Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Норильский индустриальный институт»

А.Г. Куприяшкин

ОСНОВЫ МОДЕЛИРОВАНИЯ СИСТЕМ

Учебное пособие

Норильск 2015

УДК 004.738(075.8)
ББК 22.183.4я73
К 92

Куприяшкин, А.Г. Основы моделирования систем [Текст]: учеб. пособие / А.Г. Куприяшкин; Норильский индустр. ин-т. – Норильск: НИИ, 2015. – 135 с.

ISBN 978-5-89009-628-9

Учебное пособие соответствует Федеральным государственным образовательным стандартам по направлению подготовки (квалификация (степень) «бакалавр») «Информационные системы и технологии» Норильского индустриального института и отвечает требованиям учебного процесса.

Изложены теоретические вопросы учебных курсов «Имитационное моделирование» и «Основы математического моделирования».

Основная часть материала посвящена моделированию динамических систем в среде AnyLogic. Показаны основные этапы разработки моделей – от постановки задачи до создания анимированной презентации.

Предназначено для аудиторной и внеаудиторной самостоятельной работы студентов всех профилей (бакалавров) и всех форм обучения.

Утверждено редакционно-издательским советом Норильского индустриального института в качестве учебного пособия

Рецензенты: В.В. Михайлов, д.т.н., профессор, ведущий научный сотрудник Санкт-Петербургского института информатики и автоматизации РАН;

Ю.В. Маловичко, к.т.н., доцент, декан Горно-технологического факультета Норильского индустриального института

ISBN 978-5-89009-628-9

ВВЕДЕНИЕ

Я мыслю, следовательно, моделирую.
НЕ Декарт

Руководствуясь жизненным опытом и научными знаниями, человек строит модели – от бумажных корабликов до картины мира. Чем они богаче и чем точнее мы можем ими оперировать, тем лучше наше сознание, наша «самая важная модель» соответствует реальности и находит способы ее изменения.

Моделирование – самое эффективное средство поддержки принятия решений, а по словам Ричарда Докинза – «один из самых интересных способов предсказывать будущее» [12].

Теоретические предпосылки этого утверждения формировались на протяжении веков. В основу математического моделирования легли: математический анализ, теория вероятностей, численные методы, теория подобия. В XX в. появилась база практического приложения моделей: математическое программирование; теория массового обслуживания; теория алгоритмов; теория систем; кибернетика.

Другая, «фактологическая», основа моделирования – стремительно растущий потенциал знаний фундаментальных и прикладных наук.

В сочетании с современным технологическим прорывом эти основы создают необычайные возможности построения моделей, ограниченные лишь смелостью исследователя. Перечислим только злободневные глобальные темы, которые проходят непрерывную проверку моделированием: экономика, политика, экология.

Моделирование уверенно помогает понять, как устроен мир. Можно надеяться, что с его помощью мы когда-нибудь узнаем, как работает и наша «самая важная модель».

1. ОСНОВНЫЕ ПОНЯТИЯ

Моделирование как научный метод

Моделирование дает предположительную информацию о неком фрагменте реальности. После определенных проверок она может оказаться истинной или ложной и потребовать построения новых моделей.

В науке, наряду с наблюдением, измерением, экспериментом и сравнением, эта процедура выступает как один из общенаучных методов. Однако моделирование можно рассматривать как особый интегрирующий метод. Его эффективность и универсализм возрастают по мере развития информационных технологий. В силу разных причин объект может быть недоступен (слишком мал или велик, далеко расположен, дорог, прекратил существование, например, в результате аварии). Исключительная польза моделирования заключается в том, что можно экспериментировать не с самой системой, а с его аналогом – моделью.

Моделирование – процесс отражения свойств одного объекта (оригинала) в другом объекте (модели). Это могут быть объекты «как есть» в целом и (или) их отдельные сущности – процессы и явления. Явления – например, поведение животного, состояния погоды – рассматриваются как сложные процессы.

В основу моделирования заложена процедура формализации – перевод свойств объекта на язык понятий предметной области, алгоритмов и математики.

Подобие модели объекту

Объект и модель находятся в отношении сходства, т.е. модель по каким-то признакам должна быть подобна изучаемому объекту. Это явление называют ***изоморфизмом*** (от греч. *isos* – равный и *morphe* – форма). Различают три вида подобия.

Первый вид подобия – подобное масштабирование. Примеры такого подобия: модели автомобилей, самолетов, кораблей сооружений и т.д.

Второй вид подобия – косвенное подобие *(математическая аналогия)*. Удачный математический аналог из других областей знаний может сильно упростить построение модели и ее анализ. Так, очень многие физические процессы могут быть описаны уравнениями, общий вид которых $q = -\Theta \operatorname{grad} x$ (рис. 1.1).

Перенос массы вещества – закон Фика:

$$j = -D \cdot dC/dx,$$

где D – коэффициент диффузии; C – концентрация; x – текущая координата.

Перенос тепла – уравнение Фурье:

$$q = -\lambda \cdot dT/dx,$$

где q – тепловой поток; T – температура; λ – коэффициент теплопроводности.

Перенос электричества – закон Ома:

$$i = -c \cdot du/dx,$$

где i – сила тока; c – характерная проводимость; u – потенциал.

Рис. 1.1. Тройная аналогия процессов переноса

Аналогичны законы Кулона и всемирного тяготения.

Примером также может служить подобие электрических и механических явлений:

- колебание физического маятника:

$$T = 2\pi\sqrt{l/g};$$

- пружинного маятника:

$$T = 2\pi\sqrt{m/k};$$

- колебательного контура:

$$T = 2\pi\sqrt{L \cdot C}.$$

Третий вид подобия – условное подобие или подобие по соглашению. Примерами являются когнитивные модели (рис. 1.2), географические карты, масштабированные чертежи сооружений, зданий, структурные схемы (модели системного анализа). При этом внешне сходство объекта и модели может не соблюдаться.

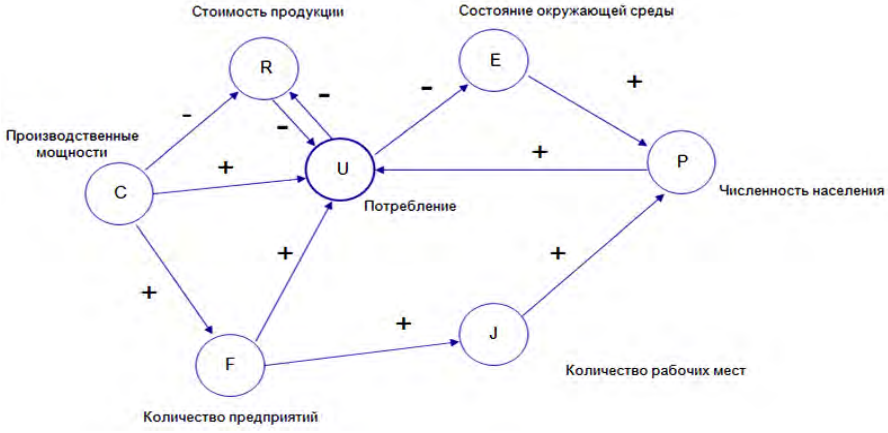


Рис. 1.2. Когнитивная модель потребления промышленной продукции (энергии, металлов и т.п.): + – положительные связи (влияния); – – отрицательные связи (влияния)

Таким образом, объект моделирования и модель могут быть любой природы – материальными или абстрактными. Например, макет самолета – это материальная модель. Схема производства – абстрактная модель. Уравнения физики – это описание абстракций разных явлений материального мира. Модели могут быть и абстракциями других моделей. Наследование (создание одних классов на базе других) в объектно-ориентированном программировании наиболее характерный пример таких построений.

Адекватность моделей

Вид и свойства будущей модели определяются целями исследователя, использующего этот инструмент. В модели отражаются свойства объекта, соответствующие этим це-

лям, которые определяют и уровни детализации модели. В первую очередь должны быть определены *существенные* свойства оригинала, характеризующие его как некую проблему, которую нужно решить с помощью модели.

При этом стоит помнить, что знать *все* свойства предмета вашего исследования нельзя. К тому же не будем забывать, метод – это инструмент, а универсальных инструментов не бывает. Означает ли это, что моделирование – ненадежный помощник? Нет. Во-первых, существует принцип множественности моделей. В соответствии с ним можно, а иногда – необходимо построить несколько моделей, позволяющих рассмотреть объект как проблему с различных позиций. К соответствующим решениям (моделям) можно идти, используя разные подходы. Например, создание модели поведения человека будет зависеть от разработки разных целей:

1) добиться антропоморфной кинематики компьютерной модели тела человека;

2) получить модель характерных психических реакций человека;

3) смоделировать реакции различных социальных групп людей.

Во-вторых, существуют специальные процедуры проверки, является ли модель точным представлением реальной системы, т.е., *адекватна* ли модель системе.

При *верификации,* т.е. проверке достоверности модели, определяется, правильно ли концептуальная модель (модельные допущения) преобразована в компьютерную программу.

Валидация – это процесс, позволяющий установить, является ли модель точным представлением системы для конкретных целей исследования.

Определяющим моментом в этих процедурах является положение: «модель и ее результаты достоверны, если … руководители проекта признают их правильными» [1]. В итоге, если модель «адекватна», ее можно использовать для принятия решений относительно системы, которую она представляет, как если бы они принимались на основании экспериментов с реальной системой.

В-третьих, итоговый результат (т.е. «хорошая» или «плохая» модель получится) зависит от личности разработчика. Моделирование, как метод научного познания, предполагает творческий подход к объекту и целям исследования.

В этом виде научного производства не обойтись без развитого воображения, умения анализировать и делать обобщения. Хорошие модели – это «минитеории» и их создание требует нестандартного мышления.

Системный подход в моделировании

Материальные сущности – это собрание взаимосвязанных и взаимодействующих элементов, образующих системы разного уровня сложности.

Топологическая сложность определяется числом элементов и связей. *Функциональная* сложность характеризуется процессами (поведением) системы и ее элементов. По этим признакам можно найти положение данного объекта в иерархии систем (вплоть до мировой) и сформировать предметную область моделирования.

На нижних уровнях главенствуют индивидуальные поведения, фиксированные физические связи, точные размеры, расстояния, скорости, времена. На верхних уровнях существенны глобальные причинные зависимости, тенденции, сценарии, динамика потоков, влияние обратных связей и окружающей среды, моделирование которой может быть выделено в отдельную и весьма непростую задачу.

В сложных системах состав элементов и типы связей могут существенно изменяться. Такие системы могут расти, стареть, умирать, перестраиваться и эволюционировать.

Систему как «организованно работающую целостность» характеризуют состояния и особенности их смены (рис. 1.3).

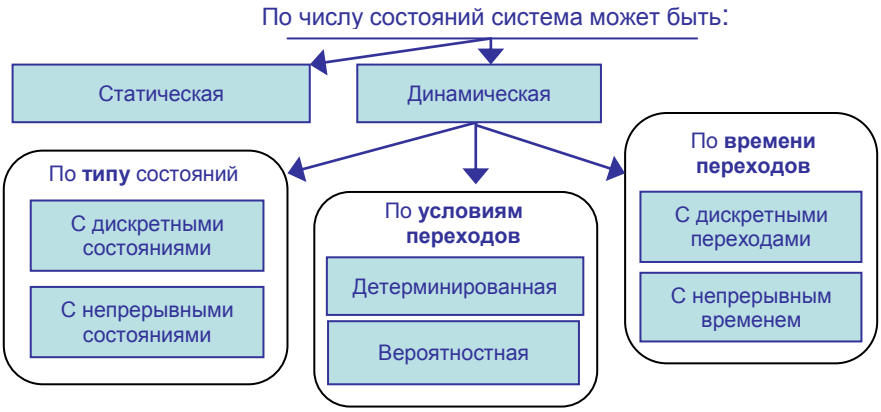


Рис. 1.3. Типы систем

При определении типа системы принимается решение, в рамках какой типовой математической схемы будет строиться модель (рис. 1.4).

Переходы / Состояния	Детерминированные	Вероятностные
Непрерывные	*D*-схемы	*Q*-схемы
Дискретные	*F*-схемы	*P*-схемы

Рис. 1.4. Типовые математические схемы моделирования систем: *D* (*dynamic*) – модели, вида *dx/dt* = *f*(*x*); *Q* (*queuing*) – модели систем массового обслуживания; *F* (*finite automata*) – конечные автоматы; *P* (*probabilistic automata*) – вероятностные автоматы

Построение математических моделей

Для реализации соответствующей математической схемы возможно применение двух методов.

1. Аналитическое моделирование предполагает использование систем алгебраических, дифференциальных, интегральных уравнений, связывающих выходные переменные с входными. Уравнения дополняются системой ограничений.

Аналитическое решение можно найти, если параметров не много и система обладает линейным поведением.

Главным препятствием в реализации этого метода может оказаться полное или частичное отсутствие «формул» процессов.

Самым доступным инструментом аналитического моделирования является Excel. Возможности этой среды для создания моделей обеспечиваются наличием инструментов анализа, большого количества математических функций, реализацией численных методов, встроенной поддержки Visual Basic for Applications. Имитационный потенциал Excel ограничен, и построение моделей динамических систем представляет определенные трудности. Использование программных возможностей этого пакета позволяет частично преодолевать эту проблему.

2. *Имитационное моделирование.* Здесь математическая модель воспроизводит алгоритм («логику») функционирования исследуемой системы во времени при различных сочетаниях значений параметров системы и внешней среды.

Имитационное моделирование – это построение компьютерных моделей и проведение экспериментов над ними [2].

Имитационный подход применяют, когда параметров много, зависимости не линейны, система имеет качественно различные состояния (непрерывные процессы прерываются дискретными переходами), траекторию во времени (объект эволюционирует), обладает вероятностным поведением и обратными связями. Имитационный подход незаменим, когда нужно сопроводить модель анимационной презентацией (*симуляцией*). При создании виртуальных тренажеров, моделей движения транспорта и пешеходов это может оказаться главной задачей моделирования.

Сочетание имитационного и аналитического методов возможно в рамках одной *комбинированной* модели и способно реализовать практически любые задачи.

Задачи моделирования систем

К любому объекту как предмету моделирования можно приложить две задачи «верхнего уровня». Во-первых, нужно определить, «из чего он сделан». Эта процедура реализуется с помощью системного анализа. Любой объект сначала рассматривается как «черный ящик», обладающий входами (вводами) и выходами (выводами) (рис. 1.5, *а*).

а) ***б)***

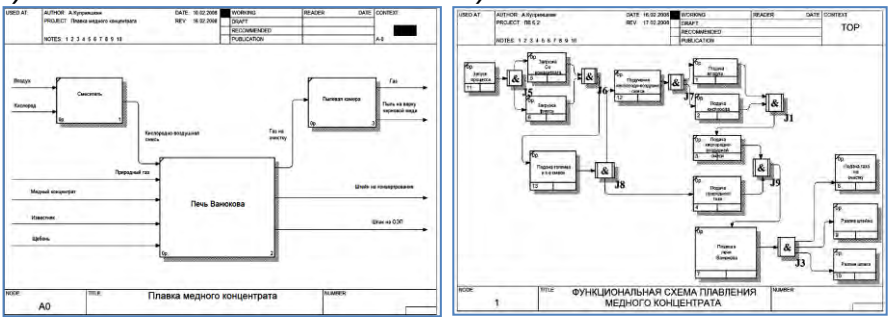


Рис. 1.5. Структурная (*а*) и функциональная (*б*) схемы металлургического процесса

Затем производится *декомпозиция* системы: находятся все ее элементы или подсистемы, связи между ними, принадлежащие им входящие и выходящие потоки и т.д. Процесс завершается построением структурной модели. В соответствии с требованием изоморфизма в структурной модели добиваются *качественной* адекватности оригиналу.

Вторая задача – выяснение, «как объект работает» – может быть реализована в несколько этапов:

1) строится функциональная, *количественно* адекватная оригиналу модель, которая отражает алгоритм работы системы (см. рис. 1.5, *б*).

2) создается имитационная модель, вид которой определяется особенностями функционирования системы и задачами визуализации;

3) планируются и проводятся модельные эксперименты;

4) в зависимости от результатов экспериментов принимаются решения по внесению изменений в модель, поиску дополнительной информации о системе, проведению новых испытаний;

5) делаются выводы и рекомендации.

Направления и инструменты имитационного моделирования

Несмотря на широту понятия «имитационное моделирование», существует определенная специализация его задач. В связи с этим выделяют следующие направления этого метода и наиболее соответствующее им программное обеспечение:

- моделирование динамических систем (MATLAB, VisSim, Lab View, Easy5);
- дискретно-событийное моделирование (GPSS, SYMULA, Arena, AutoMod, Enterprise Dynamics, FlexSim);
- агентное моделирование (Net Logo, Swarm, Repast, ASCAPE);
- системная динамика (VenSim, PowerSim, iSink).

Перечисленные программные пакеты обладают как несомненными достоинствами, так и имеют свои недостатки, к которым относятся – узкая направленность, нелокализованный интерфейс, привязка моделей к среде разработки (не автономность) и дороговизна (MATLAB). Следствие этого – формирование непредставительных сообществ разработчиков.

Инструмент имитационного моделирования AnyLogic обладает рядом преимуществ, главное из которых – возможность реализации всех направлений имитационного моделирования в одной модели.

2. ОСНОВЫ МОДЕЛИРОВАНИЯ В СРЕДЕ ANYLOGIC

2.1. Общие сведения

Использование AnyLogic предоставляет уникальную возможность войти в мир моделирования, имея лишь базовую подготовку в области информационных технологий.

Это современная среда разработки моделей на языке Java с русскоязычным графическим интерфейсом и тщательно продуманной контекстной справочной системой. AnyLogic содержит большую библиотеку визуальных компонентов. Разработчик может также создавать и добавлять в среду собственные компоненты. Модели сохраняются как Java-апплеты. В профессиональной версии работает отладчик и можно создавать автономные JAR-файлы. AnyLogic-модели обладают хорошими средствами 2D–3D симуляции, интерактивности и развитыми возможностями проведения экспериментов (в том числе оптимизационных).

После установки программы нужно пройти регистрацию на сайте компании [11] и получить ознакомительный (бесплатный) или постоянный ключ. Регистрация с постоянным ключом дает возможность получать от группы поддержки помощь в разработке моделей по Интернету, а также обновления программы. Необходимо отметить, что разработчики регулярно обновляют учебную версию AnyLogic, сближая ее возможности с профессиональной версией более ранних релизов.

При запуске AnyLogic отображается начальная страница (рис. 2.1). После этого можно выбрать пример из обширного списка учебных моделей и посмотреть ее работу.

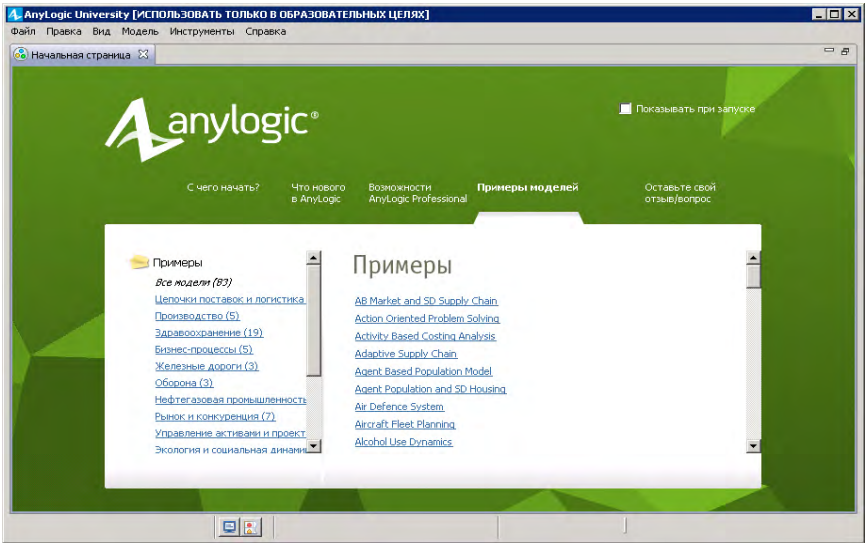


Рис. 2.1. Стартовая страница AnyLogic

Другой вариант – закрыть стартовую страницу и начать работу «с нуля». Перед этим можно снять флажок у метки *Показывать при запуске*, чтобы в следующий раз сразу же начинать работу с моделью.

2.2. Элементы пользовательского интерфейса

Окно *AnyLogic* в начале работы над новой моделью показано на рис. 2.2, где в верхней части находится строка меню и панели инструментов. В центре расположена область построения модели – *Графический редактор* (вкладка *Main*). По периферии – рабочие панели среды моделирования с соответствующими вкладками и кнопками управления размерами. Их можно вынести за пределы главного окна *AnyLogic* или закрыть совсем.

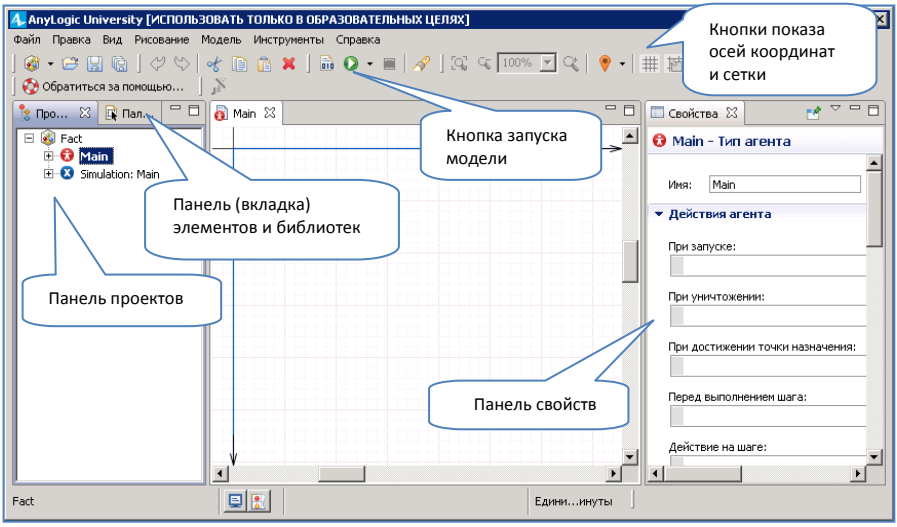


Рис. 2.2. Главное окно *AnyLogic*

Вернуть на прежнее место закрытое окно можно, активизировав соответствующий пункт главного меню *Вид* (рис. 2.3).

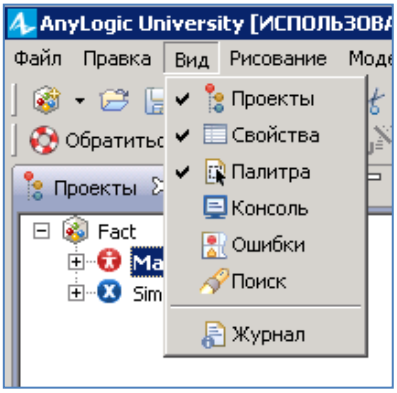


Рис. 2.3. Меню выбора рабочих окон

В современных средах программирования разработка приложений организуется в виде проектов.

В панели проектов AnyLogic отображаются древовидные диаграммы вновь созданных и загруженных с диска

моделей. Если перед окончанием сеанса работы с программой они не были закрыты специальной командой меню *Файл–Закрыть*, то при очередном запуске AnyLogic он загрузит их в рабочую среду и покажет в окне проектов. Эта опция весьма полезна, так как позволяет легко переносить (копировать) из одной модели в другую элементы, фрагменты кода и готовые классы.

В начале работы с моделью дерево проекта содержит, как минимум, два компонента: класс активного объекта *Main* (главный класс, агент верхнего уровня) и простой эксперимент *Simulation:Main.*

Модель запускается в виде имитационного эксперимента кнопкой на панели инструментов или клавишей *F5*. Процесс отображается в специальном окне презентации. На начальном этапе проект представляет собой «пустую» заготовку, поэтому интерфейс и детали настроек эксперимента и окна презентации рассмотрим позже.

Создание модели в общем виде сводится к последовательному размещению в окне графического редактора функциональных элементов, определению их свойств, разработке новых классов и блоков программного кода и проведению экспериментов. Неизбежные спутники на этом пути – ошибки. Они отображаются в специальном окне с указанием места или объекта и достаточно подробным описанием проблемы на русском языке. Полезной будет подсказка кода, вызываемая комбинацией клавиш *Ctrl-Space*, и отладчик, доступный в профессиональной версии AnyLogic.

2.3. Первый проект

Рассмотрим процесс создания моделей в среде AnyLogic на примере упрощенной версии калькулятора, вычисляющего факториал.

Последовательно выберем пункты меню *Файл–Создать–Модель* или щелкнем по пиктограмме палитры инструментов  . Открывается диалоговое окно начальных настроек модели (рис. 2.4).

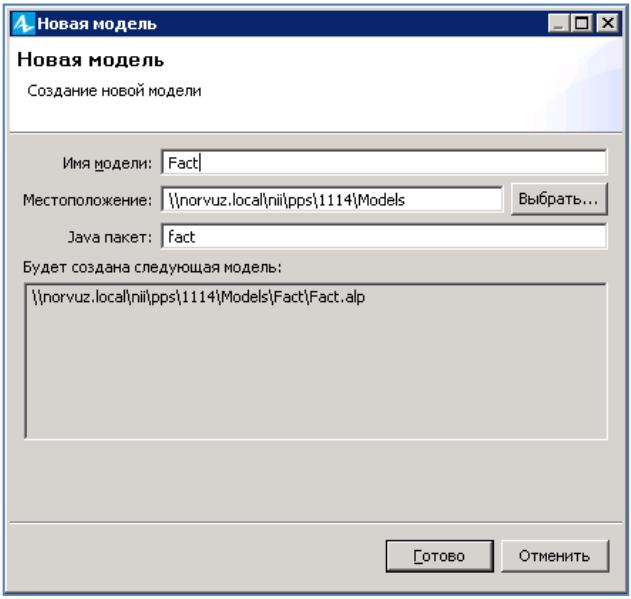


Рис. 2.4. Окно начальных настроек

По умолчанию AnyLogic предлагает имя проекта, в виде: *Model#,* где # – номер, соответствующий порядку создания модели.

Изменим его на *Fact* и, если необходимо, выберем другое местоположение, где будут сохраняться все файлы модели. Поскольку используются язык Java, компоненты приложения (классы) собираются в пакеты. Таким образом, в AnyLogic можно создавать несколько моделей с разными именами, хранящих ресурсы в одном пакете. При попытке сохранить модель под новым именем, например, *Fact_1* – пункт меню *Файл–Сохранить как* – имя пакета по умолчанию останется прежним.

Если его не изменить (например на *fact_1*), то появятся две модели с разными именами и общим пакетом. Одновременно работать с ними (загружать оба проекта) будет нельзя.

> В Java учитывается регистр букв в именах. Название модели и ее компонентов может содержать русские буквы.

Нажмем кнопку *Готово.*

В качестве основы дизайна приложения выберем *Калькулятор,* который есть в стандартном наборе программ Windows (рис. 2.5).

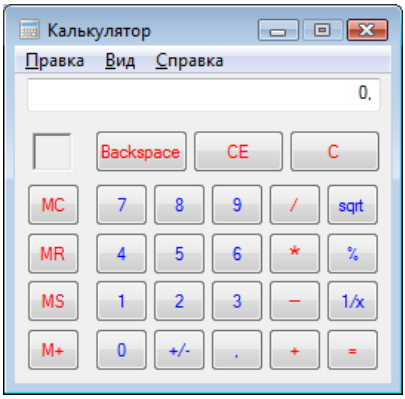


Рис. 2.5. Калькулятор Windows. Вид *«Обычный»*

В результате получится вид как на рис. 2.6.

Рис. 2.6. Вид модели калькулятора

Сформулируем задачу: построить функциональную модель калькулятора. С помощью цифровых кнопок набирается значение «*n*», которое отображается на «цифровом табло». Кнопкой *Backspace* можно последовательно убирать последние цифры. Нажатие кнопки *n!* приводит к появлению значения факториала в цифровом табло. Нажатие кнопки *C (Clear)* очищает табло.

Нам понадобятся две переменные, элементы управления, элементы презентации, а также фрагменты, определяющие логику модели.

Опишем алгоритм реализации задачи в виде последовательности шагов.

Шаг 1. Первая переменная будет принимать значение *n*, набираемое с помощью кнопок. Раскроем закладку палитры элементов. Перетащим в окно графического редактора элемент *Переменная,* как показано на рис. 2.7.

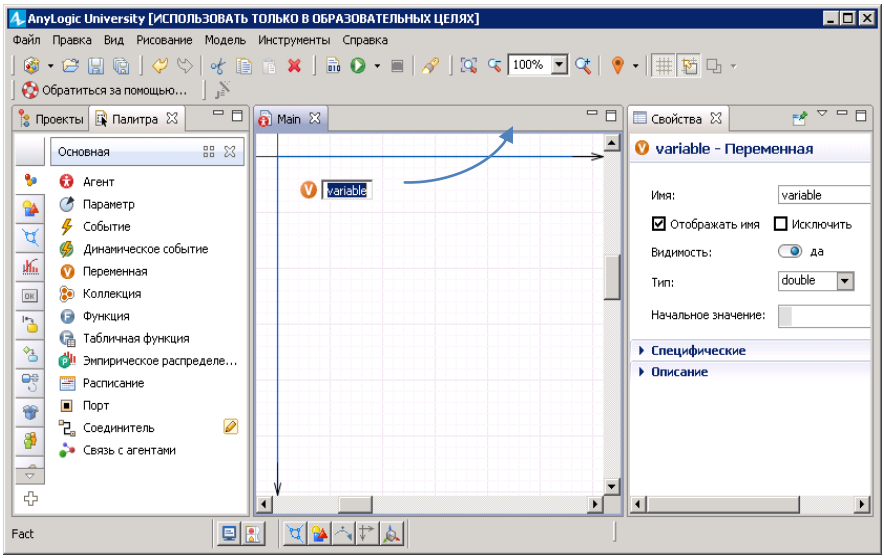


Рис. 2.7. Перетаскивание элемента *Переменная* в окно графического редактора

Изменим свойства переменой в соответствии с рис. 2.8.

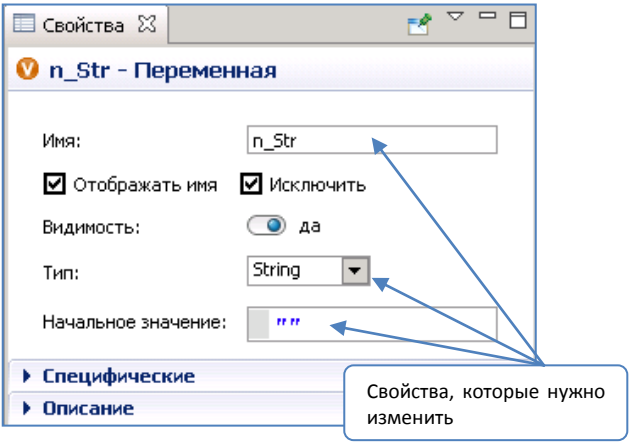


Рис. 2.8. Панель свойств переменной

Шаг 2. Изменим имя со стандартного *variable* на *n_Str.* Учтем правила языка Java. Так как это объекты, их имена (а также имена методов) должны начинаться со строчной буквы. Всем переменным AnyLogic устанавливает по умолчанию тип ***double*** (вещественный) и начальное значение, равное нулю (если его не изменить, в поле *Начальное значение* не показывается).

В данном случае нужна переменная, которая будет хранить «число», набираемое кнопками, т.е. его изображением на табло должна быть *строка* (а не сумма) цифр, позиция которых будет соответствовать их разрядам.

Изменим тип переменной на ***String*** (строковый) и установим с помощью двух кавычек ее начальное значение («пустая строка»). В противном случае оно всегда будет *null* (так как это переменная ссылочного типа).

Шаг 3. Введем в модель вторую переменную, которая будет принимать конечное значение параметра цикла, необходимое для вычисления факториала. Для этого повторим шаг 1. Изменим стандартное имя переменной на *n.* В свойствах переменной установим тип ***int*** (целый).

Шаг 4. Добавим в модель элемент *Текст* из группы палитры инструментов *Презентация*. Он будет отображать на табло введенное кнопками число *n*. Внесем изменения на панели свойств (рис. 2.9).

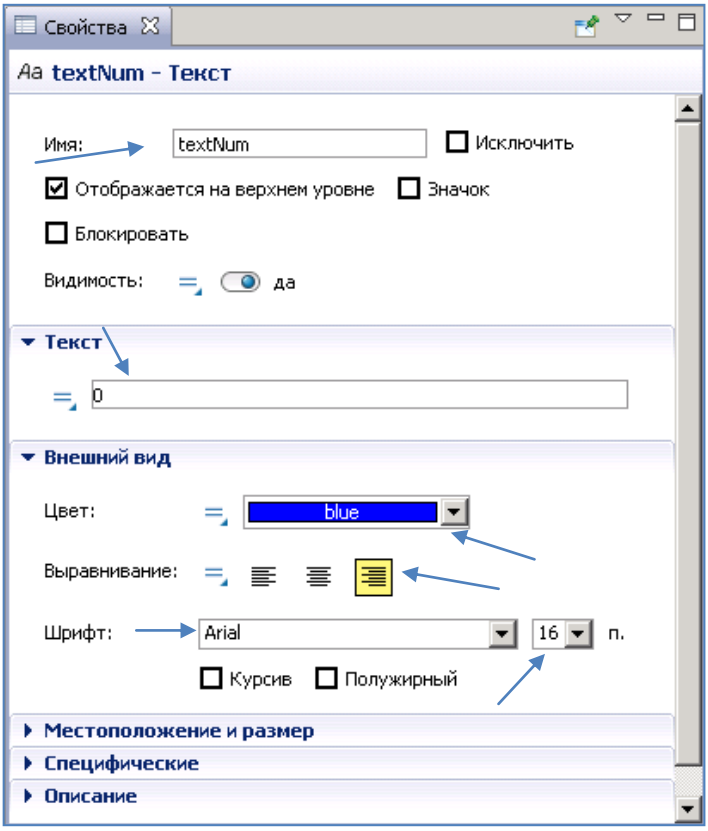


Рис. 2.9. Панель свойств элемента *Текст*
(стрелками показаны измененные свойства)

Шаг 5. Добавим в еще один элемент *Текст*. Назовем его *textFact*. Он будет показывать значение факториала на табло. Повторим для него операции шага 4. Выравнивание установим по левому краю.

Шаг 6. Поместим в поле графического редактора кнопку из группы *Элементы управления*. Чтобы сразу получить убедительный результат вычислений, пусть это будет кнопка с цифрой «*3*» (3!=6) (рис. 2.10).

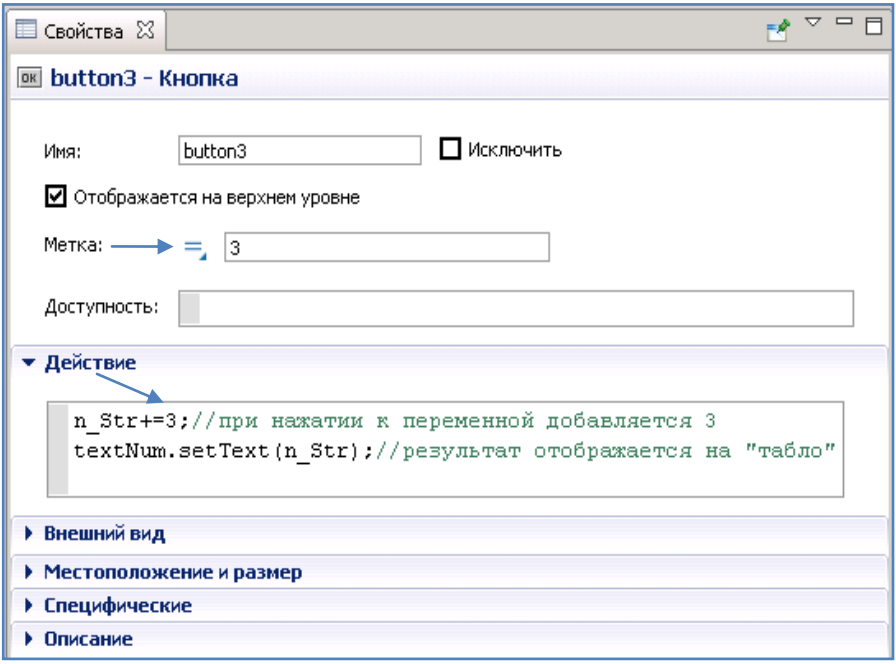


Рис. 2.10. Панель свойств элемента *Кнопка* (стрелками показаны изменения)

Впервые в модели появляется фрагмент кода на языке Java. Это всего лишь две строки. Первая изменяет значение текстовой переменной, вторая вызывает метод *setText()* элемента (объекта) *textNum*. Результат ясен из комментариев.

> В Java предусмотрены два вида комментариев: для строки кода – начинается с двух косых черт //, и для фрагмента, расположенного между парными символами /* и */.

Шаг 7. Поместим в поле графического редактора еще одну кнопку. Потянув за угловой маркер, немного увели-

чим ее размеры. Результатом ее нажатия будет вычисление факториала и вывод результата на табло. Свойства кнопки и вид окна редактора модели показаны на рис. 2.11.

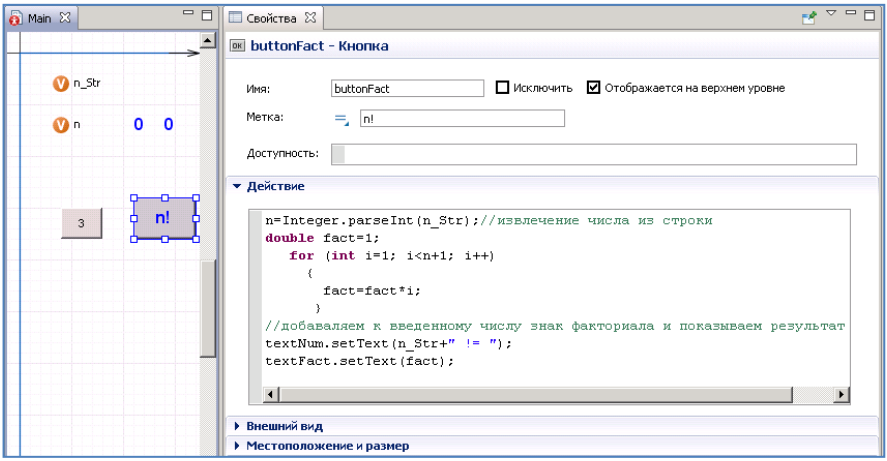


Рис. 2.11. Вычисление факториала по щелчку кнопки

При вычислении факториала невозможно напрямую использовать строковое представление числа в переменной *n_Str*, поэтому в первой строке кода «извлекаем» его с помощью класса-оболочки *Integer* и присваиваем результат переменной *n*.

Вычисление факториала будем осуществлять с помощью цикла с параметром.

В теле кода объявим вспомогательные переменные *fact* (тип вещественный) и параметр цикла *i* (целый), пробегающий значения от 1 до *n*.

Выполнение двух последних строк кода будет приводить к изменению соответствующих текстовых элементов. В *textNum* появятся значение *n*, символы «!» и «=». В *textFact* отобразится значение факториала.

Шаг 8. Запустим модель с помощью кнопки на панели инструментов или клавиши *F5*. Вначале откроется *Окно настроек эксперимента* (рис. 2.12).

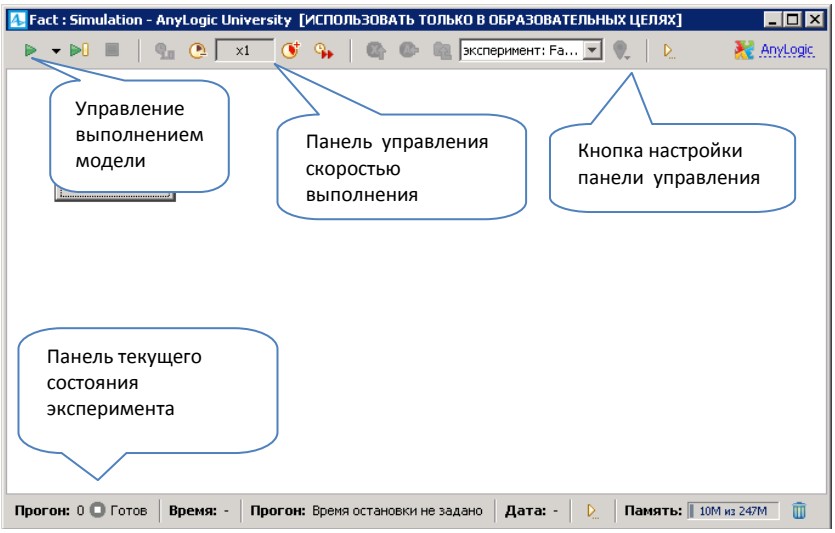


Рис. 2.12. Окно настроек эксперимента

Щелчком по кнопке запуска модели попадаем в *Окно презентации*. Модель работает. Последовательно нажимаем кнопки «*3*» и «*n*!». Текстовые элементы отображают корректный результат (рис. 2.13).

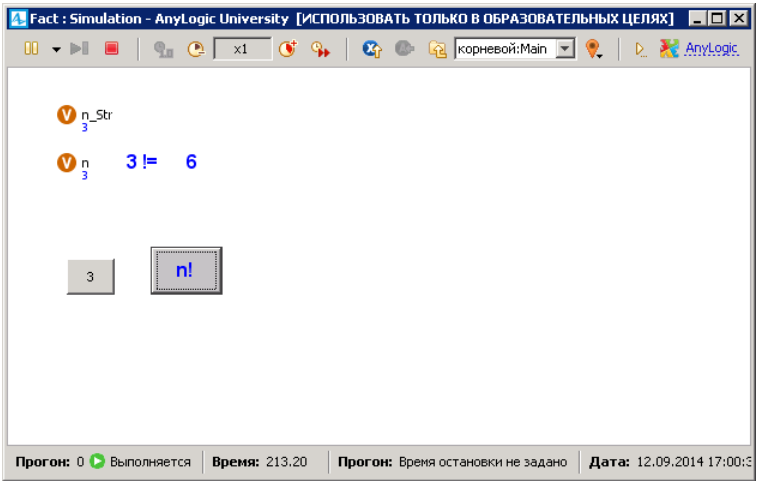


Рис. 2.13. Вид окна работающей модели

Дальнейшие действия будут сводиться к добавлению кнопок, оформлению дизайна «калькулятора» и вида готового приложения.

Шаг 9. Поместим элементы в соответствующие позиции. Цифровые кнопки «размножим» с помощью операций последовательного копирования-вставки уже имеющейся у нас кнопки «*3*». Они будут наследовать ее свойства, включая размеры и код. Необходимо изменить пункты свойств *Имя кнопки, Метка* и *Действие* (код). Так, для кнопки «*0*», код действий примет вид:

```
n_Str+=0;
textNum.setText(n_Str);
```

Шаг 10. Напишем фрагмент кода для кнопки *«C» (Clear):*

```
n_Str="";
textNum.setText("");
textFact.setText("");
```

Щелчок по ней приведет к начальным значениям строковой переменой *n_Str* и двух текстовых элементов, отображающих ввод *n* и вывод *n!*

Состояние модели на данном этапе показано на рис. 2.14.

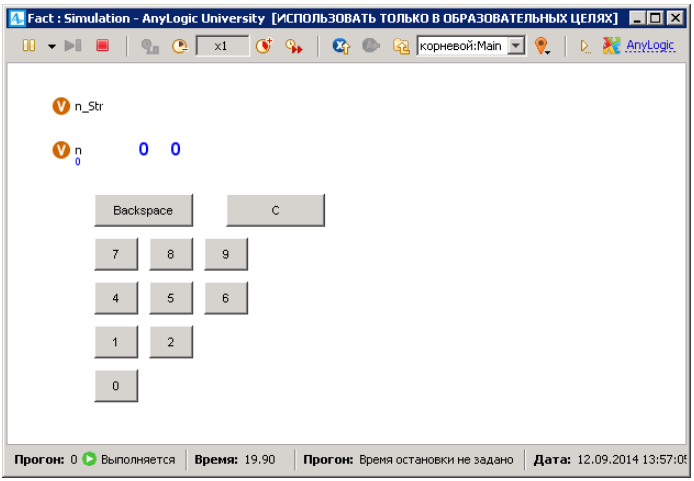


Рис. 2.14. Вид модели на стадии 9

Шаг 11. Окончательная стадия разработки модели. Поместим в окно графического редактора элемент *Скругленный прямоугольник* из группы *Презентация* панели *Палитра.* Он будет выполнять функцию электронного табло калькулятора. Подгоним его размеры под ширину клавиатуры. Чтобы он не закрывал текстовые элементы, щелчком правой клавиши мыши вызовем контекстное меню *Порядок–На задний план.*

> Новые элементы, помещаемые в окно графического редактора, последовательно располагаются на переднем плане.

В свойствах *Радиус* уменьшим радиус углов с 10 до 5 (рис. 2.15).

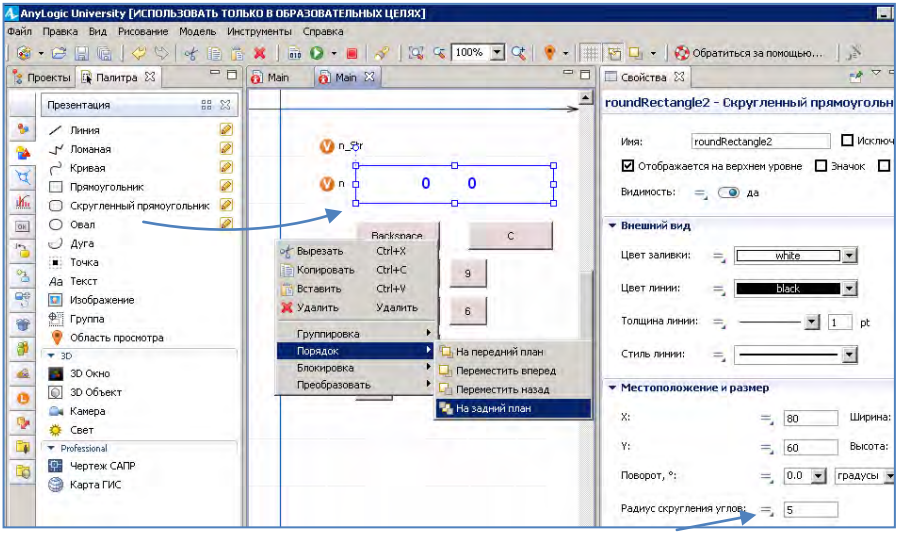


Рис. 2.15. Настройка электронного табло

Проделаем аналогичные действия с двумя другими прямоугольниками, которые будут имитировать корпус калькулятора. В свойствах первого прямоугольника уберем заливку.

Немного увеличим размер второго прямоугольника (рис. 2.16).

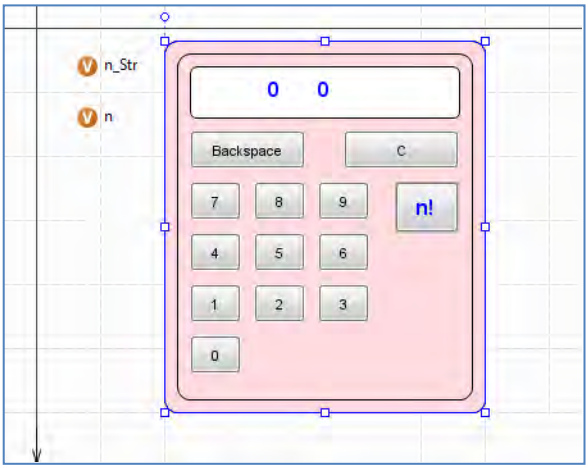


Рис. 2.16. Заключительная стадия разработки дизайна калькулятора

Шаг 12. Настройка приложения. Переместим переменные за пределы область показа, например, влево от оси *Y*. Выделим мышью все элементы калькулятора и передвинем их немного вниз и влево.

> В окне презентации по умолчанию показывается область графического редактора определенных размеров, расположенная под осями координат (ось ординат *Y* «перевернута».) Модель может быть больше или меньше этого пространства, поэтому область первоначального показа можно изменить и организовать несколько таких зон и навигацию между ними. Сделаем это позже на примерах других моделей.

В панели проектов мышью щелкнем по элементу *Simulation:Main,* затем по *frame* (рамка). Рядом с окном графического редактора появится закладка *Simulation.* Для показа калькулятора и работы с ним вместе с необходимыми полями, достаточно области размером 350 точек по горизонтали и 400 по вертикали.

Изменим *Свойства* рамки в соответствии с этими значениями, затем скорректируем другие элементы (рис. 2.17).

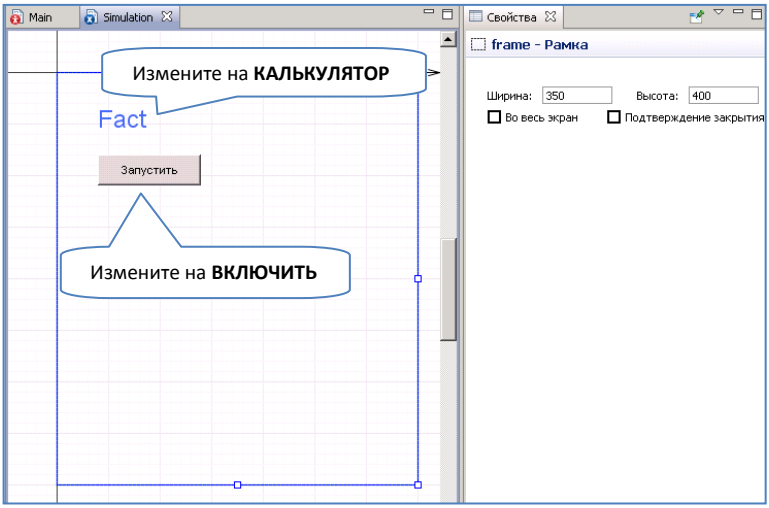


Рис. 2.17. Настройка области показа модели

В окончательном варианте приложение примет вид, как на рис. 2.18.

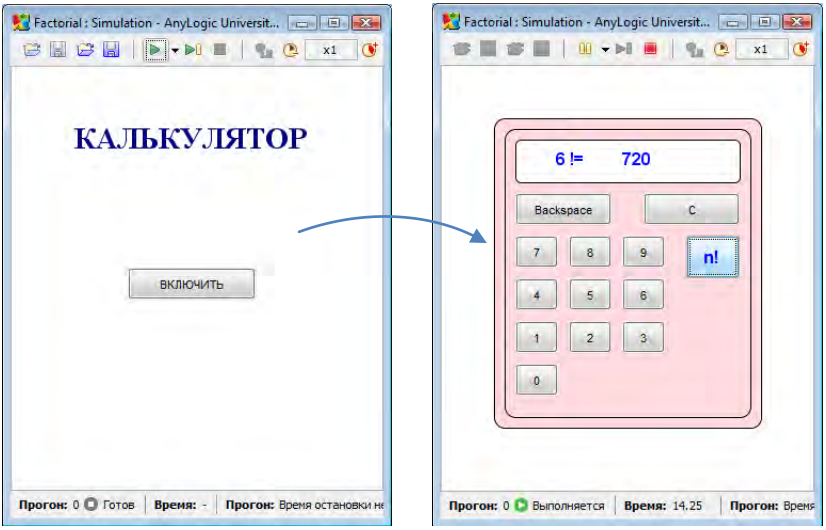


Рис. 2.18. Окончательный вариант модели

Шаг 13. Сохранение приложения. В AnyLogic можно экспортировать проект в независимый от среды разработки Java-апплет, запускаемый в браузере (рис. 2.19).

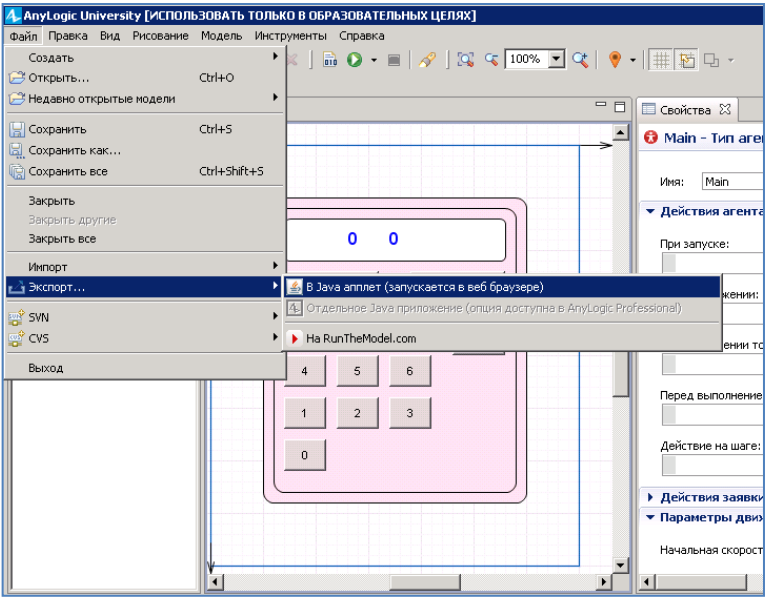


Рис. 2.19. Меню экспорта модели

В диалоговом окне выберем каталог для сохранения апплета (рис. 2.20).

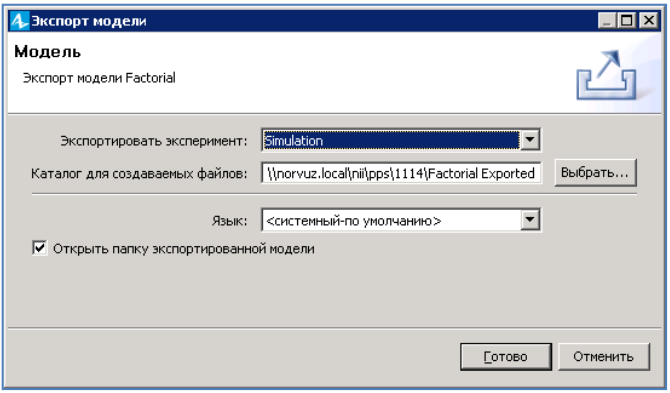


Рис. 2.20. Диалоговое окно экспорта модели в апплет

Приложение работает в браузере Internet Explorer (рис. 2.21).

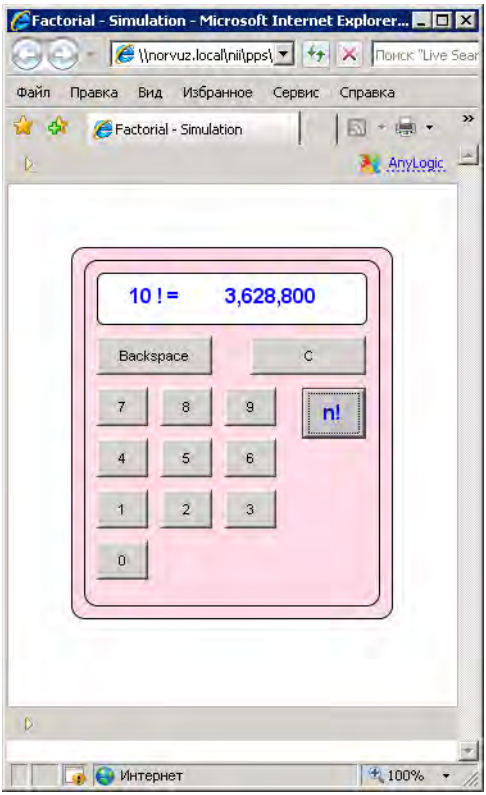


Рис. 2.21. Вид страницы с апплетом в браузере Internet Explorer

3. ДЕТЕРМИНИРОВАННЫЕ НЕПРЕРЫВНЫЕ СИСТЕМЫ

3.1. Общие сведения

Детерминированные системы с непрерывными переходами состояний моделируются в рамках математических *D*-схем. Они представляют собой системы дифференциальных и алгебро-дифференциальных уравнений. В таких моделях анализируют изменения параметров системы во времени и определяют ее качественные состояния – режимы.

Выделяют следующие режимы динамики системы:

1) стационарный – система находится в положении равновесия;

2) переходный – система движется из одного равновесного состояния в другое;

3) периодический – система через одинаковые интервалы времени проходит одинаковые состояния;

4) динамический – состояние системы непрерывно меняется во времени.

При моделировании физических систем учитывают начальные и конечные состояния, граничные условия (значения параметров на границах системы) и ограничения.

Рассмотрим простую модель реактора идеального смешения:

$$\frac{dCout}{dt} = (Cin - Cout)/tau,$$

где *Cin* и *Cout* – входная и выходная концентрации реагента; *tau* – среднее время пребывания реагента в аппарате.

Начальными состояниями данной системы будут $Cout(t = 0) = 0$; $Cin(t = 0) > 0$. Ограничение: $tau > 0$.

Если *Cin* = const, то система может иметь два режима – переходный (пока $Cout < Cin$) и стационарный, когда выходная концентрация *Cout* станет равной входной *Cin* (рис. 3.1).

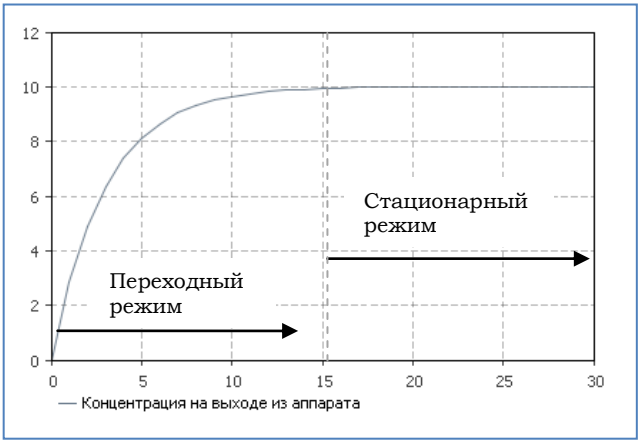


Рис. 3.1. График изменения выходной концентрации во времени

3.2. Моделирование систем с сосредоточенными параметрами

Модели с сосредоточенными параметрами строятся на основе дифференциальных уравнений в «обыкновенных» производных по одной переменной (времени или координате). Параметры в математических моделях таких систем в каждый конкретный момент времени имеют единственное значение.

В качестве примера построим имитационную модель пружинного маятника.

Груз лежит на поверхности, закрепленный на конце растянутой пружины. В горизонтальной плоскости на него действуют три силы (рис. 3.2).

Если убрать силу тяги $\vec{F}$, груз начнет колебательные движения около начала координат. Процесс будет протекать с затуханием, вызванным силой трения $\vec{F}_{тр}$, вектор которой будет менять знак.

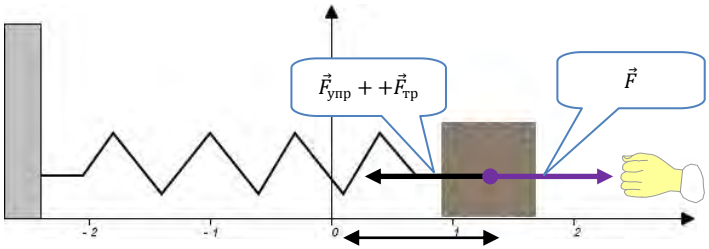


Рис. 3.2. Схема системы «Пружинный маятник»

Математическая модель, записанная в виде системы уравнений, имеет вид:

$$\begin{cases} m\dfrac{dV}{dt} = -cx - kV, \\ \dfrac{dx}{dt} = V, \end{cases}$$

где m – масса груза; x – координата; V – скорость; c – жесткость пружины; k – коэффициент трения. Начальные условия: $x(t = 0) = x_0$; $V(t = 0) = 0$.

Задачу решим в два этапа. Построим модели системы:

1) математическую («формализованную»);

2) имитационную интерактивную («неформализованную»).

Математическая модель

Для построения данной модели разработаем следующий алгоритм:

1. Создадим новую модель *Swing.*

2. В окне графического редактора поместим элементы системы дифференциальных уравнений – математическую модель. Перетащим элемент *Накопитель* из палитры *Системная динамика* (рис. 3.3).

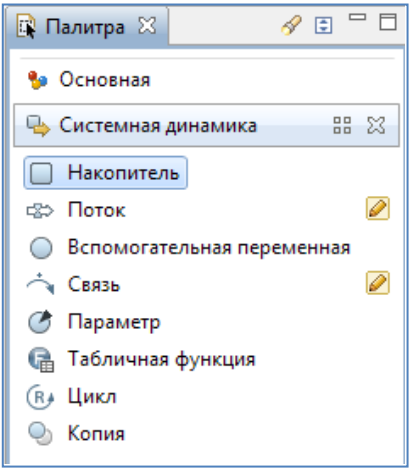


Рис. 3.3. Компоненты моделирования системной динамики

Переименуем элемент в *x*, установим *Свойства: Режим задания уравнения: Произвольный.* Введем формулу: *d(x)/dt = V.* Установим начальное значение (отклонение) (рис. 3.4).

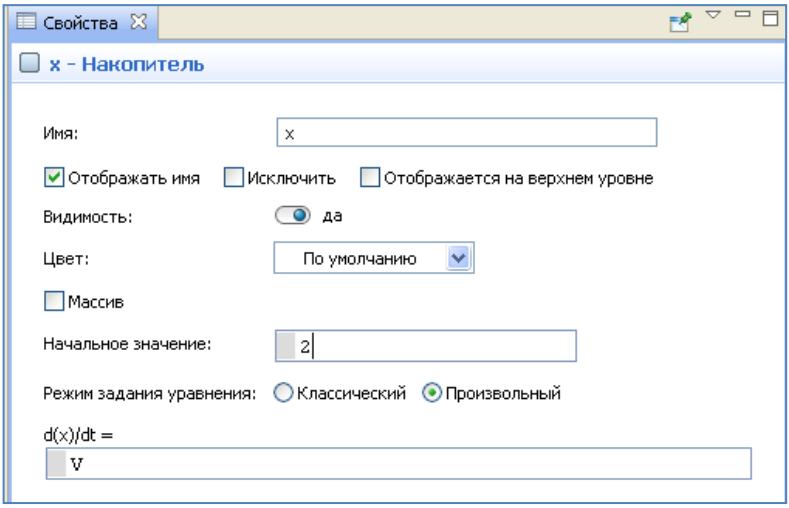


Рис. 3.4. Панель свойств координаты

3. Повторим действия предыдущего шага для элемента *V*. Вид уравнения для него:

$$d(V)/dt = -x*c-k*V.$$

(Для сокращения числа параметров примем в модели «условную» массу, равную единице).

4. Из группы *Системная динамика* введем в модель два параметра: жесткость пружины – *c* и коэффициент трения – *к*. Значения по умолчанию (равные нулю) изменим: для *c* – на 1, для *к* – на 0.1. На этом этапе логика модели показана на рис. 3.5.

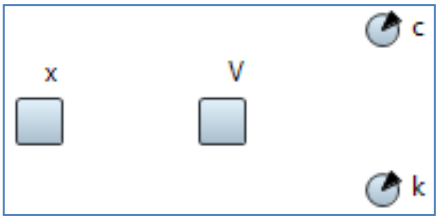


Рис. 3.5. Фрагмент окна графического редактора с элементами математической модели

5. Попробуем запустить модель. В панели ошибок появляются два сообщения (рис. 3.6).

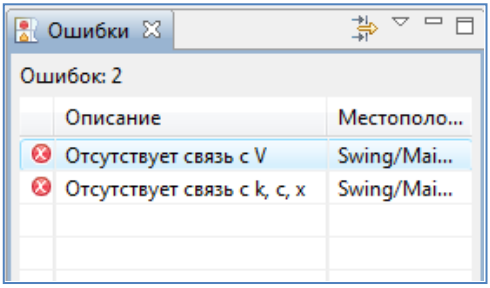


Рис. 3.6. Панель ошибок

Ситуация объясняется тем, что в нашей системе уравнений «уравнения есть, а системы нет». Элементы необходимо связать. Можно сделать это вручную с помощью элементов *Связь* (см. рис. 3.3). Однако проще щелкнуть

мышью по описанию ошибки, и AnyLogic сам направит нас в панель *Свойства элемента*, с которым связана данная проблема. Далее будем просто последовательно подтверждать предлагаемые средой действия (рис. 3.7).

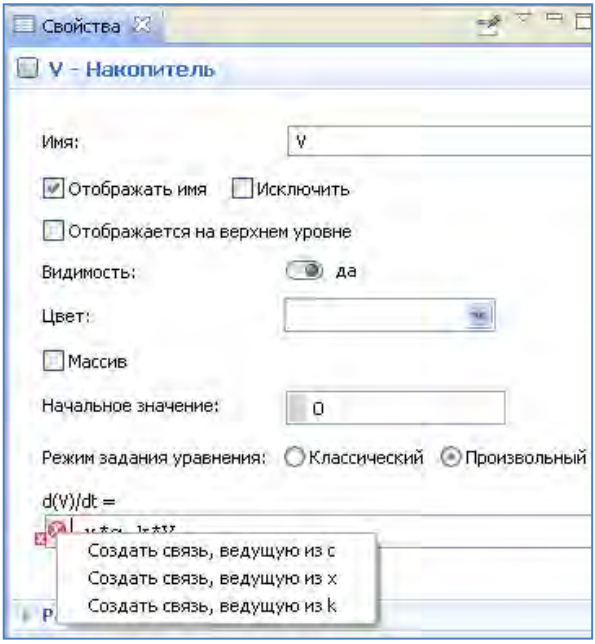


Рис. 3.7. Диалог исправления ошибок

Связи будут установлены с учетом «правильных» направлений (влияний) (рис. 3.8).

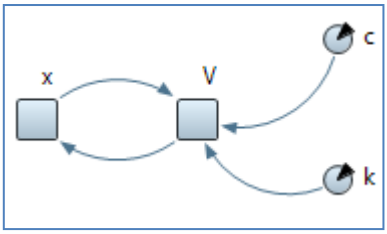


Рис. 3.8. Математическая модель с установленными связями

Снова запустим модель. В этот раз она работает без ошибок. Поставим выполнение на паузу. Щелкнем по одному из накопителей. Откроется «инспект» состояния элемента (рис. 3.9).

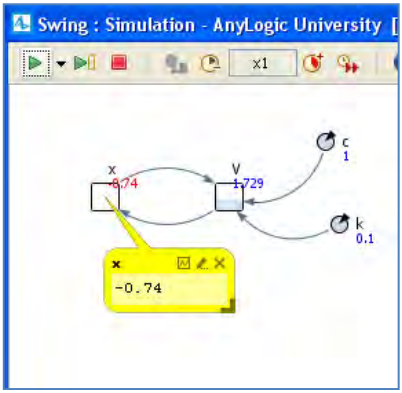


Рис. 3.9. Фрагмент презентации модели с открытым «инспектом»

С помощью мыши его можно перемещать и растягивать (изменять размеры). Кнопка … позволяет корректировать текущее числовое значение данной величины. Щелчок по кнопке … приводит к показу графика ее динамики во времени (рис. 3.10).

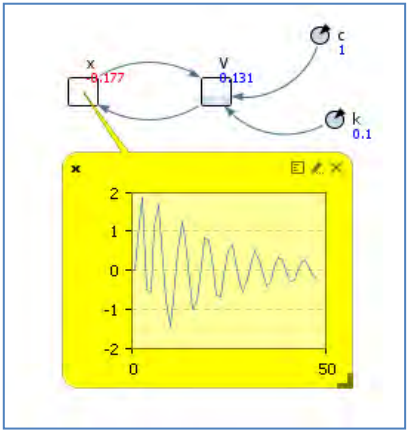


Рис. 3.10. Временной график координаты тела

Как видим, процесс протекает корректно – с затуханием. Изменение (пока вручную) числовых значений жесткости пружины *c* и коэффициента трения *k* показывает эффекты, соответствующие реальным системам.

Таким образом, получили *адекватную* модель в соответствии с задачей данного этапа.

Разработка имитационной интерактивной модели

Инспекты элементов в работающей модели позволяют получить общее представление об отдельных процессах. Однако их использование для более глубокого анализа системы вызывает очевидные неудобства и с ней трудно экспериментировать. Необходимо организовать более легкий способ взаимодействия с моделью – «дружественный интерфейс».

Перетащим все элементы математической модели («логику») в область слева от начала координат. Освободившуюся область будем использовать для размещения имитационной модели (симуляции) и элементов управления ею. Воспользуемся элементами из палитры *Презентация* (рис. 3.11).

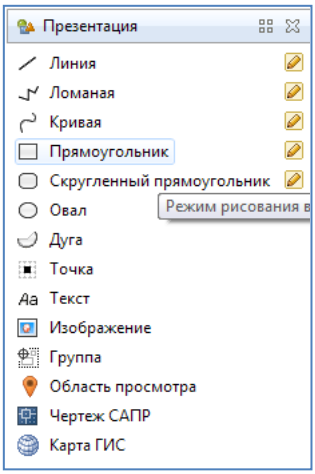


Рис. 3.11. Элементы визуализации модели

Разместим элементы системы и установим их свойства в соответствии с рис. 3.12.

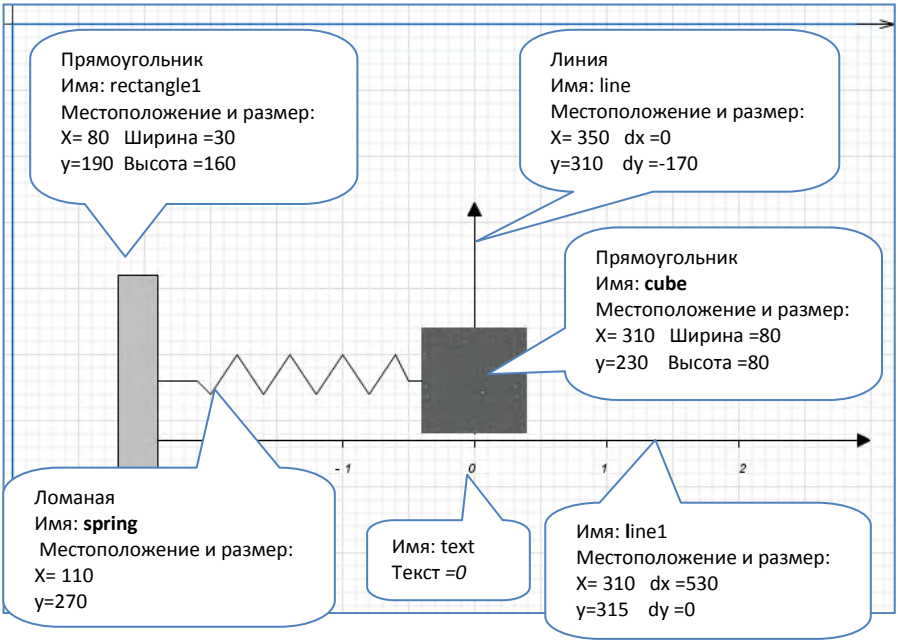


Рис. 3.12. Свойства элементов презентации модели

Имена элементов, которые будут изображать пружину и груз изменены на *spring* и *cube.* В дальнейшем будем обращаться к ним, как к объектам в программном коде.

Концы стрелок создадим с помощью компонента *Ломаная* (палитры *Презентация*) (рис. 3.13).

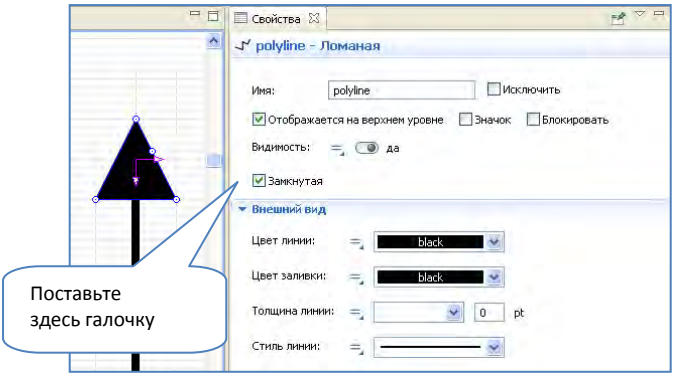


Рис. 3.13. Создание стрелки с помощью ломаной линии

Создадим алгоритм управления презентацией.

Как уже отмечалось ранее, AnyLogic поддерживает *дискретно-событийный* подход моделирования и содержит необходимые инструменты для его реализации.

Перенесем в модель элемент *Событие* из группы *Основная*. С его помощью организуем поведение объектов симуляции (рис. 3.14).

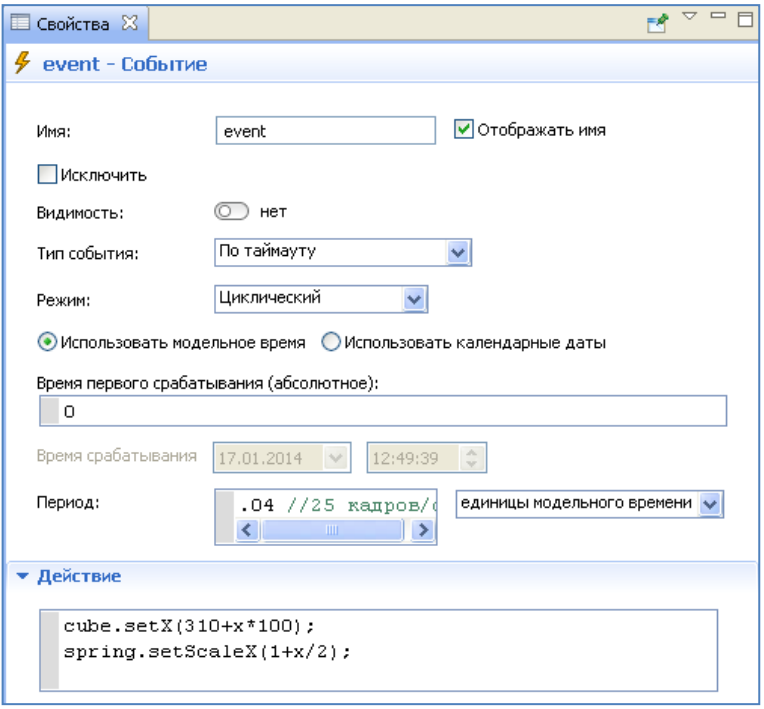


Рис. 3.14. Задание событий для объектов

По умолчанию режим работы этого элемента установлен в *Срабатывает один раз.* Нужно, чтобы проверка происходила регулярно и достаточно часто, поэтому режим был изменен на *Циклический.* Период слежения установим в 0.04 (секунды) – чтобы обеспечить *кинематографическое* качество движения (25 «кадров» в секунду).

Задача кода (*Действие*) заключается в периодическом вызове двух сеттер-методов, которые устанавливают

положение прямоугольника и степень растяжения (масштаб) пружины в зависимости от текущей координаты.

> Сеттеры и геттеры – методы, которые соответственно устанавливают и получают значения. Как правило, это значения переменных экземпляра (объекта). Ранее уже использовался подобный метод – *setText()* – когда создавалась модель калькулятора.

Запустим модель. Положение груза и масштаб пружины сразу начинают изменяться в соответствие с математической моделью.

Начальное отклонение груза имеет фиксированное ненулевое значение. Таким образом, задавать значение горизонтальной координаты груза *x* (растягивать, сжимать и отпускать пружину) неудобно. Будем это делать наиболее естественным способом – мышью. В качестве курсора используем изображение руки.

В данном проекте протекают две группы событий:

1) изменения значений накопителей и параметров в математической модели;

2) изменения графических элементов – координат, масштаба и состояния мыши в презентации.

Все эти события необходимо отслеживать и связывать.

Для работы с мышью и изменения вида курсора в модель нужно импортировать класс *Cursor* (для курсора) и библиотеку (пакет) классов *event.* Они находятся в части программного обеспечения Java-пакете *awt* (Abstract Windows Toolkit).

В свойствах Main *Java для экспертов* запишем две команды:

```
import java.awt.event.*;
import java.awt.Cursor;
```

Из палитры инструментов перетащим в модель элемент *Изображение.* Назовем его *hand.* Командой *Добавить* загрузим с диска подходящую картинку (в данном случае это снимок руки автора) (рис. 3.15).

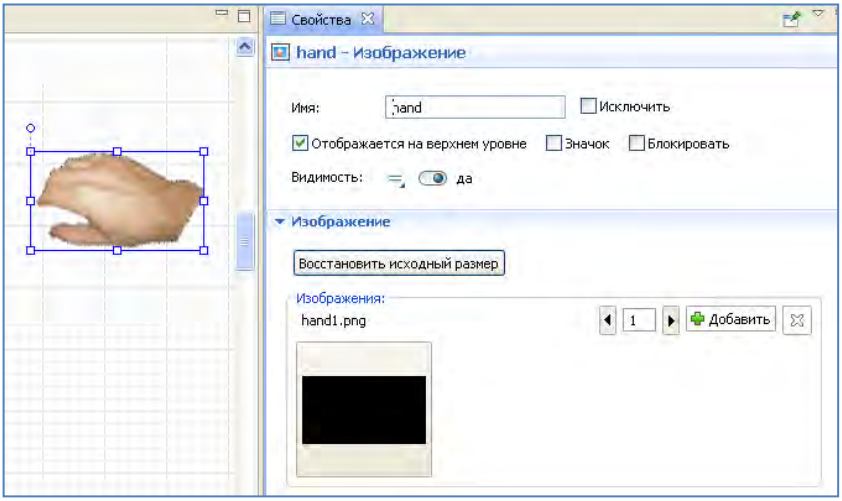


Рис. 3.15. Задание свойств изображения для курсора

Значение координат картинки можно не изменять, поскольку в дальнейшем они будут привязаны к координатам мыши.

> Все загружаемые в модель изображения сохраняются в папке проекта.

Создадим две функции, которые будут обеспечивать взаимодействие мыши и элементов презентации («руки» и «груза»).

Из палитры *Основная* перетащим в модель элемент *Функция*. Назовем ее *mouseDrag*. По умолчанию в свойствах функций активна кнопка выбора *Действие (не возвращает ничего)* (альтернативой будет вариант *Возвращает значение*). Оставим этот пункт без изменений.

В окне *Тело функции* запишем код:

```
//Переопределяем один из стандартных адаптеров мыши
MouseMotionAdapter myMouseMotionAdapter=new
MouseMotionAdapter()
 {
  public void mouseDragged(MouseEvent e) {
   super.mouseDragged(e); //перемещение мыши
```

```
        double m_X=e.getX()-100; /*вспомогательная переменная*/
        hand.setX(m_X+70); /*"рука" движется вместе с курсором*/
        cube.setX(m_X); /*связать координату груза с курсором  */
      /*масштаб пружины.200-длина пружины,310-координата ее  конца на
грузе: */
        spring.setScaleX(1+(m_X-310)/200);

      /*установить значение накопителя в соответствие с положением курсора:*/
        x=(m_X-310)/100;
         }
        };
         //получаем доступ к панели презентации
         Panel p1=getPresentation().getPanel();
         //Добавляем слушатель мыши
      p1.addMouseMotionListener(myMouseMotionAdapter)
```

Создадим еще одну функцию *mousePress*. Она будет обрабатывать реакцию элементов презентации на щелчок мыши:

```
      MouseAdapter myMouseAdapter=new MouseAdapter()
       {
      /*Далее реализуем и переопределяем только два метода, которые нас
интересуют*/
        public void mousePressed(MouseEvent e) {
          super.mousePressed(e);  //Кнопка прижата
           mouseDrag(); // Вызов первой функции
            }
        public void mouseReleased(MouseEvent e) {
          super.mouseReleased(e); // Кнопка отжата
          //Делаем событие активным:
           event.restart();
          //Убираем изображение руки в сторону и вверх:
          hand.setX(m_X+70);
          hand.setY(hand.getY()-70);
           }

        };
      Panel p=getPresentation().getPanel();
      p.addMouseListener(myMouseAdapter);
```

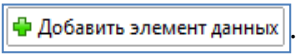
До этого момента объекты презентации начинали функционировать в соответствии с математической моделью сразу после запуска эксперимента. Процесс обеспечивает элемент модели *event*. Теперь он должен начинать работу после того, как мышью выбрано положение груза. Таким образом, при запуске модели он должен быть неактивен.

Для этого внесем в свойство Main *Действия агента–При запуске* две строки кода:

```
event.reset();
mousePress();
```

После этих изменений событие станет активным (система придет в движение), когда будет отпущена кнопка мыши.

Установим в свойствах накопителя *x* начальное значение в ноль. Теперь будем изменять его, перемещая мышью груз на презентации.

Перенесем в модель элемент *Временной график* из палитры *Статистика*. Щелкнем в панели свойств графика по кнопке 

.

Внесем соответствующие изменения в свойства (рис. 3.16).

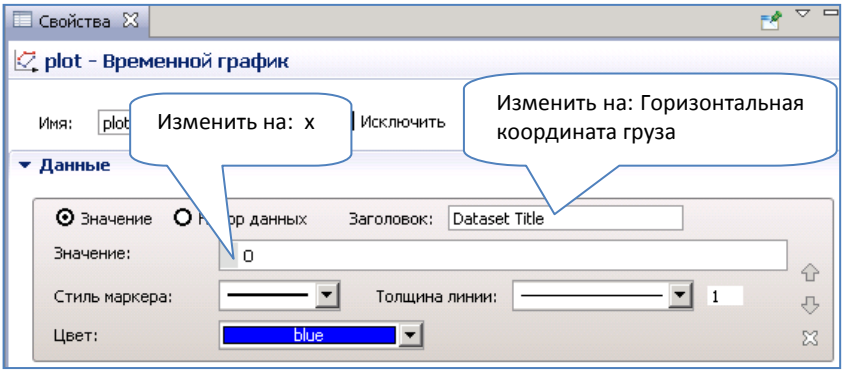


Рис. 3.16. Настойка элемента *Временной график* для координаты груза

Добавим в модель средства управления значениями параметров *c* и *k* математической модели.

Из палитры *Элементы управления* перетащим в модель два *бегунка*. Для первого – в свойстве *Связать с*: – введем *c* (латинское). Установим для него минимальное и максимальное значение (рис. 3.17).

Над бегунками разместим два текстовых элемента с соответствующими надписями.

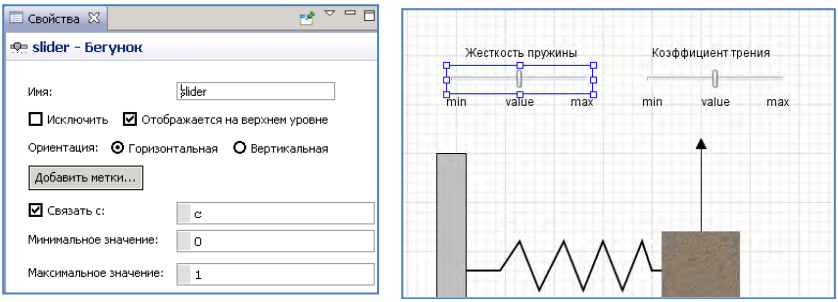


Рис. 3.17. Настройка элементов управления параметрами модели

Аналогичные действия проделаем для второго бегунка, с помощью которого будем изменять значение *k*.

Запустим модель. Если все сделано правильно, увидим интерактивную презентацию (рис. 3.18).

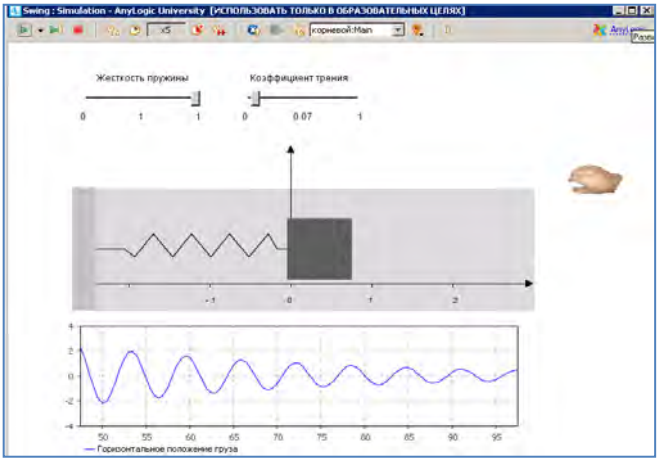


Рис. 3.18. Интерактивная презентация модели

Почему при составлении математической модели для накопителей был выбран *Режим задания уравнения: Произвольный* (а не *Классический,* установленный по умолчанию)? На самом деле их форма в AnyLogic соответствует одному из направлений имитационного моделирования.

В моделях *динамических систем* важно точно отобразить поведение отдельных переменных. В этом случае естественным будет применение дифференциальных уравнений вида *dx/dt=f(x).*

Другое направление – С*истемная динамика* – исследует сложные системы, где существенны причинно-следственные зависимости, петли обратной связи и т.п.

Использование одного набора компонентов в AnyLogic позволяет создавать эквивалентные математические модели в рамках двух подходов.

Рассмотрим простую модель радиоактивного распада:

$$\frac{dN}{dt} = -\lambda N \, .$$

В соответствующих режимах задания уравнений она будет выглядеть так, как показано на рис. 3.19.

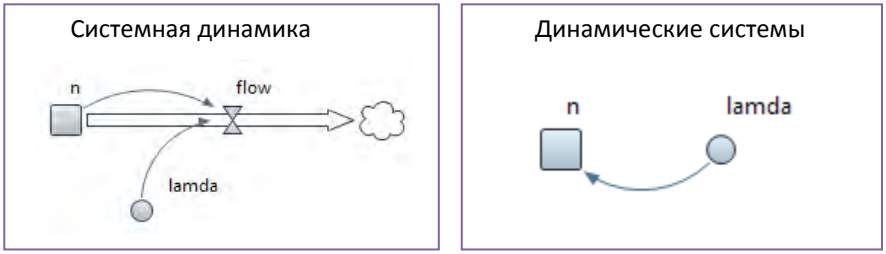


Рис. 3.19. Две формы уравнений модели радиоактивного распада

Какую форму предпочесть? Обе модели сходны, поэтому особой разницы в выборе нет.

Модель пружинного маятника в форме системной динамики представлена на рис. 3.20.

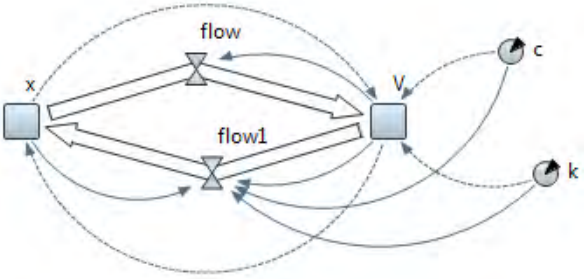


Рис. 3.20. Модель пружинного маятника

Свойства накопителей:

dx/dt = flow1–flow; начальное значение *V*.
dV/dt = flow–flow1; начальное значение *flow+k+c+x*.

Свойства потоков:

*flow = –V; flow1 = –x*c–k*V.*

В этом случае приходим к неоправданному усложнению модели.

Более полное представление о данном направлении моделирования дают учебные модели AnyLogic, например, Predator Prey (Хищник–Жертва).

3.3. Дополнительные средства визуализации модели

Добавим в модель трехмерную анимацию. В AnyLogic она отображается в специальных окнах:

1. Откроем палитру *Презентация*. Поместим элемент *3D Окно* так, чтобы он оказался ниже области окна презентации работающей модели (рис. 3.21) (по умолчанию ее размеры от начала координат 1200×800 пикселей).

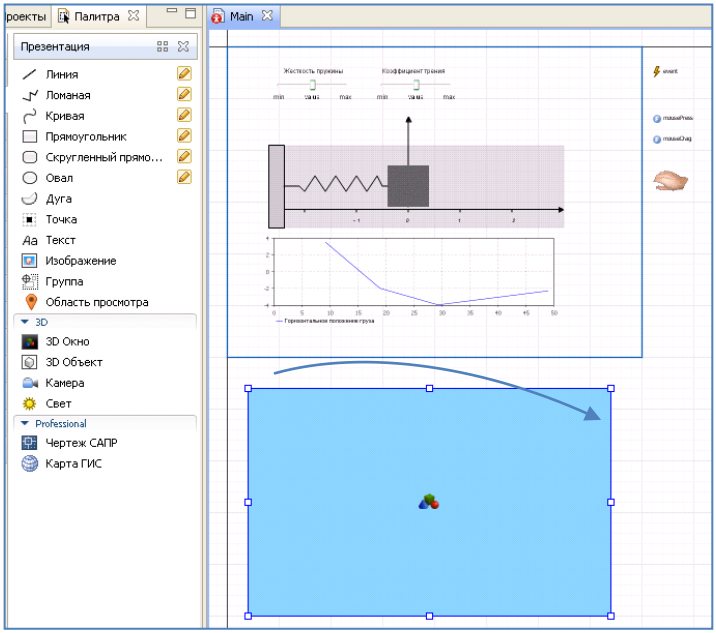


Рис. 3.21. Настройка 3D окна

2. Настроим объекты для корректного показа в 3D. У каждого из них теперь есть «вертикальная» *Z*-координата. По умолчанию ее значение 0. Пока все элементы лежат друг на друге в одной плоскости.

Покажем принцип разнесения объектов в пространстве на примере столешницы и груза (рис. 3.22, 3.23).

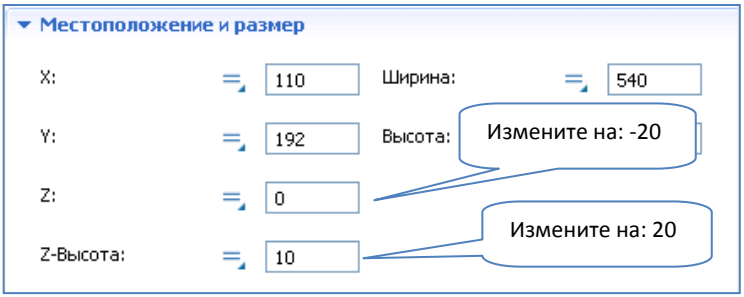


Рис. 3.22. Верхняя (по оси *Z*) плоскость столешницы с ненулевой координатой

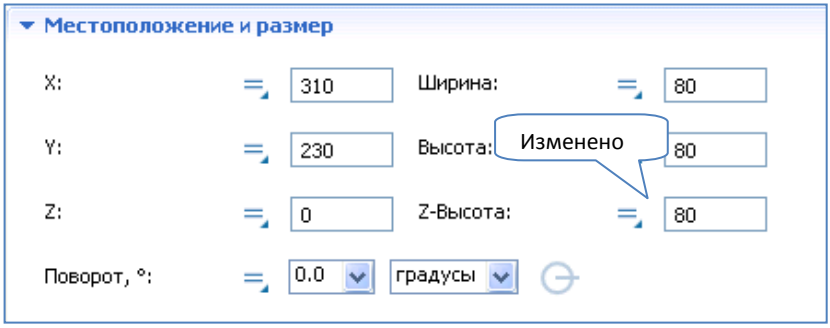


Рис. 3.23. Груз на столешнице

3. Плоская ломаная, играющая роль пружины, в 3D окне будет выглядеть как «инородное тело», поэтому *удалим* ее и заменим на «настоящую», трехмерную, пружину. В AnyLogic имеется палитра *3D Объекты*, с набором наиболее часто моделируемых элементов, однако пружины там нет. Сделаем ее с помощью бесплатного 3D-редактора Blender (рис. 3.24).

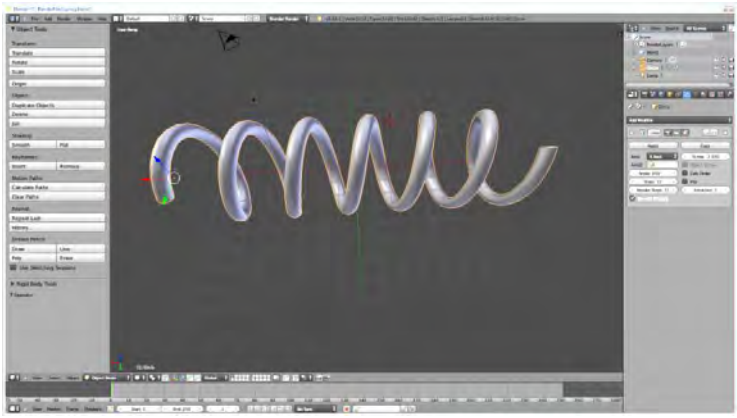


Рис. 3.24. Необходимые 3D-объекты, созданные в Blender

В AnyLogic можно использовать 3D-файлы в форматах **.x3d и *.wrl.* Сохраним файл как *spring.x3d,* и внедрим в модель через элемент *3D Объект* палитры *3D.*

Выберем направление осей и подберем масштаб картинки (рис. 3.25).

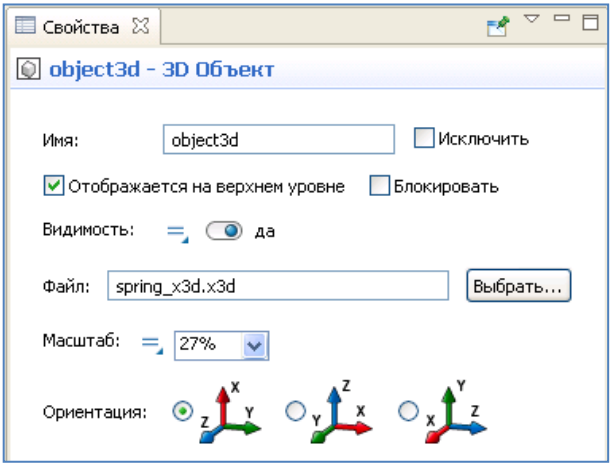


Рис. 3.25. Расположение пружины в соответствии с окружением

В свойствах 3D-объекта нет раздельных установок масштаба по осям координат. Это означает, что размеры пружины будут изменяться во *всех* направлениях.

Для исправления этой ситуации выделим объект и создадим группу (рис. 3.26).

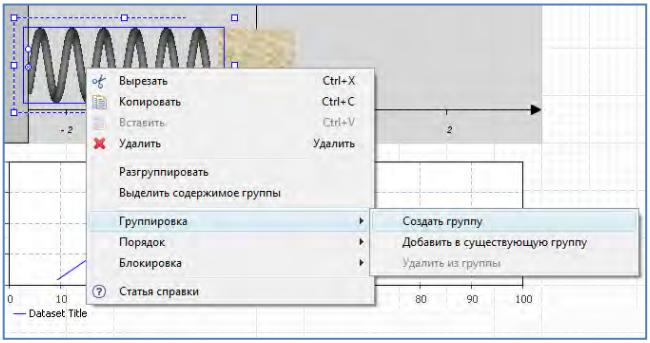


Рис. 3.26. Создание группы через контекстное меню

Назовем ее так же, как удаленную ломаную – *spring* (чтобы не изменять ранее написанный код).

Объединенные в группу элементы работают как единое целое. Так, код, написанный в свойствах группы, является общим для всех ее членов. Тем не менее свойства каждого элемента остаются доступными для изменения.

Центр группы расположен в точке, равноудаленной от всех, вошедших в нее элементов (в центре тяжести). В данной группе один элемент и центр находится в середине пружины (рис. 3.27). В этом случае динамические свойства группы (перемещение, масштаб, вращение) будут изменяться относительно центра фигуры.

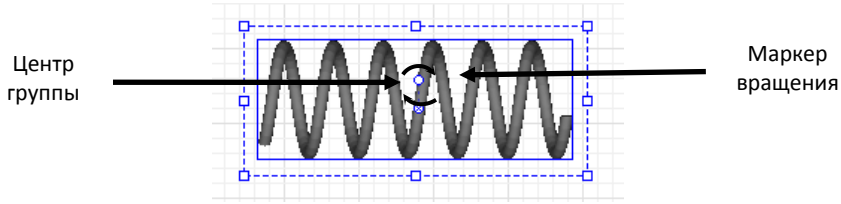


Рис. 3.27. Изначальное положение центра группы

Пружину в группе нужно сдвинуть. Щелкнем по ней. Получим доступ к свойствам внедренного 3D-объекта. Сместим его вправо (рис. 3.28).

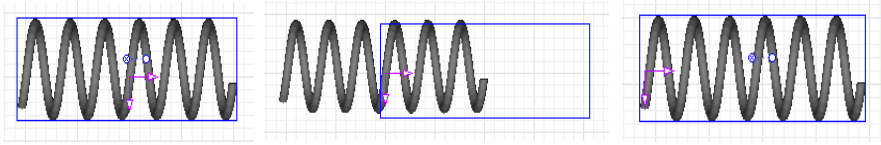


Рис. 3.28. Перемещение объекта в группе

Теперь масштаб пружины будет изменяться правильно (рис. 3.29).

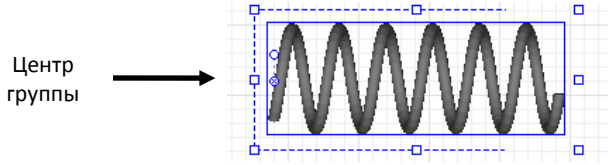


Рис. 3.29. Итоговое положение центра группы

В свойствах группы введем значения масштабов по оси *Y* и *Z* меньше единицы, например 0.7. Благодаря этому, в работающей модели у пружины будет более пропорциональный вид. Изменим *Z*-координату. (см. п. 2).

4. Добавим в модель камеру и источник света. Наличие камер позволяет разместить несколько 3D окон с разными видами на объекты. Можно также организовывать перемещение камер и переключение между ними в пределах одного окна.

Из палитры *3D* перенесем *Камеру*. В *Свойствах* установим 315.0 Значения поворота – *Z*. Расположение – *X*:840; *Y*:100; *Z*:70.

Перетащим в окно графического редактора элемент *Свет*. Настроим его свойства (рис. 3.30).

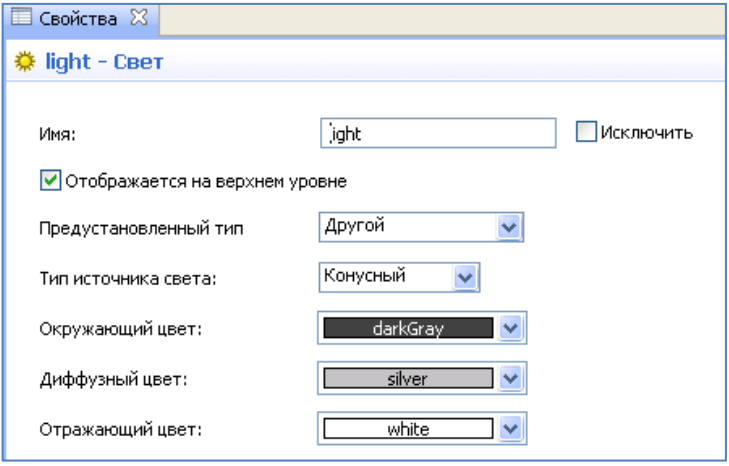


Рис. 3.30. Настройка источника света

В свойствах *Расположение* установим значения координат: *X*:620; *Y*:–20; *Z*:100 и углов: *Угол X*:0.0; *Угол Z*:205.0.

В свойствах 3D окна в поле *Камера* выберем: *camera*. Запустим модель и выберем область просмотра [*window 3d* (рис. 3.31).

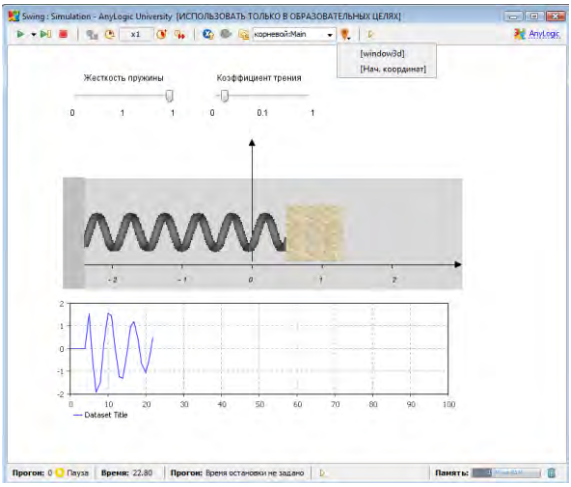


Рис. 3.31. Выбор области просмотра в работающей модели

Откроется область трехмерной модели системы (рис. 3.32).

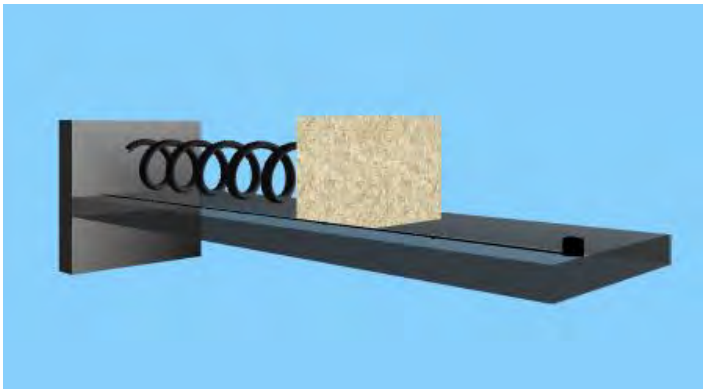


Рис. 3.32. 3D-анимация модели в AnyLogic

5. Организуем области просмотра и управление переходами между ними из окна работающей модели.

Вставим в модель объект *Область просмотра* из палитры *Презентация*, изменим название со стандартного *viewArea* на *view2D* (рис. 3.33).

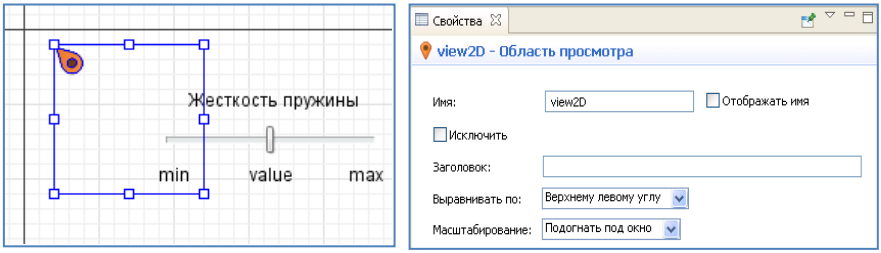


Рис. 3.33. Настройка свойств области просмотра

Растянем рамку объекта так, чтобы внутри него поместились бегунки, 2D-презентация и график.

Создадим еще одну область просмотра. Назовем ее *view3D*. Переместим ее так, чтобы 3D окно оказалось в его границах.

Добавим в первую область просмотра текстовый элемент. В свойстве *Текст* введем: *3d модель* (рис. 3.34). В свойстве *Действие по щелчку: view3D.navigateTo();*.

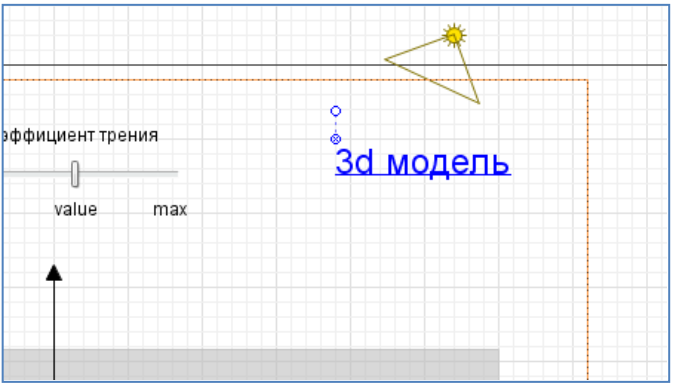


Рис. 3.34. Текстовый элемент для перехода в другую область просмотра

Аналогично настроим вторую область просмотра. Изменим текст на: *2d модель*. Код действия по щелчку: *view3D.navigateTo();*.

В результате создалась интерактивная модель, оснащенная элементами 2D–3D-визуализации, средствами навигации и управления.

3.4. Динамическое моделирование процесса управления

Существует несколько базовых принципов управления, предполагающих наличие определенных элементов и связей, объединенных в систему (рис. 3.35).

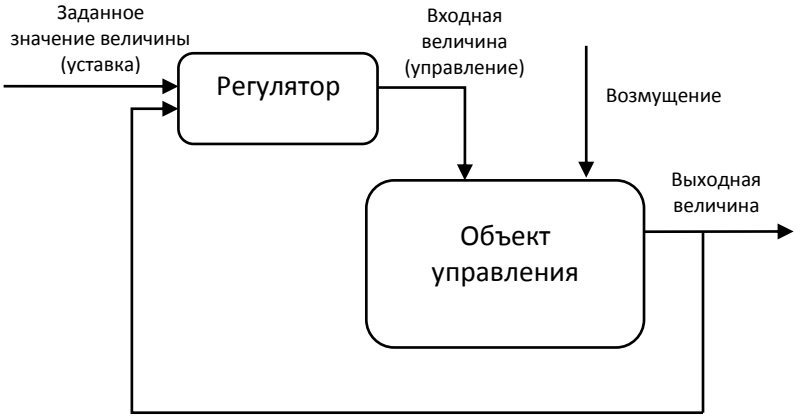


Рис. 3.35. Схема системы управления

Выходная величина типичного объекта управления описывается уравнением:

$$\frac{dn}{dt} = \frac{1}{a} \cdot (k \cdot u - z - h),$$

где a – коэффициент пропорциональности, определяемый параметрами объекта управления; k – коэффициент усиления объекта по управлению; z – изменяющееся внешнее воздействие (возмущение).

Регулятор можно строить разными способами.

- с пропорциональным законом управления (П-регулятор);
- с интегральным законом управления (И-регулятор);
- с пропорционально-дифференциальным законом управления (ПД-регулятор);

• с пропорционально-интегральным законом управления (ПИ-регулятор);

• с пропорционально-интегрально-дифференциальным законом управления (ПИД-регулятор).

Построим модель системы регулирования уровня жидкости в емкости с пропорционально-интегральным законом управления (рис. 3.36).

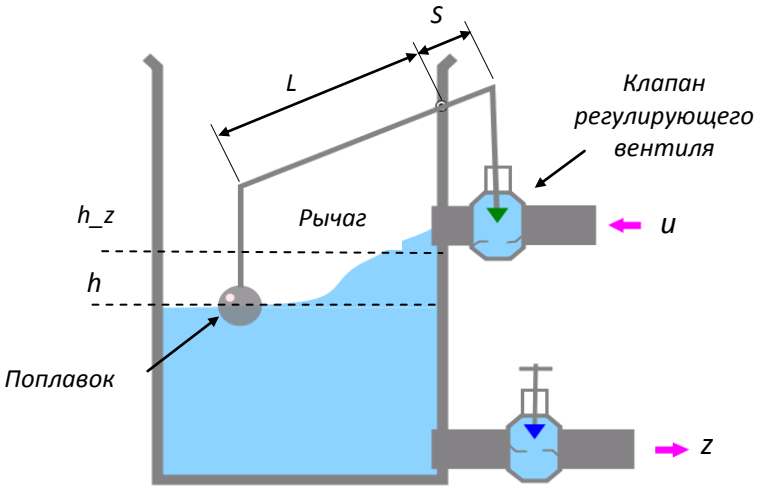


Рис. 3.36. Система регулирования уровня в емкости

Входную величину (управление) определим следующим образом:

$$u = u_i + k_p \cdot (h\_z - h),$$

где h_z – заданное значение уровня; u – управление, которое состоит из интегральной и пропорциональной части; u_i – интегральная составляющая управления; k_p – коэффициент при пропорциональной составляющей управления.

Интегральная составляющая управления может быть задана:

$$\frac{dui}{dt} = k_i \cdot (h\_z - h),$$

где k_i – коэффициент при интегральной составляющей управления.

Для создания модели выполним следующие шаги:

1. Создадим логику модели. Математическая модель – система двух дифференциальных и одного алгебраического уравнений (рис. 3.37).

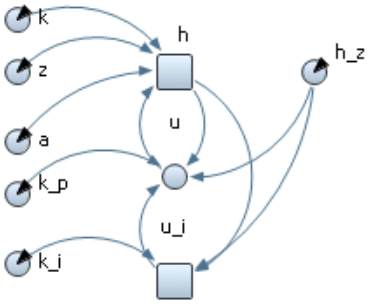


Рис. 3.37. Математическая модель

Установим начальные значения параметров: a = 10; k = 1; k_p = 0.5; k_i = 0.5; z = 80; h_z = 300.

В накопителе h вычисляется выходная величина – уровень наполнения емкости. Установим его начальное значение: 80.

2. Внесем в модель элемент *Временной график*. Настроим его свойства для показа графика накопителя h. Запустим эксперимент (рис. 3.38).

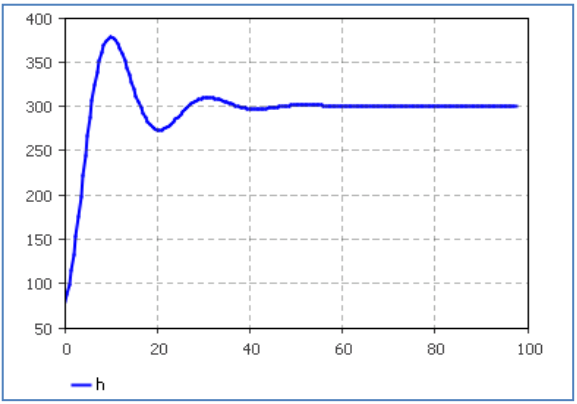


Рис. 3.38. Переходный процесс регулирования уровня наполнения емкости

Построим анимационную модель системы. Для этого необходимо связать поведение воды, рычага и поплавка с значением накопителя *h*.

3. Создадим рычаг. Он состоит из четырех звеньев: короткого, длинного плеча и двух штанг на концах.

Внесем в модель элемент *Ломаная*. В ее *Свойствах* изменим имя со стандартного *polyline* на *pL*.

Заготовка, которую предоставляет *AnyLogic*, уже содержит четыре отрезка. С помощью включенной в окне графического редактора сетки, подгоним их размеры (рис. 3.39).

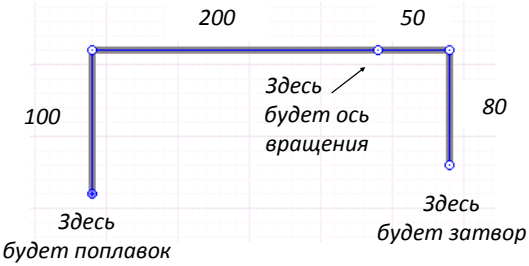


Рис. 3.39. Ломаная заготовка для презентации рычага (размеры отрезков указаны в пикселях)

> Разбить отрезок ломаной на дополнительные звенья можно, щелкнув по нему мышью. Чтобы удалить звено, нужно щелкнуть по граничной точке отрезка.

Для упрощения расчетных формул перенесем ломаную в окне редактора так, чтобы третья точка рычага оказалась в начале координат.

4. Добавим в модель четыре параметра.

В параметры *xL* и *yL* будут передаваться значения координат точек большого (*Large*) плеча рычага.

Параметры *xS* и *yS* будут отображать координаты малого (*Small*) плеча.

Запишем выражения для параметров, учитывая длины отрезков и то, что ось ординат направлена вниз («перевернута»):

xL = –200*cos((h/h_z-1));
yL = –200*sin((h/h_z-1));
xS = 50*cos((h/h_z-1));
yS = 50*sin((h/h_z-1)).

Установим связи. После этого логика модели примет вид, показанный на рис. 3.40.

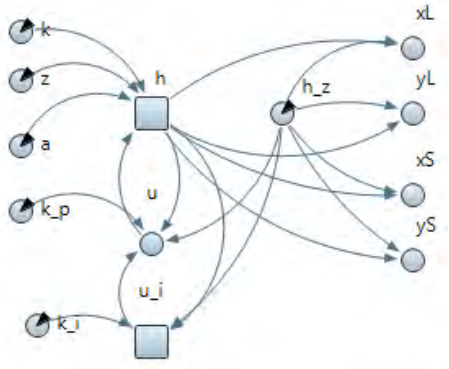


Рис. 3.40. Измененная математическая модель

Щелкнем по заголовку *Точки ломаной* окна свойств ломаной *pL*. Увидим, что точки проиндексированы и имеют свою систему координат, относительно точки с индексом 0 (рис. 3.41).

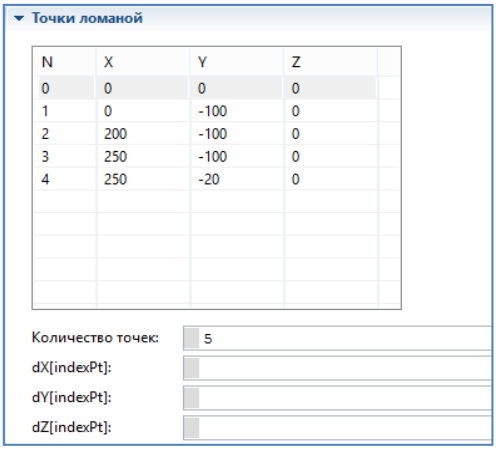


Рис. 3.41. Свойства точек ломаной

В поле *Количество точек* запишем: *5*. Добавим фрагмента кода

- в поле *dX[indexPt]:*

```
indexPt==0? 200+xL:
indexPt==1? 200+xL:
indexPt==2? 200:
indexPt==3? 200+xS:
indexPt==4? 250:0
```

- в поле *dY[indexPt]:*

```
indexPt==0? yL:
indexPt==1? pL.getPointDy(0)-100:
indexPt==2? -100:
indexPt==3? yS-100:
indexPt==4? pL.getPointDy(3)+130:0
```

Для контроля положения координат точек ломаной использована управляющая конструкция Java-цепочка проверок и соответствующих действий. Расположение точки на ломаной определяется ее параметром *indexPt.*

Координаты третьей точки (с индексом *2*) не изменяются, поскольку через нее проходит «ось вращения» рычага.

Изменение *y*-координаты у точек *1* и *3* рассчитывается с помощью метода *getPointDy(),* который определяет положение предыдущей точки – *0* и *3* соответственно.

Код можно было упростить таким образом, потому что вертикальные координаты этих четырех точек изменяются одинаково, с разницей на (неизменную) длину звена.

Конструкцию обязательно нужно закончить альтернативным действием для последней проверки (ветвью «иначе»). В данном случае просто поставим ноль.

5. Создадим поплавок. Добавим в модель две окружности из палитры *Презентация*. Установим значения радиусов для первой – 20 пикселей, для второй – 4 пикселя. Объединим окружности в группу и настроим ее свойства (рис. 3.42).

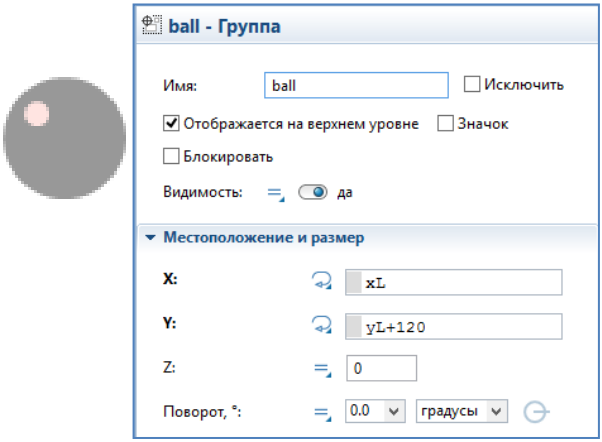


Рис. 3.42. Поплавок с бликом, полученный из двух окружностей

Горизонтальная координата поплавка привязана к первой точке рычага. Его вертикальная координата получается добавлением к значению *y*-координаты первой точки ломаной длины ее первого звена (100) и радиуса поплавка (20).

6. Добавим в презентацию затвор. Изготовим его по образцу конца стрелки из модели маятника (см. рис. 3.13). Координаты, с учетом поправок на размеры полученной фигуры, привяжем к последней точке ломаной (рис. 3.43).

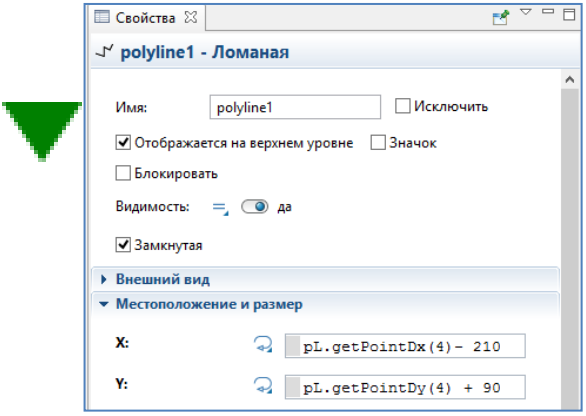


Рис. 3.43. Затвор и его свойства

7. Создадим «воду». Для этого поместим в модель прямоугольник бледно-голубого цвета, контур сделаем бесцветным и настроим его свойства (рис. 3.44).

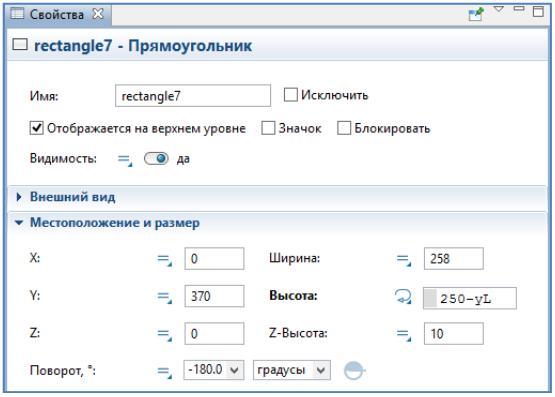


Рис. 3.44. Моделирование воды

Обратите внимание на значение свойства *Поворот*. Чтобы направление уровня воды было корректным, прямоугольник нужно перевернуть.

Добавим в модель замкнутую *Кривую* для потока воды (рис. 3.45).

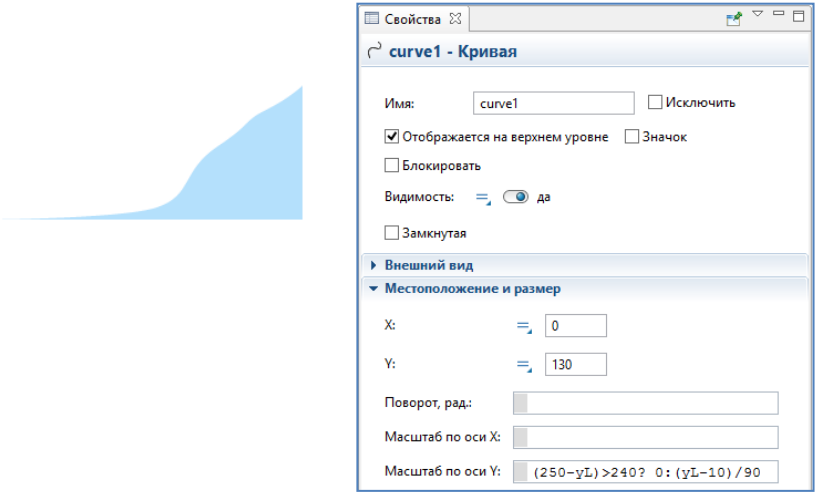


Рис. 3.45. Презентация потока воды

Ее масштаб отрегулирован так, что поток будет исчезать, когда уровень достигнет определенного значения (выше наливной трубы) и подача воды в систему приостановится.

После внесения дополнительных элементов презентация модели в графическом редакторе окажется в «разобранном» виде (рис. 3.46).

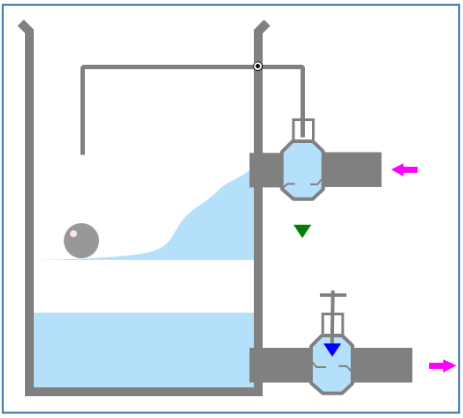


Рис. 3.46. Вид модели в графическом редакторе

После запуска эксперимента все оказывается на положенных местах и следует в соответствии с математической моделью, состояние которой отображается на графике.

3.5. Модели с распределенными параметрами

Существует область задач, когда параметры системы изменяются по времени и пространственным координатам. Для таких систем создаются модели с распределенными параметрами. Они строятся на основе систем дифференциальных уравнений в частных производных.

В основе этого подхода лежат численные методы. Для пространственных координат и времени выбираются значения шагов изменения. Затем строится конечно-разностная схема, которая связывает параметры в систему уравнений.

Рассмотрим характерный пример из этой области. Толстая однородная пластина разделяет две среды с разными температурами (рис. 3.47). Требуется построить модель распределения температуры для внутренних точек пластины.

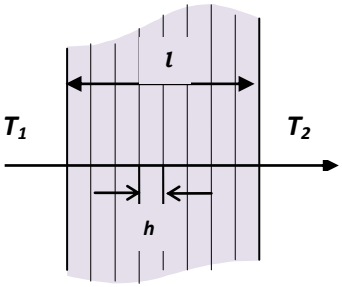


Рис. 3.47. Пластина, разделяющая две среды

Математическая модель данной системы – уравнение теплопроводности:

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2};\ \ x_0 \le x \le x_k;\ \ t > 0;\ \ a > 0,$$

где u – температура; x – пространственная координата; a – коэффициент теплопроводности пластины; t – время.

Уравнение описывает распределение температуры как функции двух переменных (времени и координаты) по толщине бесконечной плоской пластины.

Будем полагать, что в начальный момент времени температура пластины по всей толщине одинакова. Начальное условие: $u(x,t = 0) = T_0$.

Граничные условия $u(x_0,t) = T_1$; $u(x_k,t) = T_2$ (значения начальной и конечной температур не изменяются во времени).

Для построения модели в AnyLogic необходимо задать выражения накопителя u в разных сечениях пластины (рис. 3.48).

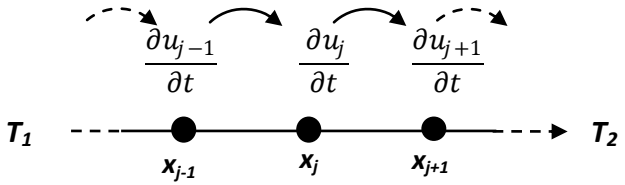


Рис. 3.48. Фрагмент разностной схемы модели

Частная производная второго порядка (правая часть уравнения теплопроводности) приближенно может быть выражена через значения функции u в точке j и в двух соседних точках j–1 и j+1:

$$\frac{\partial^2 u(j)}{\partial x^2} \approx \frac{1}{h}\left[u_{j+1}(t) - 2u_j(t) + u_{j-1}(t)\right].$$

Таких уравнений будет столько, сколько сечений было выделено в пластине. Построим модель для пяти сечений:

1. Введем в модель четыре параметра. Для температуры среды T_1 (по левую сторону пластины): *u_0*, значение по умолчанию 200. Для T_2: *u_6* – значение 0. Для коэффициента теплопроводности введем параметр a со значением 0.5. Параметр *h_Sq* (квадрат шага сетки) примем равным единице.

2. Зададим форму уравнения накопителя *u_1* для температуры первого слоя пластины:

a*(u_2-2*u_1+u_0)/h_Sq.

В качестве начального значения T_0 выберем 20. Аналогично введем в модель еще четыре накопителя.

3. Для объединения всех элементов в систему уравнений установим связи (рис. 3.49).

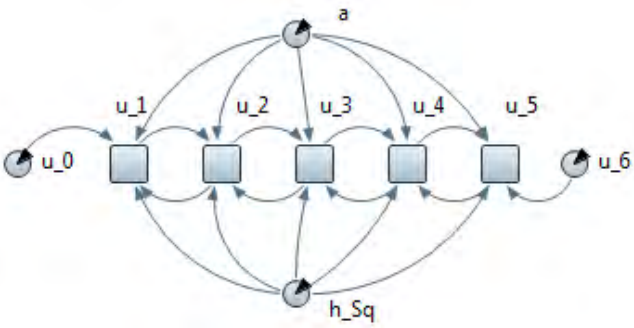


Рис. 3.49. Математическая модель теплопроводности

4. Внесем в модель *Временной график* и добавим в него пять элементов данных для отображения значений температур в соответствующих накопителях.

5. Создадим анимацию изменения температуры в слоях пластины. Поместим в модель *Прямоугольник*. Его горизонтальная координата будет соответствовать положению слоя в пластине. Настроим его свойства.

В свойствах *Специфические* введем *количество*, равное семи (пять – для слоев и два – для среды). В результате получим массив прямоугольников и можем обращаться к их номерам (целочисленной переменной *index*) (рис. 3.50).

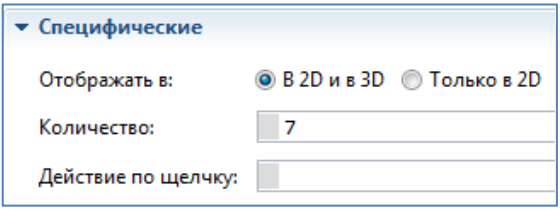


Рис. 3.50. Задание количества прямоугольников в презентации

Местоположение прямоугольника по оси *x* в презентации будет динамически зависеть от его ширины (50 пикселей) и положения в массиве (рис. 3.51).

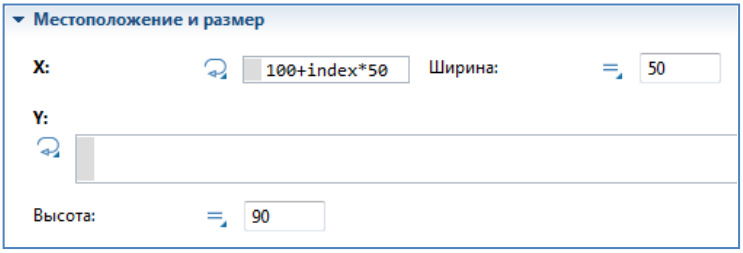


Рис. 3.51. Положение прямоугольника в презентации зависит от его индекса в массиве

Так, координата первого прямоугольника (с нулевым индексом) по горизонтали будет равна 100, второго – 150 (со сдвигом вправо на ширину прямоугольника) и т.д.

Цвет заливки прямоугольника будет отображать текущую температуру в слое (рис. 3.52).

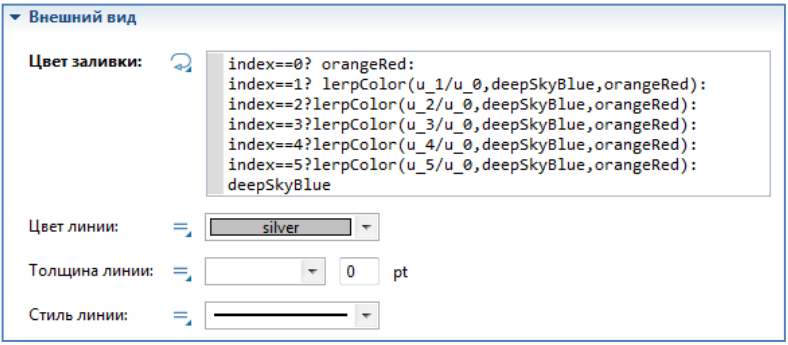


Рис. 3.52. Свойства, отображающие изменения температуры в слоях пластины

Для формирования цвета прямоугольника используется управляющая конструкция *Java.* Первый прямоугольник, его индекс 0 (температура среды T_1), имеет красноватый цвет заливки. Для определения цвета каждого слоя используется функция *lerpColor(<value>, color1,color2).* Она выдает значение цвета, в зависимости от значения вещественного параметра *value.* Если оно меньше или равно 0, то результат функции *color1.* Если больше или равно 1 – *color2.* Промежуточные значения (от 0 до 1) формируют промежуточный цвет. Таким образом, цвет прямоугольни-

ка будет динамически изменяться в зависимости от текущего значения своего накопителя.

Седьмой прямоугольник, его индекс 6 (температура среды T_2), будет синеватого цвета.

Сделаем так, чтобы высота прямоугольника отражала значение температуры в слое.

Внесем изменение в свойства *Высота* (рис. 3.53).

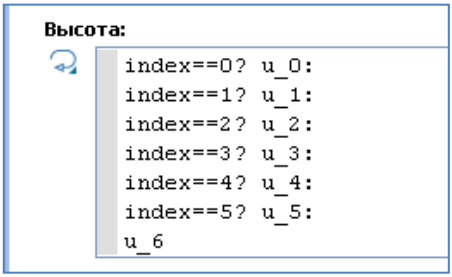


Рис. 3.53. Высота прямоугольников, зависящая от температуры

Чтобы размер прямоугольника изменялся «правильно» – снизу вверх, – его нужно вертикально перевернуть на 180°. Маркер положения фигуры должен оказаться внизу (рис. 3.54).

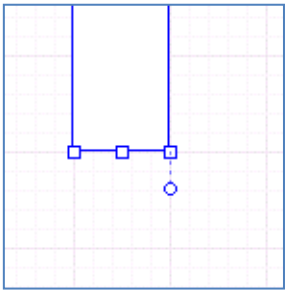


Рис. 3.54. Правильное положение маркера прямоугольника

В результате получили *нестандартную* столбиковую диаграмму переходного режима системы с отображением температуры в слоях (рис. 3.55).

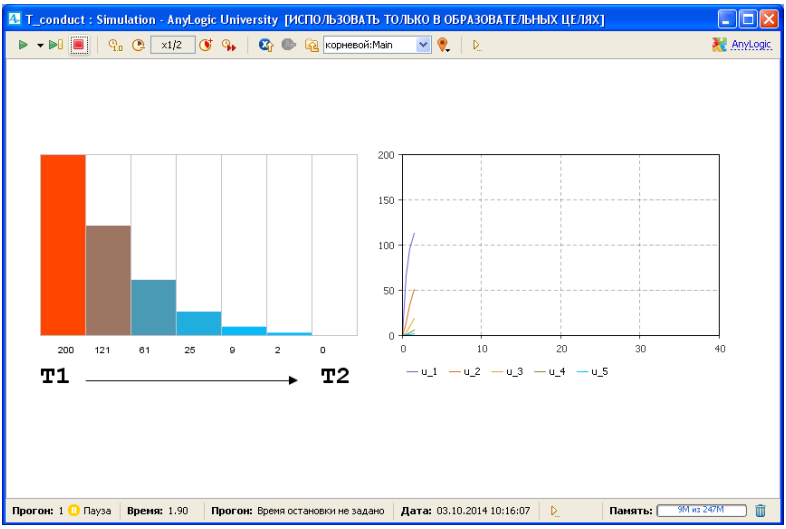


Рис. 3.55. Презентация модели теплопроводности

В соответствии с математической моделью эксперимент демонстрирует рост температуры и передачу тепла от более нагретых областей к более холодным.

3.6. Решение оптимизационной задачи

До сих пор мы строили модели систем и затем проводили простые эксперименты путем изменения значений параметров. Таким образом, решалась *прямая задача моделирования.*

Суть *обратной задачи* моделирования сводится к нахождению таких значений параметров модели, которые бы соответствовали определенному состоянию системы.

Оптимизация позволяет решать обратную задачу моделирования путем нахождения наилучших значений параметров модели, соответствующих максимуму или минимуму *целевой функции.* В качестве целевой функции может выступать один из параметров математической модели или специальный функционал, который также характеризует систему и интересует исследователя. Оптимизация – неотъемлемый этап решения экономических задач, типа «затра-

ты–эффект», когда, например, необходимо минимизировать стоимость продукта (или операции) или увеличить его выход при заданных ограничениях на ресурсы.

Существуют специальные методы реализации подобного класса задач. В AnyLogic для этих целей используется патентованный OptQuest – один из лучших на сегодняшний день профессиональных пакетов оптимизации.

Рассмотрим его возможности на примере.

При движении транспортных средств возникают периодические силы. Их действие в целом негативно, поскольку может приводить к резонансным явлениям. Для их уменьшения используются различные успокоители.

Решим задачу минимизации воздействия периодических сил на автомобильную подвеску.

Автомобиль массой m_2 испытывает воздействие силы $f(t)$, вызванной неровностями дороги (рис. 3.56). Амортизаторы (пружина жесткостью k_2 и демпфер b_2) по тем или иным причинам не создают достаточного успокоения машины. Оказывается, что добавление успокоителя массой m_1, опирающегося на пружину с жесткостью k_1 и демпфера b_2, может снять проблему.

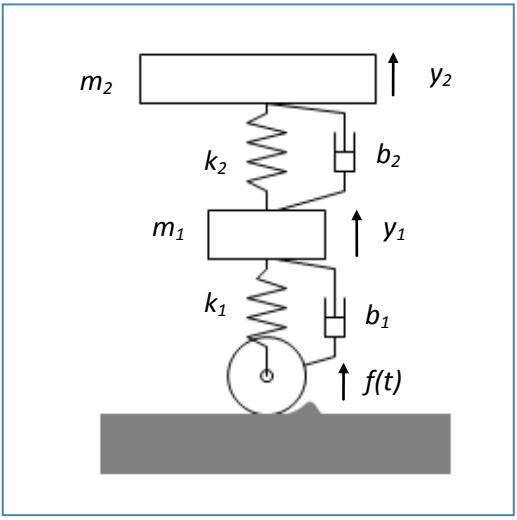


Рис. 3.56. Система подвески автомобиля

Остается найти такое значение m_1, при котором амплитуда кузова машины y_2 во время прохождения неровностей будет минимальной.

Математическая модель – система уравнений:

$$\begin{cases} m_1 \dfrac{\partial^2 y_1}{\partial t^2} = -b_1\dot{y}_1 + b_2(\dot{y}_1 - \dot{y}_2) - k_1 y_1 + k_2(y_2 - y_1) + b_1 f(t) + k_1 f(t), \\ m_2 \dfrac{\partial^2 y_2}{\partial t^2} = b_2(\dot{y}_1 - \dot{y}_2) + k_2(\dot{y}_1 - \dot{y}_2) + k\dot{y}_2, \end{cases}$$

где k – коэффициент сопротивления среды; y_i, $\dot{y}_i$ – координаты и скорости.

Ход решения задачи:

1. Понижая порядок системы уравнений, получим математическую модель в AnyLogic (рис. 3.57). Она содержит четыре уравнения накопителей:

```
d(y1)/dt=v1                                                    (1)

d(v1)/dt=(-b1*v1+b2*(v1-v2)-k1*y1+k2*(y2-1+b1*rub+
+k1*rub)/m1                                                    (2)

d(y2)/dt=v1                                                    (3)
d(v2)/dt=(b2*(v1-v2)+k2*(y1-y2)-k*v2)/m2                       (4)
```

Для всех накопителей начальные значения установлены в ноль (движения и, соответственно, возмущений нет).

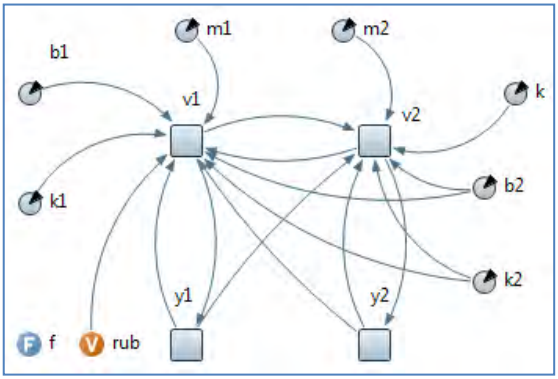


Рис. 3.57. Математическая модель системы подвески

Параметр m_2 – масса кузова, приведенная к одной рессоре. Установим его значение по умолчанию в 250.

В качестве значения по умолчанию для коэффициента сопротивления среды k выберем 0.4.

Начальные значения остальных параметров:

$k_1 = 5$, $b_1 = 0.4$, $k_2 = 1$, $b_2 = 0.2$, $m_1 = 10$.

2. Моделирование неровностей дороги (будем моделировать среду системы).

В выражении уравнения (2) модели есть параметр-переменная *rub* (неровность) с начальным значением 0. Введем в модель функцию *f(),* которая будет возвращать вещественное значение высоты неровности, изменяющееся случайным образом.

В свойстве *Тело функции* введем код:

```
double mu;
mu= abs(uniform(5)*sin(xRub*time()));
return mu;
```

В качестве значения аргумента функции *sin()* выступает расстояние, которое проходит колесо машины до столкновения с неровностью (точнее, неровность – до столкновения с колесом). Оно определяется скоростью *xRub,* умноженной на текущее модельное время (функция *time() AnyLogic*).

Для этого введем в модель накопитель *xRub:*

```
d(xRub)/dt=0.5.
```

Функция Java – *abs()* – будет «срезать» отрицательные значения функции *sin()* и возвращать только положительные значения неровности. Это сделано для упрощения анимации движения по дороге.

Введем параметр *yRub.* Он будет устанавливать высоту элементу анимации неровности.

3. Анимация модели. Сначала создадим неровность. Поместим в презентацию серый прямоугольник (он будет изображать дорожное полотно) и чуть выше замкнутую *Кривую* (рис. 3.58).

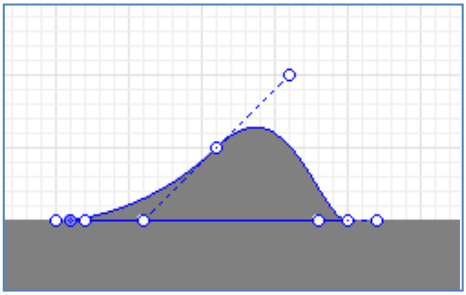


Рис. 3.58. Презентация неровности на дороге

Назовем ее *stone* (камень) и настроим свойства *Местоположение и размер:*

X:stone.getX()<220?stone.getX()>20?220-xRub*50:220:220-Rub*50

Y:215-yRub

На презентации неровность будет циклически перемещаться по отрезку оси *X* между точками 220–20, со скоростью *xRub*50* (пикселей/с). По достижению левого конца интервала она будет вновь появляться справа, в результате возникнет визуальный эффект движения системы. Высота неровности будет зависеть от параметра *yRub.* Чтобы связать его значение с динамической переменной *rub* математической модели, введем в модель элемент *Событие* и настроим его свойства (рис. 3.59).

Значение периода будет выбираться из распределения вероятностей, имеющего экспоненциальную форму. Таким образом, обеспечили дополнительную стохастичность в поведении неровности, установив неравномерность периода изменения ее параметров в математической модели (первая строка кода) и презентации.

Выполнение последней строки кода будет приостанавливать работу модели по истечении 200 единиц модельного времени. Она нужна для сравнения значений критериев системы при проведении экспериментов. В качестве таких критериев логично выбрать среднее значение амплитуды колебания кузова автомобиля и ее стандартное отклонение. Меньшие значения критериев будут более предпочтительными.

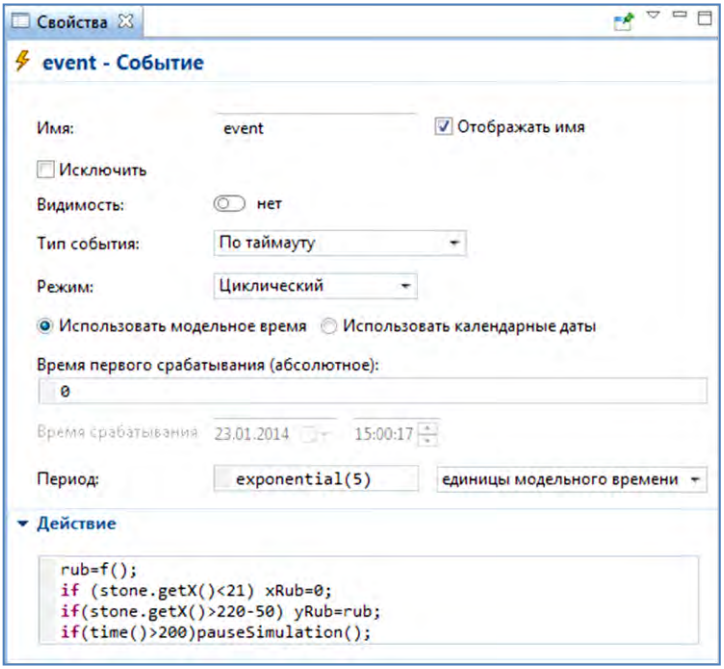


Рис. 3.59. Управление неровностью

4. Для сбора данных значений вертикального смещения кузова введем в модель элемент *Статистика* из группы *Статистика*. Назовем его *stat_yCar* : stat_yCar .

Поставим в *Свойствах*: *Непрерывная*; Значение: *y2*; *Обновлять данные автоматически*; Период: *0.1*.

Добавим в модель два параметра:

```
yRange= stat_yCar.max()-stat_yCar.min()
yDev=stat_yCar.deviation()
```

5. Внесем в презентацию *Временной график*, на котором будем наблюдать три величины: вертикальное отклонение подвески *y1* отклонение кузова *y2* и значение неровности *rub* (рис. 3.60).

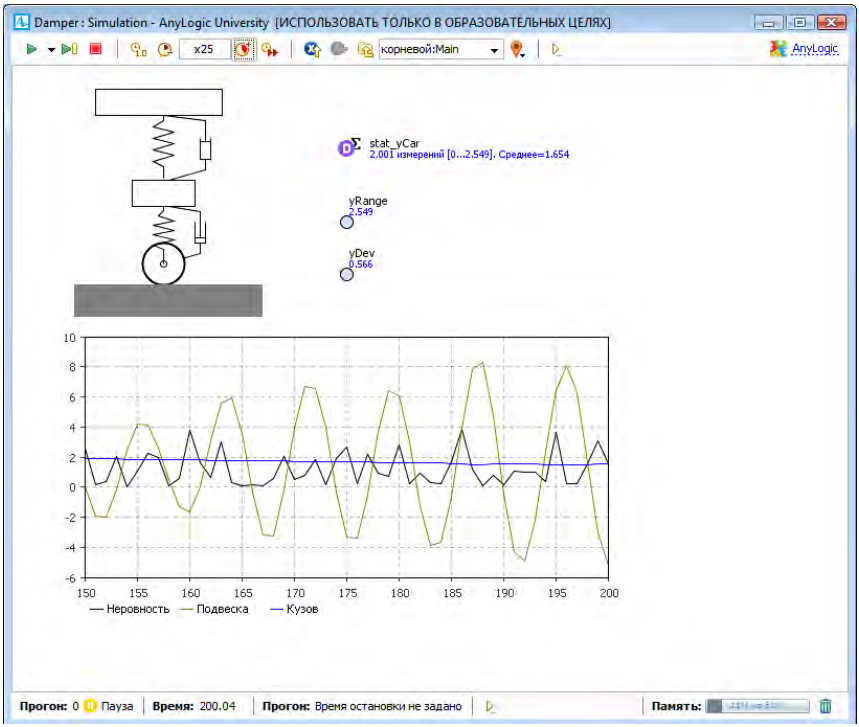


Рис. 3.60. Результат работы простого эксперимента (при заданных значениях параметров)

Как видим, подвеска выполняет свою функцию.

Однако есть такая масса успокоителя, при которой значения выбранных критериев поведения кузова будут *наименьшими*.

Для ее поиска проведем оптимизационный эксперимент.

В Главном меню AnyLogic выберем пункты *Файл–Создать–Эксперимент.* В появившемся диалоговом *Тип эксперимента* окне выберем *Оптимизация* (рис. 3.61).

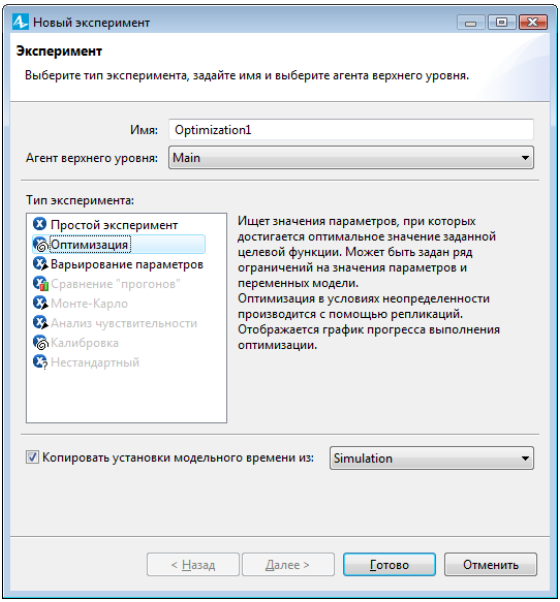


Рис. 3.61. Выбор типа эксперимента

Настроим его свойства (рис. 3.62).

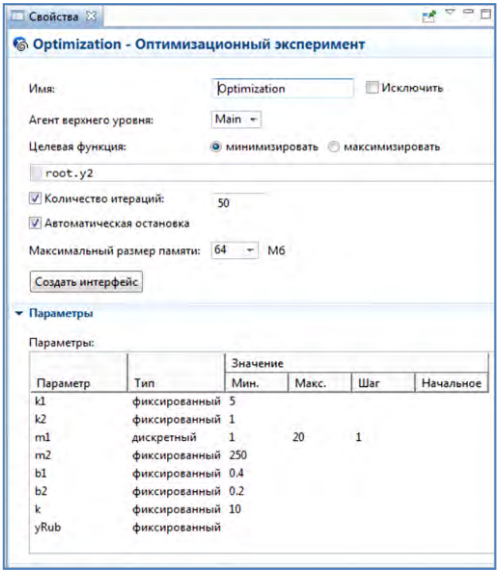


Рис. 3.62. Настройка оптимизационного эксперимента

Оптимизатор будет искать наилучшее значение массы успокоителя, приводящее к минимуму целевой функции. В поле *Целевая функция* внесем *root.y2*. Для *m*1 установим *Тип* – дискретный; минимальное значение – 1; максимальное – 20; шаг – 1.

Щелкнем по кнопке *Создать интерфейс*. Раскроем список запуска экспериментов и сделаем оптимизационный эксперимент *текущим*.

Оптимизатор начнет работать и через некоторое время решение будет найдено (рис. 3.63).

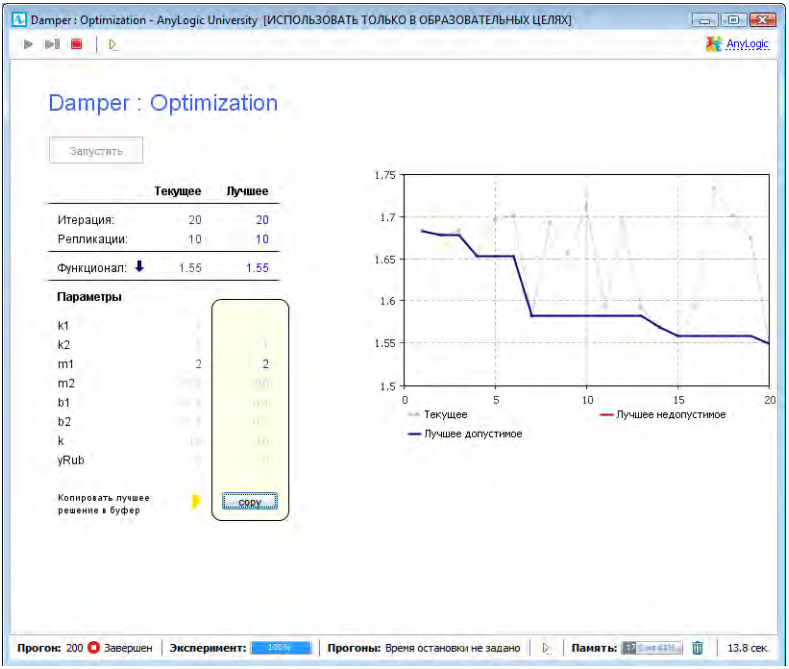


Рис. 3.63. Окно оптимизационного эксперимента

Значение целевой функции оказалось равным 1.55 при массе успокоителя 2.

Проверим, действительно ли она уменьшает значения критериев работы подвески. Вначале проведем простой эксперимент при значении массы успокоителя, равной 10. Изменим его на найденное оптимизатором.

Снова запустим *простой* эксперимент. Получим новые значения критериев (рис. 3.64).

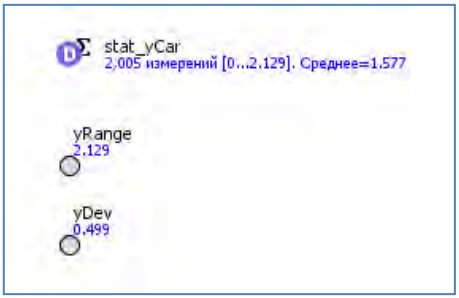


Рис. 3.64. Результат эксперимента после внесения найденного значения массы успокоителя

Как видим, диапазон колебаний кузова уменьшился на 16%, а стандартное отклонение его вертикальной координаты – на 12%. Эго свидетельствует об увеличении плавности хода кузова. Таким образом, значение массы успокоителя, найденное в оптимизационном эксперименте, удовлетворяет поставленной задаче.

4. ДИСКРЕТНО-СОБЫТИЙНОЕ МОДЕЛИРОВАНИЕ

4.1. Агентные модели конечных автоматов

У каждого своя жизнь.
Прописная истина

Дискретно-событийный подход применяется к большой и разнородной группе задач, которые формализуются тремя математическими схемами (*F, P, Q*).

Первые две описывают структуры с дискретными состояниями – автоматы.

Если система обладает разнообразным поведением, и может переходить в качественно различные состояния под воздействием входных сигналов (команд, событий), то ее можно описать в рамках концепции *Конечных автоматов* (*F*-схем).

Конечный автомат – математическая модель поведения устройств с конечной памятью. Она широко используются для описания сложных детерминированных дискретных систем, например, в проектировании ПЛК (программируемых логических контроллеров).

Конечный автомат состоит из четырёх основных элементов:

1) состояний, которые определяют поведение и могут производить действия;

2) переходов состояний, которые определяют переход от одного состояния к другому;

3) правил и условий, которые должны быть выполнены для осуществления перехода;

4) входных состояний, полученных извне или изнутри, которые могут согласно правилам привести к переходу состояний.

В настоящее время идея конечных автоматов широко используется при создании компьютерных игр. И это не удивительно, поскольку алгоритм конечного автомата реа-

лизован в одной из самых ранних разработок в этой области – игре «Жизнь».

«Жизнь» – матричная игра – *клеточный автомат*, придуманный Джоном Конвэем. Она демонстрирует пример возникновения сложных структур, в основе которого могут лежать простые правила.

Существует множество реализаций «Жизни». В примерах моделей AnyLogic она также есть.

Построим свой вариант этой по-настоящему культовой игры.

Будем использовать агентный подход, поддержка которого реализована в AnyLogic. Кратко его суть заключается в следующем. Моделируются отдельные элементы системы – агенты и среда. Агенты обладают типовым набором свойств (морфологических и реактивных), в соответствии с которым они функционируют в среде. При этом они могут образовывать популяции, взаимодействовать с другими агентами своего типа (класса) или с агентами других типов. Популяции агентов могут состоять из вариативных групп, например, возрастных категорий, социальных групп и т.п. Поскольку взаимодействие агентов со средой может носить взаимный характер, то ее также логично рассматривать как агента, в котором размещены популяции агентов нижних уровней.

Таким образом, моделируется сложная система, состояния которой являются результатом взаимодействия всех ее элементов.

Наша игра будет содержать популяцию 10000 клеток, размещенных в матрице размером 100×100 ячеек. Щелкая по клетке, можно изменять ее цвет (состояния – живая или неживая).

На каждом шаге игры будем проверять состояния ее ближайших соседей – по горизонтали, вертикали и диагоналям (в так называемом Муровом пространстве). В зависимости от результата будет определяться состояние клетки на следующем шаге. Состояния всех клеток будем сохранять в массиве размером в 10000 элементов. Живой клетке будет соответствовать элемент, хранящий 1, нежи-

вой – 0. Массив будет играть роль «кратковременной памяти» популяции клеток. На каждом шаге его содержимое будет обновляться.

Алгоритм построения следующий:

1. Создадим агента. Пункты главного меню: *Файл – Создать – Тип агента*. В открывшемся диалоге выберем пункт *Не использовать шаблоны типов агентов*. В поле *Имя нового агента* напишем *Cell*.

В структуре модели появился новый элемент – класс *Cell*. Щелкнем по нему. В окне открывшегося графического редактора класса поместим элемент *Прямоугольник* и назовем его *rect*. Прямоугольник будет отображать состояния клетки путем изменения цвета заливки. Пусть белый цвет характеризует состояние «неживая», а красный – «живая». Настроим свойства прямоугольника-клетки (рис. 4.1).

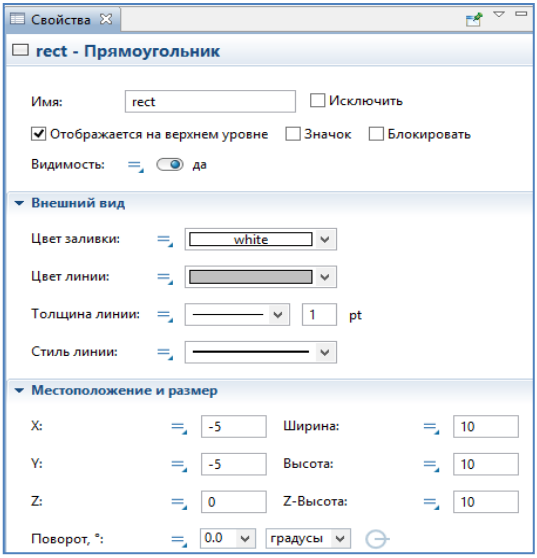


Рис. 4.1. Свойства клетки

Создаем единичную клетку («особь»). Она будет частью другого агента – «популяции», поэтому для корректного размещения клеток в пространстве популяции прямоугольник должен быть помещен в начало координат.

Добавим *переменную click* (тип – *boolean*, начальное значение – *false*).

В свойствах прямоугольника *Специфические* – *Действие по щелчку* запишем фрагмент кода:

```
this.rect.setFillColor(red);
click=!click;
if(click==true){this.rect.setFillColor(red);}
else {this.rect.setFillColor(white);};
```

Теперь можно создавать комбинации состояний клеток на игровом поле, щелкая по ним мышью.

2. Создадим новый класс *MyCell* – среду, в которой будут функционировать клетки. Откроем его в графическом редакторе. Из окна структуры модели перетащим в него элемент *Cell*. Зададим количество клеток (10000) (рис. 4.2).

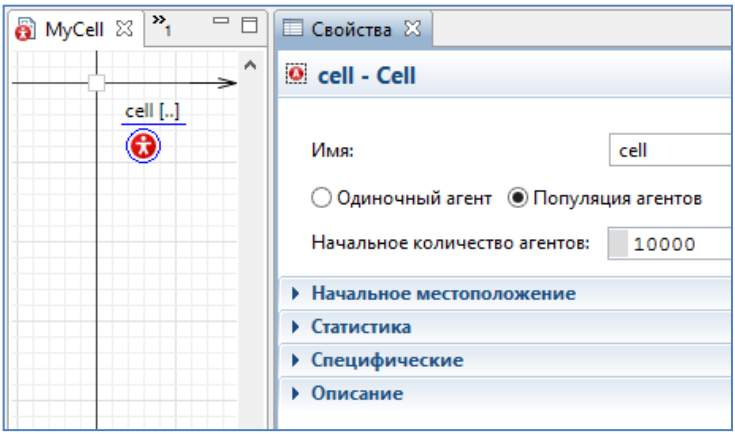


Рис. 4.2. Клетки становятся популяцией

Настроим свойства среды *MyCell*. Поскольку размер клетки 10×10 пикселей (см. рис. 4.1), то пространство, которое займет популяция из 10000 клеток, должно составлять 1000×1000 пикселей. Выберем *Тип пространства* – *Дискретное; Тип соседства* – *Мурово* (рис. 4.3).

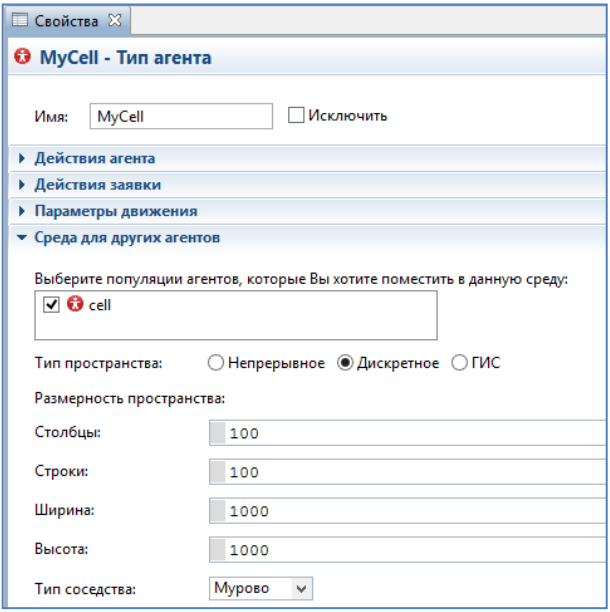


Рис. 4.3. Свойства популяции клеток

3. Создадим логику поведения клетки. Для этого понадобятся:

1) массив из 10000 элементов для хранения состояния клеток;

2) функция, с помощью которой будет определяться текущее состояние клетки, количество ее ближайших соседей и ее состояние на следующем шаге (оно будет сохраняться в массиве);

3) переменная, в которой будет сохраняться в виде суммы результат проверки количества соседей клетки;

4) функция, которая будет проверять состояние массива («памяти») и формировать состояние игры на следующем шаге, т.е. делать клетки красными или белыми.

Введем в модель среды *MyCell* переменную *a*. В свойствах переменной выберем *Тип – Другой.*

Далее запишем тип переменной: ***int*** **[]** (массив).

В поле *Начальное значение* запишем: *new int*[10000].

Введем в модель функцию *checkCell()*. В поле *Тело функции* запишем код:

```
for (int i = 0; i < cell.size(); i++)
// определяем состояния клеток в границах строк и столбцов:
if (cell.get(i).getR()>0 & cell.get(i).getR()<99 & cell.get(i).getC()>0 &
cell.get(i).getC()<99)

if (cell.get(i).rect.getFillColor()==red) /*если клетка красная(живая) */
  {sum=0;
for (CellDirection c : CellDirection.values()) /*проверка 8 ближайших соседей*/
        if(((Cell)cell.get(i).getAgentNextToMe(c)).rect.getFillColor()
==red){sum++;};

if (sum==2 || sum==3) {a[i]=1;} /* клетка остается жить, запоминаем со-
стояние в массиве*/
    else {a[i]=0;} /* иначе - на следующем шаге
    умирает*/
  }

 else //если клетка белая (неживая)
  {sum=0;
for (CellDirection c : CellDirection.values())/*проверка 8 ближайших соседей*/
if(((Cell)cell.get(i).getAgentNextToMe(c)).rect.getFillColor() ==red){sum++;};
  if (sum==3) {a[i]=1;}// клетка рождается
  }
```

Действия, вызываемые данной функцией, следуют из приведенных комментариев, однако код требует некоторого пояснения.

Агенты имеют параметр целого типа – индекс, по которому их можно однозначно идентифицировать. Конструкция *cell.size()* возвращает размер популяции агентов – *cell[…]*. Для этого использовали оператор цикла языка *Java* – *for(<условие>){<действие>;}*. Его роль – обеспечить итерацию по популяции агентов (клеток).

Дальнейшие операции осуществляются в теле этого цикла.

Конструкции *cell.get(i).getR()* и *cell.get(i).getC()* определяют номер строки и столбца *i*-го агента.

В теле функции использован еще один вид цикла *Java* – итерация по коллекции. В упорядоченном дискретном пространстве агенты имеют ближайших соседей по направлениям. Цикл с условием (*CellDirection c : CellDirec-*

tion.values()) как раз и обеспечивает проверку попадания соседа в окружение клетки. Переменная *c*, перечисляемого типа ***CellDirection,*** последовательно принимает значения, соответствующие «сторонам света» – *N, S, SW* и т.д.

В теле цикла метод

```
(Cell)cell.get(i).getAgentNextToMe(c)).rect.getFillColor()
```

находит ближайшего к клетке агента (другую клетку) и проверяет его цвет.

Введем в модель функцию *checkArray()*. Она будет проверять содержимое элементов массива *a* и закрашивать клетку в соответствующий результату (0 или 1) цвет. В поле *Тело функции* запишем код:

```
for (int i = 0; i < cell.size(); i++)
if (cell.get(i).getR()>0 & cell.get(i).getR()<99 & cell.get(i).getC()>0 & cell.get(i).getC()<99)
{
 if (a[i]==1) {cell.get(i).rect.setFillColor(red);}

 if (a[i]==0) {cell.get(i).rect.setFillColor(white);}
}
```

Откроем окно свойств агента. Выберем пункт *Среда для других агентов*. Активизируем чекбокс *Выполнять шаги* и запишем действия, которые должны выполняться (рис. 4.4). Значение периода установим равным единице.

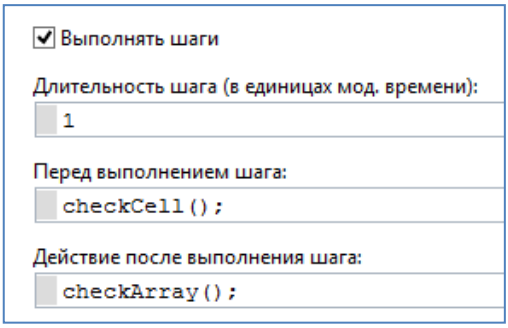


Рис. 4.4. Настройка действий в популяции клеток *MyCell*

4. Посмотрим результат. При запуске эксперимента «ничего» не происходит, т.к. по умолчанию запускается модель (агента) верхнего уровня *Main.* А он пока пуст. Но есть готовая модель популяции *MyCell.* В свойствах *Эксперимента* выберем его в качестве агента верхнего уровня (рис. 4.5).

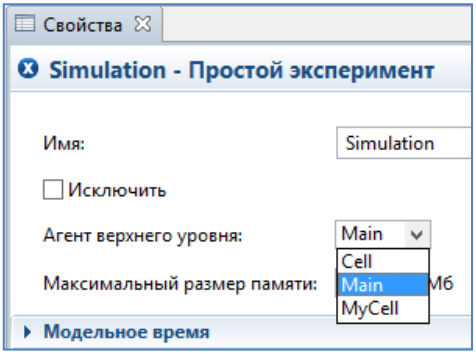


Рис. 4.5. Выбор компонента модели в качестве агента верхнего уровня

После этих настроек модель начинает работать корректно. Убеждаемся в том, что «Жизнь» удалась!

5. Вернемся к «проблеме» пустого уровня *Main.* Его вполне можно удалить. Однако он понадобится в своей изначальной роли – агента верхнего уровня.

Существует другой пример конечного автомата с простым поведением. Представим себе агента в двумерном дискретном пространстве (поле клеток). Он перемещается на одну клетку вправо или влево. Направление движения агента зависит от цвета клетки, с которой он совершает переход. При этом он «окрашивает» клетку, на которую переходит, в один из двух цветов. Цвет окрашиваемой клетки зависит от направления поворота агента.

Этот автомат получил название *муравья Лэнгмана.* Несмотря на очевидно простые правила поведения, он обладает удивительной особенностью. Поначалу муравей создает хаотически окрашенный узор, но через некоторое число переходов начинает «строить дорогу» из клеток, расположенных в одном направлении.

6. Построим двухуровневую агентную модель муравья Лэнгмана, аналогичную той, которую использовали при создании «Жизни».

Создадим новый класс модели агента *Ant.*

В окне графического редактора класса с помощью инструментов рисования создадим фигурку муравья. Назовем ее *pict* и поместим центр координат (рис. 4.6).

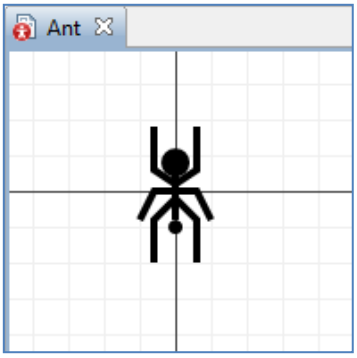


Рис. 4.6. Картинка презентации агента

Введем в модель класса агента параметр *flag*.

В свойствах параметра выберем тип ***String;*** значение по умолчанию – "*N*" (север). В этом параметре будет «запоминаться» направление, откуда муравей пришел ("*N*", "*S*", "*E*", "*W*").

Введем еще один параметр *dir*, тип – ***double.*** В нем будет храниться значение угла поворота картинки. С его помощью мы добьемся большей «реалистичности» перемещений муравья.

Создадим функцию *checkCell().* В поле *Тело функции* запишем код:

```
for (int i = 0; i < main.myCell.cell.size(); i++) {
 Cell cell=main.myCell.cell.get(i);
//определяем номера строки и столбца клетки
   int  r=cell.getR();
   int  c=cell.getC();
//останов, если муравей достиг границ
if ((getC()==1)||( getC()==99)||( getR()==1)||( getR()==99))
```

```
{get_Main().pauseSimulation();}

if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==red)&flag=="N")
 {
cell.rect.setFillColor(white);//перекрашиваем клетку
//новое направление движения, в данном случае, направо
  flag="E";
// поворачиваем фигурку муравья
  dir=dir+PI/2; pict.setRotation(dir);
  jumpToCell(r,c+1);// переход в новую клетку
  break;// выход из цикла
  };
if
((getC()==c)&(getR()==r)&(cell.get(i).rect.getFillColor()==white)&flag=="N")
{cell.rect.setFillColor(red);flag="W";dir=dir-PI/2;  pict.setRotation(dir);  jump-
ToCell(r,c-1);break;};
if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==red)&flag=="E")
{cell.rect.setFillColor(white);flag="S";dir=dir+
PI/2; pict.setRotation(dir); jumpToCell(r+1,c);break;};
if      ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==white)&flag=="E")
{cell.rect.setFillColor(red);flag="N";dir=dir-PI/2;  pict.setRotation(dir);  jumpToCell(r-
1,c);break;};
if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==red)&flag=="S")
{cell.rect.setFillColor(white);flag="W";dir=dir+
PI/2; pict.setRotation(dir); jumpToCell(r,c-1);break;};
if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==white)&flag=="S")
{cell.rect.setFillColor(red);flag="E";dir=dir-PI/2;  pict.setRotation(dir);  jump-
ToCell(r,c+1);break;};
if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==red)&flag=="W")
{cell.rect.setFillColor(white);flag="N";dir=dir+
PI/2; pict.setRotation(dir); jumpToCell(r-1,c);break;};
if ((getC()==c)&(getR()==r)&(cell.rect.getFillColor()==white)&flag=="W")
{cell.rect.setFillColor(red);flag="S";dir=dir-PI/2;  pict.setRotation(dir);  jump-
ToCell(r+1,c);break;};
}
```

Примечание. Мы не создаем для муравья своего игрового поля. Он будет проверять и изменять состояния клеток «Жизнь». Для этого он должен их найти. Клетки будут располагаться на третьем уровне иерархии агентов, по-

стольку первым уровнем в нашей модели будет *пока* пустой класс (агента) *Main.* Обратиться к нему можно с помощью метода *get_Main()* или по ссылке *main,* которую AnyLogic размещает в теле каждого агента.

Муравей будет находить клетки так:

• определение размера популяции клеток

```
main.myCell.cell.size();
```

• обращение к *i*-й клетке

```
main.myCell.cell.get(i).
```

Настроим поведение муравья. В *Свойствах* агента Ant *Действия при запуске* запишем фрагмент кода:

```
flag="N";
dir=0;
```

Действия на шаге: checkCell().

Сформируем среду для муравья. Создадим новый класс *MyAnt.* Поместим в него *Ant* (популяцию агентов). В свойстве *Начальное количество агентов* зададим значение 1.

В свойстве *Среда для других агентов* класса *MyAnt* установим такие же значения параметров, как у среды клеток «Жизнь».

Почему бы сразу не поместить муравья в среду клеток? Дело в том, что два агента не могут находиться в одной точке пространства (иметь совпадающие значения координат), поэтому муравей будет перемещаться в своей среде (в агенте *MyAnt*) и из него контролировать состояния клеток. Параметры среды муравья совпадают со средой «Жизнь» за исключением одного. В свойстве *Длительность шага* для муравья нужно установить другое значение, например 0.3. Так, поведение сделаем агентов асинхронным и позволим клеткам «Жизнь» создавать комбинации на основе тех, что успеет наделать муравей.

7. Заключительная стадия проекта. Помещаем популяции агентов *myCell* и *myAnt* в *Main* (рис. 4.7).

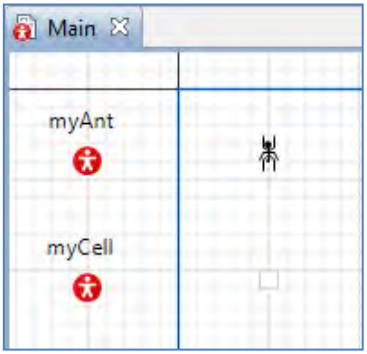


Рис. 4.7. Среда для популяций клеток и муравьев – *Main*

Поместим в центр игрового поля одного муравья. Для этого в *Свойствах* (*Main*) – *Действия агента при запуске* запишем:

```
myAnt.ant.get(0).jumpToCell(49,49);
```

Запустим эксперимент. Муравей начнет перекрашивать клетки, а «Жизнь» «оживлять» их. Для последней известны многочисленные устойчивые («неумирающие») комбинации клеток. Они бывают двух типов: стационарные (статические) и динамические – возобновляющиеся после определенного числа переходов.

Довольно быстро модель построила несколько простых устойчивых комбинаций, включая передвигающегося по полю «лемминга» (рис. 4.8).

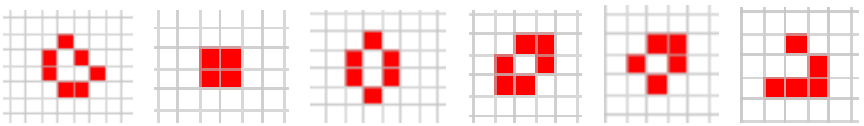


Рис. 4.8. Простые устойчивые комбинации, найденные в модели. Крайний справа – «лемминг» (*walker*)

После примерно 4000 шагов в игре появилась сложная конструкция, которая возвращается в прежнее состояние после трех переходов (рис. 4.9).

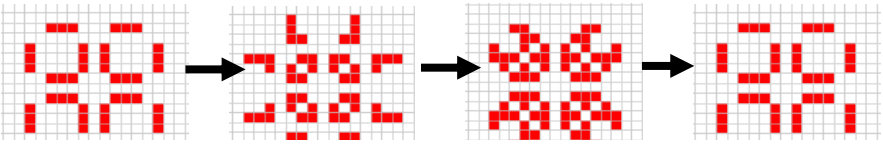


Рис. 4.9. Устойчивая комбинация клеток, «найденная» в модели

Эта структура также известна поклонникам игры как «Пульсар CP 48–56–72». Но в данном случае ее нашла модель, а не человек. Полученная система представляет собой вариант машины Тьюринга – универсального вычислительного устройства, способного в идеальном случае решить задачу любой сложности.

8. Сохранение игрового поля. Добавим в модель возможность загружать и сохранять понравившиеся состояния поля в файле.

Предварительно создадим на диске файл Excel с именем, например *Ant_Life*. Раскроем палитру *Внешние данные* и перетащим в *Main* элемент *Файл Excel*.

Переименуем его в *data*. В *Свойстве элемента – Файл* выберем сохраненный на диске файл *Ant_Life* (свяжем файл с элементом модели).

Из палитры *Управление* перетащим в модель две кнопки (рис. 4.10).

Рис. 4.10. Кнопки работы с файлом

В свойстве *Действие* кнопки *Сохранить поле* запишем код:

```
int k=0;
for (int i=1; i<101;i++)
for (int j=1; j<101;j++)
{
 data.createCell(1, i, j);
  if (myCell.a[k]==1)
   data.setCellValue(myCell.a[k],"Лист1",i,j);
 k++;
}
```

В двойном цикле (по строкам и столбцам листа Excel) с помощью метода *createCell* (*1*, *i*, *j*) будет создаваться ячейка в первом листе таблицы, затем в нее помещаться единица из массива *a*, принадлежащего агенту *myCell*. Нумерация элементов массива сквозная, поэтому итерацию по массиву будет обеспечивать переменная *k*.

Код действия для кнопки *Загрузить поле*:

```
int k=0;
for (int i=1; i<101;i++)
for (int j=1; j<101;j++)
{
 if  data.getCellNumericValue(1, i, j)==1)
  {myCell.cell.get(k).rect.setFillColor(red);}
 k++;
}
```

Метод *getCellNumericValue* (*1, i, j*) возвращает значение, хранимое в ячейке таблицы. Если это – единица, клетка на поле закрашивается в красный цвет.

Проверим работу кнопок (рис. 4.11).

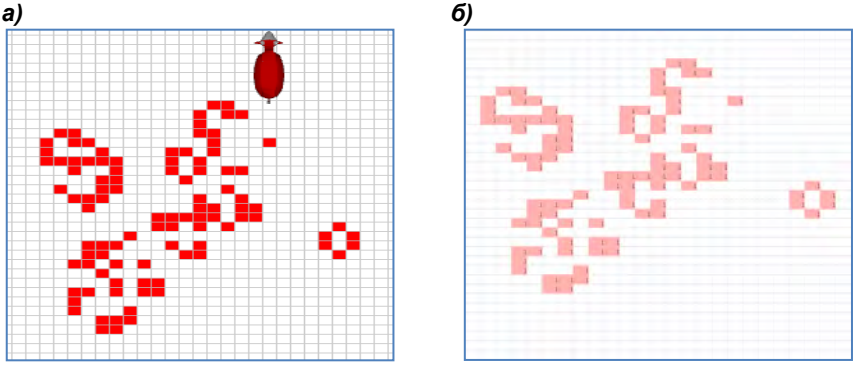


Рис. 4.11. Фрагмент игрового поля (*а*) и его состояния, сохраненного на листе Excel (*б*)

4.2. Моделирование систем методами статистических испытаний

Если переходы состояний автомата носят стохастический характер, то система называется *вероятностным автоматом.*

Значение ее параметров в данный момент времени точно предсказать невозможно. Однако можно найти множество простых однородных вероятностных («хаотических») процессов, которые приводят к построению сложно организованных упорядоченных структур.

Оказалось, что зная о характере распределения вероятностей событий и имитируя их с помощью генераторов случайных чисел, адекватные модели таких систем построить можно.

Подходы, с помощью которых можно реализовать подобные задачи, получили название ***методов статистических испытаний***, или ***методов Монте-Карло.***

Нахождение определенного интеграла методом Монте-Карло

По геометрической интерпретации значение интеграла равно площади подынтегральной функции.

Пусть задана функция *f(x)*, отличная от нуля в заданном интервале от *a* до *b*. Построим ее график в координатной плоскости. Выделим на ней прямоугольную область высотой *h*, большей абсолютного максимума функции в искомом интервале и основанием *b–a* (рис. 4.12).

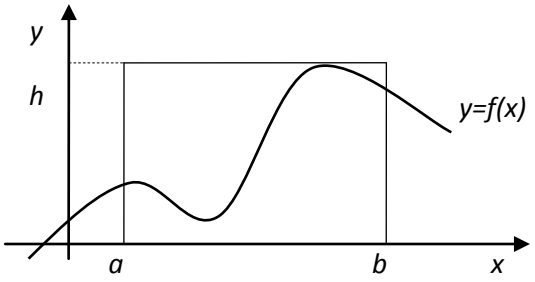


Рис. 4.12. Нахождение интеграла

Рассмотрим методику нахождения определенного интеграла с помощью процедуры, основанной на методе статистических испытаний.

Разыграем два равномерно распределенных случайных числа *x* и *y* на интервалах от *a* до *b* и от 0 до *h*. Они определяют координаты точек, попадающих внутрь прямоугольника, охватывающего кривую. Повторяем опыт *N* раз. Вероятность попадания точек в область ниже подынтегральной функции вычисляется по формуле:

$$P = \frac{S_{f(x)}}{S_{прям}} = \frac{S_{f(x)}}{h(b-a)} = \frac{N_{f(x)}}{N_{прям}}.$$

Отсюда площадь подынтегральной функции

$$S_{f(x)} = h(b-a)\frac{N_{f(x)}}{N_{прям}},$$

где $N_{прям}$ – общее число точек, попавших в прямоугольник, равное числу испытаний (*N*); $N_{f(x)}$ – число точек, попавших под функцию, для его определения, координаты всех точек нужно проверять на соответствие условию: $y \le f(x)$.

Построим имитационную модель нахождения определенного интеграла:

$$\int_0^{\pi} \sin(x)dx = -\cos(\pi) - (-\cos(0)) = 2.$$

Создадим логику модели. Внесем в нее два параметра: *n* и *n_inF*. В первом будем сохранять число точек, попавших в прямоугольник, во втором – число точек под функцией.

Добавим *Динамическую переменную integral*. Она будет накапливать искомое значение площади под функцией. В поле выражения для нее запишем: *Math.PI*n_inF/n*.

Установим связи между параметрами и переменной.

Подготовим область вывода результатов имитационного эксперимента. В свойстве *Main – Действия агента при запуске* напишем фрагмент кода:

```
//Рисование осей координат
Color myColor=new Color(0,0,0);
//Y-axe
ShapeLine myY=new ShapeLine  (true, 100, 100,
myColor, 0, 300+20, 1, LineStyle.LINE_STYLE_SOLID);
presentation.add(myY);
//X-axe
ShapeLine myX=new ShapeLine  (true, 100-20, 400,
myColor, PI*150+40, 0, 1, LineSyle.LINE_STYLE_SOLID);
presentation.add(myX);
```

В точку презентации с координатами *x*: 300, *y*: 200 введем элемент *Текст*. Он будет отображать текущее (рассчитываемое) значение интеграла.

Для выбора цвета рисования используется класс *Java Color* и класс *AnyLogic ShapeLine* – для рисования осей. Экземпляры класса (объекты) создаются командой-конструктором ***new***.

Команда *presentation.add()* помещает в окне эксперимента линии черного цвета. Оси координат будут появляться после запуска модели в позициях, указанных в параметрах объекта *ShapeLine*.

Добавим в модель элемент *Событие event*. Установим значения его свойств:

- тип события – по таймауту;
- режим – циклический;
- время первого срабатывания – 0;
- период – 1;
- действие

Тогда

```
for (int i=0; i<100; i++){
  n++;
double x;
double y;
  x=uniform(0,Math.PI);
  y=uniform(0,1);
if (y<=Math.sin(x) && y>0) {
n_inF++;
```

```
        ShapeOval dot=new ShapeOval(true,x*150+100,-
y*150+400,0,red,red,0.1,1,1,LineStyle.
        LINE_STYLE_SOLID);
          presentation.add(dot);
        }else{
        ShapeOval dot=new ShapeOval(true,x*150+100,-
y*150+400,0,lightGrey,lightGrey,0.1,1,1,
        LineStyle.LINE_STYLE_SOLID);
          presentation.add(dot);
          }
         }
        text.setText(integral);
```

Примечание. В течение одного периода сразу определяется положение 100 точек.

В параметре *n* сохраняется текущее значение числа экспериментов.

С помощью функции Java *uniform ()* на каждом шаге цикла два раза последовательно вызывается («разыгрывается») случайное равномерно распределенное вещественное число на интервале от 0 до π – для *x*, и от 0 до 1 – для *y*.

Если эти значения находятся в области под линией функции или на ней, то ставится красная точка, если над функцией – серая. Число красных точек подсчитывается в параметре *n_inF*.

Выполнение команды *text.setText(integral)* приводит к появлению в текстовом элементе рассчитанного значения интеграла.

Внесем в модель *Временной график*. Настроим его для показа одного элемента данных – переменной *integral.* На графике будет отображаться процесс поиска решения.

После выполнения примерно 15 шагов эксперимента модель находит приближенное значение интеграла с относительной погрешностью 0,1% (рис. 4.13). Затем результат остается практически неизменным.

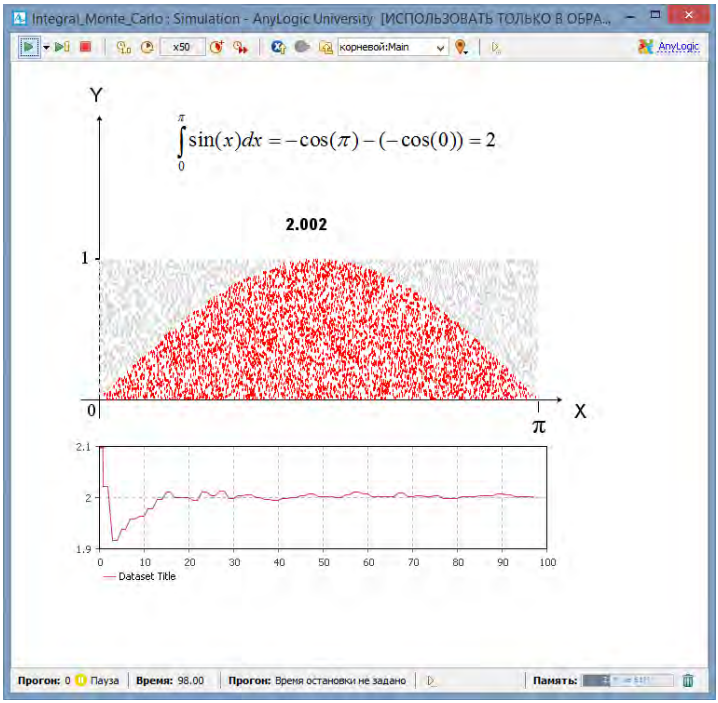


Рис. 4.13. Окно презентации имитационного эксперимента

Дискретная вероятностная модель теплопроводности

Метод статистических испытаний с успехом используется при решении уравнений параболического типа [4].

Одномерный вариант таких задач – модель теплопроводности – рассмотрен в гл. 3.4.

На атомно-молекулярном уровне изменение температуры в веществе носит дискретный характер и определяется изменением отношения количеств возбужденных и невозбужденных частиц [7].

Оценку значения и направление температурного градиента конечно-разностной схемы в модели теплопроводности (см. рис. 4.2) можно связать с вероятностью нахождения частицы в данном слое в определенный момент времени. Распределение температуры можно получить, моделируя вероятности переходов частиц между слоями.

Для построения дискретной вероятностной модели этой системы используем разностную схему Кранка-Николсона, которая дает вероятности нахождения частиц в момент времени *t:*

- в *j*-м слое

$$p_1 = \frac{h - 2\tau(1-\lambda)}{h^2 + 2\lambda\tau};$$

- в *j+1*-м слое

$$p_2 = \frac{\tau(1-\lambda)}{h^2 + 2\lambda\tau};$$

- в *j–1*-м слое

$$p_3 = \frac{\tau\lambda}{h^2 + 2\lambda\tau},$$

при $\lambda = 0{,}5$.

При соответствующих значениях временного и пространственного интервала получим: $p_1 = 0{,}6$; $p_2 = 0{,}1$; $p_3 = 0{,}1$.

В качестве основы моделирования воспользуемся моделью «Жизнь» (см. гл. 4.1). Для этого:

1. Сохраним модель под другим именем, например, *T_conduct_Monte_Carlo* и переименуем пакет.

2. Вернем *MyCell* роль агента верхнего уровня (см. рис. 4.5).

3. Удалим переменную *a* и функцию *chekArray.*

4. Заменим код функции *chekCell:*

```
double check;
for (int i = 0; i < cell.size(); i++)
{  if  (cell.get(i).rect.getFillColor()==red)
  {
   if (cell.get(i).getC()==1)
   { check=uniform_discr(1,10);
    if (check>4) {
      cell.get(i).rect.setFillColor(red);}
    if (check==3) {
      cell.get(i+1).rect.setFillColor(red);}
        }
      else
        {
```

```
        check=uniform_discr(1,10);
        if (check>4) {
          cell.get(i).rect.setFillColor(red);}
         check=uniform_discr(1,10);
        if (check==2){
         cell.get(i+1).rect.setFillColor(red);}
         check=uniform_discr(1,10);
        if (check==1){
         cell.get(i).rect.setFillColor(white);
         cell.get(i-1).rect.setFillColor(red);}
       } //else
    }
}
```

Теперь функция *chekCell* будет закрашивать клетки в соответствии с вероятностями их пребывания в слое (столбце клеток).

В первом столбце будем задавать температуру $T1$.

Упростим задачу и условимся, что $T1 > T2$ и $u_i = 0$. В этом случае движение клеток из первого столбца возможно только «вперед» – в столбец с номером 2.

С помощью функции *uniform_discr(1,10)* разыгрываются равномерно распределенные целые числа (от 1 до 10). Числа в интервале от 5 до 10 выпадают с вероятностью 0,6. Она определяет временную задержку клетки красного цвета в текущем слое.

Осуществляется проверка:

```
if uniform_discr(1,10)>4 {…}.
```

После выпадения одного числа (1 или 2) красная клетка с вероятностью 0,1 переходит в следующий слой (столбец) или в предыдущий.

5. Добавим в модель семь *Переменных*. Назовем их *u_0, u_1,…,u_6.* В них будем сохранять количества красных клеток – значение «температуры» в соответствующем слое.

6. Добавим функцию, с помощью которой будем считать красные клетки, *countCell:*

```
u_1=0;u_2=0;u_3=0;u_4=0;u_5=0;
for (int i = 0; i < cell.size(); i++)
 {
  if (cell.get(i).getC()==2 & cell.get(i).
    rect.getFillColor()==red) {u_1++;}
  if (cell.get(i).getC()==3 & cell.get(i).
    rect.getFillColor()==red) {u_2++;}
  if (cell.get(i).getC()==4 & cell.get(i).
    rect.getFillColor()==red) {u_3++;}
  if (cell.get(i).getC()==5 & cell.get(i).
    rect.getFillColor()==red) {u_4++;}
  if (cell.get(i).getC()==6 & cell.get(i).
    rect.getFillColor()==red) {u_5++;}
  if (cell.get(i).getC()==7 & cell.get(i).
    rect.getFillColor()==red) {pauseSimulation();}
  }
```

Поскольку решается краевая задача, то определим правую границу пластины. В поле клеток это столбец с номером 6. После «прорыва» единственной клетки через правую границу пластины (в столбец 7) работа эксперимента приостанавливается.

7. Добавим функцию *fillCell:*

```
for (int i = 1; i < 10000; i+=100)
{if(cell.get(i).getC()==1)cell.get(i).
rect.setFillColor(red);}
```

Её роль – обеспечить автоматическое закрашивание первого столбца клеток, которое будет имитировать температуру слева от пластины, равную 100 условным единицам.

8. Откроем свойства *myCell.* В поле *Действия агента – При запуске* запишем *fillCell();.* В свойстве *Среда для других агентов* в чекбоксе *Выполнять шаги* у нас уже стоит галочка. В поле *Перед выполнением шага* запишем *chekCell();.* В поле *Действие после выполнения шага – countCell();.*

9. Откроем модель *T_conduct* и скопируем из нее в *myCell* элементы презентации. Запустим эксперимент. Ре-

зультат аналогичен полученному в непрерывной детерминированной модели (рис. 4.14).

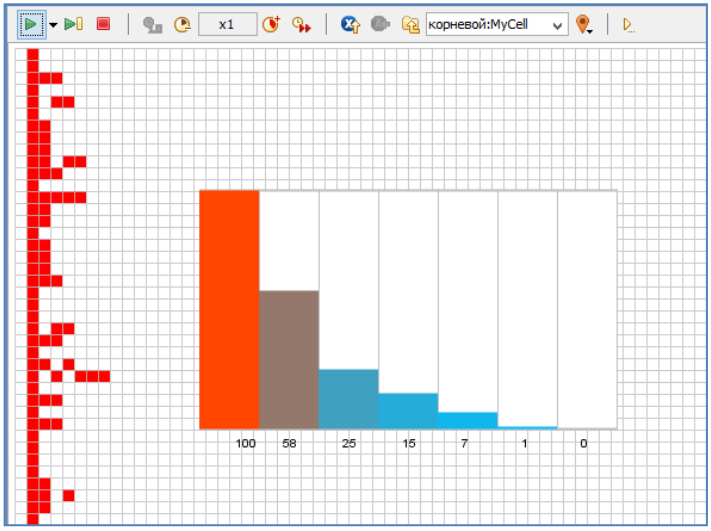


Рис. 4.14. Фрагмент вероятностной модели теплопроводности

Построение фрактала

Покажем на эффектном примере появление сложной упорядоченной структуры, основанной на вероятностных правилах.

Смоделируем фрактал – лист папоротника. Для этого:

1. Создадим новую модель.

2. Внесем в модель элемент *Функция*. Назовем ее *drawFem*. Свойство *Действие – Не возвращает ничего* оставим без изменений. Запишем код:

```
double x;
double y;
double i;
int c;
x=0;
y=0;

for (i=0;i<2000000;i++){
    c=uniform_discr(1,100);
```

```
        if (c==7) {x=0.20*x-0.26*y; y=0.23*x+0.22*y+1.6;
        ShapeOval dot = new ShapeOval(true,370+x*70,450-
y*50,0,springGreen,springGreen,.5,.5,.1,LineStyle.
    LINE_STYLE_SOLID);
            presentation.add(dot);
            }
        c=uniform_discr(1,100);
        if (c==1) {x=.0; y=0.16*y;
                ShapeOval dot = new ShapeOval(true,370+x*70,450-
y*50,0,springGreen,springGreen,.5,.5,.1,LineStyle.
    LINE_STYLE_SOLID);
            presentation.add(dot);
            }
        c=uniform_discr(1,100);
        if (c==85) { x= 0.85*x+0.04*y; y=-0.04*x+0.85*y+1.6 ;
            ShapeOval dot = new ShapeOval(true,370+x*70,450-
y*50,0,springGreen,springGreen,.5,.5,.1,LineStyle.
    LINE_STYLE_SOLID);
            presentation.add(dot);
            }
        c=uniform_discr(1,100);
        if (c==7) { x= -0.15*x+0.28*y; =0.26*x+0.24*y+0.44;
            ShapeOval dot = new ShapeOval(true,370+x*70,450-
y*50,0,springGreen,springGreen,.5,.5,.1,LineStyle.
    LINE_STYLE_SOLID);
            presentation.add(dot);
            }

     }
```

3. В свойстве *Main Действия агента при запуске* запишем *drawFem();*.

После запуска модели результат появится не сразу (рис. 4.15), поскольку функция осуществляет два миллиона итераций. На каждом шаге цикла четыре раза последовательно вызывается («разыгрывается») случайное целое число на интервале от 1 до 100. В зависимости от результата рассчитываются координаты точки.

Для рисования используется класс AnyLogic *ShapeOval*. Экземпляры класса (объекты) создаются командой-конструктором *ShapeOval dot* = ***new*** *ShapeOval()*.

Далее команда *presentation.add(dot)* помещает в окне презентации кружок ярко-зеленого цвета.

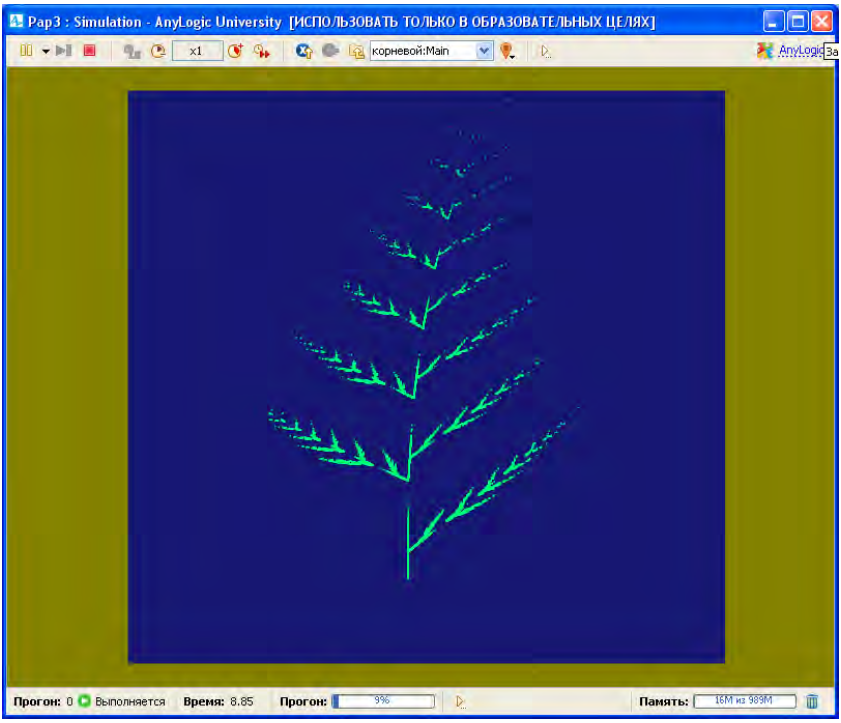


Рис. 4.15. Лист папоротника – символ фрактальной геометрии

5. МОДЕЛИРОВАНИЕ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ

5.1. Общие сведения

Существуют вероятностные системы, которые имеют три специфических признака (рис. 5.1):

- объекты, у которых может возникнуть потребность в удовлетворении некоторых требований (заявок);
- агрегаты, предназначенные для удовлетворения заявок на обслуживание;
- специальная организация приема заявок и их обслуживания.

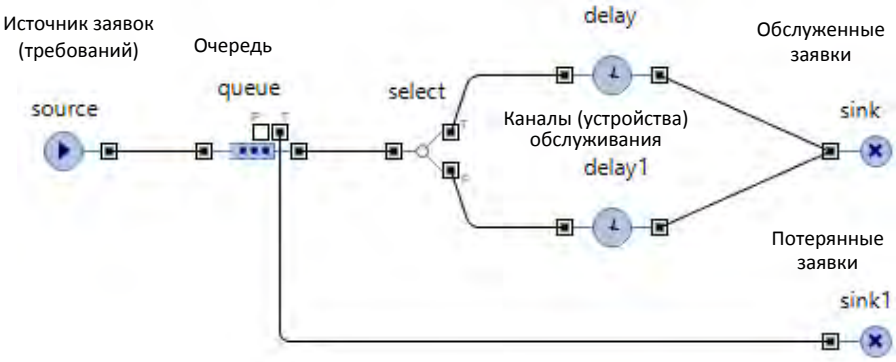


Рис. 5.1. Схема системы массового обслуживания

Такие структуры называются ***системами массового обслуживания*** (СМО). Примерами СМО могут служить: организация медицинских услуг пациентам в поликлинике; банковское обслуживание клиентов; работа АЗС; работа мобильной связи; производственные процессы в промышленной сфере; выполнение задач в вычислительных сетях и т.п.

Совокупность заявок рассматривают как поток событий, происходящих в случайные моменты времени. Для простейшего дискретного потока частота поступления требований в систему подчиняется закону Пуассона. Вероятность поступления за время t ровно k требований задается формулой:

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t},$$

где λ – число требований, поступающих в систему в единицу времени (интенсивность потока).

Распределение Пуассона описывает возникновение редких событий с неизменной частотой. Его максимум приходится на область около λ, при удалении в обе стороны оно быстро спадает (рис. 5.2).

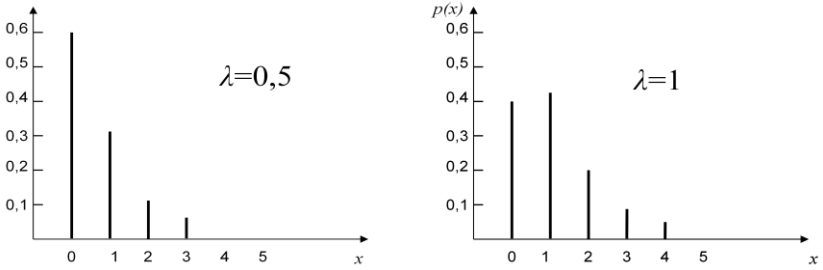


Рис. 5.2. Распределение Пуассона при различных значениях λ. Функция AnyLogic: *poisson (double lamda)*

Простейший поток обладает тремя основными свойствами: ординарность, стационарность и отсутствие последействия.

Ординарность потока означает практическую невозможность одновременного поступления двух и более требований. Например, достаточно малой является вероятность того, что из группы станков, обслуживаемых бригадой ремонтников, одновременно выйдут из строя сразу несколько станков.

Стационарным называется поток, для которого математическое ожидание интенсивности потока (λ) не меняется во времени.

Таким образом, вероятность поступления в систему определенного количества требований в течение заданного промежутка времени Δt зависит от его величины и не зависит от начала его отсчета на оси времени.

Экспоненциальное распределение часто используется для представления интервалов времени между случайны-

ми событиями, например, времени между прибытиями заявок в модели СМО или времени между отказами в моделях надежности (рис. 5.3).

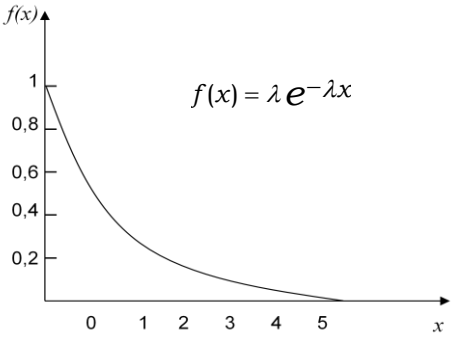


Рис. 5.3. Экспоненциальное распределение. Функция AnyLogic: *exponential (double lambda, double min)*

Процесс обслуживания направлен «навстречу» потоку заявок и тоже может характеризоваться интенсивностью (часто обозначаемой μ) или временем обслуживания.

Время обслуживания также является случайной величиной. В простых моделях (например, при недостатке информации о работе канала обслуживания) для ее моделирования применяют треугольное распределение (рис. 5.4).

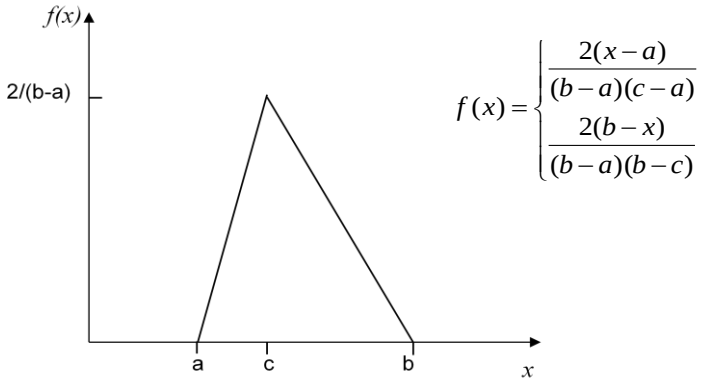


Рис. 5.4. Треугольное распределение. Функция AnyLogic: *triangular (double a, double b, double c)*

Существует несколько разновидностей СМО:

1) по числу каналов обслуживания делятся на одноканальные и многоканальные;

2) по числу фаз (последовательно соединенных агрегатов) – на однофазные и многофазные;

3) по наличию обратной связи – на разомкнутые (с бесконечным числом заявок) и замкнутые (с конечным числом заявок);

4) по наличию отказов – на системы с отказами и системы без отказов;

5) по наличию очереди – на системы без очередей (с потерями заявок), системы с ограниченным ожиданием (по времени или длине очереди) – тоже с потерями (*вытеснением* заявок); существуют системы без потерь заявок – с неограниченным ожиданием (по времени или длине очереди);

6) по принципу формирования очередей – на системы с общей очередью и системы с несколькими очередями.

Формирование очередей зависит от вида *дисциплины обслуживания*. В рамках очереди заявки могут обслуживаться либо в порядке *FIFO* (First In First Out – «первым пришел, первым обслужен»), либо *LIFO* (Last In First Out – «последним пришел, первым обслужен»).

Дисциплиной обслуживания также может быть *SJF* (Shortest Job First – самое короткое по времени выполнения задание обрабатывается первым) и обслуживание по приоритету.

Приоритет обслуживания имеет три разновидности:

• относительный приоритет (заявка высокого приоритета ожидает окончания обслуживания заявки с более низким приоритетом;

• абсолютный приоритет (заявка высокого приоритета при поступлении немедленно вытесняет заявку с более низким приоритетом);

• смешанный приоритет (если заявка с низшим приоритетом обслуживалась в течение времени, меньше критического, то используется относительный приоритет, в противном случае исполнится абсолютный приоритет).

Значение (назначение) приоритета заявки может быть статическим или вычисляться и изменяться во времени.

Система массового обслуживания – это динамические *непрерывные* системы. При постановке простых задач они имеют аналитическое решение.

Пусть, система *S* (например, сборочный цех) имеет два устройства (например, роботов на конвейере). Каждое такое устройство может иметь два состояния: (1,0). Например, «работает» и «не работает», «занято-свободно» и т.п.

Тогда *S* может находиться в одном из четырех состояний:

- *S*1 – оба устройства работают;
- *S*2 – первое устройство не работает, второе работает;
- *S*3 – первое устройство работает, второе не работает;
- *S*4 – оба устройства не работают.

Введем обозначения для потоков событий:

- λ_i – интенсивности перехода *i*-го устройства в состояние 0;
- μ_i – интенсивности перехода *i*-го устройства в состояние 1.

Данную систему можно рассматривать как двухканальную СМО с потоками требований λ_i и потоками обслуживания μ_i. Построим граф состояний системы (рис. 5.5).

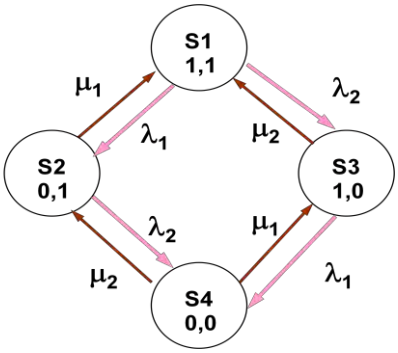


Рис. 5.5. Граф состояний системы

Пусть p_i – вероятность (переходная вероятность) того, что система находится в состоянии S_i.

Тогда для состояния S_1 выражение

$$p_1(\lambda_1 + \lambda_2) = p_2 \mu_1 + p_3 \mu_2$$

является вероятностью ухода системы из состояния $S1$, отнесенной к единице времени.

Рассматривая баланс приходов и уходов для каждого из четырех состояний, получаем систему *уравнений Колмогорова:*

$$\begin{cases} p_1(\lambda_1 + \lambda_2) = p_2 \mu_1 + p_3 \mu_2, \\ p_2(\lambda_2 + \mu_1) = p_1 \lambda_1 + p_4 \mu_2, \\ p_3(\lambda_1 + \mu_2) = p_1 \lambda_2 + p_4 \mu_1, \\ p_4(\mu_1 + \mu_2) = p_2 \lambda_2 + p_3 \mu_1. \end{cases}$$

Последнее уравнение можно заменить, так как оно может быть получено сложением первых трех. Поскольку система с вероятностью 1 находится в одном из четырех состояний, то

$$p_1 + p_2 + p_3 + p_4 = 1.$$

Решение данной системы в AnyLogic, при $\lambda_1 = 1$, $\lambda_2 = 2$, $\mu_1 = 2$, $\mu_2 = 3$, примет вид, как на рис. 5.6.

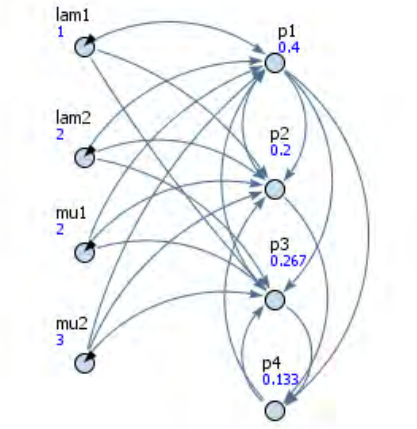


Рис. 5.6. Аналитическое решение простой СМО в AnyLogic

Моделирование сложных СМО с помощью систем уравнений часто оказывается трудоемким и сложно поддающимся анализу. При этом оказывается естественным рассматривать такие системы с точки зрения дискретно-событийного подхода.

5.2. Моделирование процесса выполнения задач компьютером

Компьютер работает в мультипрограммном режиме обработки пакетов, т.е. одновременно начинает обработку нескольких (вплоть до установленного максимума $k = 4$) заданий, но не может начать обработку новых заданий, пока не будет выполнен этот пакет. В пакете каждое задание имеет собственное время выполнения, которое покидает центральный процессор (ЦП) по его истечении. В системе существует три класса приоритетов: высший приоритет имеют задания типа 3, средний – задания типа 2, низший – задания типа 1. Когда ЦП завершает выполнение последнего задания в пакете, он сначала обращается к заданиям из очереди класса 3 и берет на выполнение как можно больше заданий, вплоть до указанного максимума k. Если в очереди класса 3 было меньше k заданий, ЦП принимает как можно больше заданий из очереди класса 2, чтобы сумма заданий класса 1 и класса 2 составила максимальный размер пакета k. Если в пакете еще остается место, ЦП переходит к очереди класса 1. Процессор не может начать обработку любых поступающих заданий, пока не завершит выполнение всех заданий в текущем пакете.

Установленное время обслуживания задания класса i равномерно распределено между константами a_i и b_i. Для каждого класса заданий действует отдельный процесс поступления, т.е. интервалы времени между поступлениями двух последовательных заданий класса i экспоненциально распределены со средним значением r_i.

Построим модель работы системы со следующими значениями параметров (табл. 5.1).

Таблица 5.1

Модель работы системы

i	r_i	a_i	b_i
1	5	4	8
2	1,6	1	2
3	0,2	0,5	1

Для моделирования логики модели будем использовать компоненты *Библиотеки моделирования процессов* (рис. 5.7).

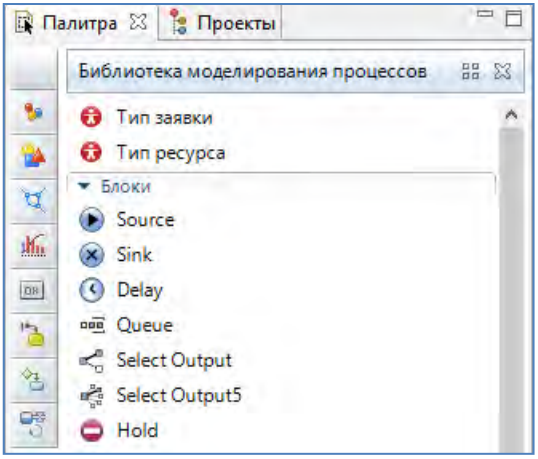


Рис. 5.7. Панель библиотеки моделирования процессов (фрагмент)

Библиотека моделирования процессов изначально поддерживает агентную модель заявок. Для моделирования СМО в рамках традиционного подхода в AnyLogic имеется аналогичная *Основная библиотека (старая):*

1. Создадим новую модель с именем *Proceccor* и далее – свой тип заявки. Для этого перетащим в *Main* элемент *Тип заявки* и выполним шаги по его созданию.

Изменим имя агента на *Entity* (заявка). Откажемся от предложения выбрать анимацию агента (позже создадим свою). Добавим агенту один параметр *priority* вещественного типа (рис. 5.8).

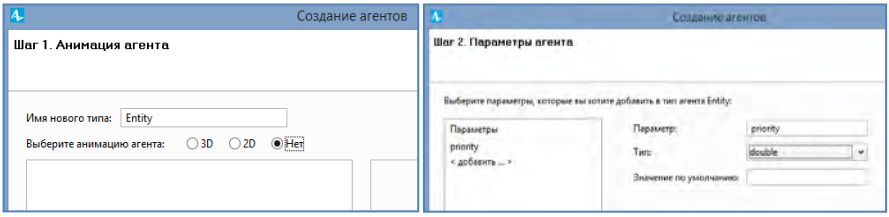


Рис. 5.8. Создание своего типа заявки

2. Агент *Entity* появился в структуре проекта как класс модели. Щелкнем по нему. Раскроется закладка окна графического редактора заявки. Добавим в него элемент *oval* из палитры *Презентация* и настроим его свойства. Выберем *Цвет заливки – silver, радиус* – 10. Перетащим агента в *Main* (рис. 5.9).

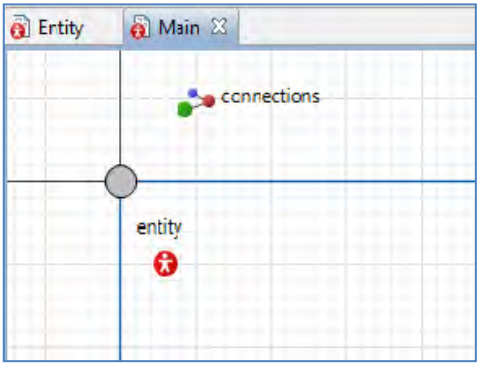


Рис. 5.9. Агент, использующийся как заявка

3. Создадим источник заявок первого типа (с наименьшим приоритетом). Для этого поместим в модель элемент *source* (источник). Переменуем в *source1*. Настроим его свойства. В поле *Тип заявки* напишем *Entity*. Заявки будут прибывать по экспоненциальному закону согласно времени между прибытиями. В качестве новой заявки будет фигурировать агент *Entity*. При подходе к выходу из источника заявке будет назначаться приоритет, при этом кружок-презентация заявки будет светло-зеленого цвета (рис. 5.10).

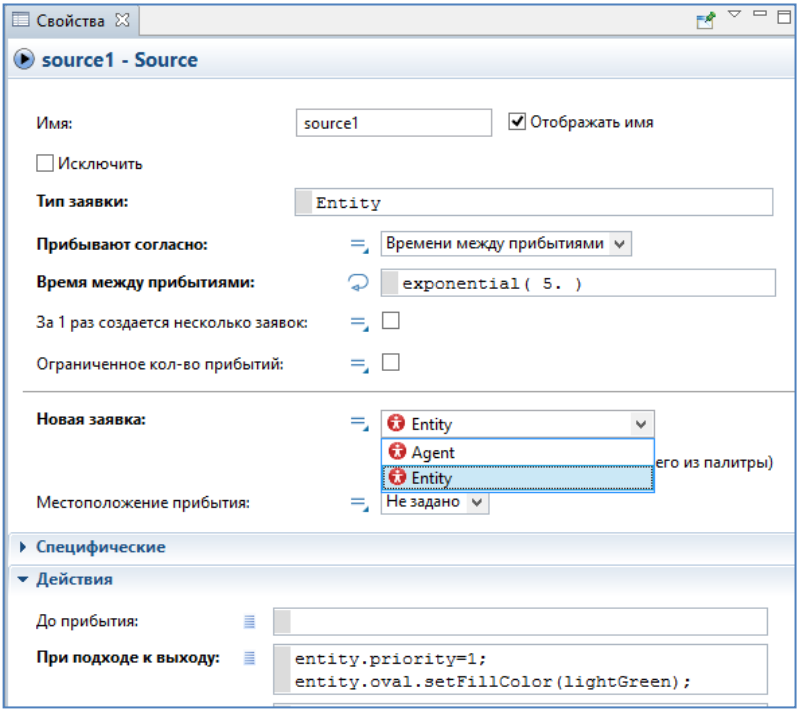


Рис. 5.10. Настройка свойств источника заявок с низшим приоритетом

4. Аналогично создадим источники заявок с приоритетом 2 и 3. Назовем их, соответственно, *source2* и *source3.* В поле свойств *Время между прибытиями* для источника *source2* запишем *exponential(1.6),* а для источника *source3: exponential(.2).* Внесем соответствующие изменения в код в поле *Действия – При подходе к выходу.* Из *source2* будут выходить кружки синего цвета, а из *source3* – красного.

> Выбрать вид распределения вероятностей и настроить его параметры можно, щелкнув по кнопке  панели инструментов *AnyLogic* (рис. 5.11). Данная опция активна в момент редактирования соответствующего поля в панели свойств элементов СМО.

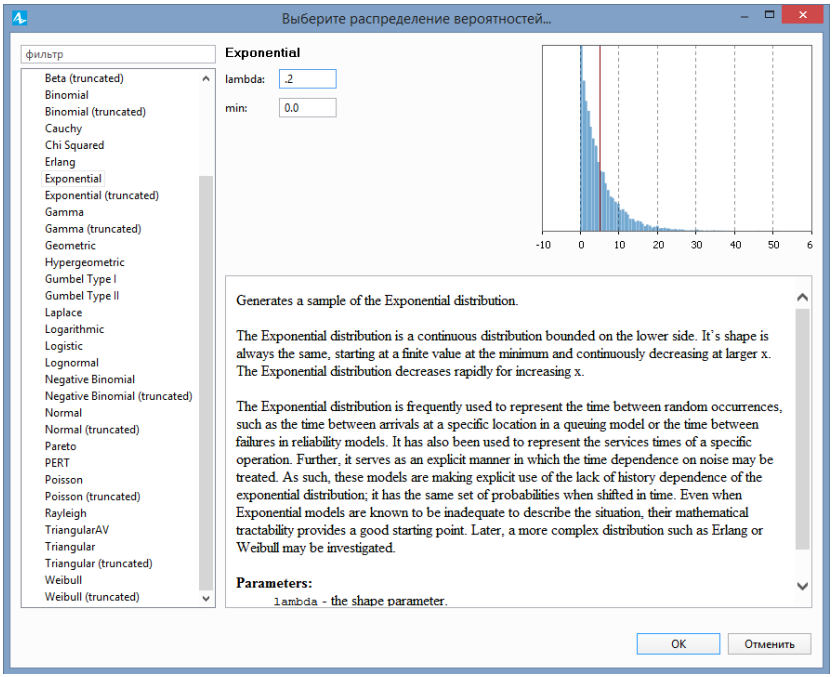


Рис. 5.11. Окно выбора распределения вероятностей

5. Внесем в модель три параметра целого типа. Назовем их *count_1, count_2, count_3*. В них будут сохраняться количество заявок соответствующего типа в пакете. Для контроля общего числа заявок в пакете (их должно быть ровно 4) понадобится еще один параметр – *pack.*

6. Раскроем список элементов *Разметка пространства* библиотеки процессов. Перетащим в модель два элемента *Прямоугольный узел.* В свойствах первого узла (*node*) *Аттракторы* выберем количество и расположение точек, в которые будут помещаться презентации заявок (кружки), находящихся в очереди.

Для второго узла (*node1*) установим *Количество аттракторов* 4 и сетку 1×4 (рис. 5.12).

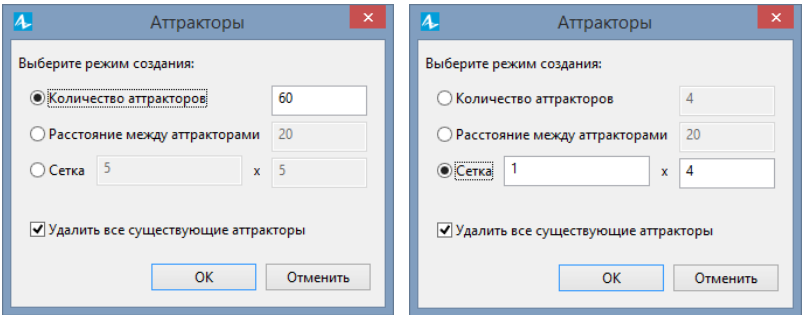


Рис. 5.12. Настройка области показа заявок в очереди и в процессоре

Результат показан на рис. 5.13.

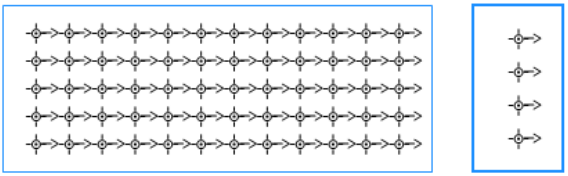


Рис. 5.13. Области показа заявок, находящихся в очереди и в процессоре

7. Добавим в модель элементы *queue* (очередь), *hold* (захват), *delay* (задержка), *sink* (сток). Далее последовательно соединим их. Ко входу очереди подключим источники заданий (рис. 5.14).

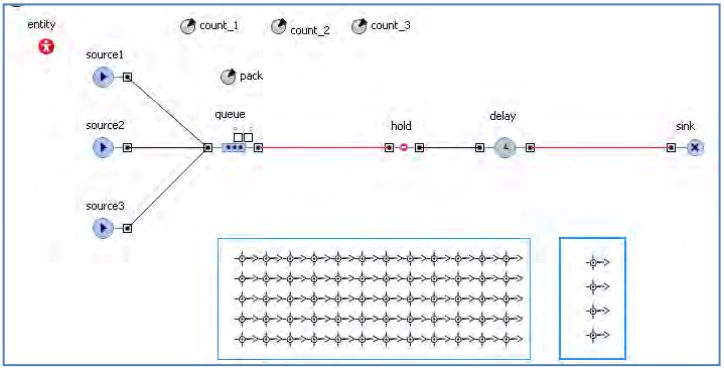


Рис. 5.14. Объединение элементов модели

8. Настроим свойства очереди. В качестве места заявок укажем элемент *node*. Попадая в очередь, заявки должны выстраиваться в соответствии с приоритетом. В свойстве *Очередь* выберем *По приоритету*. При этом необходимо явно указать место, в котором хранится значение данного свойства, *entity.priority*. При выходе заявки из очереди они будут подсчитываться. Когда сформируется пакет из четырех заданий, элемент *hold* приостановит их поступление к процессору (рис. 5.15).

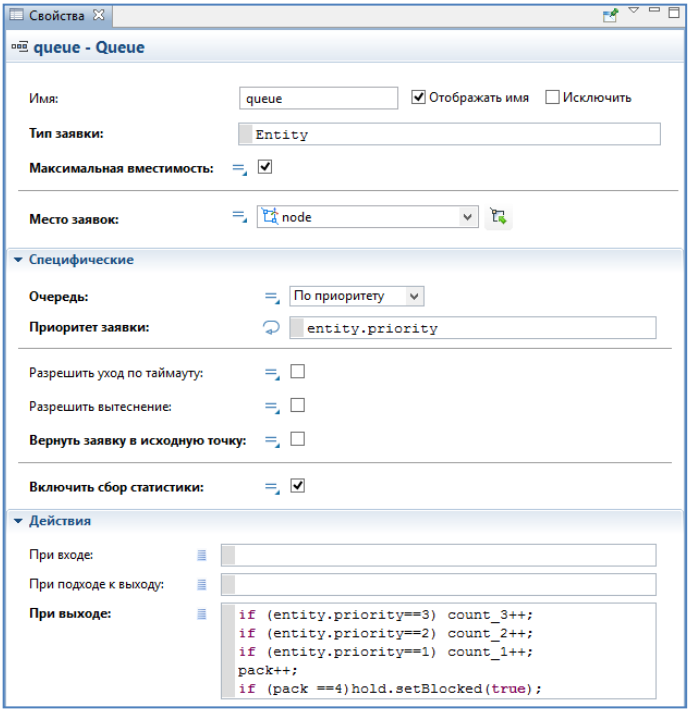


Рис. 5.15. Свойства очереди

8. Настроим работу процессора (рис. 5.16).

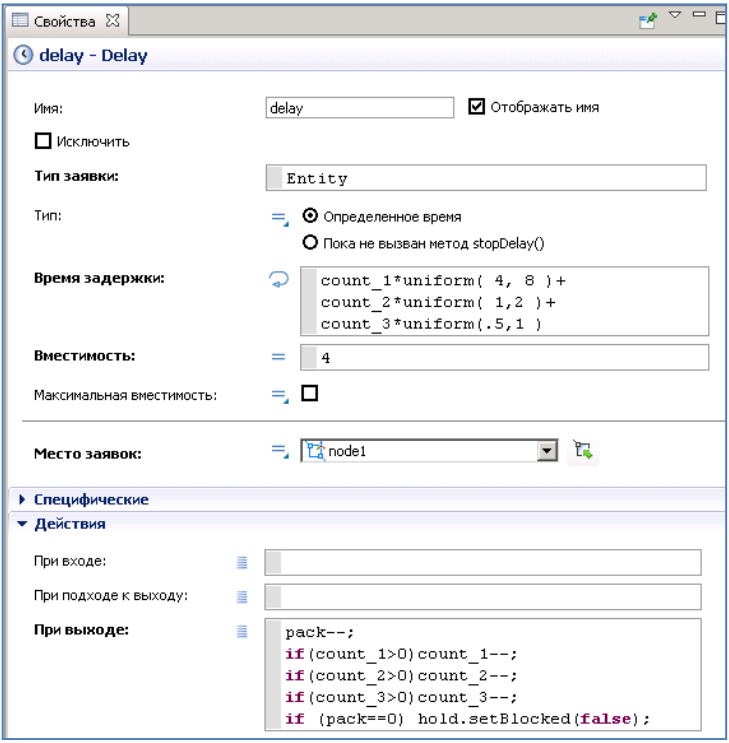


Рис. 5.16. Свойства процессора

Полное время обработки пакета заданий будет рассчитываться, исходя из количества попавших в него заявок с соответствующим приоритетом. По мере выхода заданий из блока *delay* счетчик соответствующей заявки, попавшей на обработку, будет уменьшаться, пока не станет равным нулю. После обработки всего пакета параметр *pack* станет равен нулю и элемент *hold* будет разблокирован. Затем заявки снова начнут поступать в процессор из очереди до тех пор, пока не будет сформирован новый пакет из четырех заданий.

Запустим модель (рис. 5.17).

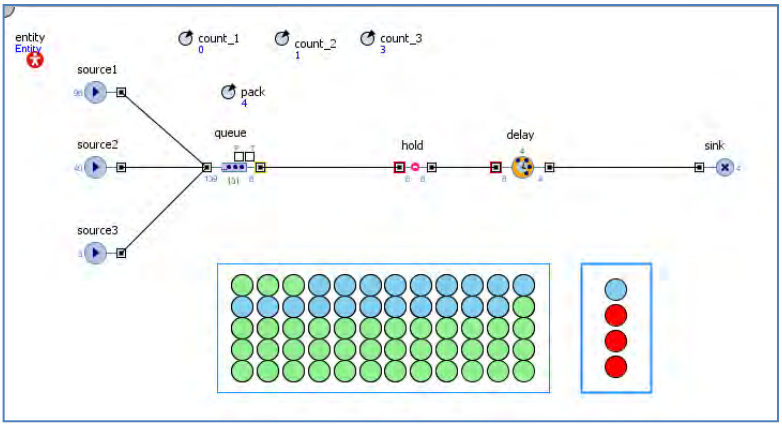


Рис. 5.17. Имитационная модель выполнения заданий процессором

На работающей модели видно, как изменяется размер очереди. Заявки сортируются и подсчитываются по признаку приоритета, затем обслуживаются процессором и покидают его в полном соответствии с заданными значениями параметров.

10. Для анализа динамических свойств модели организуем сбор данных о работе очереди и процессора. Для этого в свойстве элемента *delay Специфические – Включить сбор статистики* поставим галочку.

Для определения времени, потраченного процессором на выполнение задания, понадобятся блоки *TimeMeasureStart* и *TimeMeasureEnd.* Вставим эти блоки перед *hold* и после *delay* (рис. 5.18).

Рис. 5.18. Положение блоков TimeMeasureStart и TimeMeasureEnd для определения времени выполнения задания

В свойствах *tmEnd* укажем стартовый блок, определяющий место в модели, с которого начнутся измерения. В данном случае – это единственный элемент *tmStart.*

> Измерения времени можно начинать в одном или нескольких местах модели и заканчивать в разных ветвях процессов. Каждому блоку TimeMeasureEnd нужно указать в его свойствах, с какого именно TimeMeasureStart производить эти измерения.

Ниже элемента *node1* поместим в модель *Гистограмму* из палитры *Статистика.*

В свойствах гистограммы укажем, что на ней будут отображаться две текущие характеристики распределения (*tmEnd.distribution*) времени обработки пакета – плотность вероятности и среднее значение (рис. 5.19).

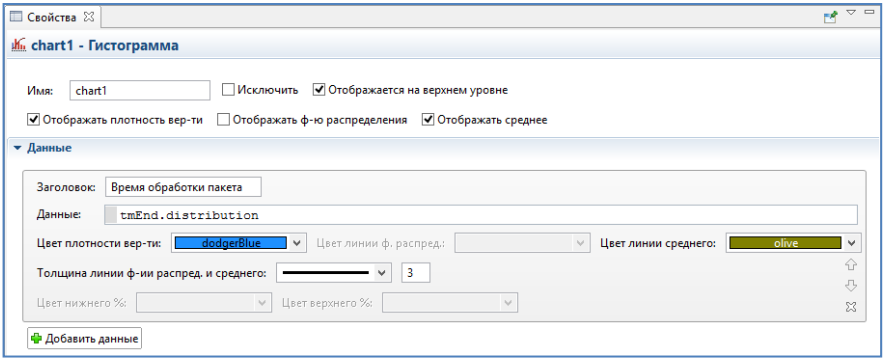


Рис. 5.19. Свойства гистограммы распределения времени обработки пакета

Правее элемента *node1* поместим *Столбиковую диаграмму.*

В окне свойств диаграммы отредактируем свойство *Данные* (рис. 5.20).

Данный элемент будет показывать текущую среднюю загруженность процессора.

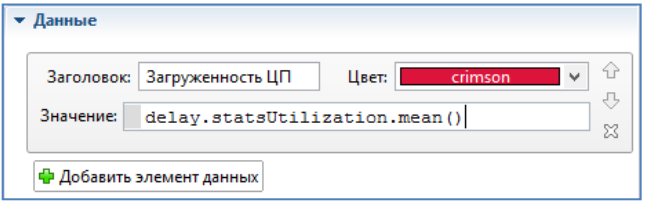


Рис. 5.20. Настройка диаграммы занятости процессора

Для сбора информации о работе очереди перетащим в модель элемент *Статистика.* Переименуем его в *stQueue.* Настроим свойства (рис. 5.21).

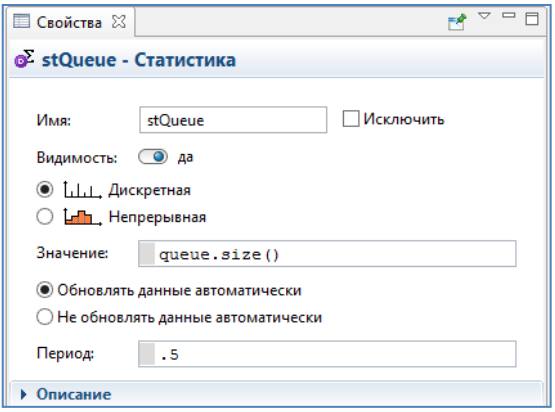


Рис. 5.21. Свойства элемента сбора статистики

В поле *Значение* введем *queue.size().* Это позволит отслеживать статистические характеристики размера очереди.

Внесем в модель еще одну столбиковую диаграмму и разметим ее под очередью. В *Свойствах* диаграммы добавим один элемент данных (рис. 5.22).

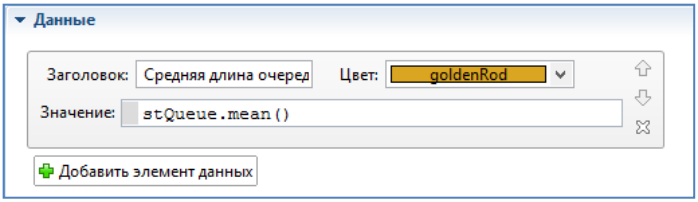


Рис. 5.22. Настройка свойств диаграммы очереди

Диаграмма будет отображать один статистический показатель – среднюю на данный момент времени длину очереди. Направление столбца установим горизонтальным.

Запустим модель и проанализируем работу системы (рис. 5.23).

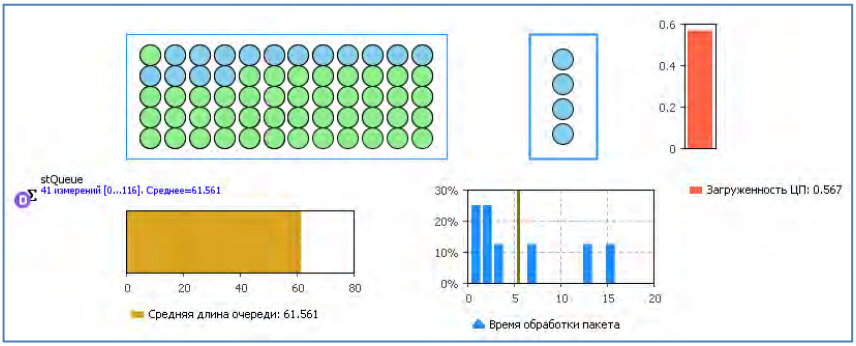


Рис. 5.23. Фрагмент эксперимента с показателями работы системы

5.3. Моделирование разгрузки кораблей в порту

Суда прибывают в гавань и время между прибытиями, представленное независимыми одинаково экспоненциально распределенными случайными величинами со средним значением, равно 1,25 дня. В гавани имеется два дока с якорными стоянками и двумя кранами для разгрузки судов.

Корабли, прибывшие тогда, когда обе якорные стоянки заняты, становятся в очередь с дисциплиной обслуживания FIFO. Время, необходимое одному крану для разгрузки судна, равномерно распределено между 0,5 и 1,5 дня.

Если в гавани всего одно судно, разгрузкой занимаются оба крана, и время разгрузки уменьшается вдвое. Если в гавани два судна, то каждый из двух кранов работает с каждым судном. Если оба крана разгружают одно судно, то по прибытии второго судна один из кранов немедленно начинает его обслуживание, а оставшееся время обслуживания первого судна увеличивается вдвое.

Для модели разгрузки выполним следующие действия:

1. Сначала настроим модель на показ в режиме 3D-симуляции.

В качестве фоновой сцены используем приморский ландшафт *landscape.x3d* из учебной модели AnyLogic Air Defence System.

Поместим его в основание координат и увеличим масштаб до 125%. После этого вызовем контекстное меню, выберем пункты *Блокировка–Блокировать фигуру.* Это нужно сделать, потому что на ней будем размещать и настраивать другие графические элементы модели.

«Построим порт». Для этого воспользуемся коллекцией трехмерных фигур из библиотеки *3D-Объекты.*

Поместим ниже картинки *камеру* и настроим ее свойства:

- имя – *camera*;
- поворот *X* – +45°;
- поворот *Z* – –90°;
- расположение: *X* – 620; *Y* – 800; *Z* – 400.

2. Разместим на ландшафте графические элементы, которые позже свяжем с функциональными блоками модели (рис. 5.24).

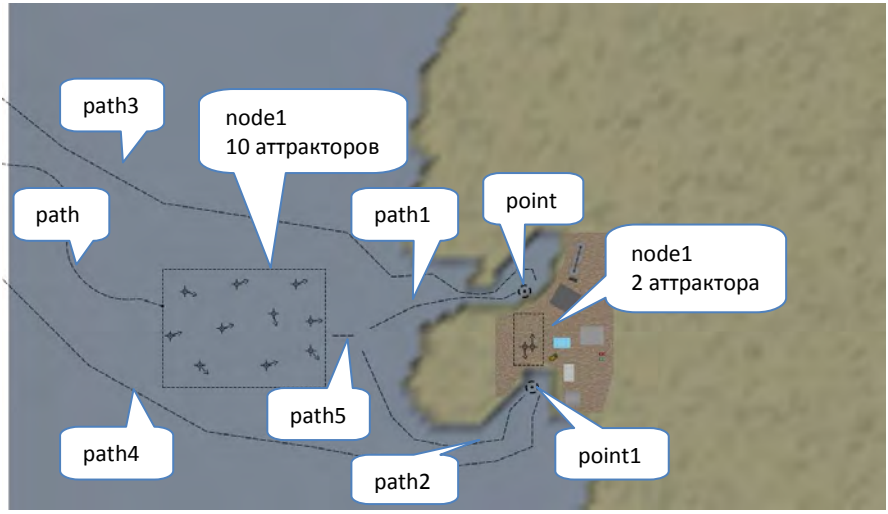


Рис. 5.24. Элементы презентации модели

3. Создадим свой тип заявки: *MyBoat.* Откроем ее в графическом редакторе.

Поместим в начало координат заявки два корабля из библиотеки *3D-Объекты: container_ship_loaded.x3d* и *container_ship_empty.x3d.* Их имена оставим прежними: *ship_loaded* и *ship_empty.* Изменим масштаб фигурок до 25%.

Чтобы корабль приходил в порт загруженным, а после разгрузки уходил пустым, в свойствах MyBoat *Действия агента – При запуске* запишем:

```
ship_empty.setVisible(false);
```

а при занятии ресурса

```
ship_loaded.setVisible(false);
ship_empty.setVisible(true);
```

Добавим камеру. Изменим ее имя на *camOnBoat.*

Настроим другие свойства камеры:

- поворот X – +10°;
- поворот Z – –25°;
- расположение – X – –70; Y – 50; Z – 40.

С ее помощью будем периодически смотреть на процессы «изнутри». При таком положении камеры можно видеть корабль, с которого ведется наблюдение.

Для управления камерой создадим диаграмму состояний (стейтчарт).

Откроем палитру *Диаграмма состояний* и соберем ее. Настроим свойства элементов (состояний и переходов) стейтчарта (рис. 5.25).

> ***Стейтчарты*** – графический язык диаграмм, мощное средство AnyLogic. Диаграмма включает (обязательно) указатель начального состояния и переходы от одного состояния к другому. Стейтчарты широко используются в моделировании систем управления и в определении поведения объектов в UML (Universal Modeling Language).

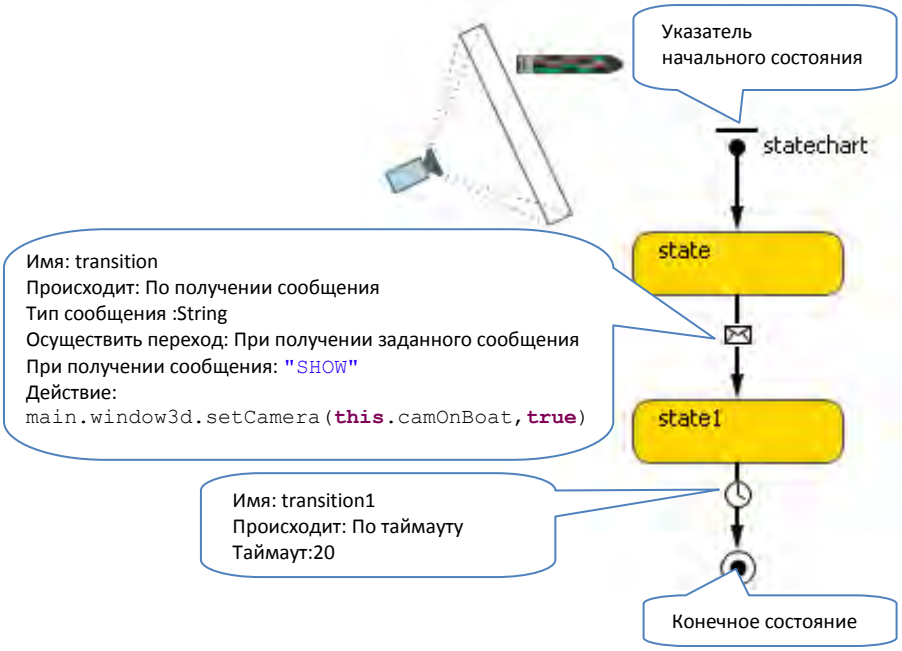


Рис. 5.25. Логика управления камерой на корабле

Переключение на камеру корабля происходит, когда он получает сообщение «*SHOW*», которое посылается ему из главной модели (агента верхнего уровня *Main*).

4. Создадим тип ресурса *MyCrane*. Поместим в него фигурку портового крана. Установим масштаб 50%.

5. Поместим заявку и ресурс в *Main*.

6. Создадим логику модели (рис. 5.26).

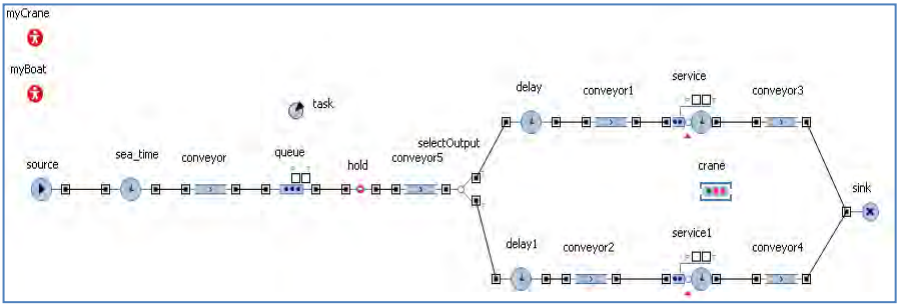


Рис. 5.26. Функциональная модель порта

В свойстве *Новая заявка* источника заявок *(source)* выберем *myBoat*. Заявки будут прибывать согласно *Времени между прибытиями*, распределенному по экспоненциальному закону – *exponential(.5)*.

Пусть корабли будут идти в порт от 2 до 7 дней. Для блока *sea_time* (тип ***Delay***) установим *Время задержки triangular(2, 7, 4)* (согласно треугольному распределению).

Первый конвейер (*conveyor*) будет моделировать проход кораблей к месту стоянки. Настроим его свойства (рис. 5.27). В качестве места заявок укажем линию *path* на презентации (см. рис. 5.24).

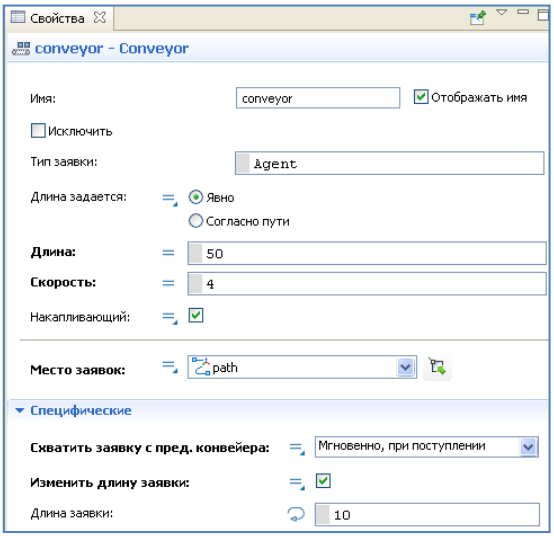


Рис. 5.27. Свойства пути на стоянку кораблей

Настроим работу очереди (*queue*). Она будет накапливать корабли на рейде и пропускать их к докам. Внесем в модель параметр *task*, в котором будет сохраняться количество заявок (максимум две), пропущенных на обслуживание.

В свойствах очереди для места заявок: укажем *node1*. Дисциплина очереди: *FIFO*.

Действия при выходе:

```
task++; if(task==2)hold.setBlocked(true);
```

Далее идет короткий конвейер (*conveyor5*, место заявок – *path5*), имитирующий подход кораблей, идущих на разгрузку к берегу. Длину пути для конвейера сделаем равной 20. Скорость корабля оставим прежней.

Селектор (*selectOutput*) будет проверять, какое направление на облуживание свободно и таким образом регулировать его выбор кораблем. При этом нужно «обследовать» на наличие корабля все узлы, где он может оказаться: на конвейере и на подходе к нему, в сервисе и на походе к нему (очереди сервиса). В свойстве *Условие* (для выбора *true*) запишем:

```
conveyor1.size()==0 & service.size()==0 &
service.queueSize()==0 & delay.size()==0.
```

Далее корабли попадают в параллельные процессы со сходными значениями параметров задержек и конвейеров. В качестве мест размещения кораблей, идущих к докам, для всех элементов укажем *path1* для левого пути и *path2* – для правого.

Необходимые компоненты работы – два портовых крана. Они являются *ресурсами* доков и хранятся в объекте модели *crane* (тип ***ResourcePool***) (см. рис. 5.26 и 5.28).

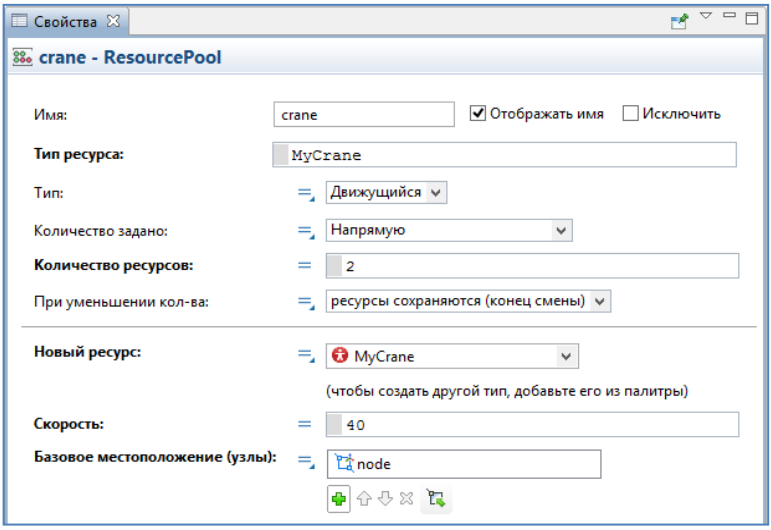


Рис. 5.28. Свойства портовых кранов

Чтобы сервис (док) мог работать с кранами, их нужно выбрать в качестве типа ресурсов, а также определить порядок использования (захвата) и высвобождения. По окончанию операции кран будет возвращаться в середину перешейка между заливами. На разгрузку одного корабля будет уходить от 12 часов до полутора дней (рис. 5.29).

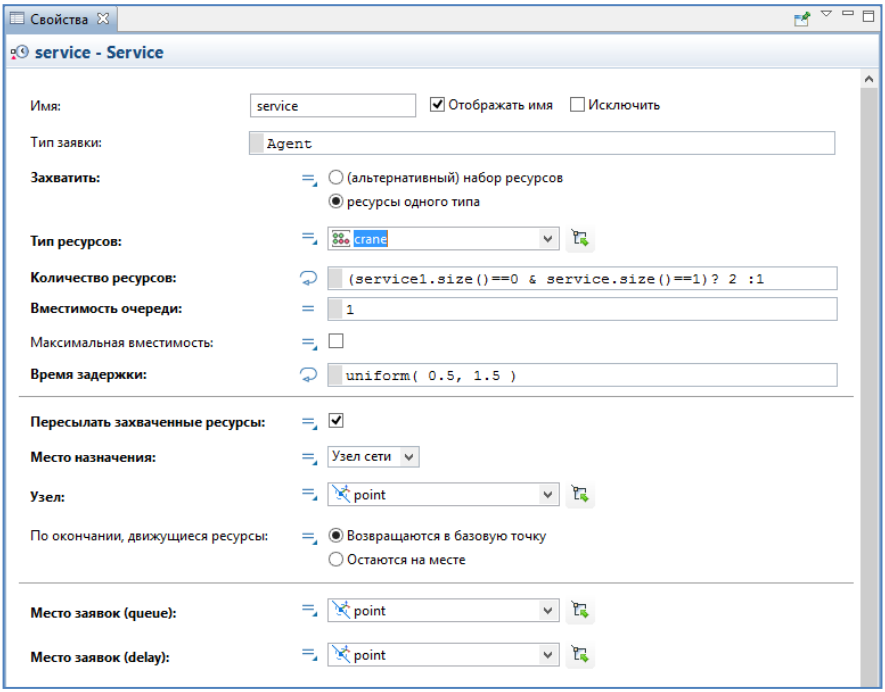


Рис. 5.29. Настройка разгрузки корабля в первом сервисе

В свойстве сервиса *Действия при выходе* запишем:

```
task--; if(task==0)hold.setBlocked(false);
```

Другой док (*service1*)настроим аналогично. В свойстве *Количество ресурсов* запишем выражение:

```
(service.size()==0 & service1.size()==1)? 2:1
```

Теперь работа доков настроена так, что если в них будет стоять один корабль, его будут разгружать оба кра-

на. Если будут заняты оба дока – один кран будет работать с одним кораблем.

После обслуживания блок *hold* разрешит пропуск к докам очередной пары кораблей с контейнерами.

Пустые корабли разворачиваются и покидают модель через конвейеры (*conveyor3* и *conveyor4*, длиной по 100 единиц каждый) по маршрутам *path3* и *path4.*

7. Заключительная стадия проекта. Создадим область *view2D* для просмотра элементов презентации («плоской модели»).

Поместим в модель *3D-Окно*. Имя (*window3d*) изменять не будем. Размеры окна установим такими же, как для *view2D*. Организуем для него область просмотра *view3D.*

Добавим в 3D-окно *Переключатель* из палитры *Элементы управления*. В *Свойствах* добавим новый элемент. Элементы переименуем (рис. 5.30).

Рис. 5.30. Вид переключателя после внесенных изменений

В поле *Действие* запишем код:

```
if( value == 0 ){window3d.setCamera( camera,true );}
if( value == 1 ){if (queue.getFirst() instanceof MyBoat)send("SHOW",
queue.getFirst());}
if( value == 2 ){view2D.navigateTo();}
```

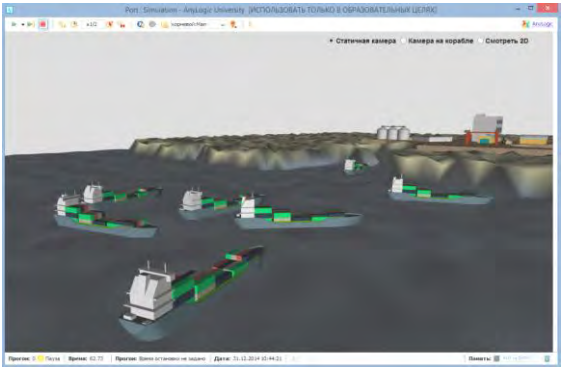
При первом выборе вид в модели будет переключаться на основную камеру. При втором – кораблю, стоящему в очереди первым (на рейде), будет посылаться сообщение «*SHOW*».

После этого стейтчарт агента *myBoat* установит вид с камеры корабля (см. рис. 5.25).

Вернемся в область презентации модели в 2D, поместим в нее текстовый элемент *Смотреть 3D.* В свойстве *Действие* по щелчку запишем: *view3D.navigateTo();.*

Запустим модель (рис. 5.31).

а)



б)

в)

Рис. 5.31. Симуляция модели порта в 2D и виды с камеры корабля на рейде и в доке

Вопросы и задания на моделирование

1. Как модели проходят проверку на адекватность? Что является ее главным критерием?

2. В чем суть простановки прямой и обратной задачи моделирования?

3. Какие признаки системы определяют выбор типовой математической схемы моделирования?

4. Назовите основные методы построения моделей.

5. Какие направления имитационного моделирования поддерживает AnyLogic?

6. Какие элементы будут отображаться в панели *Проекты* в процессе создания модели?

7. Как нужно сохранить проект, чтобы иметь возможность одновременно работать с двумя копиями модели?

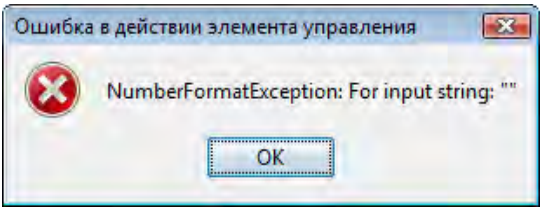
8. Почему в модели калькулятора для ввода числа *n* использована переменная строкового типа?

9. Почему для нахождения факториала использована переменная вещественного, а не целого типа?

10. При больших значениях *n,* как видно на рис. 2.21, формат числа (значение факториала), не соответствует национальному стандарту России. Как исправить эту проблему?

11. Сделайте так, чтобы значения *n* и *n!* последовательно отображались в одной области табло.

12. Если перед вычислением факториала нажимать кнопку «C», то будет появляться информационная панель с сообщением об ошибке:



При возобновлении правильной последовательности действий (сначала вводить *n*, потом нажать кнопку «*n!*»), ошибка исчезнет. Какие изменения нужно внести в модель, чтобы исправить эту проблему?

13. В модели калькулятора есть неработающая кнопка. Сделайте так, чтобы она выполняла свою функцию.

14. В модели маятника сделайте ограничение для левого положения груза.

15. В модели теплопроводности сделайте так, чтобы входная температура изменялась по периодическому закону.

16. В модели подвески скорость машины постоянна. Сделайте ее переменной. Создайте анимацию перемещения элементов подвески. Проведите оптимизационные эксперименты с варьированием других параметров. Массу кузова оставьте прежней. Увеличилась ли при этом плавность его хода? В примере использована неровность «бугры». Создайте модель неровности «бугры-ямы». Проведите эксперименты с моделью.

17. Как в модели AnyLogic можно изменить иерархию агентов? В модели «Жизнь» сделайте агентом верхнего уровня *MyAnt.* Поэкспериментируйте с моделью. Поместите в модель несколько муравьев. Сделайте так, чтобы можно было сохранять в файле отдельные фрагменты игрового поля. В модели клетка останавливает рост, когда достигает границы поля, а когда это происходит с муравьем – останавливается игра. Снимите это ограничение, представив, что поле не плоскость, а сфера. Сделайте трехмерные варианты модели «Жизнь».

18. Модифицируйте задачу нахождения определенного интеграла методом Монте-Карло так, чтобы можно выбирать функцию из списка и изменять значения интервалов.

19. Для каких типов событий в моделях СМО используются дискретные распределения вероятностей и для каких – непрерывные? Назовите критерии выбора характера распределения событий.

20. Проведите оптимизацию работы компьютера. В качестве целевого функционала примите максимальную загруженность процессора при минимальном времени обслуживания.

21. Представим, что в морском порту работает один буксир и три портовых крана. Буксир проводит корабли с

рейда к свободному доку. На его борту есть рация. Корабль, вставший первым в очередь на рейде, посылает сообщение диспетчеру порта о прибытии и об объеме груза. Порт принимает его, пересылает буксиру и портовым кранам. Буксир начинает обслуживать корабль после того, как выведет корабли из доков порта. Количество портовых кранов, направленных в док, зависит от объема груза на корабле, но их не может быть меньше одного на корабль. Если один док пустой, в занятом могут работать два крана. Третий кран дежурит в пустом доке. Постройте имитационную модель этой системы.

22. Добавьте в модель порта работы на территории. Краны переносят контейнеры с корабля на один тип грузовиков, которые увозят их на материк. Другой тип грузовиков подвозит в порт автомобили. Третий тип – животных (овец). Корабли на рейде посылают сообщение в порт о типе груза, который они будут забирать в порту после разгрузки. Корабли, забирающие овец, обслуживаются первыми, максимальным числом кранов и могут забрать максимум тысячу животных. Если в порт прибыли грузовики с овцами, и эти корабли стоят на рейде, они становятся первыми в очереди на проход к докам. Если корабли в порту разобрали всех подвезенных овец и оказались недогруженными, они могут забрать несколько автомобилей в количестве, эквивалентном пятидесяти овцам на одну машину.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Кельтон, Д. Имитационное моделирование. Классика CS [Текст] / Д. Кельтон, А. Лоу; [пер. с англ.]. – СПб.: Питер, 2004. – 847 с.

2. Карпов, Ю. Имитационное моделирование систем. Введение в моделирование с AnyLogic 5 [Текст] / Ю. Карпов. – СПб.: БХВ-Петербург, 2005. – 400 с.

3. Советов, Б.Я. Моделирование систем [Текст]: учебник для вузов / Б.Я. Советов, С.А. Яковлев. – М.: Высшая школа, 2007. – 343 с.

4. Рено, Н.Н. Численные методы [Текст]: учеб. пособие / Н.Н. Рено. – М.: КДУ, 2007. – 100 с.

5. Маликов, Р.Ф. Основы математического моделирования [Текст]: учеб. пособие / Р.Ф. Маликов. – М.: Горячая линия–Телеком, 2010. – 368 с.

6. Королев, А.Л. Компьютерное моделирование [Текст] / А.Л. Королев. – М.: Бином, 2010. – 230 с.

7. Тарасов, Л.В. Мир, построенный на вероятности [Текст] / Л.В. Тарасов. – М.: Просвещение, 1984. – 191 с.

8. Эткинс, П. Порядок и беспорядок в природе [Текст] / П. Эткинс; [пер. с англ.]. – М.: Мир, 1987. – 224 с.

9. Божокин, С.В. Фракталы и мультифракталы [Текст] / С.В. Божокин, Д.А. Паршин. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001. – 128 с.

10. Сьерра, К. Изучаем Java [Текст] / К. Сьерра, Б. Бэйтс; [пер. с англ.]. – М.: Эксмо, 2012. – 720 с.

11. Официальный сайт компании AnyLogic [Электронный ресурс]. – Режим доступа: www.anylogic.ru

12. Докинз, Р. Эгоистичный ген [Текст] / Р. Докинз; пер с англ. Н.О. Фоминой. – М.: АСТ, 2013. – 512 с.

ОГЛАВЛЕНИЕ

Введение.....3

1. ОСНОВНЫЕ ПОНЯТИЯ..... 4

2. ОСНОВЫ МОДЕЛИРОВАНИЯ В СРЕДЕ ANYLOGIC..... 13

2.1. Общие сведения..... 13

2.2. Элементы пользовательского интерфейса..... 14

2.3. Первый проект..... 16

3. ДЕТЕРМИНИРОВАННЫЕ НЕПРЕРЫВНЫЕ СИСТЕМЫ..... 31

3.1. Общие сведения..... 31

3.2. Моделирование систем с сосредоточенными параметрами..... 32

3.3. Дополнительные средства визуализации модели..... 47

3.4. Динамическое моделирование процесса управления..... 55

3.5. Модели с распределенными параметрами..... 63

3.6. Решение оптимизационной задачи 69

4. ДИСКРЕТНО-СОБЫТИЙНОЕ МОДЕЛИРОВАНИЕ..... 79

4.1. Агентные модели конечных автоматов 79

4.2. Моделирование систем методами статистических испытаний..... 93

5. МОДЕЛИРОВАНИЕ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ..... 104

5.1. Общие сведения..... 104

5.2. Моделирование процесса выполнения задач компьютером..... 110

5.3. Моделирование разгрузки кораблей в порту..... 121

Вопросы и задания на моделирование..... 130

Библиографический список..... 133

Учебное издание

Андрей Геннадьевич Куприяшкин

ОСНОВЫ МОДЕЛИРОВАНИЯ СИСТЕМ

Учебное пособие

Редакторы: Н.А. Прозур, Т.А. Овдиенко
Художественный редактор И.В. Беляева
Выпускающий редактор Л.П. Котенко

Темплан ФГБОУВПО «НИИ» 2015 г., поз. 26. Подписано в печать 30.01.2015.
Формат 60х84 1/16. Бум. для копир.-мн.ап. Гарнитура *Bookman Old Style*.
Печать плоская. Усл.п.л. 8,4. Уч.-изд.л. 8,4. Тираж 50 экз. Заказ 2.

Редакционно-издательский отдел ФГБОУВПО «НИИ»
663310, Норильск, ул. 50 лет Октября, 7. E-mail: RIO@norvuz.ru

Отпечатано с готового оригинал-макета в отделе ТСОиП ФГБОУВПО «НИИ»