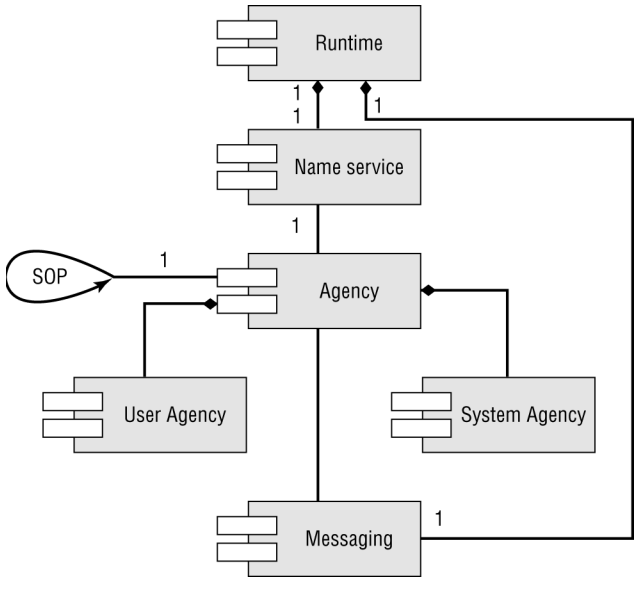

Рис. 7. Распределённое взаимодействие

В SObjectizer агенты взаимодействуют между собой только посредством асинхронных сообщений. Поэтому несложно обеспечить условия, при которых агенту-получателю неважно, откуда сообщение поступило. Необходимо только, чтобы сообщение содержало в себе копию всех необходимых данных. В этом случае сообщение может быть сериализовано, передано по какому-то IPC (Inter Process Communication) каналу в другой процесс, десериализовано и доставлено агенту-получателю. В SObjectizer для этого есть базовые средства.

В SObjectizer существует т.н. SOP (SObjectizer Protocol), определяющий, как будут сериализованы сообщения агентов. SOP позволяет передавать сообщения, которые содержат поля примитивных типов (char, int, short,...) или векторов этих типов, а также std::string и сложных объектов, сериализуемых с помощью ObjESSty (http://eao197.narod.ru/objessty). Для этого нужно всего лишь описать структуру сообщения для SObjectizer в виде набора макросов:

```
// Вот как эта структура описывает сообщение.
struct msg_alarm
{
// Имя датчика, инициировавшего данное сообщение.
std::string m_sensor_name;
// Текущие показания датчика.
float m_current_value;
// Предельное значение, после которого датчик объявляет тревогу.
float m_max;
};

// А вот так это сообщение должно быть описано для SObjectizer.
SOL4_MSG_START(msg_alarm, msg_alarm)
SOL4_MSG_FIELD(m_sensor_name)
SOL4_MSG_FIELD(m_current_value)
SOL4_MSG_FIELD(m_max)
SOL4_MSG_FINISH()
```

Такое описание позволяет SObjectizer на стороне отправителя взять значения полей m_sensor_name, m_current_value и m_max из объекта-сообщения и сериализовать их в коммуникационный канал. На принимающей стороне SObjectizer создаёт объект типа msg_alarm и восстанавливает значения его полей, затем доставляет созданный объект получателю.

SObjectizer вводит понятие коммуникационного канала (экземпляр конкретного IPC-соединения или сессии). Коммуникационные каналы обслуживают специальные транспортные агенты. Например, для организации взаимодействия двух процессов по TCP/IP в одном из них нужно объявить транспортного агента, который создаст серверный TCP/IP-сокет, а во втором процессе – транспортного агента, который создаст клиентский TCP/IP-сокет. Эти агенты будут устанавливать и обслуживать TCP/IP-соединения, объявляя каждую новую сессию отдельным коммуникационным каналом. Например, так выглядит создание транспортного агента для серверного сокета:

```
so_4::rt::comm::a_srv_channel_t a_channel(
«a_channel»,
so_4::socket::channels::create_server_channel(ip_address));
```

а вот так – создание транспортного агента для клиентского сокета:

```
so_4::sop::std_filter_t * filter = so_4::sop::create_std_filter();
filter->insert(a_common_t::agent_name());

so_4::rt::comm::a_cln_channel_t a_channel(
«a_channel»,
so_4::socket::channels::create_client_factory(ip_address),
filter,
// Вот этот объект будет указывать агенту повторять попытки
```

```
// установления соединения каждые 5 секунд, и инициировать
// повторное соединение после разрыва без задержек.
so_4::rt::comm::a_cln_channel_t::
create_def_disconnect_handler(5000, 0));
```

Над транспортными агентами в SObjectizer стоит еще один специальный агент-коммуникатор. Он отвечает за отслеживание всех доступных коммуникационных каналов и проверку их жизнеспособности. Одна из задач коммуникатора – пингование каналов, если в них нет активности, и принудительное закрытие канала в случае отсутствия ответов на пинги. Вторая задача коммуникатора – отслеживание сообщений, которые должны быть переданы в коммуникационные каналы, и восстановление сообщений, полученных из коммуникационных каналов. Какие именно сообщения коммуникатор будет передавать в коммуникационный канал, а какие игнорировать? В SObjectizer существует понятие т.н. глобального агента. Это агент, владеющий только сообщениями, но не имеющий ни состояний, ни событий. Он даже регистрируется в SObjectizer особым образом. Зато его сообщения коммуникатор отслеживает и рассылает в доступные коммуникационные каналы. Чтобы два процесса могли взаимодействовать между собой посредством SOP, им необходимо всего лишь зарегистрировать у себя одного и того же глобального агента (отсюда и его название, он как бы существует без оглядки на границы процесса). После этого любое сообщение глобального агента, порождённое в одном процессе, будет автоматически транслироваться в другой процесс.

Тестовая модель HeatBugs

Для демонстрации и испытания возможностей прототипа реализована модель HeatBugs – одна из классических демонстрационных моделей в агентно-ориентированных вычислениях.

Модель HeatBugs представляет собой прямоугольное поле, населённое агентами-жуками. Одновременно в одной ячейке поля может находиться не более одного жука. Каждый жук излучает в пространство некоторое количество тепла, которое с течением времени рассеивается в пространстве. Один жук не может обеспечить комфортной температуры для себя, поэтому он начинает движение в том направлении, которое ему кажется наиболее предпочтительным. Выбор предпочтительного направления осуществляется как выбор ближайшей к жуку незанятой точки, где температура ближе к идеальной, чем в текущем местоположении. Если много жуков сбиваются в одну кучу, вместе они так нагревают пространство, температура перестаёт быть комфортной, и жуки начинают стремиться уйти от кластера. Для каждого жука на каждом шаге моделирования определяется его «несчастливость», как функция от модуля разницы между текущей температурой среды и идеальной температурой, к которой данный жук стремится.

На *рис. 8* отображены два состояния, в которых может находиться агент-жук:

- st_initial – в нём жук находится сразу после запуска приложения и при перезапуске модели. В этом состоянии жук пытается найти незанятое место на поле, в котором он начнёт свои передвижения;
- st_sim_run – это основное и единственное состояние жука в течение эксперимента. Находясь в этом состоянии, жук пытается найти оптимальное место. Стоит обратить внимание, что параметры среды жук узнаёт у специального агента – a_heat_space; всё взаимодействие со средой – запросы и перемещения реализованы как посылка/приём сообщений.

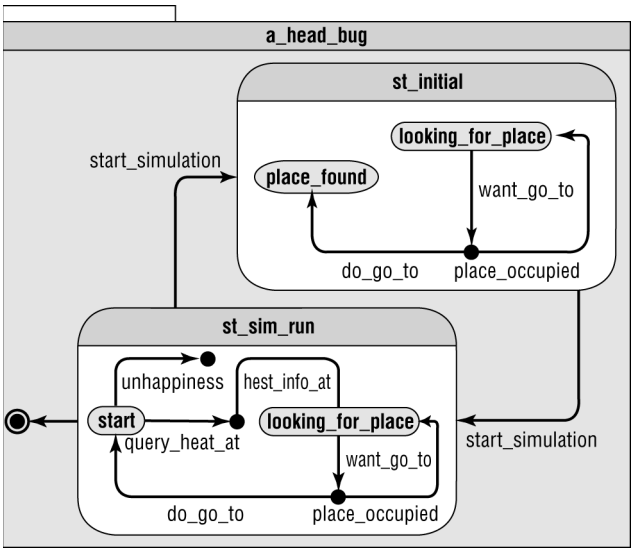

Рис. 8. Состояния агента a_heat_bug

Агент a_heat_space отображен на *рис. 9*. Агент может находиться в трёх базовых состояниях, привязанных к управлению экспериментом. В состоянии st_initial агент a_heat_space «размещает» агентов на поле. После успешного размещения всех агентов-жуков, модель переходит в состояние st_sim_run, где происходит основная часть взаимодействия в течение эксперимента. Состояние st_sim_paused введено, чтобы исследователь (или среда) мог в любой момент прервать эксперимент и просмотреть состояние отдельных агентов, участвующих в эксперименте.

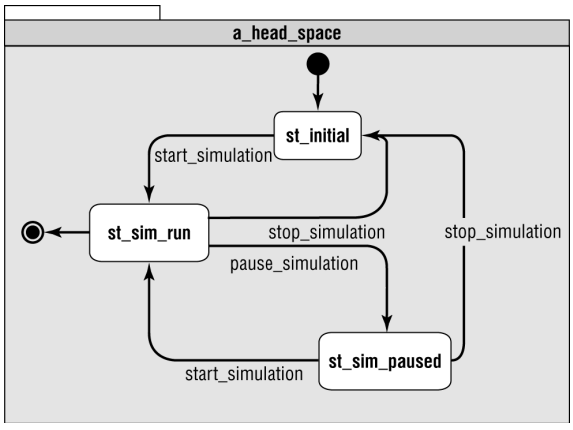

Рис.9. Состояния агента a_heat_space

Реализация графики

Одна из основных черт ASF – распределённость. В качестве иллюстрации этой концепции в прототип введён еще один сервис HeatBugsGui, обеспечивающий рендеринг графики. Этот сервис при запуске создаёт одного (или пару) из двух агентов: a_hb_gui_sender и a_hb_gui_receiver. Назначение этих агентов в сборе информации о текущем состоянии модели и отображении этой информации. Ни один из агентов не знает, запущен ли его компаньон локально, в одном адресном пространстве, или же работает на другой машине. Сообщения этих агентов помещаются на общую информационную шину и обрабатываются всеми, агентами, подписавшимися на эти сообщения. Если клиент, отображающий данные, запущен в другом процессе (и возможно на другой машине), работу по сериализации/десериализации сообщений этих агентов берёт на себя SObjectizer. Если модель запущена локально, никакой сериализации не происходит, и данные передаются практически прямым вызовом функции. Под локальностью понимается, что все агенты запущены в рамках одного процесса.

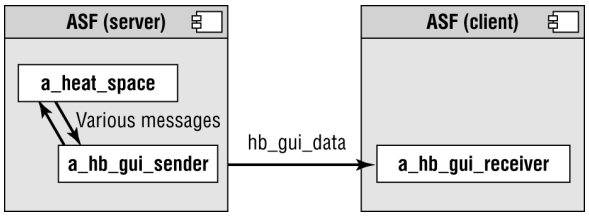

Рис. 10. Реализация удалённого GUI

5. Анализ архитектуры

Pro

Анализ работы прототипа показывает, что выбранная архитектура обеспечивает:

- ✧ **гибкость.** Приложение собирается как кубики из различных сервисов, причём переконфигурирование может происходить в runtime. Поддержка скриптинга позволяет держать систему «на кончиках пальцев», и формулировать запросы на языке высокого уровня (Python);
- ✧ **распределённость.** Создается виртуальная распределённая среда взаимодействия агентов. Важной отличительной особенностью является то, что агент, не знает и не должен знать, пришло сообщение к нему от агента находящегося с ним в одном адресном пространстве или с другой машины, возможно находящейся в тысяче километров от первой;
- ✧ **быстродействие.** Язык C++ позволяет воспользоваться преимуществами современных оптимизирующих компиляторов, генерирующий весьма эффективный код. Отсутствие сборщика мусора, характерного для управляемых (managed) языков, компенсируется использованием техник RAII, что позволяет осуществлять контроль не только за памятью, но и за другими ресурсами;
- ✧ **возможность «мягкой» интеграции с внешними системами.** При условии, что агенты должны взаимодействовать только сообщениями, мы обеспечиваем возможность мягкой интеграции с внешними системами. Например, на предприятии существует система поддержки принятия решений. Её часть – система имитационного моделирования, реализованная на ASF. С точки зрения ASF, интерфейсы к внешним системам – агенты, генерирующие свои сообщения в случае изменения внешних условий. Управляющие сигналы передаются во внешний мир через сообщения.

Contra

Было бы ошибкой утверждать, что выбранная архитектура лишена недостатков. Далее перечислены несколько проблем, к которым приводит следование архитектуре ASF.

Грамотный подход к выбору модели

Распределение вычислительных ресурсов может оказать эффект обратный, планируемому в случае моделей с большим уровнем связности. Если вычислительный эксперимент построен на десяти компьютерах объединённых сетью, и один шаг работы системы подразумевает получение реакции ото всех агентов, участвующих в эксперименте, причем каждый отдельных шаг вычислений достаточно прост, то накладные расходы по организации сетевого взаимодействия могут оказаться на несколько

порядков выше, чем сложность вычислений. Это приведёт к тому, что общее быстродействие снизится в соответствующее количество раз.

Внутренняя сложность (COM, Corba)

Внутренняя сложность фреймворка достаточно велика. Поддержка и развитее ASF требуют от программиста глубокого знания C++, а также специфики работы таких механизмов, как потоки, процессы, динамические библиотеки и сети. С точки зрения поддержки модульности и асинхронности, а также работы в распределенных средах, ASF частично реализует функциональность CORBA, обеспечивая динамическую переконфигурацию, и отказываясь от жесткой привязки к интерфейсам. Похожую функциональность в случае среды MS Windows обеспечивают дуальные COM-интерфейсы. Однако использование COM ограничило бы список доступных платформ для ASF. Использование C++ заставляет строго следить, чтобы настройки компилятора для среды и dll-расширений в точности совпадали, что делает возможным использование C++ интерфейсов через границу модуля [10][14].

Асинхронность усложняет реализацию

Для некоторых задач навязанная асинхронность может усложнить и замедлить реализацию. Однако реальный мир асинхронен. Если задача удобнее решается в синхронных терминах, имеет смысл рассмотреть альтернативы ASF.

Ненадежность соединения

Сетевые соединения ненадёжны. Поэтому прикладная логика модели должна содержать реакцию на выпадение одного или нескольких хостов из общей сети, что усложняет реализацию как сервера, так и модели.

Сравнение производительности

Чтобы получить численную характеристику выигрыша в производительности от использования ASF, проведено сравнение с распространёнными фреймворками для агентного моделирования Swarm и RePast (.NET).

Swarm

Тестовая модель HeatBugs с параметрами: количество жуков (агентов) 100, размер мира 100×100, количество шагов – 10000. ASF затратил на проход в среднем в 1,3 раза больше времени, чем Swarm. Это может показаться странным, но с другой стороны Swarm-модель использует прямые вызовы для определения температуры окружающего мира и выбора направления движения. Приблизительная оценка количества сообщений, обрабатываемых ASF и Swarm, показывает разницу на два порядка. Один шаг эмуляции в ASF-модели приводит к генерации и обработке примерно 500 сообщений, тогда как в Swarm – порядка 3-х. Стоит отметить, что модель HeatBugs – не самая удачная с точки зрения демонстрации преимуществ ASF, однако это классическая модель и авторы решили остановить свой выбор на ней.

RePast

RePast (.NET) на модели HeatBugs при параметрах, совпадающих с указанными в предыдущей главе, показал производительность в два раза меньшую, чем ASF. Отдельно стоит отметить, что модель RePast построена на прямых вызовах, что теоретически снижает накладные расходы на синхронизацию.

6. Выводы, планы

Созданная архитектура позволяет эффективно проводить имитационные эксперименты. Производительность системы намного выше, чем у аналогов, но выигрыш в производительности теряется в случае неудачно выбранной модели (в качестве мерила производительности используется количество сгенерированных и обработанных сообщений в секунду). Идеальный случай для ASF – модель, в которой небольшое количество агентов взаимодействуют асинхронно в силу природы модели, причём объём вычислений на каждом шаге эксперимента сравнительно высок, по сравнению с объёмом результатов вычислений.

Дальнейшее направление развития системы – создание DSL, позволяющего описывать агентов на высокоуровневом, графическом языке, и кодогенератора, для транслирования DSL-описания в C++ код. Помимо кодогенератора, в систему должен входить набор скриптов, содержащий настройки для компиляции расширений ASF, написаных вручную. ■

Литература

1. Gilbert, N. Terna P, How to Build and Use Agent-Base models in Social Science / N. Gilbert, P. Terna. // Mind & Society. 1999. № 1. P. [57–72].

2. Tobias, R. Evaluation of Free Java-libraries for Social-Scientific Agent Based Simulation / R. Tobias, C. Hofmann. // Journal of Artificial Societies and Social Simulation. 2004. № 7. P. [3–5].
3. Horling, B. The Soft Real-Time Agent Control Architecture / B. Horling, V. Lesser, V. Regis, T. Wagner. // Autonomous Agents and Multi-Agent Systems. 2006. № 12. P. [35–91].
4. Шутов, А. Создание распределенной системы экономического моделирования на основе RePast и Swarm / А. Шутов А, С. Капустин. 2006. 10 с.
5. Leow, R. Running C++ model under Swarm Environment / R. Leow, R. Standish. // 2005. 20 p.
6. SObjectizer. Online: http://sobjectizer.sourceforge.net/. 2008.
7. Fowler, M. Patterns of Enterprise Application Architecture / M. Fowler. Addison-Wesley Professional: 2002. 560 p.
8. Bulka, D., Mayhew D, Efficient C++ Performance Programming Techniques / D. Bulka, D. Mayhew. Addison-Wesley Professional: 1999. 336 p.
9. Bellas, F. An Agent-based Architecture for Building CORBA Distributed Systems. / F. Bellas, R. Juanes, N. Rodriguez, A. Viia. // Annual ACM Symposium on Principles of Distributed Computing. 1998.
10. Hofmeister, C. Applied Software Architecture / C. Hofmeister, R. Nord, D. Soni. Addison Wesley: 2000. 432 p.
11. Vandevoorde, D. C++ Templates: The Complete Guide / D. Vandevoorde, N Josuttis. Addison-Wesley Professional: 2002. 552 p.
12. Madina, D. A system for reflection in C++ / D. Madina, R. Standish // Proceedings AUUG 2001: Always on and everywhere. № 12. P. [207–207].
13. Jacobson, I. The Unified Software Development Process / I. Jacobson, G. Booch, D. Rumbaugh. Addison-Wesley Professional: 1999. 512 p.
14. Russian Software Developer Network. Online: http://www.rsdn.ru. 2008.
15. Karlsson, B. Beyond the C++ Standard Library: An Introduction to Boost / B. Karlsson. Addison Wesley Professional: 2005. 432 p.
16. Boost C++ libraries. Online: http://www.boost.org. 2008.
17. Huston, S. ACE Programmer's Guide, The: Practical Design Patterns for Network and Systems Programming / S. Huston, J. Johnson, U. Syyid. Addison Wesley: 2003. 544 p.
18. Охотников, Е. Соотношение SObjectizer и FIPA Abstract Architecture / Е. Охотников. 2006. 12 с.
19. Охотников, Е. SObjectizer: I Love This Game! / Е. Охотников. // RSDN Magazine. 2005. № 5. С. [19–21].

ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ – ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

представляет свои периодические издания

ЭКОНОМИЧЕСКИЙ ЖУРНАЛ ВЫСШЕЙ ШКОЛЫ ЭКОНОМИКИ

ЕЖЕКВАРТАЛЬНЫЙ НАУЧНО-ИНФОРМАЦИОННЫЙ ЖУРНАЛ

Издается с 1997 г.

Главный редактор –
Евгений Евгеньевич Гавриленков

Журнал освещает теоретические и прикладные проблемы экономической науки. В каждом* номере – статьи ведущих российских экономистов. Рецензии, методологические и лекционные материалы. Эксклюзивные статьи зарубежных экономистов. Данные официальной статистики по широкому кругу вопросов.

Каталог Агентства «Роспечать» – индекс 79264
Объединенный каталог «Пресса России» – индекс 29233

Координаты редакции:
101990 Москва, ул. Мясницкая, 20, офис 235
e-mail: redact@hse.ru, тел./факс: (495) 628-0442

РОССИЙСКАЯ ЭКОНОМИКА: ПРОГНОЗЫ И ТЕНДЕНЦИИ

ЕЖЕМЕСЯЧНЫЙ СПРАВОЧНО-АНАЛИТИЧЕСКИЙ ЖУРНАЛ

Издается с 1993 г.

Главный редактор –
Елена Анатольевна Иванова

Журнал освещает состояние, динамика и дает прогноз основных социально-экономических индикаторов. В каждом номере – хроника событий экономики. Результаты конъюнктурных опросов предприятий. Самые свежие данные. Аналитический материал представлен с использованием таблиц, графиков и диаграмм.

Каталог Агентства «Роспечать» – индекс 79275
Объединенный каталог «Пресса России» – индекс 40548

Координаты редакции:
117312 Москва, ул. Вавилова, 7, офис 203
E-maii: id.hse@mail.ru Тел./факс: (495) 772-9571

РАСПРОСТРАНЯЮТСЯ ПО РОССИИ И ДРУГИМ СТРАНАМ СНГ. ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ – НА САЙТЕ: www.hse.ru