Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра “Компьютерные технологии”

Б.М. Ярцев, А.А. Шалыто

Система программной эмуляции роботов *Lego Mindstorms* (Проект *Isenguardemu*)

Проектная документация

Проект создан в рамках

"Движения за открытую проектную документацию"

http://is.ifmo.ru/

Санкт-Петербург

2006

Оглавление

Введение

Проходя по базару, Банзан услыхал разговор с покупателем.
– Дай мне самый лучший кусок мяса, – сказал покупатель.
– В моей лавке все самое лучшее, – отвечал мясник. – Ты не найдешь здесь ни одного куска, который не был бы самым лучшим.
При этих словах Банзан стал просветленным.

"101 дзэнская притча, или Мясо и кости дзен"

Проект *Isenguardemu* является логическим продолжением проекта *Isenguard* [1]. Для начала вкратце напомним о том, что собой представляет проект *Isenguard*. Любой желающий ознакомиться с ним поближе, может всегда зайти в интернете по адресу http://is.ifmo.ru/projects/lego/ и скачать последнюю версию документации, программных кодов и схем сборки.

Итак, вернемся к истории и описанию проекта *Isenguard*. Основная работа над ним велась летом и осенью 2004 года. Впрочем, некоторые доработки, сделанные в частности при работе над проектом *Iseuguardemu*, делались и делаются практически постоянно. Летом 2004 года автор проходил практику в компании *Arcadia* (Санкт-Петербург). Целью практики было собрать что-нибудь из нескольких комплектов роботов *Lego Mindstorms*, затем написать программы для их микроконтроллеров, и, наконец, описать все в проектной документации. После нескольких не очень удачных попыток, один из авторов остановился на одной идее, описанной в книге [2]. В этой книге был представлен проект системы для доставки конфет *M&M's* из одного места в другое по полю, на котором был нарисован путь доставки, с помощью двух роботов – транспортного и поставщика. Этот автор взял данную идею за основу, но разработку конструкций роботов и написание программ выполнил самостоятельно. Наибольший интерес представлял собою процесс написания программ – после долгих и не очень успешных попыток сделать что-нибудь работоспособное, один из авторов решил прибегнуть к автоматному программированию [3 – 5]. Результатом явились две управляющие программы под операционную систему *lejos* [6] для роботов *Lego Mindstorms* – одна для транспортного робота, а другая для робота-поставщика. Отладка этих программ была выполнена за день.

В силу того, что время практики было ограничено, то через некоторое время авторы проекта оказались без комплекта *Lego Mindstorms* (на самом деле без двух комплектов, необходимых для проекта). Поэтому для сопровождения проекта и его усовершенствования необходимо было разработать эмулятор, для того, чтобы в дальнейшем (при покупке комплектов конструкторов) можно было загружать в роботы отработанные на эмуляторе управляющие программы.

Авторы приняли решение назвать программную среду эмуляции как *Isenguardemu* – сокращение от словосочетания "Isenguard Emulator".

В рамках проекта *Isenguardemu* разработан программный эмулятор процесса погрузки на языке *Java* (http://java.sun.com/). Также спроектирован и реализован пользовательский интерфейс для управления этим процессом. В этот интерфейс был встроен редактор, при помощи которого можно задавать параметры поля – путь, места для остановок и начальные позиции обоих роботов. Стоит отметить, что в программном эмуляторе, как графический интерфейс, так и процесс эмуляции были выполнены при помощи автоматного подхода.

1. Постановка задачи

Разработать программный эмулятор роботов из проекта *Isenguard* в соответствии с описанием, приведенным ниже.

Ввиду того, что управляющие программы в проекте *Isenguard* разрабатывались на языке *Java*, то и новый проект выполнен на языке *Java*.

Программный эмулятор должен включать в себя визуализатор процесса погрузки. Ввиду того, что весь этот процесс проходит в горизонтальной плоскости, желательно визуализировать процесс при помощи “вида сверху” – хорошо виден путь, по которому движется транспортный робот, а также легко можно показать процесс погрузки.

Для контроля за эмуляцией процесса погрузки следует ввести в интерфейс эмулятора соответствующие элементы управления – "Старт", "Пауза", "Перезапустить".

Для задания пути, мест остановки (фольга) и начальных положений роботов требуется спроектировать встроенный редактор.

Необходимо иметь возможность легко вносить изменения в управляющие программы транспортного робота и робота-поставщика. Для этого необходимо разработать удобный интерфейс, в рамках которого пользователь может написать свою собственную программу для доставки предметов.

В процессе эмуляции необходимо выводить на экран текущие состояния управляющих систем роботов, значения световых датчиков и датчиков касания, а также информацию выводимую на дисплеи обеих роботов.

Помимо эмуляции двух роботов необходимо эмулировать работу пульта дистанционного управления (ДУ).

Также стоит учесть тот факт, что инфракрасные сообщения, которыми обмениваются роботы друг с другом и с пультом ДУ, распространяются на ограниченное расстояние и в пределах определенного угла. При визуализации следует отображать "конус", в котором распространяется сообщение для того, чтобы пользователь, к примеру, мог проследить причину того, что один из роботов не получил это сообщение. При этом процесс визуализации будет нагляднее.

Стоит, по возможности, реализовать функцию увеличения/уменьшения масштаба визуализации и функцию "слежения" за транспортным роботом. При этом изображение всегда будет центрировано относительно транспортного робота, в том числе и при его движении.

Как отмечалось во введении, при проектировании эмулятора будет использоваться автоматный подход

2. Результат

После двух месяцев работы был разработан эмулятор транспортной системы, созданной в рамках проекта *Isenguard*. Этот эмулятор удовлетворял всем перечисленным в предыдущем разделе требованиям.

На основе эмулятора были протестированы управляющие системы роботов из проекта *Isenguard*. При этом авторы знали, что один из автоматов в проекте *Isenguard* спроектирован не по правилам автоматного программирования. Так как к этому времени комплектов *Lego Mindstorms* в нашем распоряжении уже не было, то для проверки внесенных изменений необходимо было разработать эмулятор. Он также будет полезен и при дальнейшем совершенствовании роботов в проекте *Isenguard*.

В настоящем проекте для эмулятора создан программный интерфейс, при помощи которого пользователь может писать свои собственные управляющие системы в рамках проекта *Isenguard*, а также изменять конфигурацию роботов. Но, к сожалению, при каждом изменении, вносимом в исходный код управляющих программ или же в конфигурации роботов, приходится перекомпилировать весь проект. Возможность динамической загрузки управляющих программ и конфигураций роботов является одной из приоритетных задач для будущего развития проекта *Isenguardemu*.

Практически все компоненты эмулятора разработаны при использовании автоматного подхода. В данном проекте использовались два типа автоматов: активные и пассивные.

Пассивные автоматы – это такие автоматы, которые вызываются постоянно в цикле. Они использовались для эмуляции физических процессов – движения роботов, поворотов штанги, передачи инфракрасных сообщений.

Активные автоматы – это такие автоматы, которые вызываются только по мере необходимости. Они также нашли широкое применение в проекте. К примеру, если произошло какое-то событие, например, пользователь направил мышь в определенную зону, или был нажат датчик касания, то вызывается соответствующий активный автомат. При помощи активных автоматов был реализован пользовательский интерфейс и логическая "начинка" различных компонентов роботов – датчиков, передатчиков и приемников инфракрасных сообщений, пульта дистанционного управления.

В итоге разработан работоспособный эмулятор, при помощи которого была проэмулирована и протестирована система доставки предметов, полученная в результате рефакторинга системы управления транспортным роботом из проекта *Isenguard*. Этот эмулятор будет использован в дальнейшем при модификации системы доставки.

3. Общее описание эмулятора

3.1. Внешний вид

Внешний вид эмулятора представляет с окно программы, разделенное на две части: область, где визуализируется процесс доставки предметов роботами по полю, и панель управления. У эмулятора есть два базовых режима работы – непосредственно эмуляция и режим редактирования параметров поля и положения роботов. Панель визуализации в обеих режимах одна и та же, но панели управления разные. На рис. 1 представлен снимок программы в режиме эмуляции, а на рис. 2 – в режиме редактирования.

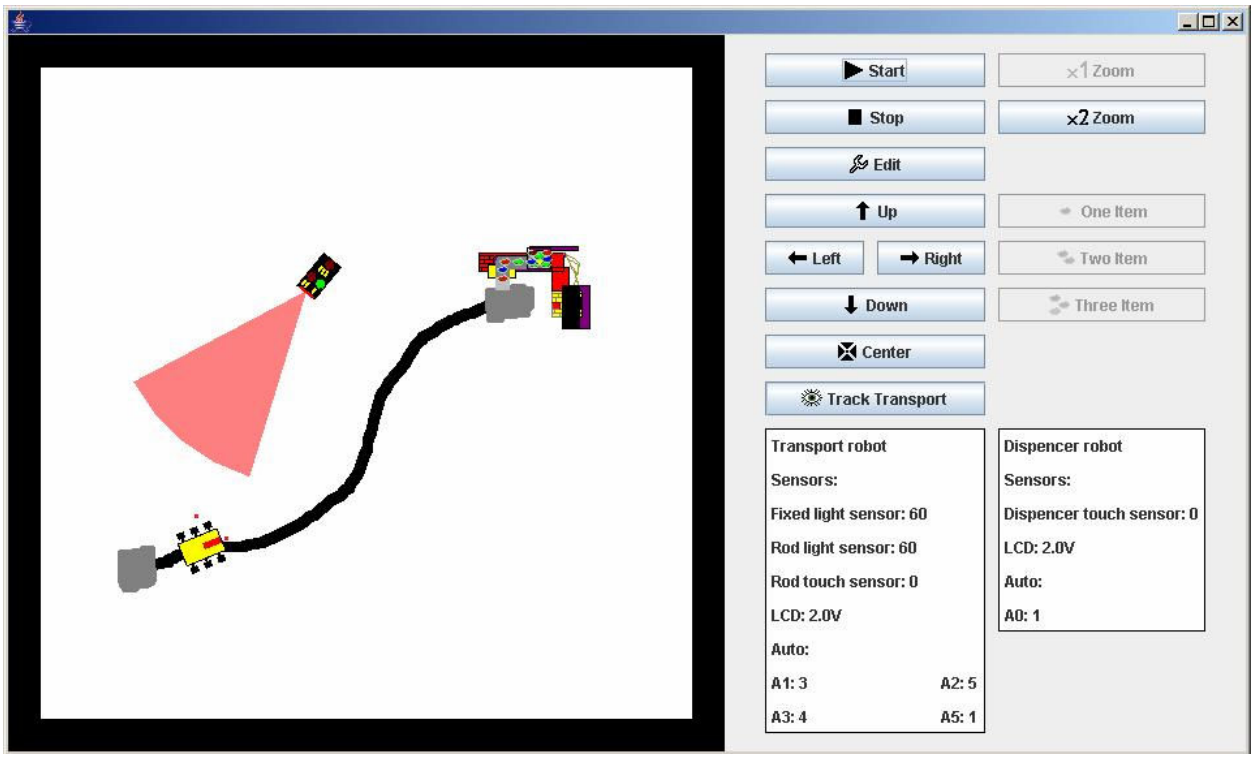

Рис. 1. Снимок программы в режиме эмуляции

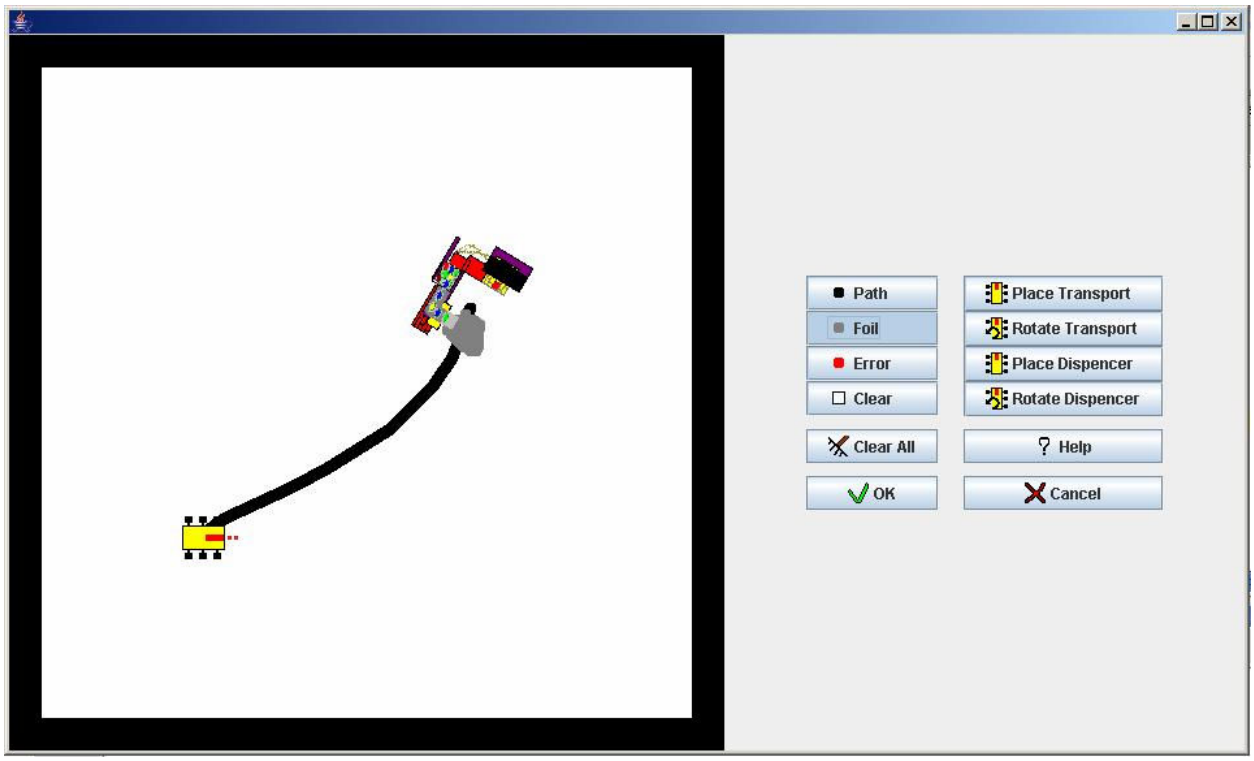

Рис. 2. Снимок программы в режиме редактирования

3.2. Область визуализации

3.2.1. Область визуализации в режиме эмуляции

Область визуализации находится в левой части окна программы эмулятора. В зависимости от выбранного размера окна программы и коэффициента увеличения она либо представляет собой белый квадрат (поле, на котором происходит доставка) с черной окантовкой, либо полностью белую область. Черная окантовка обозначает границу поля. Если в режиме эмуляции робот попадет за эту границу, то его световые датчики сообщат ему, что он заехал на неопределенное место (место ошибки) и должен предпринять соответствующие действия.

На поле расположены два робота – транспортный робот (рис. 3) и робот-поставщик (рис. 4). Две красные точки перед транспортным роботом соответствуют его световым датчикам. Точка, отстоящая дальше от робота, соответствует датчику, закрепленному на штанге, а отстоящая ближе – датчику на корпусе робота.

На конвейере робота-поставщика изображены разноцветные предметы. При выдаче они начинают двигаться по конвейеру и затем грузятся на транспортный робот. На рис. 5 показан транспортный робот с двумя погруженными на него предметами.

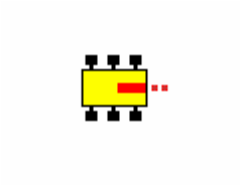

Рис. 3. Транспортный робот

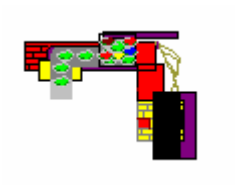

Рис. 4. Робот-поставщик

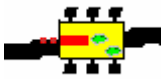

Рис. 5. Погружено два предмета

С помощью пульта дистанционного управления при эмуляции можно посылать сообщения транспортному роботу. Положение пульта на области визуализации совпадает с положением указателя мыши. Всего пульт может посылать три типа сообщений (как и в проекте *Isenguard*).Выбор типа сообщения осуществляется с панели управления. Для того, чтобы начать посылать сообщение с пульта, достаточно щелкнуть левой кнопкой мыши на области визуализации при запущенной эмуляции. Пульт начинает посылать сообщение. Повторное нажатие на левую кнопку мыши прекращает посылку сообщения. При помощи нажатой правой кнопки мыши пользователь может поворачивать пульт влево и вправо.

В зависимости от типа выбранного сообщения и текущего состояния пульта (посылает / не посылает) изображение пульта изменяется. Кнопка на изображении, соответствующая выбранному сообщению, выделяется ярко-красным цветом, а при посылке сообщения – зеленым. На рис. 6 изображен пульт с выбранным сообщением “2”, на рис. 7 – идет посылка сообщения “2”, а на рис. 8 – посылка сообщения “3”.

Рис. 6. Выбрано сообщение “2”

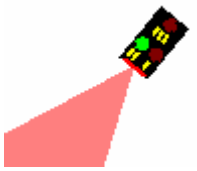

Рис. 7. Посылка сообщения “2”

Рис. 8. Посылка сообщения “3”

Помимо роботов и пульта на поле могут присутствовать путь, места остановки (фольга в проекте *Isenguard*) и место ошибки. Путь изображен на рис. 9. Место остановки с примыкающим к нему путем изображено на рис. 10. Место ошибки, пересекающее путь – на рис. 11.

Рис. 9. Путь

Рис. 10. Место остановки

Рис. 11. Место ошибки

Изменять начальные позиции роботов, путь, места остановок и ошибок можно в режиме редактирования.

Также был визуализирован “конус” передачи инфракрасного сообщения. Это делает процесс эмуляции более наглядным. На рис. 12 демонстрируется передача сообщения об окончании погрузки от робота-поставщика транспортному роботу, а на рис. 13 – передача сообщения от пульта к транспортному роботу.

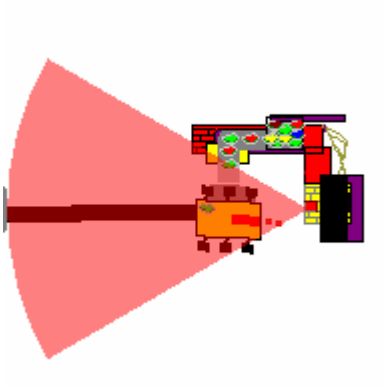

Рис. 12. Передача сообщения от поставщика

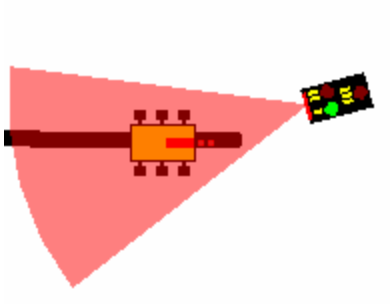

Рис. 13. Передача сообщения от пульта

Пользователь может включить на панели управления режим двукратного увеличения. В этом случае поле и все изображения на нем увеличиваются в два раза. На рис. 14 изображены роботы при двукратном увеличении.

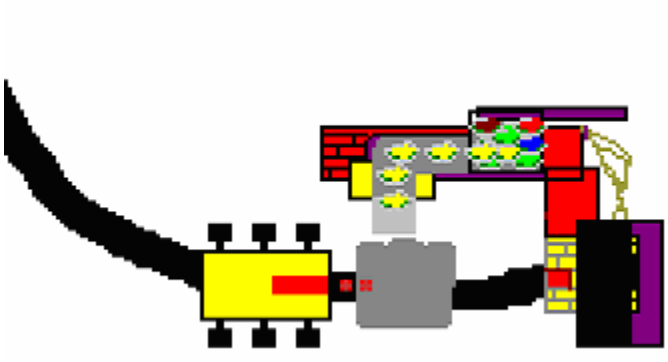

Рис. 14. Двукратное увеличение

3.2.2. Область визуализации в режиме редактирования

В режиме редактирования область визуализации находится без увеличения. В этом режиме появляются четыре новых указателя для мыши, при помощи которых можно

изображать путь, места остановки и ошибки, а также стирать уже нарисованные объекты. Подробнее о том, как производится рисование, описано в разд. 3.4. В настоящем разделе продемонстрируем новые указатели для мыши. На рис. 15 изображен указатель мыши для рисования пути и участок уже нарисованного пути. На рис. 16 – указатель для рисования места остановки, на рис. 17 – места ошибки, а на рис. 18 – указатель мыши для стирания.

Больше различий в визуализации между режимом редактирования и режимом эмуляции нет.

Рис. 15. Рисование пути

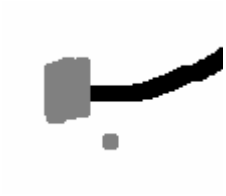

Рис. 16. Рисование места остановки

Рис. 17. Рисование места ошибки

Рис. 18. Стирание

3.3. Панель управления эмуляцией

Панель управления эмуляцией находится в правой части окна программы-эмулятора. В ней можно выделить две области – область управляющих кнопок и область, в которой отображается информация о транспортном роботе и роботе-поставщике: состояния их управляющих систем, показания датчиков и информация, выводимая на LCD-дисплей (информационная область).

Рассмотрим сначала область управляющих кнопок. На рис. 19 эта область показана при остановленной эмуляции, а на рис. 20 – для случая, когда она запущена.

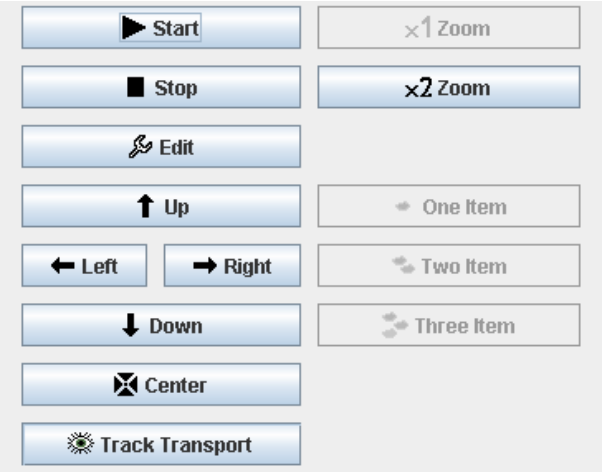

Рис. 19. Эмуляция остановлена

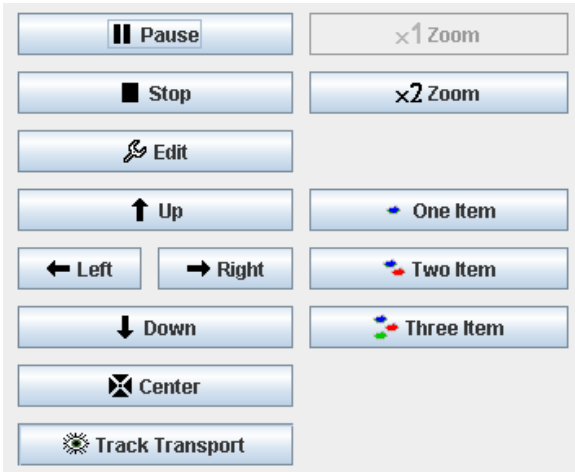

Рис. 20. Эмуляция запущена

Опишем кнопки сверху вниз с левого столбца. Первой идет кнопка "Старт"\"Пауза" ("Start"\"Pause"). Она отвечает за запуск и приостановку процесса эмуляции. При нажатии на нее запускается эмуляция, у кнопки изменяется название на "Пауза" ("Pause"). Повторное нажатие на эту кнопку меняет ее название на "Старт" ("Start") и приостанавливает эмуляцию. Если еще раз щелкнуть по этой кнопке, то эмуляция возобновится с этого же места, и.т.д.

Кнопка "Стоп" ("Stop") предназначена для полной остановки эмуляции и возврата роботов на начальные позиции (определенные в результате редактирования). Нажатие на эту кнопку при остановленной эмуляции не оказывает никакого действия.

Кнопка "Редактировать" ("Edit") переводит эмулятор в режим редактирования. Если эмуляция была запущена, то сначала эмуляция будет полностью остановлена, а затем уже будет произведен переход в режим редактирования.

Кнопки "Влево" ("Left"), "Вправо" ("Right"), "Вниз" ("Down") и "Вверх" ("Up") работают в случае, если поле превышает размеры окна программы (это обычно происходит, когда пользователь включает режим двукратного увеличения, о котором будет рассказано). При помощи этих кнопок можно осуществлять навигацию по полю – сдвигать видимую область влево, вправо, вверх и вниз.

Кнопка "Центр" ("Center") центрирует видимую область, если она была сдвинута при помощи предыдущих четырех кнопок.

Кнопка слежения за транспортом ("Track Transport") включает режим "слежения" за транспортным роботом. В этом режиме область визуализации всегда центрирована относительно транспортного робота. При этом транспортный робот постоянно находится по центру этой области, независимо от того стоит он или движется. При этом становятся неактивными кнопки "Влево" ("Left"), "Вправо" ("Right"), "Вниз" ("Down"), "Вверх" ("Up") и "Центр" ("Center"). Повторное нажатие на кнопку выключает этот режим и возвращает старое значение сдвига области визуализации.

Перейдем ко второму столбцу кнопок. Здесь располагается кнопка однократного увеличения ("1x Zoom") и кнопка двукратного увеличения ("2x Zoom"). С их помощью можно уменьшать и увеличивать изображение поля в области визуализации.

Следующие три кнопки активны только при запущенной визуализации. Это кнопки выбора количества предметов для доставки ("One Item", "Two Item" и "Three Item"). Они соответствуют типам сообщений, которые могут быть посланы с пульта транспортному роботу.

Рассмотрим теперь информационную область (рис. 21). Она состоит из двух окон: в одно выводятся данные о транспортном роботе (левое окно), а в другое данные о роботе-поставщике (правое окно).

Для транспортного робота выводятся следующие данные (сверху вниз): его название, показания светового датчика на корпусе, показания светового датчика на штанге, показания датчика касания и информация с *LCD*-дисплея. После этого следуют состояния автоматов управляющей системы (для управляющей системы из проекта *Isenguard* – это автоматы *A1*, *A2*, *A3*, *A5*). Таким образом, пользователь в любой момент времени в течение эмуляции может узнать текущее состояние каждого из автоматов и использовать данную информацию при отладке и разработке управляющей системы.

Для робота-поставщика выводится несколько меньший объем данных: его название, показания датчика касания и информация с *LCD*-дисплея. После этого следуют состояния автоматов управляющей системы (для управляющей системы из проекта *Isenguard* только одного автомата *A0*).

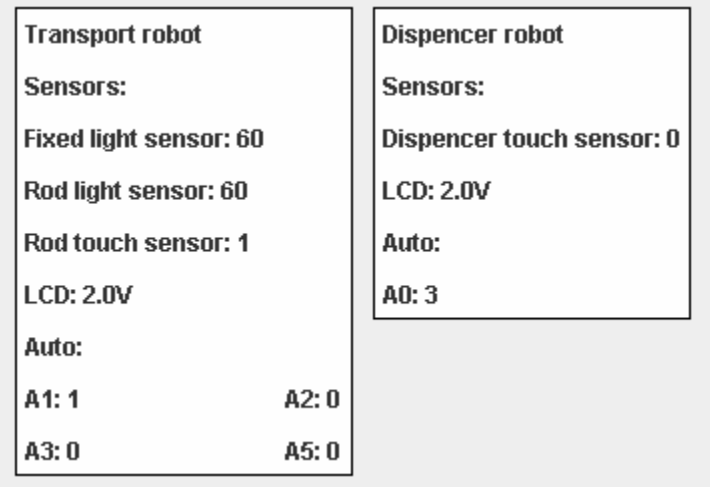

Рис. 21. Информационная область

3.4. Панель редактирования

Панель редактирования (рис. 22) заменяет панель управления эмуляцией при переходе в режим редактирования. Она состоит из нескольких кнопок, при помощи которых можно изменить начальное положение и направление роботов, а также нарисовать путь, места остановки и ошибки, либо стереть уже нарисованные объекты на поле.

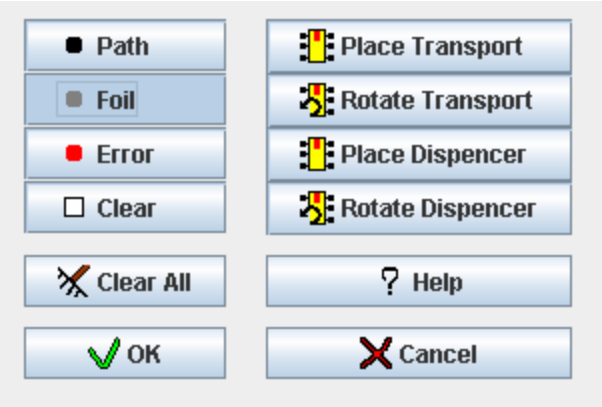

Рис. 22. Панель редактирования

Рассмотрим назначение кнопок на панели редактирования. Кнопка "Рисовать путь" ("Path") выбирается по умолчанию при переходе в режим редактирования. После того, как эта кнопка выбрана, пользователь может перевести указатель мыши в область визуализации и начать рисовать путь точно также как при использовании инструмента типа "Карандаш" или "Кисть" в популярных графических программах типа *Gimp* или *Paint*.

Функциональность кнопок "Рисовать места остановки" ("Foil") и "Рисовать места ошибки" ("Error") аналогична функциональности предыдущей кнопки, за тем исключением, что при помощи этих кнопок можно рисовать места остановки и ошибки.

Кнопка "Стереть" ("Clear") позволяет стирать нарисованные на поле объекты таким же способом, как это делает инструмент "Ластик" в графических программах.

При нажатии на кнопку "Стереть все" ("Clear All") с поля удаляются все нарисованные до этого объекты.

Кнопка "OK" сохраняет все изменения, произведенные в результате редактирования, заканчивает редактирование и переводит программу в режим эмуляции.

Кнопка "Cancel", восстанавливает параметры поля и начальные позиции роботов на те, которые были до редактирования, заканчивает редактирование и также переводит программу в режим эмуляции.

Кнопка "Установить транспорт" ("Place Transport") позволяет изменять начальную позицию транспортного робота. После нажатия на эту кнопку пользователь может указателем мыши переместить транспортный робот на новое место, а затем щелчком левой кнопкой мыши закрепить его на этой позиции.

Кнопка "Вращать транспорт" ("Rotate Transport") позволяет изменить начальное направление транспортного робота. После нажатия на эту кнопку пользователь может указателем мыши определить новое направление, а затем щелчком левой кнопкой мыши закрепить это новое направление.

Функциональность кнопок "Установить поставщик" ("Place Dispencer") и "Вращать поставщик" ("Rotate Dispencer") аналогична функциональности кнопок "Установить транспорт" и "Вращать транспорт", только изменения позиции и направления производятся для робота-поставщика.

Кнопка "Справка" открывает диалог с рекомендациями по поводу рисования пути (рис. 23).

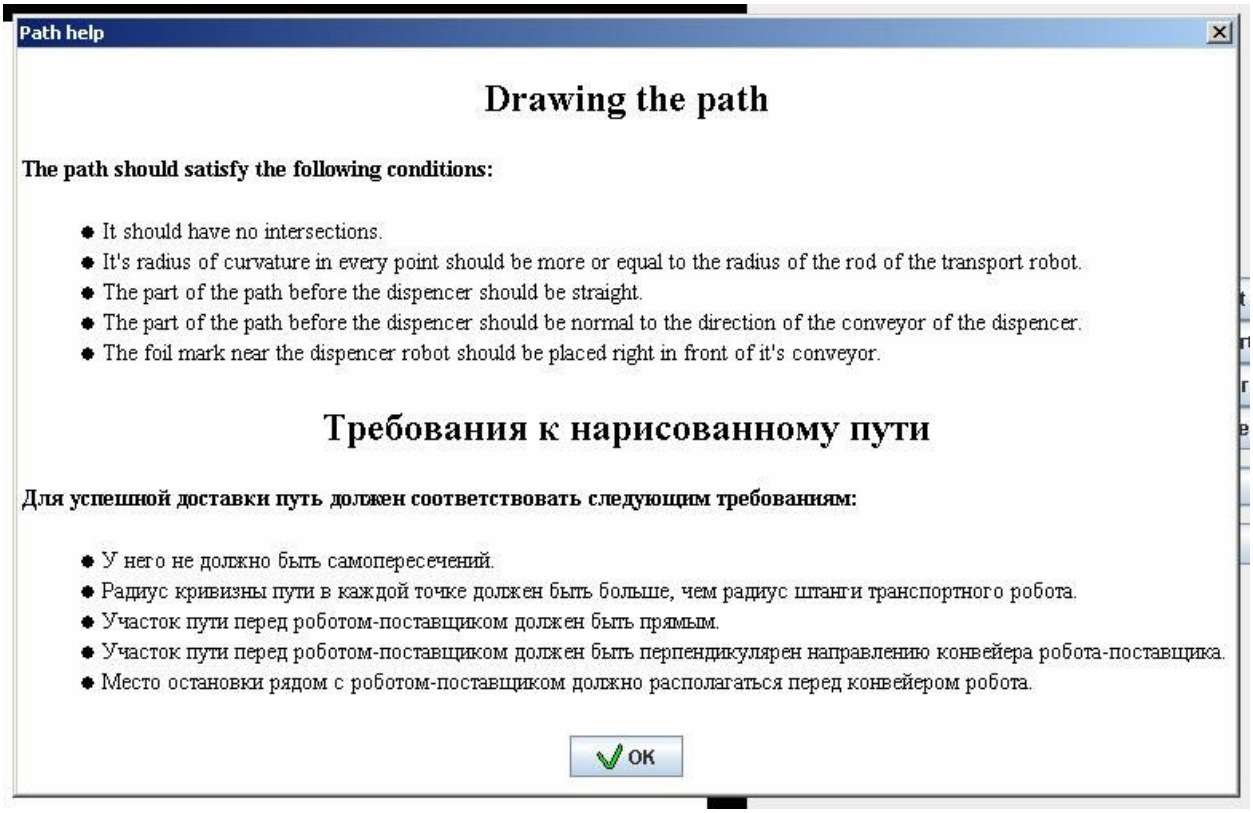

Рис. 23. Диалог с рекомендациями по поводу рисования пути

3.5. Работа с эмулятором

В данном разделе будет рассмотрена работа с эмулятором. После запуска программы пользователь должен задать параметры поля, по которому будет производиться доставка. Для этого он переходит в режим редактирования при помощи кнопки "Редактировать" ("Edit"). В режиме редактирования, прежде всего надо нарисовать путь. Для этого выбираем режим рисования пути при помощи кнопки "Рисовать путь" ("Path").

Изображаемый путь должен соответствовать условиям из проекта *Isenguard*:

- у него не должно быть самопересечений;
- радиус кривизны пути в каждой точке должен быть больше, чем радиус штанги транспортного робота;
- участок пути перед роботом-поставщиком должен быть прямым;
- участок пути перед роботом-поставщиком должен быть перпендикулярен направлению конвейера робота-поставщика;
- место остановки рядом с роботом-поставщиком должно располагаться перед конвейером робота.

Рисуем путь. Если он вдруг оказался не совсем таким, как ожидали – "ластик" и возможность полностью очистить экран всегда помогут. На рис. 24 представлен типичный путь, по которому может происходить доставка.

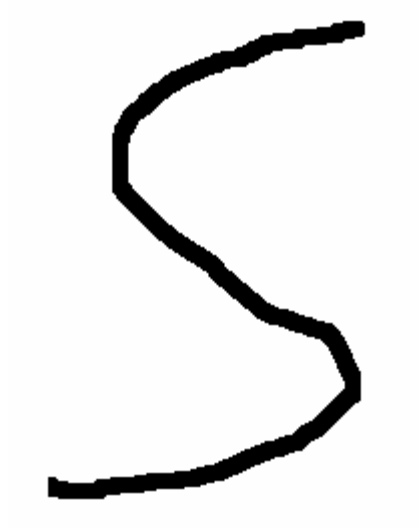

Рис. 24. Путь

После того как путь нарисован, следует нанести места для остановок. Выбираем соответствующий режим рисования кнопкой "Рисовать место остановки" ("Foil"). После этого на каждом из двух концов пути рисуем область "из фольги". Лучше ее сделать по ширине раза в два больше, чем ширина пути, чтобы транспортный робот при подъезде к этой области не взял случайно левее или правее и не проехал мимо. На рис. 25 изображен наш путь с местами для остановки.

Рис. 25. Путь с двумя местами для остановки

Нарисовав путь, следует расставить и задать нужные направления роботам. Выбираем режим установки транспорта кнопкой "Установить транспорт" ("Place Transport"), и ставим его на одно из мест для остановки (рис. 26).

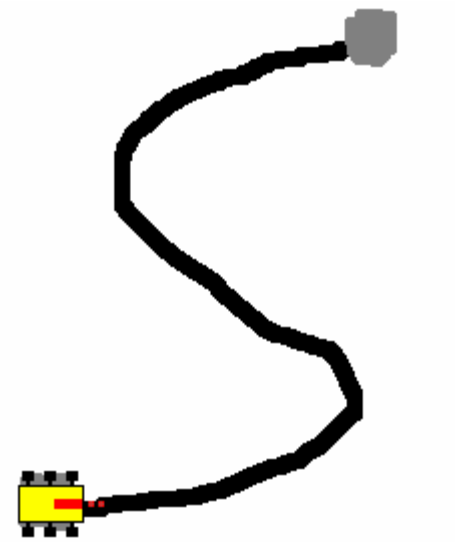

Рис. 26. Транспорт установлен

После того, как транспорт установлен, его надо повернуть в сторону, противоположную направлению движения (в проекте *Isenguard*, прежде чем начать движение, транспорт поворачивается на 180º). Теперь транспорт будет направлен, как изображено на рис. 27.

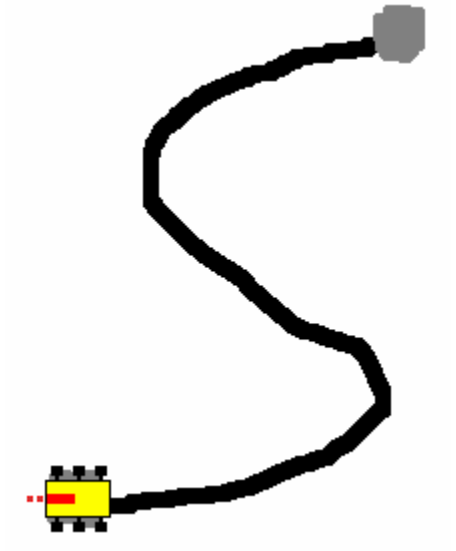

Рис. 27. Транспорт направлен

Перейдем к установке робота-поставщика. Конвейер робота-поставщика должен находиться напротив места остановки и чуть-чуть его покрывать. Приемник инфракрасных сообщений (красный прямоугольник на желтом микроконтроллере) должен находиться на продолжении пути. На рис. 28 изображен установленный соответствующим образом робот-поставщик при помощи кнопок "Установить поставщик" ("Place Dispencer") и "Вращать поставщик" ("Rotate Dispencer").

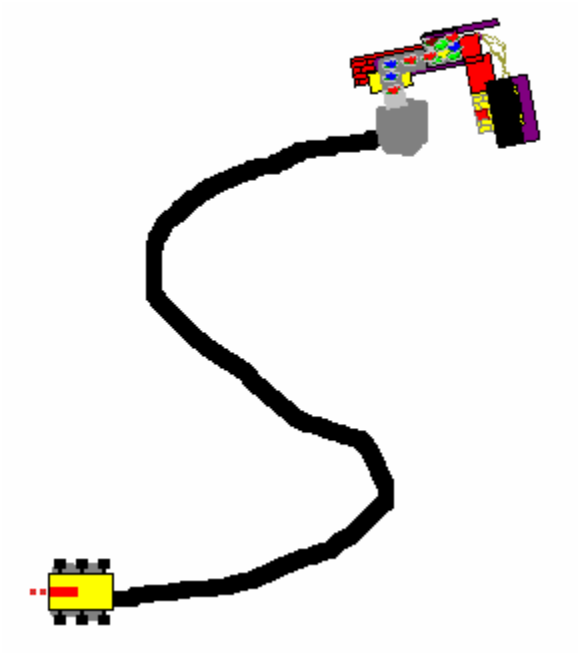

Рис. 28. Робот-поставщик установлен и направлен

Итак, роботы установлены, путь и места остановки изображены. Нажимаем кнопку "OK", для того чтобы все изменения сохранились.

Теперь программа переведена в режим эмуляции. Нажимаем кнопку "Старт"\"Пауза" ("Start"\"Pause"). Эмуляция запущена – теперь, если перевести указатель мыши в область визуализации, то на месте указателя мыши появится изображение пульта дистанционного управления (рис. 29).

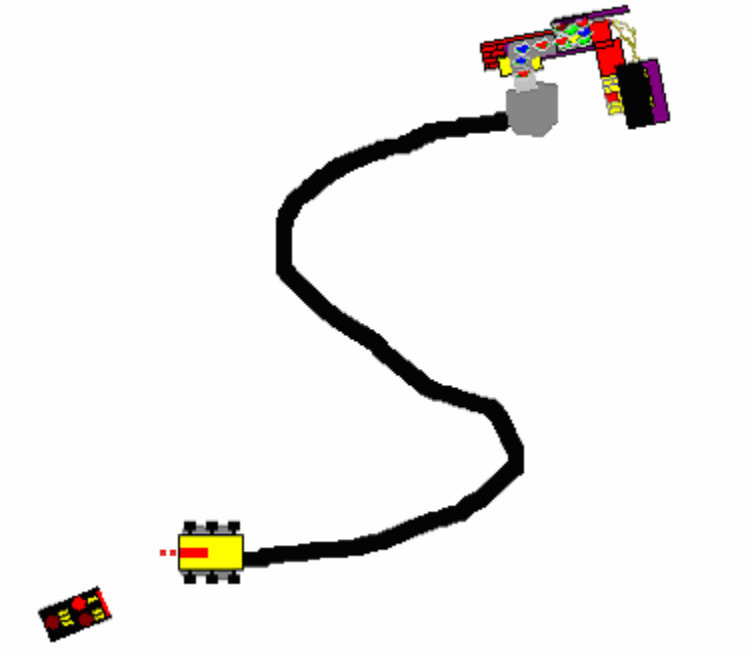

Рис. 29. Режим эмуляции

При помощи кнопок доставки одного, двух или трех предметов ("One Item", "Two Item", "Three Item") можно выбрать тип сообщения, посылаемого пультом дистанционного управления. Выберем, к примеру, сообщение о доставке двух предметов. После этого

наведем пульт управления на транспортный робот (вращать пульт можно при зажатой правой кнопке мыши) и, щелкнув по левой кнопке мыши, начнем посылать сообщение (рис. 30). Повторный щелчок мыши прекращает передачу сообщения от пульта.

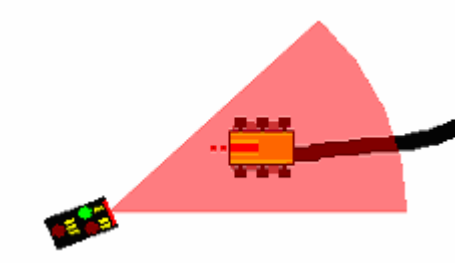

Рис. 30. Посылка сообщения о доставке

Управляющая программа транспортного робота, как только она обработает сообщение, разворачивает его в противоположном направлении и начинает вести по пути . При этом можно наблюдать как движется штанга со световым датчиком. Сама штанга не изображается – ее движение совпадает с движением светового датчика, который выглядит как небольшой красный квадрат. На рис. 31 показано, как транспорт ищет путь слева от себя.

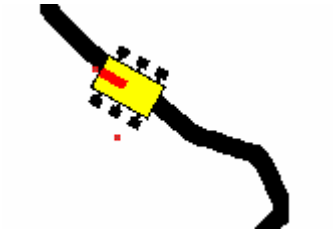

Рис. 31. Транспорт движется по пути

После того, как транспорт доедет до места остановки, расположенного рядом с роботом-поставщиком, он еще немного проедет вперед и пошлет сообщение о своей готовности к погрузке (рис. 32). В сообщении будет закодировано количество доставленных предметов. Робот-поставщик, получив сообщение, начинает процесс погрузки.

Рис. 32. Транспортный робот готов к погрузке

Как только предмет сбрасывается с конвейера на транспорт, то он появляется на "спине" транспорта (рис. 33).

Рис. 33. Первый предмет погружен

Как только все предметы будут погружены, робот-поставщик посылает сообщение транспортному роботу, о том, что погрузка завершена (рис. 34).

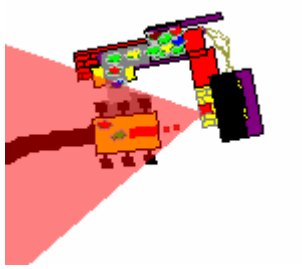

Рис. 34. Погрузка завершена

После этого транспортный робот отъезжает от робота-поставщика и начинает свой путь назад, к месту доставки. После того, как он доедет до этого места, произойдет остановка (рис. 35). Транспортный робот ждет следующее сообщение от пульта, чтобы привезти пользователю другие предметы.

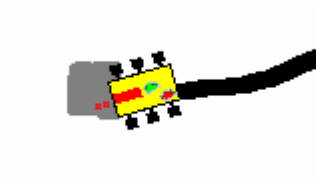

Рис. 35. Доставка завершена

Так как эмуляция пользователя, который бы забирал доставленные предметы в программе, не предусмотрена, то было сделано следующее конструктивное решение – как только транспортный робот получает новое сообщение с пульта, его поддон с предметами снова становится пустым.

При эмуляции можно использовать еще несколько возможностей, предоставляемых программой. Например, сделать двукратное увеличение и включить режим слежения (рис. 36). К сожалению, показать как работает режим слежения в документации нельзя, так что рекомендуем читателю обратиться непосредственно к программе, расположенной на этом же сайте (http://is.ifmo.ru/, раздел "Проекты").

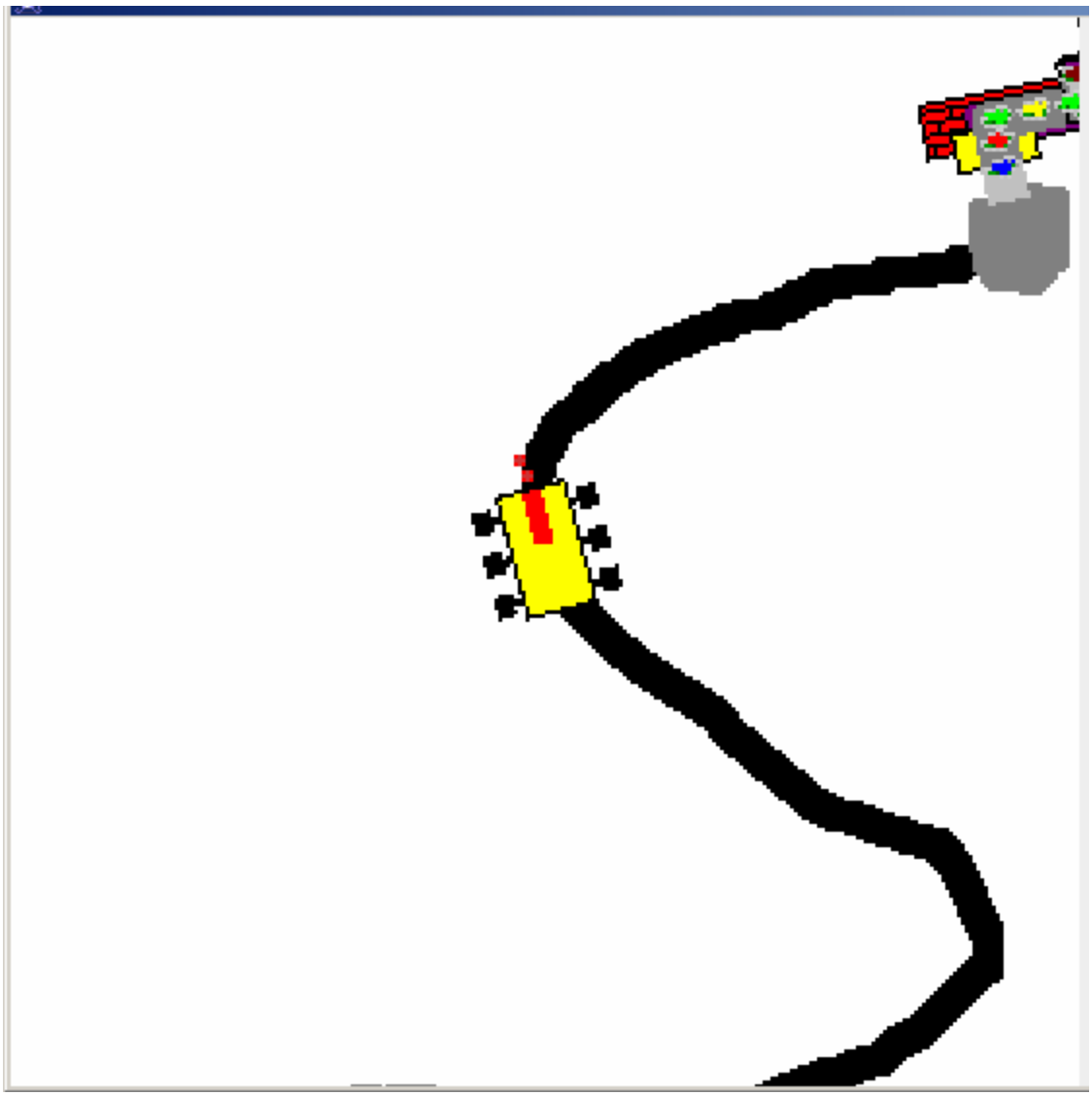

Рис. 36. Слежение при двукратном увеличении

При эмуляции было выявлено несколько недостатков системы доставки. Первый из них состоит в том, что если транспорт неточно подъезжал к роботу-поставщику, то сообщение зачастую не попадало на приемник робота-поставщика, и транспортный робот оставался навечно стоять и ждать ответного сообщения от робота-поставщика (рис. 37). Конечно, не совсем навечно – до того момента, пока эмуляция не будет перезапущена.

Рис. 37. Транспорт подъехал неточно

Подобная ситуация иногда возникает, если прямой участок пути перед роботом-поставщиком недостаточной длины. Хотя скорее это проблемы выбранного метода остановки – транспортный робот не может точно определить свое направление и позицию

относительно робота-поставщика. Одна из важных задач – сделать более "продвинутую" систему точного подъезда к роботу-поставщику.

Сбоев в системе при движении по пути обнаружено не было. Недостаток системы – ее медлительность.

Помимо изучения работы системы в "нормальном" режиме, были специально промоделированы несколько аварийных ситуаций, для проверки системы управления роботов. В первой ситуации путь неожиданно обрывался (рис. 38).

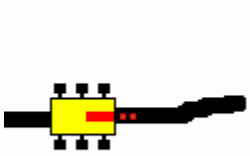

Рис. 38. Путь неожиданно обрывается

Транспортный робот действовал, как и предполагалось – поискав путь своей штангой слева и справа, он останавливался (рис. 39), выдав сообщение об ошибке (рис. 40).

Рис. 39. Транспорт остановился

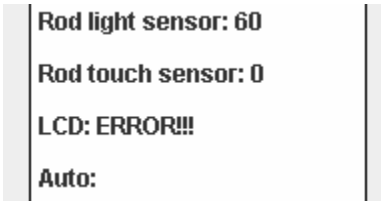

Рис. 40. Ошибка! Путь не найден

Потом была проведен эксперимент, в котором использовалась возможность редактора изображать места ошибки. В этом эксперименте место ошибки пересекает путь (рис. 41). Робот, наехав на это место, как и в предыдущем эксперименте, остановился с аналогичным сообщением об ошибке (рис. 42).

Рис. 41. Ошибка на пути

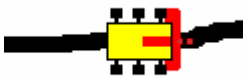

Рис. 42. Робот остановился

После этого была проверена еще одна возможность, заложенная в систему управления транспортным роботом – возможность проскакивать через небольшие промежутки в пути. На рис. 43 показан как раз такой промежуток в пути. Пользователь, не ознакомившийся со всеми возможностями управляющей системы транспорта, может подумать, что робот остановится на месте и начнет искать путь, но нет! На самом же деле, транспортный робот спокойно наедет на этот промежуток (рис. 44) и продолжит свое движение (рис. 45). Для того, чтобы более подробно изучить данную особенность управляющей системы транспортного робота, следует посмотреть документацию на автомат *A2* из проекта *Isenguard*.

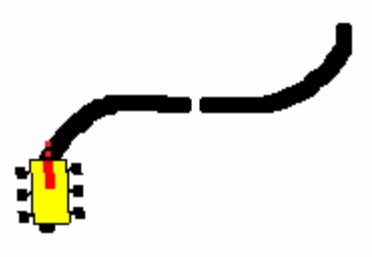

Рис. 43. Промежуток

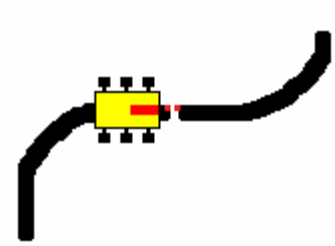

Рис. 44. Робот наезжает…

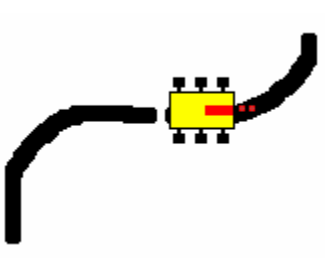

Рис. 45. ... и проезжает!

4. Архитектура эмулятора

4.1. Технология разработки эмуляторов на основе автоматного подхода

Основные положения этого подхода были сформулированы в ряде работ [3–5]. Для написания эмуляторов мультиагентных систем (одной из таких мультиагентных систем и является проект *Isenguard*) этот подход может быть сформулирован следующим образом.

1. Выполняется сбор информации. Изучается мультиагентная система, работу которой следует эмулировать. Необходимо определить какая эмуляция требуется – некоторые мультиагентные системы необходимо эмулировать в режиме реального времени. К таким системам относится и система из проекта *Isenguard*. Для других мультиагентных систем может использоваться событийно-ориентированная эмуляция. К примеру, произошло какое-то событие – пользователь нажал кнопку, или по сети пришло сообщение, система каким-то образом отреагировала на это сообщение В частности, при разработке проекта *IsenguardEmu* использовалась как эмуляция в режиме реального времени (физические процессы – движение транспортного робота по пути, отгрузка и погрузка предметов), так и событийно-ориентированная эмуляция (логика работы различных компонентов транспортного робота и робота-поставщика).

2. Разрабатывается физическая модель мультиагентной системы (если требуется).

3. Для каждого агента разрабатывается диаграмма классов.

4. Разрабатывается диаграмма классов и интерфейсов, обеспечивающая взаимодействие между агентами (среда взаимодействия агентов).

5. Разрабатывается пользовательский интерфейс программы и его диаграмма классов.

6. Определяются шаблоны построения изоморфного кода классов, управляемых автоматами, по графам переходов. Для этих классов разрабатываются структурные схемы, отражающие структурные особенности реализации данных классов.

7. Если логика работы класса описывается несколькими автоматами, то строится схема взаимодействия этих автоматов. При разработке проекта *IsenguardEmu*, авторы придерживались правила – один класс может либо не реализовывать ни один из автоматных интерфейсов, либо реализовывать один их них, либо оба (природа некоторых классов *IsenguardEmu*, реализующих компонент системы, дуальна – она имеет как физическую, так и логическую составляющую. Поэтому некоторые классы реализуют оба автоматных интерфейса).

8. Для каждого автомата создается четыре документа:

- словесное (вербальное) описание поведения автомата;
- схема связей автомата, задающая его интерфейс, в которой указываются символьные обозначения входных переменных, событий и выходных воздействий и их полные названия, а также названия источников и приемников информации;

- граф переходов, в котором состояния имеют названия, а все остальные составляющие помечены символами. Это позволяет компактно и формально описывать даже весьма сложное поведение агента;

- по графу переходов формально и изоморфно строится исходный код программы, реализующий логику работы автомата.

Отметим, что входные переменные могут быть функциями, в том числе и весьма сложными.

9. Пишется исходный код каждого из классов. Для классов, логика которых описывается автоматами, код изоморфно реализуется по графам переходов.

10. Выполняется отладка эмулятора. Отладка упрощается за счет возможности наблюдения за состояниями каждого автомата.

11. В эмуляторе изучается работа полученной мультиагентной системы.

12. В случае если необходимо внести какие-то изменения в эмулятор или в мультиагентную систему, то всегда возможно повторить шаги с первого вплоть до одиннадцатого и добавить или изменить необходимую функциональность.

13. Осуществляется разработка и выпуск проектной документации [7].

5. Процесс разработки эмулятора

В этом разделе будет рассмотрен поэтапный процесс разработки эмулятора в рамках предложенного подхода.

5.1.1. Сбор информации

В настоящем проекте в качестве мультиагентной системы для эмуляции выступает система из проекта *Isenguard*. Она состоит из двух роботов-агентов: транспортного робота и робота-поставщика. Так что помимо логики работы данных агентов, необходимо также эмулировать и сами роботы – динамику физического процесса движения роботов.

Транспортный робот считывает информацию о цвете поля при помощи световых датчиков. Необходимо спроектировать сущность "поле", а также предусмотреть механизм передачи информации о цвете поля в определенной точке световым датчикам.

Роботы также обмениваются сообщениями между собой и с пультом. Необходимо разработать механизм передачи сообщений, который в зависимости от взаимного положения передатчика и приемника будет либо передавать между ними сообщение, либо нет.

В каждый из роботов проекта *Isenguard* загружается управляющая программа. Требуется реализовать интерфейс, позволяющий легко вносить изменения в управляющую программу в эмуляторе, а также разработать механизм передачи данных от датчиков робота и механизм передачи данных от управляющей системы к двигателям.

5.1.2. Физическая модель

Разработка физической модели мультиагентной системы сводится к разработке физической модели следующих объектов: транспортного робота, робота-поставщика и передачи сообщений по инфракрасным портам.

5.1.2.1. Физическая модель транспортного робота

В проекте *Isenguard* транспортный робот может двигаться вперед и назад, а также поворачивать влево и вправо. Управляющая система транспортного робота спроектирована таким образом, что прямолинейное движение вперед и назад практически не коррелирует с вращательным движением влево и вправо. Таким образом, робот по замыслу разработчиков проекта *Isenguard* либо движется прямолинейно, либо вращается на месте. Выделяем основные состояния транспортного робота: стоит на месте, движется вперед, движется назад, поворачивает влево, поворачивает вправо.

Рассмотрим сначала прямолинейное движение транспортного робота. Для каждого режима мощности работы двигателей, транспортный робот будет двигаться с соответствующей скоростью, возрастающей при увеличении мощности двигателей. При старте с места транспортный робот не сразу начнет движение с соответствующей скоростью, а некоторое время будет до нее разгоняться. Также, если робот двигался со одной скоростью, а затем изменили мощность работы двигателей, то робот также не сразу начнет двигаться с новой скоростью, а будет до нее разгоняться или замедляться.

Учитывая, что транспортный робот двигался сравнительно медленно (порядка 5–6 см/с), а также его малую инертность (привод от двигателей к колесам организован при помощи червячной передачи), то было решено использовать следующую физическую модель транспортного робота:

- для каждого значения мощности работы двигателя $P_i; i \in 0..7$ определить скорость $\vec{v}_i$, с которой робот будет двигаться;

- при изменении мощности работы двигателя, а также при изменении направления движения (вперед или назад) робот будет двигаться с постоянным ускорением $\vec{a}$, пока не достигнет соответствующей скорости. Модуль ускорения одинаков для всех значений мощности работы двигателей ($\vec{v} = \vec{v}_0 + \vec{a}dt$);

- при торможении робот движется с тем же по модулю ускорением, что и при изменении мощности работы двигателей.

На графике (рис. 46) показана зависимость скорости движения робота от времени. Робот сначала покоился на месте, затем начинает движение с мощностью P_1, разгоняется до скорости $\vec{v}_1$, и некоторое время двигается с ней. После этого мощность понижается до значения $P_2 < P_1$. Робот замедляет скорость своего движения до $\vec{v}_2$. После движения с этой скоростью, робот поменяет направление своего движения и т.д. В конце концов, робот останавливается.

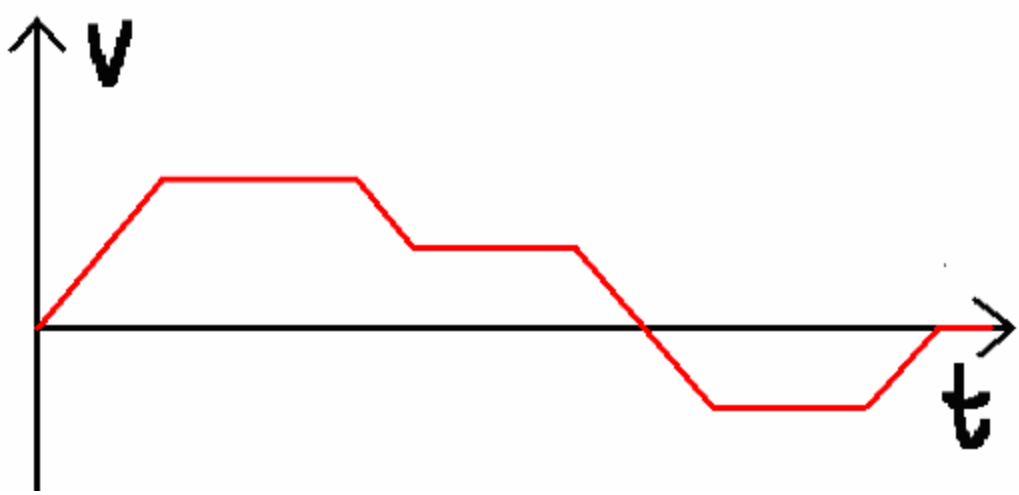

Рис. 46. Зависимость скорости от времени

Теперь рассмотрим вращательное движение транспортного робота. Тут ситуация практически такая же, как и при прямолинейном движении – для каждого из значений мощности работы двигателей $P_i; i \in 0..7$ определена соответствующая скорость вращения ω_i и угловое ускорение ε, которое разгоняет или замедляет вращающийся робот при изменении мощности. График изменения угловой скорости представлен на рис. 47.

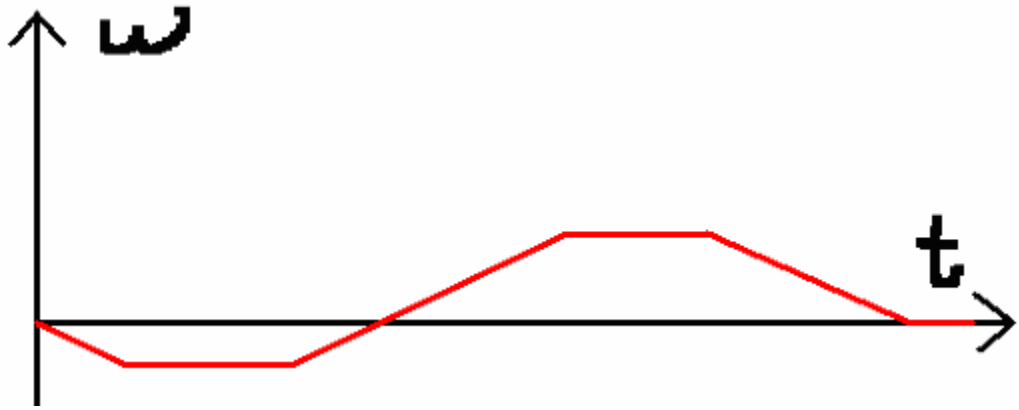

Рис. 47. График зависимости угловой скорости от времени

Теперь рассмотрим "комбинированную" ситуацию: предположим, что робот вращался, а управляющая система сформировала команду "двигаться прямо". Как действовать в этом случае?

Угловая скорость, имевшаяся у робота, не исчезает моментально, а будет уменьшаться по модулю, как если бы робот, вращаясь на месте, начал тормозить. Аналогично будет, если при прямолинейном движении робота, управляющая система даст команду повернуть влево и вправо. Линейная скорость робота будет уменьшаться по модулю в соответствии с уже сформулированными законами движения транспортного робота.

В данной физической модели предлагается связать мощность работы двигателей с линейной скоростью следующим образом: $v_i = iC$, где C – константа, соответствующая линейной скорости при мощности работы двигателей P_1. С угловой скоростью мощность

связывается аналогичным образом: $\omega_i = iD$, где D – некая константа, соответствующая угловой скорости при мощности работы двигателей P_1.

У транспортного робота есть еще одна важная движущаяся деталь – штанга. Она может вращаться вправо и влево, когда транспортный робот ищет путь. Вращение штанги обеспечивается еще одним двигателем. Для эмуляции вращения штанги использовался тот же подход, что и при эмуляции вращения транспортного робота. Угловая скорость штанги также связывается с мощностью работы двигателя по закону $\omega_i = iK$, где K – константа, соответствующая скорости при мощности работы двигателя штанги P_1. Для штанги также определяется угловое ускорение $\varepsilon_{шт}$, которое воздействует на угловую скорость штанги при торможении или изменении мощности работы двигателя.

Расчет положения робота выполняется при помощи явной схемы Ньютона. В вычислительной математике данный метод является одним из самых плохих с точки зрения точности, но в данной задаче, где самым сложным типом движения является равноускоренное движение, точности, обеспечиваемой этим методом, вполне хватает.

5.1.2.2. Физическая модель робота-поставщика

Робот поставщик использует двигатель для того, чтобы сбрасывать предметы со своего конвейера на транспортный робот. Для того чтобы сбросить один предмет, ручка-сбрасыватель должна сделать полный оборот вокруг своей оси. Зависимость скорости вращения ручки-сбрасывателя от мощности работы двигателя поставщика (P_i) определяется следующим образом: $\omega_i = iK$. Инертностью ручка-сбрасыватель не обладает.

5.1.2.3. Физическая модель передачи данных по инфракрасному порту

Как известно, по инфракрасному порту данные можно передавать лишь на незначительное расстояние. Сообщение распространяется в пределах определенного угла. Также приемник может принять сообщение, если он находится под определенным углом к линии распространения этого сообщения. Соответственно для передатчика определяется угол распространения сообщения и дистанция распространения сообщения. Для приемника определяется угол приема сообщения. На рис. 48 изображен передатчик. Дистанция распространения сообщения помечена буквой *D*, угол распространения сообщения помечен буквой *C*. На рис. 49 показаны два приемника *E* и *F*. Приемник *F* получит сообщение, а приемник *E* сообщение не получит.

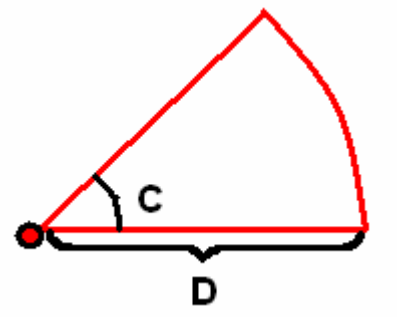

Рис. 48. Передатчик и сообщение

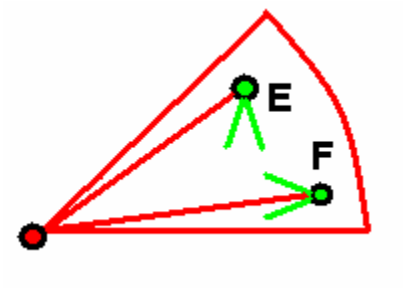

Рис. 49. Сообщение и приемники

5.1.3. Программные интерфейсы

Для программной реализации агентов, среды их взаимодействия, а также для пользовательского интерфейса было разработано несколько программных интерфейсов. В данном разделе они будут кратко описаны.

5.1.3.1. Интерфейс *Automaton*

С помощью данного интерфейса реализуются активные автоматы внутри класса.

Методы:

- `void updateAutoState(int event, int data)`. Это аналог автоматной функции [3]. Этот метод вызывается с параметром `event` – событием, по которому был вызван этот метод, и дополнительным параметром `data`, который может содержать некую дополнительную информацию. Эта информация может быть передана в методы, реализующие выходные воздействия. Более подробно об этом механизме сказано ниже.

- `void setAutoState(int state)`. Этот метод формирует текущее состояние автомата.

- `int getAutoState()`. Этот метод возвращает текущее состояние автомата.

5.1.3.2. Интерфейс *CalcObject*

С помощью данного интерфейса реализуется логика пассивных (вычислительных) автоматов внутри класса.

Методы:

- `void updateCalcState(int event, int data, double timeDelta, Object objData)`. Это автоматная функция пассивного автомата. Отличается от интерфейса `Automaton` большим количеством параметров. Во-первых, параметром `timeDelta` – промежутком времени, а, во-вторых, параметром `objData`, при помощи которого можно передать в автоматную функцию объект.

- `void setCalcState(int calcState)`. Этот метод выставляет текущее состояние автомата.

- `int getCalcState()`. Этот метод возвращает текущее состояние автомата.

5.1.3.3. Интерфейс *CoordinateObject*

Этот интерфейс позволяет получать и изменять координаты объекта. Его реализуют классы обеих роботов, а также компоненты роботов, координаты которых необходимы для физической эмуляции.

Методы:

- `double getX()`. Возвращает текущую абсциссу объекта.

- `double getY()`. Возвращает текущую ординату объекта.

- `double getAngle()`. Возвращает текущий угол объекта в радианах (от оси X, против часовой стрелки).

- `void setX(double x)`. Изменяет текущую абсциссу объекта.

- `void setY(double y)`. Изменяет текущую ординату объекта.

- `void setAngle(double angle)`. Изменяет текущий угол объекта.

5.1.3.4. Интерфейс *Sensor*

Этот интерфейс реализуют объекты, соответствующие световому датчику и датчику касания.

Методы:

- `int getValue()`. Возвращает текущее значение, считываемое датчиком.

- `String getDescription()`. Возвращает краткое описание датчика.

5.1.3.5. Интерфейс *ControlSystem*

Это интерфейс реализуют управляющие системы роботов.

Методы:

- `void restart()`. Перезапускает управляющую систему.

- `List getControlAutomata()`. Возвращает список управляющих автоматов. (объекты типа `ControlAutomaton`).

5.1.3.6. Интерфейс *FieldRobot*

Этот интерфейс реализуют оба класса, соответствующих роботам. При помощи данного интерфейса можно получить текущее сообщение на *LCD*-дисплее робота. Эмулятор с помощью соответствующего метода может получить список датчиков робота с целью считывания с них информации.

Методы:

- `List getSensors()`. Возвращает список из всех датчиков данного робота (объектов реализующих интерфейс `Sensor`).
- `ControlSystem getControlSystem()`. Возвращает объект, соответствующий управляющей системе робота.
- `String getDescription()`. Возвращает краткое описание робота.
- `String getLcdMessage()`. Возвращает текущее сообщение *LCD*-дисплея.
- `RobotIRDAModule getIrdaModule()`. Возвращает объект, отвечающий за прием и передачу сообщений по инфракрасному порту.

5.1.3.7. Интерфейс *DrawableObject*

Этот интерфейс предназначен для отрисовки агентов и их компонентов в окне программы. При помощи него можно получить объект типа `Image`.

Методы:

- `Image getObjectImage()`. Возвращает изображение объекта для отрисовки.
- `void setObjectImage(Image img)`. Устанавливает изображение объекта.
- `boolean isShown()`. Возвращает текущий статус объекта (видимый или невидимый).
- `void setShown(boolean shown)`. Устанавливает текущий статус объекта (видимый или невидимый).

5.1.4. Диаграммы классов агентов

Помимо интерфейсов для реализации агентов и их компонентов было разработано несколько классов. Вначале рассмотрим классы, общие для роботов, а затем классы, соответствующие каждому из роботов в отдельности.

5.1.4.1. Абстрактный класс *PhysicalObject*

Этот класс предоставляет базовую функциональность для движущихся физических объектов (таких в эмуляторе три: два робота и штанга транспорта). Он реализует

интерфейсы `CoordinateObject`, `CalcObject` и `ControlSystem`. При помощи методов `double getlSpeedX()`, `double getlSpeedY()`, `double getwTorque()`, `double getAccX()`, `double getAccY()`, `double getAccTorque()` можно получить текущую скорость, ускорение, угловую скорость и угловое ускорение объекта. Методы `double getMass()` и `double getInertiaMoment()` позволяют получить массу и момент инерции относительно центральной оси. Метод `double getSize()` возвращает характерный размер объекта. Конструктор `PhysicalObject(double mass, double inertia, double size)` создает объект с соответствующей массой, моментом инерции и характерным размером.

5.1.4.2. Абстрактный класс *ControlAutomaton*

Данный класс реализует интерфейс `Automaton` и используется как базовый класс для автоматов из управляющих систем. В нем реализованы следующие методы:

Методы:

- конструктор `ControlAutomaton(int id)`. Создает новый объект с уникальным номером `id`.
- `void init()`. Инициализирует управляющий автомат начальным состоянием и все его внутренние переменные начальными значениями.
- `int getId()`. Возвращает уникальный номер управляющего автомата.
- `abstract void addCommand(int autoId, int commandId)`. Добавляет команду `commandId` данному автомату от автомата с уникальным номером `autoId`. Предназначен для реализации методики передачи сообщений от одного автомата другому в проекте *Isenguard*.

5.1.4.3. Класс *AutomatonUpdator*

Функциональность данного класса похожа на функциональность `Listener` в языке *Java*. Он выступает в качестве обработчика событий для автоматов. Предположим, что в неком классе происходит какой-то процесс и требуется узнать о том, когда данный процесс изменит свой статус (из активного в неактивный, к примеру). Существует также другой класс, который реализует интерфейс `Automaton`. Требуется, чтобы при изменении статуса процесса, автомат другого класса вызывался с определенным событием. Именно для этого и предназначен класс `AutomatonUpdator`. Более подробно данная методика вызова автоматной функции одного автомата по некоему условию или событию, произошедшему в другом классе, будет рассмотрена ниже.

Методы:

- `AutomatonUpdator(Automaton auto, int event)`. Создает новый объект, который будет вызывать автомат `auto` с событием `event`.

- `void updateAutomaton(int param)`. Вызывает автоматную функцию автомата с параметром `param`.

5.1.4.4. Класс *LightSensor*

Класс `LightSensor` является программной реализацией светового датчика. Реализует интерфейсы `Automaton`, `CoordinateObject` (нам необходимы координаты датчика, чтобы загрузить в него соответствующее значение цвета поля), `Sensor` и `DrawableObject`. Метод `getValue()` возвращает число, соответствующее текущему цвету поля.

Методы:

- `void activate()`. Активитует световой датчик.
- `void deactivate()`. Деактивирует световой датчик.
- `void intensityChanged(int value)`. Обеспечивает обновление значения светового датчика.
- `LightSensor(String description)`. Создает новый объект с описанием `description`.

5.1.4.5. Класс *TouchSensor*

Класс `TouchSensor` является программной реализацией датчика касания. Реализует интерфейсы `Automaton` и `Sensor`. Если датчик касания нажат, то метод `getValue()` возвращает единицу, иначе он возвращает ноль.

Методы:

- `void touch()`. Нажать датчик касания.
- `void untouch()`. Провести с датчиком касания операцию, обратную нажатию.
- `TouchSensor(String description)`. Создает новый объект с описанием `description`.

5.1.4.6. Класс *InfraredEmitter*

Данный класс обеспечивает передачу сообщений по инфракрасному порту. Реализует интерфейсы `Automaton`, `CalcObject`, `CoordinateObject` и `DrawableObject`.

Методы:

- `InfraredEmitter(Color emittingColor, double radius, double alpha, double emittingPeriod)`. Создает новый объект. Параметр `emittingColor` определяет какого цвета будет конус распространения

сообщения при отрисовке. Параметр `radius` – дистанция распространения, `alpha` – угол распространения, `emittingPeriod` – время между двумя импульсами генерации сообщения.

- `void startSending(byte value)`. Начать передачу сообщения `value`.
- `void stopSending()`. Окончить передачу сообщения.
- `byte getSendingValue()`. Текущее передаваемое сообщение.
- `boolean isSending()`. Возвращает значение `true`, если объект находится в режиме передачи.
- `boolean isSignalReady()`. Возвращает значение `true`, если объект готов к передаче очередного импульса сообщения.
- `void signalSent()`. Сбросить текущий импульс сообщения.
- `Color getEmitterColor()`, `double getInfraredRadius()`, `double getInfraredAlpha()`. Данные методы возвращают соответственно цвет конуса сообщения, дистанцию и угол передачи.

5.1.4.7. Класс *InfraredReceiver*

Данный класс позволяет принимать сообщения по инфракрасному порту.

Методы:

- `InfraredReceiver(double angle)`. Создает новый объект с углом приема `angle`.
- `boolean hasMessage()`. Возвращает значение `true`, если получено сообщение.
- `byte getMessage()`. Возвращает полученное сообщение.
- `void addMessage(byte message)`. Добавляет новое сообщение в очередь приемника.
- `double getReceiveAngle()`. Возвращает угол приема сообщений.
- `void addAutomatonUpdatorOnReceiving(Automaton auto, int event)`. По параметрам `auto` и `event` создает объект типа `AutomatonUpdator` и инициирует его каждый раз, когда сообщение получено.

5.1.4.8. Класс *InfraredReceiverEmitter*

Класс объединяет в себе функциональность как передатчика, так и приемника, включая в себя в качестве членов объекты `InfraredEmitter` и `InfraredReceiver`. Этот класс

реализует интерфейсы `DrawableObject` и `CoordinateObject`. Разработан класс для удобства – передатчик и приемник сообщений в проекте *Isenguard* реализуются при помощи одного инфракрасного порта, находящегося в микроконтроллере каждого из роботов.

Методы:

- `InfraredReceiverEmitter(double receiveAngle, Color color, double radius, double angle, double period)`. Создает новый объект, передатчику и приемнику передаются соответствующие параметры.
- `InfraredReciever getReceiver()`. Возвращает объект, соответствующий приемнику.
- `InfraredEmitter getEmitter()`. Возвращает объект, соответствующий передатчик.

5.1.4.9. Класс *RobotIRDAModule*

Этот класс является программной реализацией инфракрасного порта, находящегося в каждом из микроконтроллеров. Позволяет, как передавать, так и получать сообщения. Умеет классифицировать тип сообщения – от пульта или от другого порта на микроконтроллере.

Методы:

- `RobotIRDAModule()`. Создает новый объект с параметрами, соответствующими параметрам из проекта *Isenguard*.
- `void sendMessage(byte message)`. Послать сообщение по инфракрасному порту.
- `boolean isAbleToSend()`. Возвращает значение `true`, если порт готов к передаче.
- `boolean isMessageFromRemoteReceived()`. Возвращает значение `true`, если получено сообщение от пульта управления.
- `boolean isMessageFromRobotReceived()`. Возвращает значение `true`, если получено сообщение от микроконтроллера на другом роботе.
- `int readMessage()`. Считывает и возвращает полученное сообщение. Если сообщение не было получен, то возвращает минус единицу.
- `void flushMessages()`. Сбрасывает полученные сообщения.

5.1.5. Транспортный робот

5.1.5.1. Класс *SixWheeledTransport*

Этот класс является программной реализацией транспортного робота из проекта *Isenguard*. Он предоставляет возможность управления движением, погрузкой и разгрузкой предметов, и посылкой сообщений. В этом классе в качестве внутреннего класса реализована система управления (`LineTraceControlSystem` – о ней далее), аналогичная системе управления транспортным роботов из проекта *Isenguard*. Данный класс наследуется от класса `PhysicalObject` и реализует интерфейс `FieldRobot`.

Методы:

- `SixWheeledTransport()`. Создает новый объект с параметрами, соответствующими транспортному роботу из проекта *Isenguard*.
- `void moveForward(int power)`. Формирует команду двигаться вперед с мощностью `power`.
- `void moveBackward(int power)`. Формирует команду двигаться назад с мощностью `power`.
- `void turnLeft(int power)`. Формирует команду поворачивать влево с мощностью `power`.
- `void turnRight(int power)`. Формирует команду поворачивать вправо с мощностью `power`.
- `void setPower(int power)`. Выставляет новой значение мощности работы двигателей.
- `int getPower()`. Возвращает текущее значение мощности работы двигателей.
- `void stop()`. Останавливает робот.
- `void restart()`. Возвращает все параметры робота к исходным значениям.
- `void loadItem(int type)`. Погрузить предмет типа `type` на транспорт.
- `void unloadAllItems()`. Выгрузить все предметы из транспорта.

5.1.5.2. Класс *Rod*

Этот класс отвечает за управление штангой транспортного робота.

Методы:

- `Rod(double rodRadius, double rodAngle)`. Создает штангу с радиусом `rodRadius` и исходным углом `rodAngle`.

- `void turnLeft(int power)`. Обеспечивает поворот налево с мощностью двигателей `power`.
- `void turnRight(int power)`. Обеспечивает поворот направо с мощностью двигателей `power`.
- `void reverse()`. Позволяет изменить направление поворота штанги на противоположное.
- `void restart()`. Позволяет вернуть штангу в исходное положение.
- `void stop()`. Останавливает штангу.
- `void setPower(int power)`. Выставляет новой значение мощности работы двигателя штанги.
- `int getPower()`. Возвращает текущее значение мощности работы двигателя штанги.
- `double getRadius()`. Возвращает радиус штанги.
- `double getInitialAngle()`. Возвращает исходный угол поворота штанги.

5.1.5.3. Класс *LineTraceControlSystem*

Этот класс реализует интерфейс `ControlSystem` и реализует управляющую систему транспортного робота. В качестве членов данного класса выступают автоматы `A1Auto`, `A2Auto`, `A3Auto`, `A5Auto`, которые наследуют класс `ControlAutomaton`.

5.1.5.4. Диаграмма классов транспортного робота

Полученная диаграмма классов транспортного робота представлена на рис. 50.

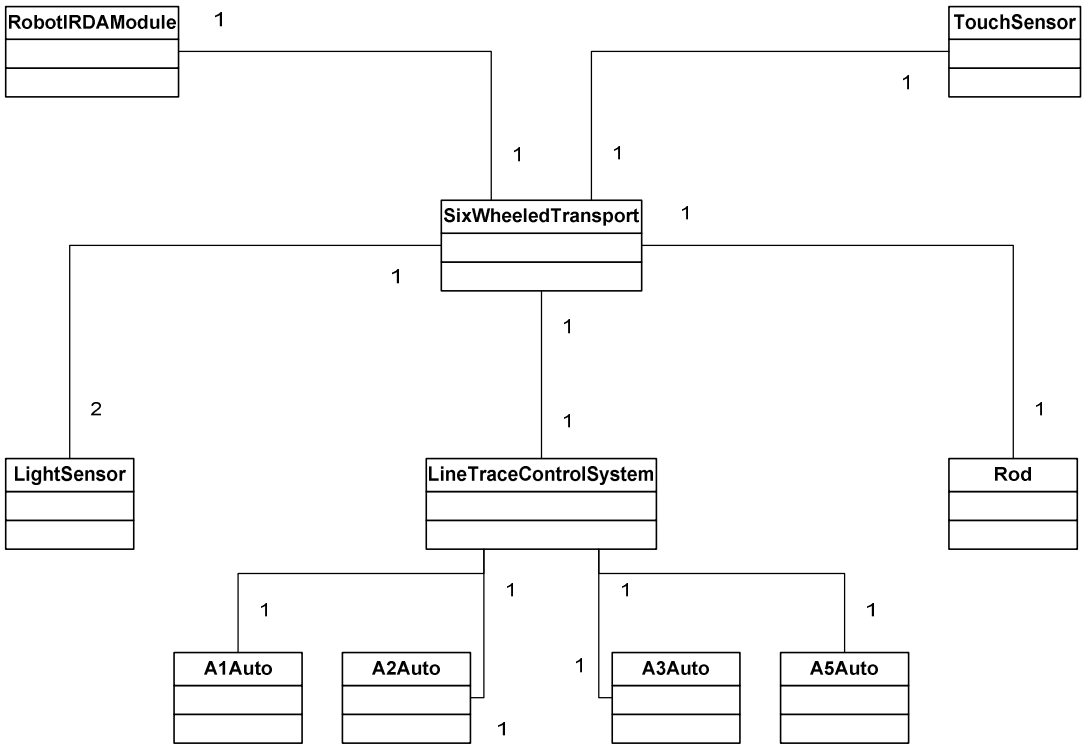

Рис. 50. Диаграмма классов транспортного робота

5.1.6. Робот-поставщик

5.1.6.1. Класс *Dispencer*

Данный класс является программной реализацией робота-поставщика из проекта *Isenguard*. При помощи данного класса можно управлять погрузкой предметов. Управляющая система (`DispencerControlSystem`) также реализована как внутренний класс.

Методы:

- `void startDispencing(int power)`. Начать отгрузку предметов при мощности двигателя `power`.
- `void stopDispencing()`. Окончить погрузку предметов.
- `void restart()`. Возвращает робот-поставщик в исходное состояние.
- `int getPower()`. Возвращает текущее значение мощности работы двигателя штанги.

5.1.6.2. Класс *DispencerControlSystem*

Данный класс реализует интерфейс `ControlSystem` и является управляющей системой для робота-поставщика. В качестве членов данного класса выступает автомат `A0Auto`, который наследует класс `ControlAutomaton`.

5.1.6.3. Диаграмма классов робота-поставщика

Диаграмма классов робота-поставщика представлена на рис. 51.

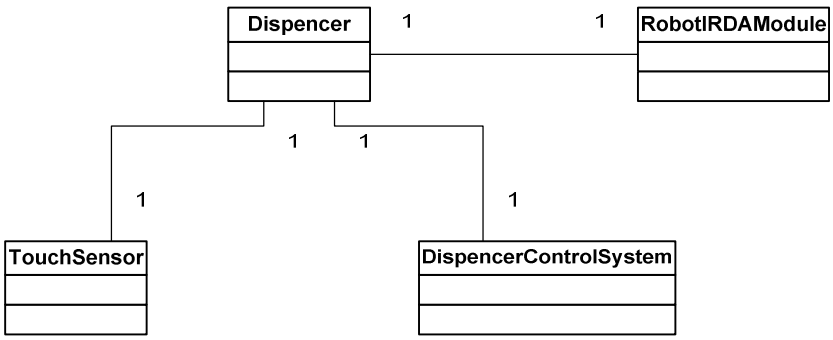

Рис. 51. Диаграмма классов робота-поставщика

5.1.7. Пульт управления

Несмотря на то, что пульт дистанционного управления обычно не рассматривается как отдельный агент, но авторы считают целесообразным рассмотреть диаграмму его классов именно в этом разделе.

5.1.7.1. Класс *RemoteControl*

Данный класс является программной реализацией пульта дистанционного управления.

Методы:

- `RemoteControl(double rad, double angle)`. Создает новый объект. Параметры `rad` и `angle` определяют положение источника инфракрасных сообщений относительно пульта управления.
- `void prepareButton(int button)`. Позволяет выбрать активной кнопку `button`.
- `void buttonPressed()`. Нажать кнопку.
- `InfraredEmitter getEmitter()`. Возвращает объект типа `InfraredEmitter`, соответствующий источнику инфракрасных сообщений пульта управления.

5.1.7.2. Диаграмма классов пульта управления

Диаграмма классов пульта управления представлена на рис. 52.

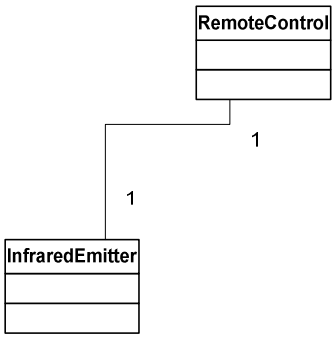

Рис. 52. Диаграмма классов пульта управления

5.1.8. Среда взаимодействия

Взаимодействие роботов-агентов друг с другом производится посредством двух основных классов `BattleField` и `InfraredField`.

5.1.8.1. Класс *BattleField*

Этот класс соответствует полю из проекта *Isenguard*. Он предназначен для того, чтобы вычислять траекторию движения транспортного робота, хранить данные о цвете поля в каждой точке, загружать эти данные в световые датчики транспортного робота, а также отгружать предметы с робота-поставщика на транспортный робот. Данный класс реализует интерфейс `CalcObject`.

Методы:

- `BattleField(int width, int height)`. Создает новый объект с размерами поля `width` и `height`.
- `void start()`. Формирует команду начать эмуляцию
- `void stop()`. Останавливает эмуляцию и возвращает систему с исходное состояние.
- `void pause()`. Приостанавливает систему. При вызове метода `start()` система продолжит эмуляцию с места, на котором она была приостановлена.
- `boolean isSimulation()`. Возвращает значение `true`, если в данный момент идет эмуляция.
- `int getWidth()`. Возвращает ширину поля.
- `int getHeight()`. Возвращает высоту поля.
- `void loadNewData(BattleFieldData data)`. Позволяет загрузить новые параметры поля (путь, места остановки и ошибки, начальные координаты роботов). Если размер загружаемых данных не совпадает с размером поля, то ничего не происходит. При загрузке новых данных агенты и пульт управления возвращаются

в свое исходное состояние. Загрузка производится посредством класса `BattleFieldData`.

- `SixWheeledTransport getTransport()`. Возвращает объект, соответствующий транспортному роботу
- `Dispencer getDispencer()`. Возвращает объект, соответствующий роботу-поставщику.
- `RemoteControl getRemoteControl()`. Возвращает объект, соответствующий пульту дистанционного управления.
- `InfraredField getInfraredField()`. Возвращает объект типа `InfraredField`.
- `BattleFieldData createBattleFieldData()`. Создает объект типа `BattleFieldData` с размером, совпадающим с размером поля.

5.1.8.2. Класс *BattleFieldData*

Класс `BattleFieldData` предназначен для хранения данных о поле и исходных координат роботов. Он является внутренним классом `BattleField`.

Методы:

- `BattleFieldData(int xmin, int xmax, int ymin, int ymax)` Создает новый объект, где координата x может принимать значения от `xmin` до `xmax`, а y – от `ymin` до `ymax`.
- `int getPointAt(int x, int y)`. Получить тип поля (поле, путь, место остановки или ошибки) в точке с координатами `x` и `y`.
- `void setPointAt(int x, int y, int type)`. Установить тип поля в точке с координатами `x` и `y`.
- `double getTransportX()`. Возвращает исходную координату x транспортного робота.
- `double getTransportY()`. Возвращает исходную координату y транспортного робота.
- `double getTransportAngle()`. Возвращает исходное направление транспортного робота.
- `double getDispencerX()`. Возвращает исходную координату x робота-поставщика.
- `double getDispencerY()`. Возвращает исходную координату y робота-поставщика.

- `double getDispencerAngle()`. Возвращает исходное направление робота-поставщика.

- `void setTransportCoords(double x, double y, double angle)`. Выставляет исходные координаты транспортного робота.

- `void setDispencerCoords(double x, double y, double angle)`. Выставляет исходные координаты робота-поставщика.

- `boolean compareSize(BattleFieldData data)`. Возвращает значение `true`, если размеры объектов совпадают.

5.1.8.3. Класс *InfraredField*

Этот класс используется для расчета и передачи инфракрасных сообщений. Реализует интерфейс `CalcObject`. Для работы с ним, в него необходимо загрузить все источники и приемники инфракрасных сообщений.

Методы:

- `InfraredField()`. Создает новый объект. Размеры не требуются, так как никакой информации о поле хранить `InfraredField` не требуется.

- `void addEmitter(InfraredEmitter emitter)`. Добавить источник сообщений.

- `void addReceiver(InfraredReceiver receiver)`. Добавить приемник сообщений.

- `void addReceiverEmitter(InfraredReceiverEmitter emrec)`. Добавить источник и приемник сообщений.

- `List getEmitters()`. Возвращает список источников.

- `List getReceivers()`. Возвращает список приемников.

5.1.8.4. Диаграмма классов среды взаимодействия

Диаграмма классов среды взаимодействия показана на рис. 53.

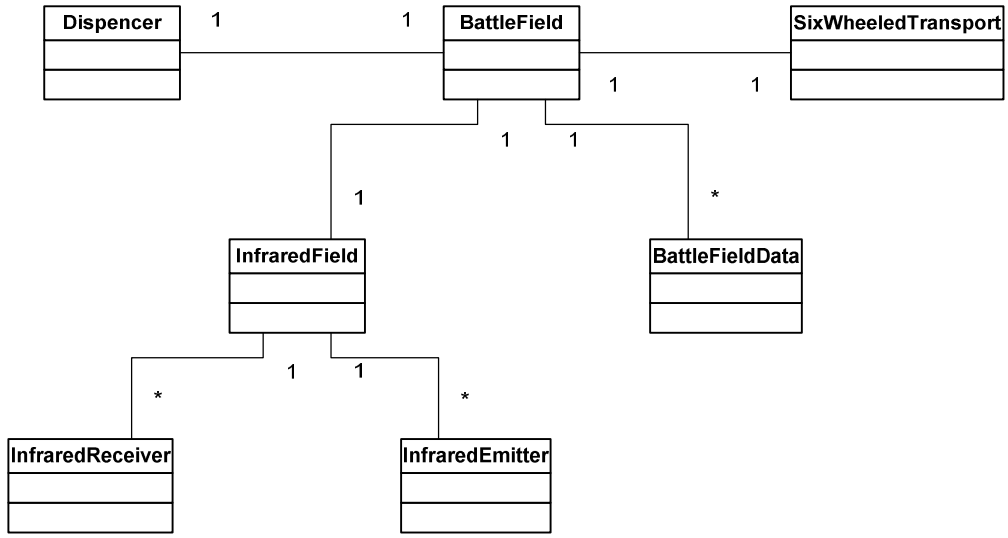

Рис. 53. Диаграмма классов среды взаимодействия

5.1.9. Пользовательский интерфейс

Пользовательский интерфейс реализуется при помощи класса `BattleFieldView` (непосредственно представления `BattleField`). В нем задаются панели управления (внутренние классы `EditPanel` и `ProcessControlPanel`), выполняется отрисовка поля с роботами (внутренний класс) При помощи класса `BattleFieldEditor` выполняется редактирование поля. Класс `RobotDataOutput` используется в управляющей панели (`ProcessControlPanel`) для вывода информации о состоянии роботов.

5.1.9.1. Класс *BattleFieldView*

Реализует интерфейс `Automaton` и наследуется от класса `JFrame`.

Методы:

- `BattleFieldView(int battleWidth, int battleHeight, int viewPortWidth, int viewPortHeight, int controlWidth, int controlHeight)`. Создает новый объект типа `BattleFieldView`. В этом конструкторе по входным параметрам (`battleWidth` и `battleHeight`) создается объект типа `BattleField`, а также инициализируются управляющие панели.

- `BattleFieldView createDefaultBattleFieldView()`. Создает новый объект с параметрами, заданными по умолчанию.

5.1.9.2. Класс *ProcessControlPanel*

Управляющая панель для процесса эмуляции. Реализует интерфейс `Automaton`. Наследуется от класса `JPanel`.

5.1.9.3. Класс *EditPanel*

Управляющая панель для редактирования поля. Реализует интерфейс `Automaton`. Наследуется от класса `JPanel`.

5.1.9.4. Класс *RobotDataOutput*

Панель для вывода информации о роботе. Наследуется от класса `JPanel`.

Методы:

- `RobotDataOutput(FieldRobot robot)`. Создает панель для вывода информации о роботе `robot`.

5.1.9.5. Класс *BattleFieldEditor*

Данный класс описывает логику работы редактора поля. Реализует интерфейс `Automaton`.

Методы:

- `BattleFieldEditor(Image pi, Image fi, Image foiI, Image erI, Color fieldColor)`. Создает новый объект. `pi` – рисунок кисти для поля, `fi` – рисунок кисти для поля, `foiI` – рисунок кисти для фольги, `erI` – рисунок кисти для места ошибки, `fieldColor` – цвет поля.

- `void loadNewImage(Image offscreen)`. Загрузить в редактор изображение, на котором будет производиться отрисовка при редактировании.

- `int getRobotX(int robot)`. Получить текущую координату *x* робота `robot`.

- `int getRobotY(int robot)`. Получить текущую координату *y* робота `robot`.

- `double getRobotAngle(int robot)`. Получить текущее направление робота `robot`.

- `void setRobotCoords(int robot, int x ,int y, double angle)`. Выставить координаты в редакторе для робота `robot`.

5.1.9.6. Класс *BattleFieldDrawPanel*

Этот класс объявлен как внутренний класс `BattleFieldView`. Он наследуется от класса `JPanel` и реализует интерфейсы `MouseListener` и `MouseMotionListener`. В нем выполняется отрисовка поля, роботов и пульта.

5.1.9.7. Диаграмма классов пользовательского интерфейса

Диаграмма классов пользовательского интерфейса изображена на рис. 54.

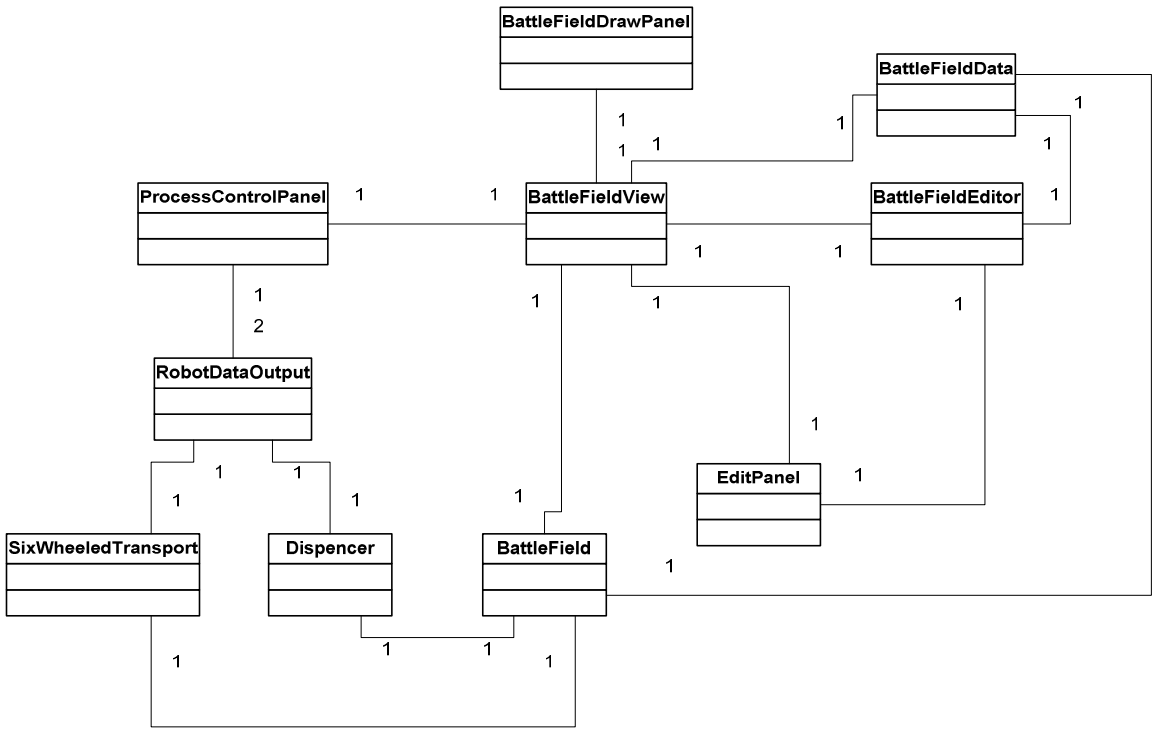

Рис. 54. Диаграмма классов пользовательского интерфейса

5.1.10. Структурные схемы классов

В данном разделе описываются шаблоны построения изоморфного кода по графам переходов, а также структурные особенности реализации классов эмулятора.

5.1.10.1. Построение изоморфного кода по графам переходов

Опишем процесс создания кода по графам переходов автомата. Данный процесс практически идентичен, как для интерфейса `Automaton`, так и для интерфейса `CalcObject`. Небольшие различия будут описаны отдельно.

Рассмотрим граф переходов двух автоматов *A1* и *A2*, изображенных на рис. 55. Пусть им будут соответствовать классы с именами `A1` и `A2`, которые реализуют интерфейс `Automaton`. Пусть объект класса `A2` будет членом класса `A1`. (`private A2 a2auto;`). В этом проекте имеется возможность вызывать из одного автомата другой с определенным событием на переходах. То есть обозначение `A2(e11)` на графе переходов автомата *A1* означает, что автоматная функция автомата *A2* вызывается с событием *e11* на переходах из состояния 1 в состояние 2 и обратно.

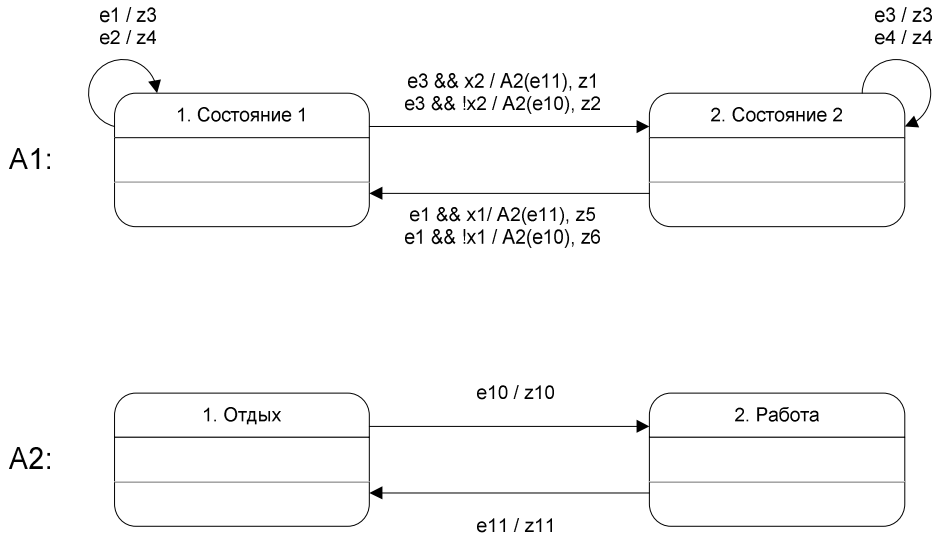

Рис. 55. Графы переходов автоматов

Первым делом в теле каждого из классов необходимо задать константы, соответствующие состояниям автоматов. Значение такой константы совпадает с номером соответствующего состояния на графе переходов. При написании кода для интерфейса `Automaton` к основной произвольной части имени константы добавляется суффикс `_STATE`. При написании кода для интерфейса `CalcObject` к основной произвольной части имени константы добавляется суффикс `_CALC_STATE`.

Константы класса `A1`:

```
public static final int STATE_1_STATE = 1;
public static final int STATE_2_STATE = 2;
```

Константы класса `A2`:

```
public static final int REST_STATE = 1;
public static final int WORK_STATE = 2;
```

После этого следует задание константы для каждого события в системе. Тут можно пойти двумя путями. Если для каждого автомата есть строго определенный набор событий, то можно задавать константы для событий в каждом классе отдельно, иначе можно задать все константы в головном классе (в данном случае в классе `A1`). Реализуем в данном случае второй вариант. При написании кода для интерфейса `Automaton` к основной произвольной части имени константы события добавляется суффикс `_EVENT`. При написании кода для интерфейса `CalcObject` к основной части имени константы события добавляется суффикс `_CALC_EVENT`. Значения констант событий могут быть произвольными, главное, чтобы все они были уникальными. В классе автомата `A1` появляются следующие константы:

```
public static final int E1_EVENT = 1;
public static final int E2_EVENT = 2;
public static final int E3_EVENT = 3;
public static final int E4_EVENT = 4;
public static final int E10_EVENT = 271;
public static final int E11_EVENT = 314;
```

После того, как все константы, соответствующие событиям и состояниям реализованы, перейдем к входным переменным и выходным воздействиям.

Так как входные переменные и выходные воздействия непосредственно друг с другом не связаны, то их можно реализовывать в произвольной последовательности. Начнем со входных переменных.

Каждой входной переменной соответствует один метод класса, возвращающий значение типа `boolean`. Этот метод объявляется как `private` метод. У него нет входных параметров. Его имя образовывается следующим образом: сначала идет имя входной переменной, затем символ подчеркивания "_", и произвольная часть, которая характеризует тип входной переменной. Внутри входной переменной может находиться произвольный код.

Входной переменной `x1` соответствует следующий код:

```
private boolean x1_signalActive() {
    //здесь может быть некий код
}
```

Входной переменной `x2` соответствует следующий код:

```
private boolean x2_receiveComplete() {
    //здесь может быть некий код
}
```

Теперь перейдем к реализации выходных воздействий. Каждому выходному воздействию также соответствует один метод, объявленный как `private` и возвращающий пустое (`void`) значение. Имя метода образовывается следующим образом: сначала идет имя входной переменной, затем символ подчеркивания "_" и, наконец, произвольная часть, которая характеризует тип входной переменной. В отличие от входных переменных, выходные воздействия могут принимать один или несколько различных параметров. Рассмотрим выходное воздействие `z1`, которое не принимает никаких входных параметров. Ему соответствует следующий метод:

```
private boolean z1_startEngine() {
    //здесь может быть некий код
}
```

Выходное воздействие `z2`, принимает, к примеру, два параметра типа `int`. Ему соответствует следующий код:

```
private boolean z2_setAimLocation(int x1, int x2) {
    //здесь может быть некий код
}
```

Для чего была введена такая функциональность, как передача параметров в выходные воздействия? К примеру, необходимо сделать так, чтобы некий автомат обрабатывал события "щелчок" от мыши. Толку от такого события, без координат места, по которому был произведен щелчок немного. При помощи параметров возможно передать в автоматную функцию координаты щелчка, и вызывая метод, соответствующий выходному воздействию обработки события от мыши, передать в него координаты

щелчка. До конца не известно, дает ли использование такой возможности какие-либо преимущества. Но почему бы не попробовать сделать это при разработке эмулятора?

Далее следует написание кода методов `void setAutoState(int state)` и `int getAutoState()` интерфейса `Automaton`, и соответствующих им методов `void setCalcState(int calcState)` и `int getCalcState()` интерфейса `CalcObject`. Прежде всего, надо завести переменную типа `int` для того, чтобы хранить текущее состояние. Для классов, реализующих интерфейс `Automaton`, она должна называться `state`, а для классов, реализующих интерфейс `CalcObject`, она должна называться `calcState`. Данная переменная должна инициализироваться начальным значением состояния автомата. Для автомата `A1` полученный код будет выглядеть следующим образом:

```
private int state = STATE_1_STATE;
public int getAutoState() {
    return state;
}
public void setAutoState(int state) {
    this.state = state;
}
```

Если бы автомат `A1` реализовывал интерфейс `CalcObject`, то в этом случае код выглядел бы следующим образом:

```
private int calcState = STATE_1_CALC_STATE;
public int getCalcState() {
    return calcState;
}
public void setCalcState(int calcState) {
    this.calcState = calcState;
}
```

Теперь перейдем к реализации автоматных функций. Вначале рассмотрим реализацию автоматных функций для интерфейса `Automaton`. Следует отметить, что при проектировании эмулятора не использовались переходы только по входным переменным – каждый переход из одного состояния в другое сопровождается каким-либо событием (за исключением автоматов для управляющих программ роботов, но о них будет рассказано далее в этом разделе). Автоматные функции по графам переходов для интерфейса `Automaton` записываются по схеме, изображенной на рис. 56.

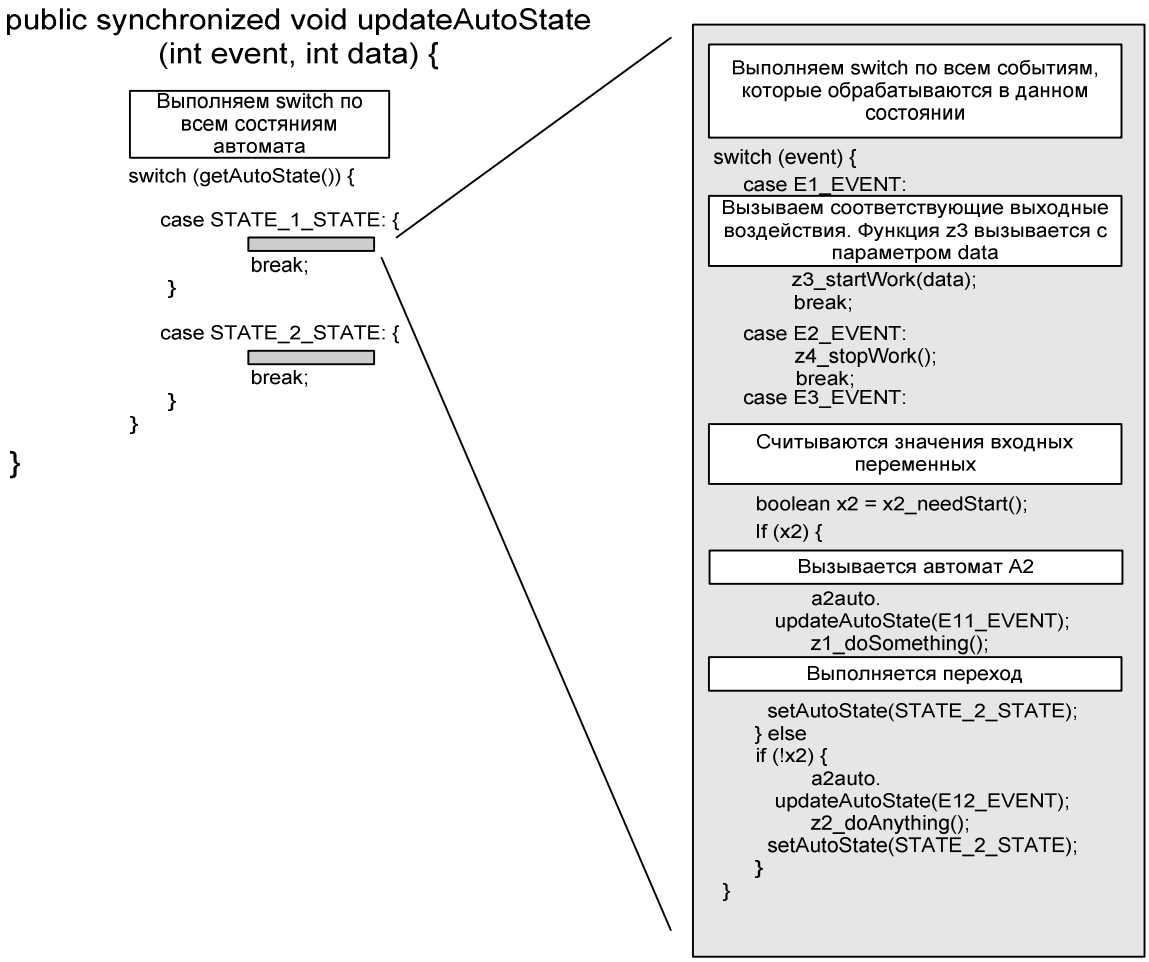

Рис. 56. Схема реализации для интерфейса `Automaton`

По той причине, что в эмуляторе может быть несколько потоков выполнения (основной поток выполнения функции `main`, а также потоки обработки событий от различных графических компонентов), то автоматные функции всегда задаются с ключевым словом `synchronized`. Это необходимо, так как в противном случае, если вдруг несколько потоков одновременно вызовут автоматную функцию, то результат выполнения может быть непредсказуемым.

Отличие в реализации автоматных функций между интерфейсом `Automaton` и `CalcObject` заключаются только в том, что в интерфейсе `CalcObject` функция `void updateCalcState(int event, int data, double delta, Object objData)` принимает большее количество параметров. Это объясняется тем, что сравнительно многим выходным воздействиям при расчете движения требуется промежуток времени, в течение которого необходимо производить расчеты. В качестве данного параметра обычно выступает параметр `delta`. Также была обеспечена возможность передачи объекта в качестве параметра – параметр `objData`. Предположим, что автомат `A2` реализовывал интерфейс `CalcObject`, тогда схема построения автоматной функции для него была бы следующей (рис. 57):

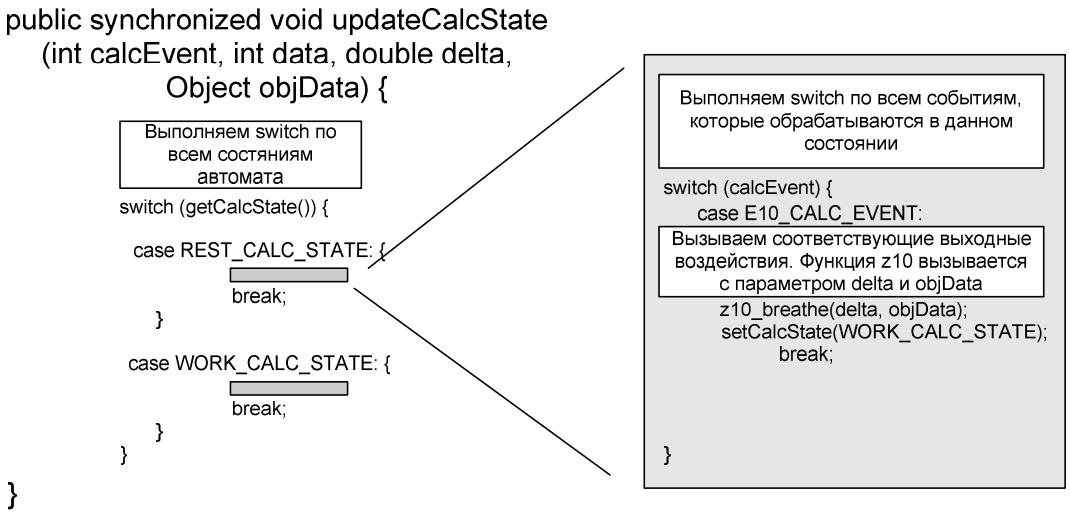

Рис. 57. Схема реализации для интерфейса `CalcObject`

Стоит отметить один недостаток данного подхода – при проектировании пользователю приходится самому проверять, являются ли классы, реализующие автоматы, безопасными от взаимоблокировки (*deadlock*). Так к примеру может произойти, если из одной автоматной функции мы пытаться вызвать другую автоматную функцию, а из другой функции, находящейся в другом потоке выполнения, – первую. В этом случае, а также в некоторых других, может возникнуть взаимоблокировка (*deadlock*).

Теперь рассмотрим построение кода для автоматов управляющих систем роботов. По той причине, что *Java* под платформу *lejos*, применяемой для рассматриваемых роботов, была немного урезанной и измененной, шаблон, по которому составлялись автоматы управляющих систем для эмулятора, был также изменен. В данном случае был взят автомат, который использовался в качестве примера в проекте *Isenguard* (рис. 58). Шаблоны для этого автомата изображены на рис. 59 и на рис. 60.

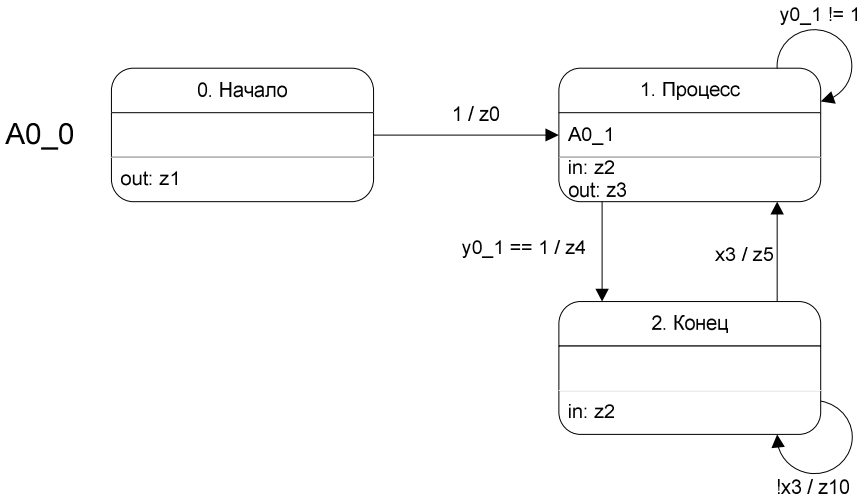

Рис. 58. Пример управляющего автомата

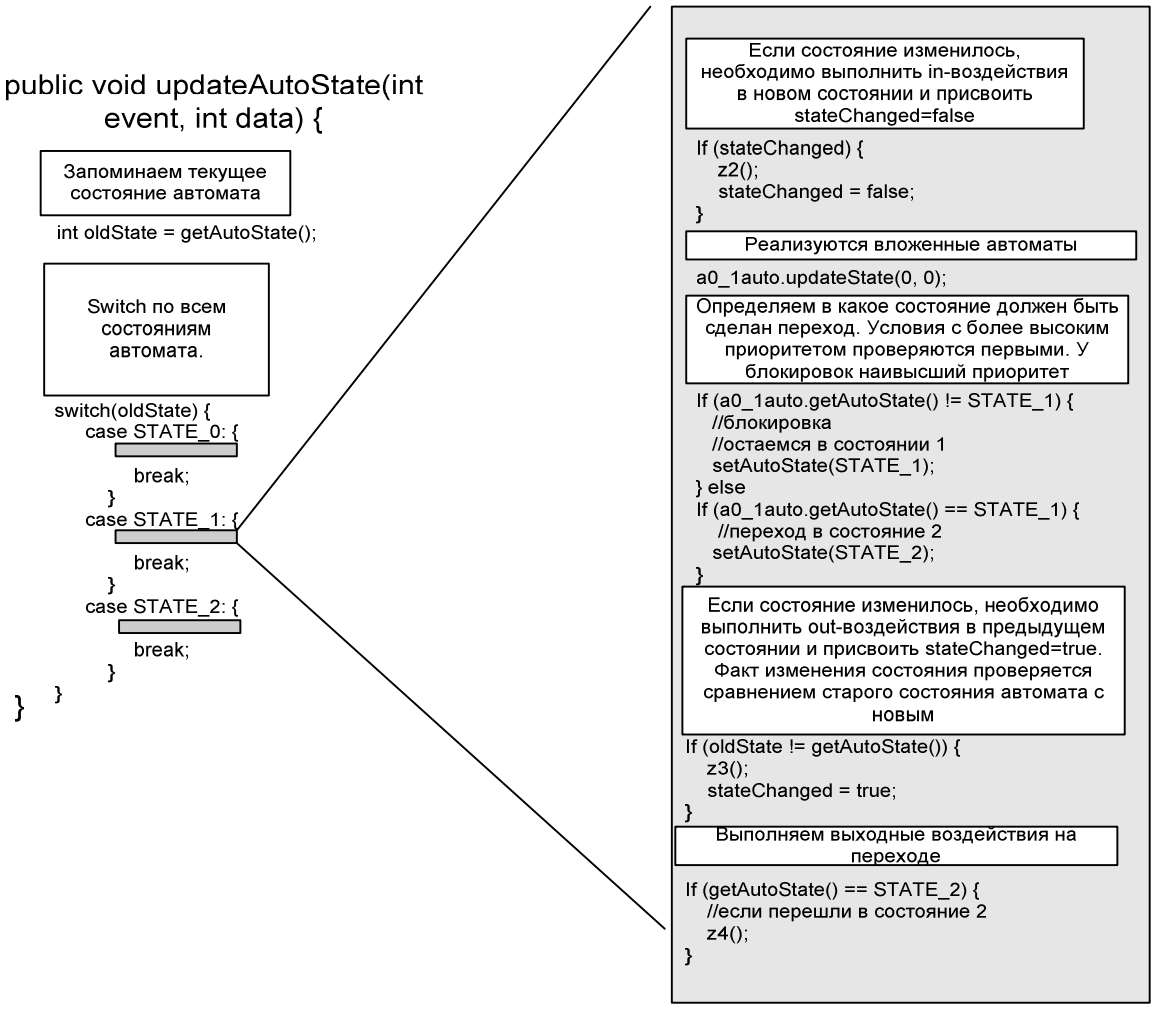

Рис. 59. Схема реализации для автоматов управляющих программ

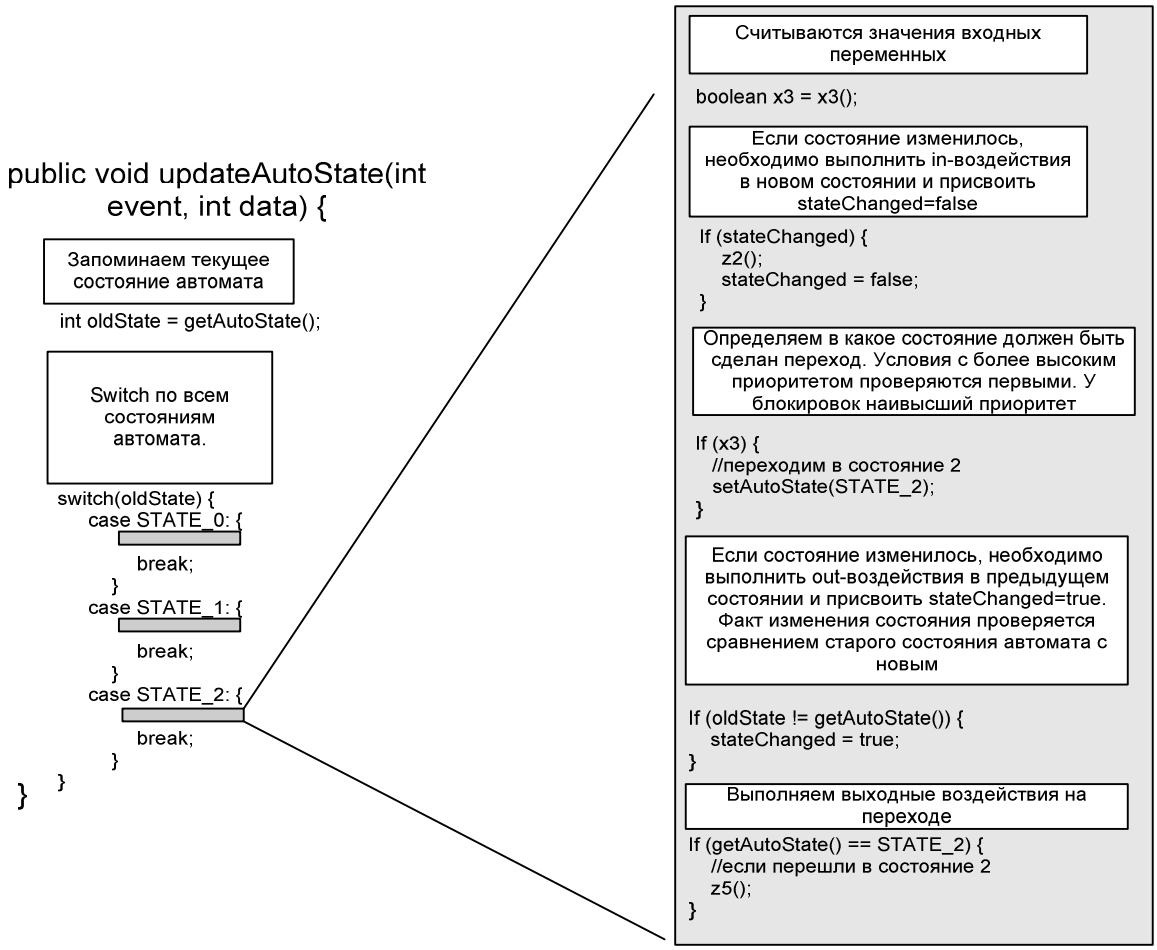

Рис. 60. Схема реализации для автоматов управляющих программ

Состояния управляющих автоматов называются просто: `STATE_1`, `STATE_2`, `STATE_3`. Входные переменные и выходные воздействия также реализуются как `private` методы соответствующего класса. Стоит отметить один важный момент – в проекте *Isenguard* была реализована система передачи команд от одного автомата к другому. Она была не очень удобна для использования. Вызывать автоматные функции другого автомата с определенным событием на переходах гораздо удобнее. Но, несмотря на это, механизм передачи команд все равно пришлось написать.

Механизм реализуется при помощи метода `void addCommand(int autoId, int command)`. Этот метод присутствует в каждом управляющем автомате. Переменная `autoId` – это уникальный код автомата, от которого дается команда, а переменная `command` – это код самой команды. В этом методе сначала с помощью оператора `switch` идет определение автомата-источника посылаемой команды, а затем следует определение типа команды, что также выполняется при помощи оператора `switch`. Когда определена команда, то соответствующей булевской переменной присваивается значение `true`.

Рассмотрим, для примера, код данного метода для управляющего автомата `A2` транспортного робота.

```
public synchronized void addCommand(int autoId, int commandId) {
    switch(autoId) {
        case A1_ID: //Получили команду от автомата A1
```

```
            if (A2_A1_Listener) { //Команды от A1 разрешены
                switch(commandId) {

                    //Команда "Следовать по пути"
                    case A1Auto.A1_TRACE_PATH_COMMAND:
                        A2_TracePath = true;
                        break;
                }
            }
            break;
        case A3_ID: //Получили команду от автомата A3
            if (A2_A3_Listener) { //Команды от A3 разрешены
                switch(commandId) {
                    //Команда "Путь найден слева"
                    case A3Auto.A3_PATH_LEFT_COMMAND:
                        A2_PathLeft = true;
                        break;
                    //Команда "Путь найден справа"
                    case A3Auto.A3_PATH_RIGHT_COMMAND:
                        A2_PathRight = true;
                        break;
                    }
                }
            break;
    }
}
```

5.1.10.2. Структурная схема транспортного робота

Основной класс – это класс `SixWheeledTransport`. Объекты классов `LineTraceControlSystem`, `Rod`, `LightSensor`, `TouchSensor` и `RobotIRDAModule` являются его членами. Объекты `A1Auto`, `A2Auto`, `A3Auto`, `A5Auto` являются членами класса `LineTraceControlSystem`. Также для датчиков и штанги хранятся начальные координаты как члены класса.

```
private LightSensor fixedLightSensor;
private double fixedLightRadius = 20;
private double fixedLightAngle = 0;
private LightSensor rodLightSensor;
private Rod rod;
private double rodRadius = 25;
private double rodAngle = 0.0;
private TouchSensor touchSensor;
private RobotIRDAModule irdaModule;
private double irdaModuleRadius = 15;
private double irdaModuleAngle = 0;
private LineTraceControlSystem system;
```

Для получения изображения транспорта с объектами для доставки, в методе `Image getObjectImage()` на копию изображения транспорта наносятся изображения предметов к доставке.

Для расчета процесса движения были введены две константы, которые и являются теми константами, которые используются для пересчета мощности в скорость движения, соответствующую этой мощности:

```
private double speedPerPowerUnit = 5;
private double torquePerPowerUnit = 0.05;
```

Еще две константы определяют ускорение (линейное и угловое) которое возникает, когда робот движется линейно или поворачивает:

```
private double speedAcc = 20;
private double torqueAcc = 0.1;
```

Факт того, что датчик касания нажат, определяется по положению штанги. Таким образом, если штанга отклоняется от своего исходного положения не больше чем на некоторый заранее определенный угол, то датчик считается нажатым, в противном случае он нажатым не считается.

```
if (Math.abs(rod.getAngle() - rod.getInitialAngle())
    % (Math.PI * 2) < sensorTouchAngle) {
    touchSensor.touch();
} else {
    touchSensor.untouch();
}
```

Расчет положения робота выполняется при помощи явной схемы Ньютона:

```
setX(getX() + speed * Math.cos(getAngle()) * timeDelta);
setY(getY() + speed * Math.sin(getAngle()) * timeDelta);
```

Перед расчетом новых координат производится расчет новой линейной и угловой скоростей робота. Покажем, как рассчитывается линейная скорость при движении вперед:

```
if (speed < speedPerPowerUnit * power) {
    speed += speedAcc * timeDelta;
} else {
    speed -= speedAcc * timeDelta;
}
```

Расчет движения штанги производится аналогичным способом. Вся оставшаяся логика работы классов, реализующих агента, соответствующего транспортному роботу, заключена в автоматах.

5.1.10.3. Структурная схема робота поставщика

Основной класс – это класс `Dispencer`. Объекты классов `DispencerControlSystem`, `RobotIRDAModule` являются его членами.

Объект `A0Auto` является членом класса `DispencerControlSystem`.

```
private TouchSensor touchSensor;
private RobotIRDAModule irdaModule;
private double irdaModuleRadius = 22;
```

```
private double irdaModuleAngle = 7 * Math.PI / 4;
private double irdaModuleDirection = Math.PI;
private DispencerControlSystem system = null;
```

Для получения изображения робота-поставщика с объектами, движущимися по конвейеру в методе `Image getObjectImage()` на копию изображения робота-поставщика наносятся изображения предметов к доставке. Для хранения координат изображений предметов, движущихся по конвейеру поставщика, используется вспомогательный класс `Conveyor`. Данный класс является внутренним классом класса `Dispencer`. Он хранит текущие координаты предметов.

5.1.10.4. Структурная схема среды взаимодействия

Основным классом является класс `BattleField`. Объект класса `InfraredField` является членом этого класса. Объекты, соответствующие транспортному роботу, роботу-поставщику и пульту дистанционного управления также являются членами данного класса. Данные о пути и положениях роботов загружаются в объект `BattleField` при помощи объекта `BattleFieldData`.

```
…
private SixWheeledTransport transport;
private Dispencer dispencer;
private RemoteControl remoteControl;
private BattleFieldData battleData;
private InfraredField infraredField;
…
```

Данные о поле хранятся в двумерном массиве, являющимся членом класса `BattleFieldData`.

```
private byte [][] points;
```

Размер создаваемого массива точек поля задается в конструкторе объекта `BattleFieldData`.

```
private BattleFieldData(int xmin, int xmax, int ymin, int ymax);
```

Передача инфракрасных сообщений осуществляется в классе `InfraredField`. Возможность передачи сообщения от источника приемнику определяется при помощи следующего метода, возвращающего значение типа `boolean`:

```
private boolean transmitPossible(double emX, double emY, double
emAngle, double emAlpha, double emRadius, double recX, double
recY, double recAngle, double recAlpha)
```

Добавлять передатчики и приемники в класс `InfraredField` можно при помощи следующих методов:

```
public void addEmitter(InfraredEmitter emitter);
public void addReceiver(InfraredReceiver receiver);
public void addReceiverEmitter(InfraredReceiverEmitter irrecem);
```

5.1.10.5. Структурная схема классов пользовательского интерфейса

Основным классом является класс `BattleFieldView`. Объекты классов `BattleField`, `EditPanel`, `ProcessControlPanel`, `BattleFieldDrawPanel`, `BattleFieldData` являются его членами. Объект класса `RobotDataOutput` является членом класса `ProcessControlPanel`.

```
…
private BattleField battleField;
private BattleField.BattleFieldData data;
private BattleFieldDrawPanel drawPanel;
private ProcessControlPanel processControlPanel;
private EditPanel editPanel;
private BattleFieldEditor editor;
…
```

Данные об объектах на поле при редактировании в классе `BattleFieldEditor` хранятся как `Image`. Исходные координаты роботов – как `int` и `double`. Далее, если пользователь подтвердил свои изменения, сделанные в редакторе, то в классе `BattleFieldView` данные преобразуются из формата `Image` в объект `BattleFieldData` и загружаются в объект `BattleField`.

В классе `BattleFieldDrawPanel` с помощью метода `paintComponent(Graphics g)` производится отрисовка поля. В данном методе последовательно считывается информация обо всех объектах на поле, и при необходимости данные объекты отрисовываются. Вот как в этом классе выполняется отрисовка транспортного робота (методы `transFieldToViewX(double)` и `transFieldToViewY(double)` являются методами класса `BattleFieldView` и преобразуют координаты поля к экранным координатам):

```
if (transport.isShown()) {
    drawImageAtPosition(ig, transport.getObjectImage(),
        transFieldToViewX(transport.getX()),
        transFieldToViewY(transport.getY()),
        transport.getAngle());
}
```

Для того чтобы при отрисовке не приходилось постоянно преобразовывать изображение поля из `BattleFieldData`, изображение поля отдельно хранится еще в формате `Image`.

Двукратное увеличение выполняется путем преобразования каждого пикселя полученного изображения в четыре при помощи растяжения в объекте `ImageRaster`, а затем вывода этого объекта на экран.

Передача параметров о координатах мыши в автоматную функцию выполняется следующим образом:

```
//Обработчик события "нажатие" от мыши
public void mousePressed(MouseEvent e) {
    if (e.getButton() == MouseEvent.BUTTON1) {
        //Преобразование координат
        int xCoord = transScreenToViewX(e.getX());
```

```
            int yCoord = transScreenToViewY(e.getY());

            //Вызов автоматной функции
            BattleFieldView.this.updateAutoState(
            E19_LEFT_MOUSE_PRESSED_EVENT, (xCoord << 16) + yCoord);
        }
}
```

5.1.11. Описание автоматов транспортного робота

При проектировании модели транспортного робота было использованы следующие автоматы: `LightSensor`, `TouchSensor`, `InfraredEmitter`, `InfraredReceiver`, `RobotIRDAModule` (на графах переходов сокращенно `IMC` (пассивный автомат) и `IMA` (активный автомат)), `Rod` (на графах переходов сокращенно `R`), `LineTraceControlSystem` (на графах переходов сокращенно `S`) и `SixWheeledTransport` (основной автомат). У каждого автомата свой собственный список событий. Есть события, являющихся общими для всех пассивных автоматов (события интерфейса `CalcObject`).

5.1.11.1. Общие события

- `e100` – Обновить координаты. Параметр `delta` (типа `double`) – интервал времени.
- `e101` – Загрузить исходные параметры.

5.1.11.2. Автомат *LightSensor*

Данный автомат реализует интерфейс `Automaton`. В его задачи входит определение текущего режима работы светового датчика – активирован или не активирован.

Список событий

- `e1` – Активировать датчик.
- `e2` – Деактивировать датчик.
- `e3` – Обновить значение. Параметр `data` (типа `int`) – новое значение, считаваемое датчиком.

Выходные воздействия

- `z1 (value)` – Обновить значение светового датчика. Параметр `value` – новое значение.

Схема связей

Схема связей автомата `LightSensor` изображена на рис. 61.

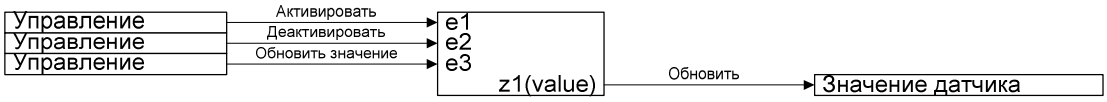

Рис. 61. Схема связей автомата `LightSensor`

Граф переходов

Граф переходов автомата `LightSensor` изображен на рис. 62.

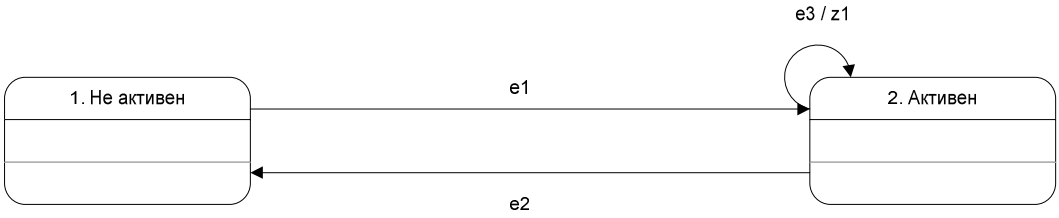

Рис. 62. Граф переходов автомата `LightSensor`

5.1.11.3. Автомат *TouchSensor*

Данный автомат реализует интерфейс `Automaton`. Он определяет текущее состояние датчика касания – нажат или не нажат.

Список событий

- `e1` – Нажать датчик.
- `e2` – Отпустить датчик.

Схема связей

Схема связей автомата `TouchSensor` изображена на рис. 63.

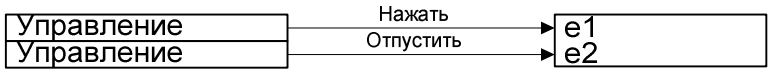

Рис. 63. Схема связей автомата `TouchSensor`

Граф переходов

Граф переходов автомата `TouchSensor` изображен на рис. 64.

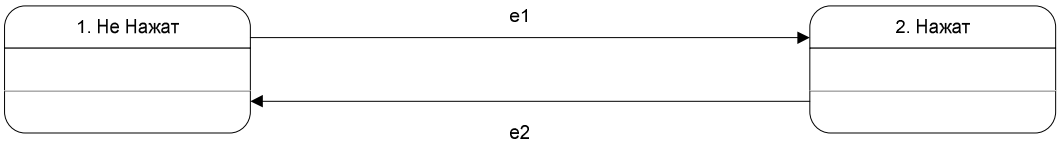

Рис. 64. Граф переходов автомата `TouchSensor`

5.1.11.4. Автомат класса *Rod*

Данный автомат реализует интерфейс `CalcObject`. Данный автомат отвечает за расчет движения и координат штанги при поиске пути.

Список событий

- `e10` – Начать поворот штанги влево. Параметр `data` – мощность двигателя.
- `e11` – Начать поворот штанги вправо. Параметр `data` – мощность двигателя.
- `e12` – Остановить штангу.
- `e13` – Обновить мощность работы двигателя штанги. Параметр `data` – мощность двигателя.
- `e14` – Начать движение штанги в обратном направлении.

Выходные воздействия

- `z0(power)` – Обновить значение мощности двигателя. Параметр `power` – новое значение мощности двигателя.
- `z1` – Запустить с исходными параметрами.
- `z3_1(delta)` – Рассчитать угловую скорость при движении влево. Параметр `delta` (типа `double`) – временной интервал.
- `z3_2(delta)` – Рассчитать угловую скорость при повороте вправо. Параметр `delta` (типа `double`) – временной интервал.
- `z3_3(delta)` – Рассчитать угловую скорость при торможении. Параметр `delta` (типа `double`) – временной интервал.
- `z3_4(delta)` – Рассчитать новые угловые координаты штанги. Параметр `delta` (типа `double`) – временной интервал.

Схема связей

Схема связей автомата `Rod` изображена на рис. 65.

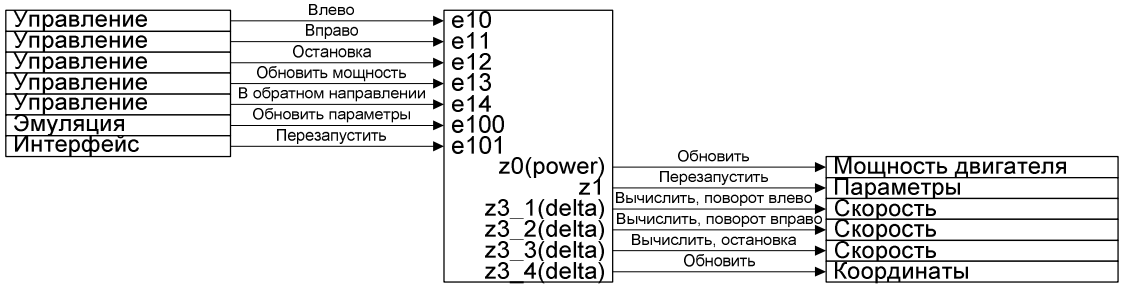

Рис. 65. Схема связей автомата `Rod`

Граф переходов

Граф переходов автомата `Rod` изображен на рис. 66.

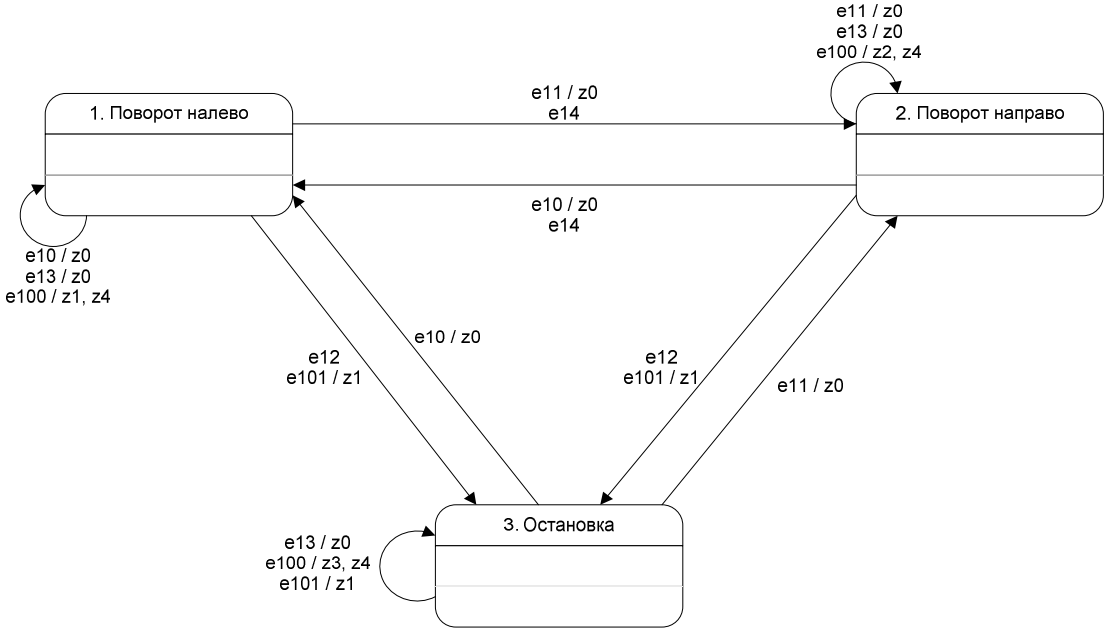

Рис. 66. Граф переходов автомата `Rod`

5.1.11.5. Автомат *InfraredEmitter* (пассивный автомат)

Данный автомат реализует интерфейс `CalcObject` и отвечает за генерацию временных импульсов, для того чтобы осуществлять периодическую передачу инфракрасного сообщений.

Список событий

- `e1` – Сбросить таймер.
- `e2` – Остановить таймер.
- `e3` – Запустить таймер.

Список входных переменных

- `x10` – Таймер превысил значение `emittingPeriod`.

Список выходных воздействий

- `z10(delta)` – Обновить таймер. Параметр `delta` (типа `double`) – временной интервал.
- `z11` – Сбросить текущее значение таймера.
- `z12` – Вызвать автомат `InfraredEmitter` (активный) с событием `e4`.

Схема связей

Схема связей автомата `InfraredEmitter` (пассивный) изображена на рис. 67.

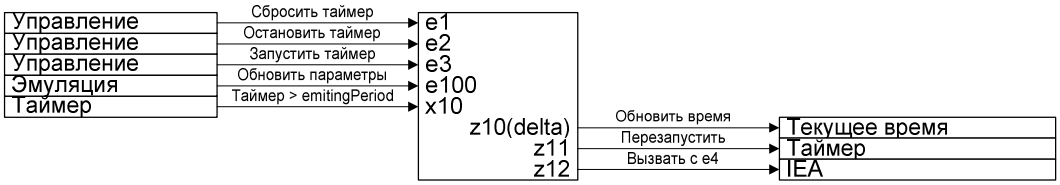

Рис. 67. Схема связей автомата `InfraredEmitter` (пассивный)

Граф переходов

Граф переходов автомата `InfraredEmitter` (пассивный) изображен на рис. 68.

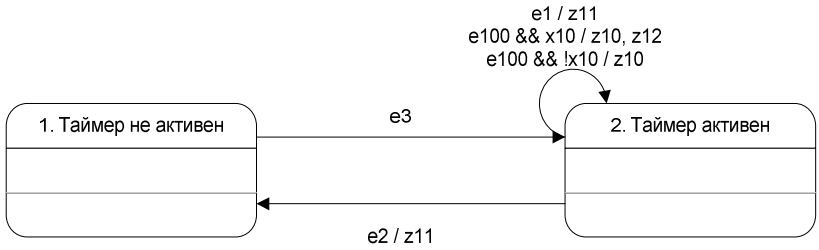

Рис. 68. Граф переходов автомата `InfraredEmitter` (пассивный)

5.1.11.6. Автомат *InfraredEmitter* (активный автомат)

Данный автомат реализует интерфейс `Automaton` и управляет логикой процесса посылки сообщений по инфракрасному порту. Он получает события от пассивного автомата.

Список событий

- `e1` – Начать передачу сообщений. Параметр `data` (типа `int`) – значение для передачи.
- `e2` – Окончить передачу.
- `e3` – Передаваемое сообщение успешно считано.
- `e4` – Сработал таймер (событие от автомата `InfraredEmitter` (пассивный)).

Список выходных воздействий

- `z0(data)` – сохранить значение для передачи. Параметр `data` (типа `int`) – значение для передачи.

Схема связей

Схема связей автомата `InfraredEmitter` (пассивный) изображена на рис. 69.

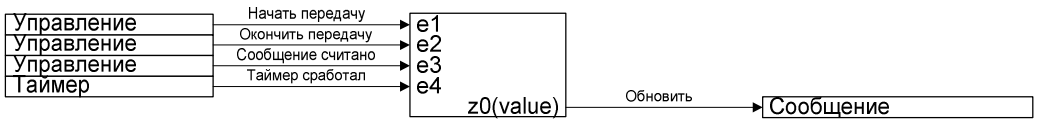

Рис. 69. Схема связей автомата `InfraredEmitter` (пассивный)

Граф переходов

Граф переходов автомата `InfraredEmitter` (пассивный) изображен на рис. 70.

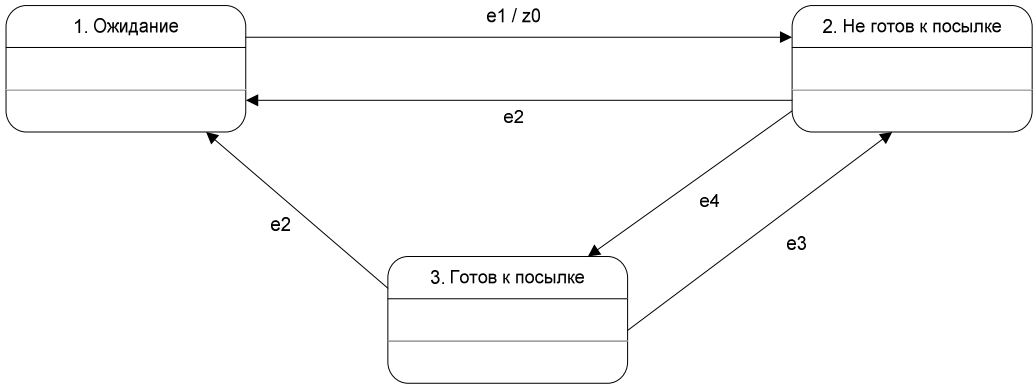

Рис. 70. Граф переходов автомата `InfraredEmitter` (пассивный)

5.1.11.7. Автомат *InfraredReceiver*

Данный автомат управляет логикой процесса прием сообщений от инфракрасного порта.

Список событий

- `e1` – Получено новое сообщение. Параметр `data` (типа `int`) – значение, передаваемое в сообщении.
- `e2` – Убрать сообщение из очереди.
- `e3` – Деактивировать приемник.
- `e4` – Активировать приемник.

Список выходных воздействий

- `z0(data)` – Добавить сообщение в очередь. Параметр `data` (типа `int`) – значение, переданное в сообщении.
- `z1` – Вызвать объекты, реализующие интерфейс `AutomatonUpdator`.
- `z2` – Убрать сообщение из очереди.

Схема связей

Схема связей автомата `InfraredReceiver` представлена на рис. 71.

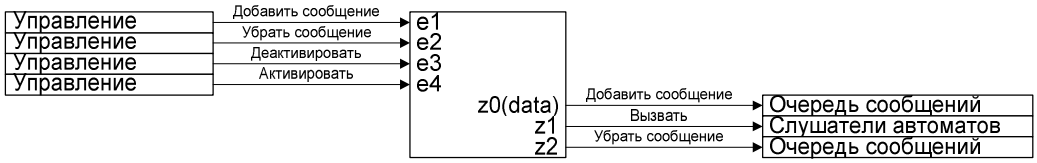

Рис. 71. Схема связей автомата `InfraredReceiver`

Граф переходов

Граф переходов автомата `InfraredReciever` представлен на рис. 72.

Рис. 72. Граф переходов автомата `InfraredReceiver`

5.1.11.8. Автомат *RobotIRDAModule* (пассивный автомат)

Данный автомат реализует интерфейс `CalcObject` и управляет таймерами – при его помощи отсчитывается время, в течение которого производится ожидание, а затем выполняется передача сигнала по инфракрасному порту.

Список входных переменных

- `x11` – Таймер ожидания превысил значение `waitPeriod`.
- `x12` – Таймер передачи превысил значение `sendPeriod`.

Список выходных воздействий

- `z11(delta)` – Обновить значения таймеров. Параметр `delta` (типа `double`) – временной интервал.
- `z4` – Сбросить таймер ожидания.
- `z5` – Сбросить таймер передачи.

Схема связей

Схема связей автомата `RobotIRDAModule` (пассивный) представлена на рис. 73.

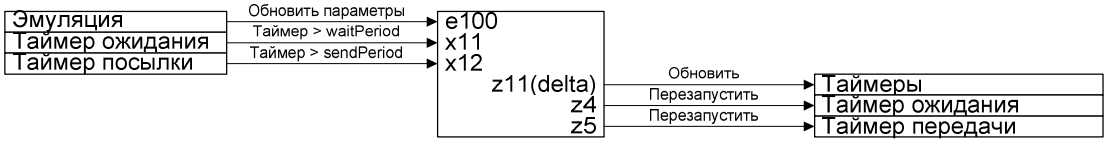

Рис. 73. Схема связей автомата `RobotIRDAModule` (пассивный)

Граф переходов

Граф переходов автомата RobotIRDAModule (пассивный) представлен на рис. 74.

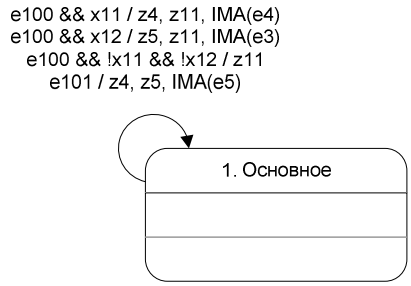

Рис. 74. Схема связей автомата `RobotIRDAModule` (пассивный)

5.1.11.9. Автомат *RobotIRDAModule* (пассивный автомат)

Данный автомат реализует интерфейс `Automaton` и управляет логикой процесса посылки сообщений по инфракрасному порту. Он получает события от пассивного автомата.

Список событий

- `e1` – Послать сообщение по инфракрасному порту. Параметр `data` (типа `int`) – значение, переданное в сообщении.
- `e2` – Получено сообщение от приемника.
- `e3` – Сработал таймер посылки.
- `e4` – Сработал таймер ожидания.
- `e5` – Перезагрузить с исходными параметрами.

Список выходных воздействий

- `z1` – Начать передачу сообщения.
- `z2` – Прекратить передачу сообщения.
- `z3` – Сохранить полученное сообщение.
- `z4` – Перезапустить таймер ожидания.
- `z5` – Перезапустить таймер посылки.
- `z7(message)` – Сохранить сообщение для передачи. Параметр `message` (типа `int`) – значение для передачи.

Схема связей

Схема связей автомата `RobotIRDAModule` изображена на рис. 75.

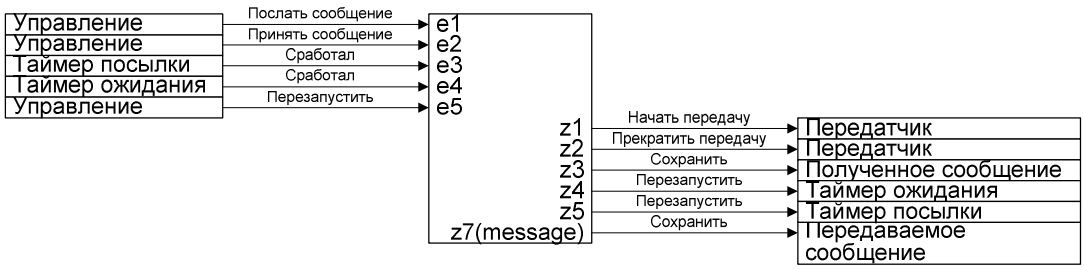

Рис. 75. Схема связей автомата `RobotIRDAModule` (активный)

Граф переходов

Граф переходов автомата `RobotIRDAModule` изображен на рис. 76.

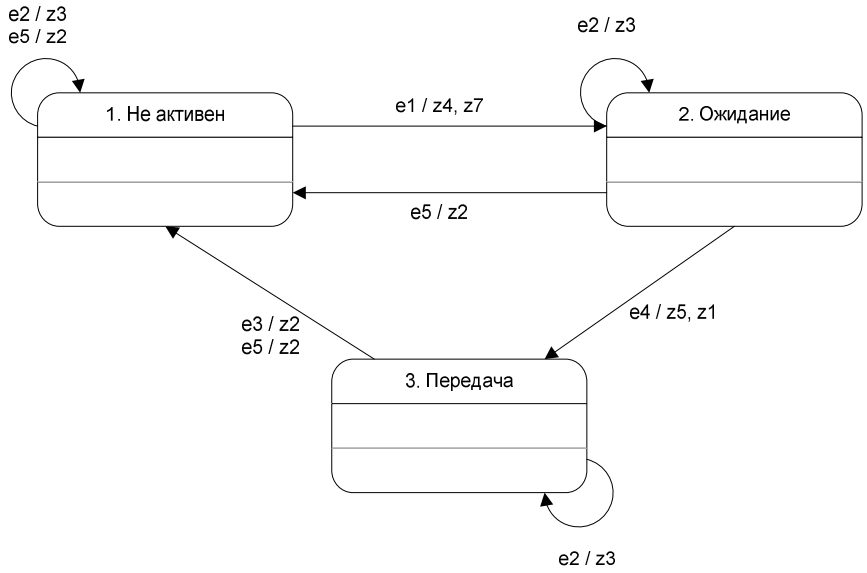

Рис. 76. Граф переходов автомата `RobotIRDAModule` (активный)

5.1.11.10. Автомат *LineTraceControlSystem*

Данный автомат реализует интерфейс `CalcObject`. С помощью этого автомата производится работа с управляющей системой транспортного робота.

Выходные воздействия

- `z0(delta)` – Обновить параметры системы. Параметр `delta` (типа `double`) – временной интервал.
- `z2` – Вызвать автоматные функции управляющих автоматов.
- `z3` – Перезапустить с исходными параметрами.

Схема связей

Схема связей автомата `LineTraceControlSystem` изображена на рис. 77.

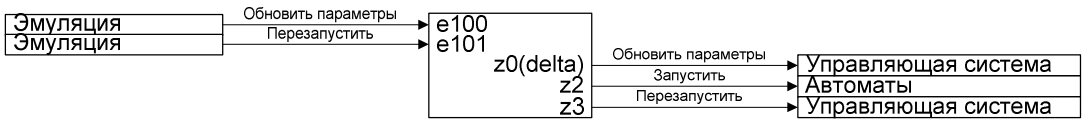

Рис. 77. Схема связей автомата `LineTraceControlSystem`

Граф переходов

Граф переходов автомата `LineTraceControlSystem` изображен на рис. 78.

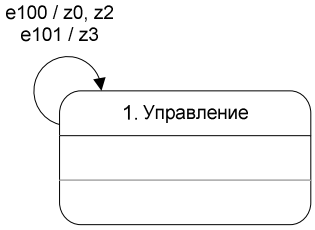

Рис. 78. Граф переходов автомата `LineTraceControlSystem`

5.1.11.11. Автомат *SixWheeledTransport*

Данный автомат реализует интерфейс `CalcObject` и отвечает за расчет движения транспортного робота.

Список событий

- `e1` – Начать движение вперед. Параметр `data` (типа `int`) – мощность двигателей.
- `e2` – Начать движение назад. Параметр `data` (типа `int`) – мощность двигателей.
- `e3` – Начать поворот влево. Параметр `data` (типа `int`) – мощность двигателей.
- `e4` – Начать поворот вправо. Параметр `data` (типа `int`) – мощность двигателей.
- `e5` – Остановиться.
- `e6` – Обновить мощность. Параметр `data` (типа `int`) – мощность двигателей.

Список выходных воздействий

- `z0(power)` – Обновить значение мощности работы двигателей. Параметр `power` (типа `int`) – мощность двигателей.
- `z1` – Перезапустить с исходными параметрами.
- `z2` – Обновить текущее время.
- `z3_1(delta)` – Рассчитать скорость при движении вперед. Параметр `delta` (типа `double`) – временной интервал.

- `z3_2(delta)` – Рассчитать скорость при движении назад. Параметр `delta` (типа `double`) – временной интервал.

- `z3_3(delta)` – Рассчитать скорость при повороте влево. Параметр `delta` (типа `double`) – временной интервал.

- `z3_4(delta)` – Рассчитать скорость при повороте вправо. Параметр `delta` (типа `double`) – временной интервал.

- `z3_5(delta)` – Рассчитать скорость при торможении. Параметр `delta` (типа `double`) – временной интервал.

- `z3_6(delta)` – Рассчитать новые координаты робота. Параметр `delta` (типа `double`) – временной интервал.

Схема связей

Схема связей автомата `SixWheeledTransport` изображена на рис. 79.

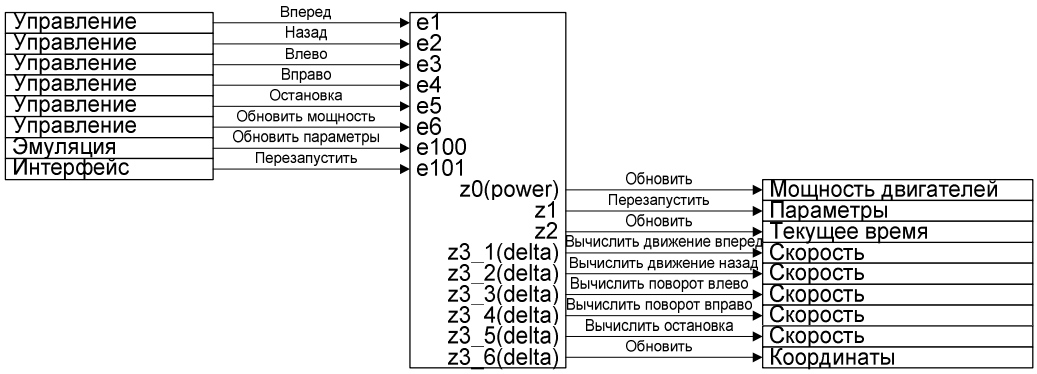

Рис. 79. Схема связей автомата `SixWheeledTransport`

Граф переходов

Граф переходов автомата `SixWheeledTransport` изображен на рис. 80.

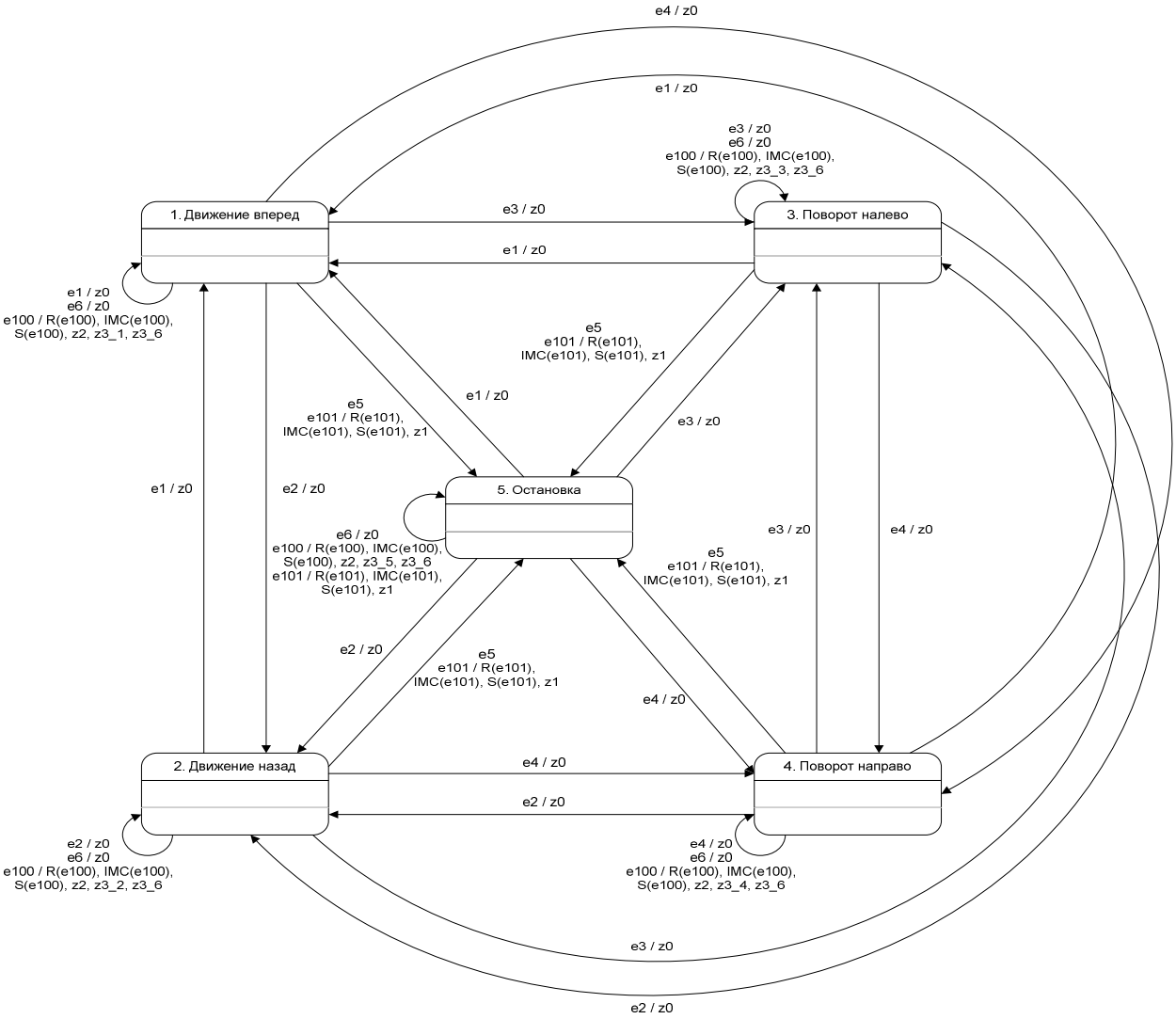

Рис. 80. Граф переходов автомата `SixWheeledTransport`

5.1.12. Описание автоматов робота-поставщика

При проектировании модели робота-поставщика были использованы следующие автоматы: `TouchSensor`, `InfraredEmitter`, `InfraredReceiver`, `RobotIRDAModule` (на графах переходов сокращенно `IMC` (пассивный автомат) и `IMA` (активный автомат)), `DispencerControlSystem` (на графах переходов сокращенно `DS`) и основной автомат `Dispencer`. У каждого автомата свой собственный список событий. Есть события, являющихся общими для всех пассивных автоматов (события интерфейса `CalcObject`). Они совпадают с общими событиями, объявленными в предыдущем разделе.

5.1.12.1. Общие события

- `e100` – Обновить координаты. Параметр `delta` (типа `double`) – интервал времени.
- `e101` – Загрузить исходные параметры.

5.1.12.2. Автомат *DispencerControlSystem*

Данный автомат реализует интерфейс `CalcObject`. Через него производится работа с управляющей системой робота-поставщика.

Выходные воздействия

- `z0(delta)` – Обновить параметры системы. Параметр `delta` (типа `double`) – временной интервал.
- `z2` – Вызвать автоматные функции управляющих автоматов.
- `z3` – Перезапустить с исходными параметрами.

Схема связей

Схема связей автомата `DispencerControlSystem` изображена на рис. 81.

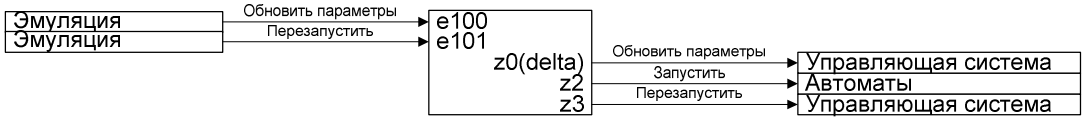

Рис. 81. Схема связей автомата `DispencerControlSystem`

Граф переходов

Граф переходов автомата `DispencerControlSystem` изображен на рис. 82.

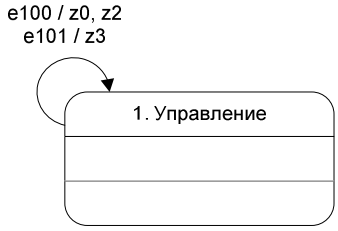

Рис. 82. Граф переходов автомата `DispencerControlSystem`

5.1.12.3. Автомат *Dispencer*

Данный автомат реализует интерфейс `CalcObject` и отвечает за работу робота-поставщика – отгрузку предметов.

Список событий

- `e1` – Начать выдачу предметов. Параметр `data` (типа `int`) – мощность двигателей.
- `e2` – Окончить выдачу предметов.

Список выходных воздействий

- `z0(power)` – Обновить значение мощности работы двигателей. Параметр `power` (типа `int`) – мощность двигателей.
- `z1` – Перезапустить с исходными параметрами.
- `z2` – Обновить текущее время.
- `z3(delta)` – Рассчитать положение конвейера. Параметр `delta` (типа `double`) – временной интервал.

Схема связей

Схема связей автомата `Dispencer` изображена на рис. 83.

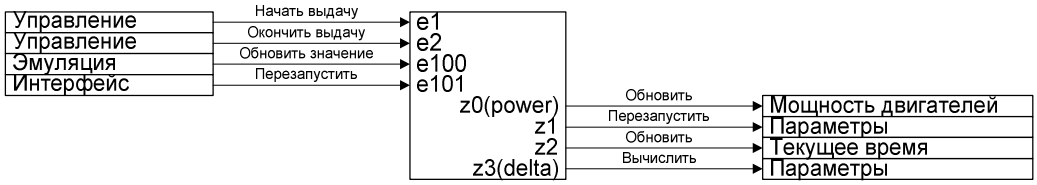

Рис. 83. Схема связей автомата `Dispencer`

Граф переходов

Граф переходов автомата `Dispencer` изображен на рис. 84.

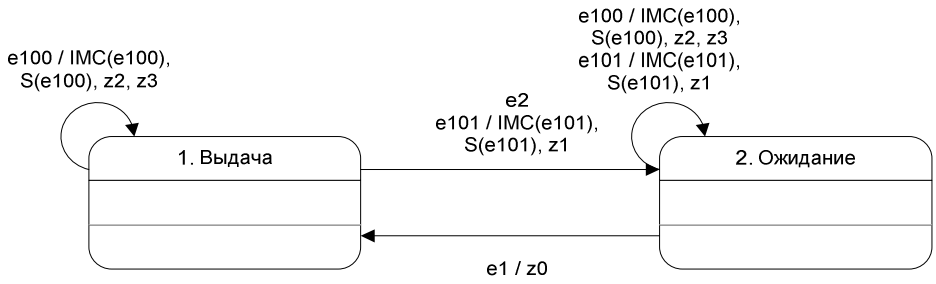

Рис. 84. Граф переходов автомата `Dispencer`

5.1.13. Описание автомата пульта

При проектировании модели пульта были использованы следующие автоматы: `InfraredEmitter`, и основной автомат `RemoteControl`.

5.1.13.1. Автомат *RemoteControl*

Данный автомат реализует интерфейс `Automaton`. Данный автомат отвечает за работу пульта дистанционного управления.

Список событий

- `e1` – Нажата кнопка на пульте.
- `e2` – Выбрана кнопка на пульте. Параметр `data` (типа `int`) – выбранная кнопка.

Список выходных воздействий

- `z3(data)` – Выбрать кнопку. Параметр `data` (типа `int`) – кнопка.
- `z5` – Начать передачу сигнала, соответствующего выбранной кнопке.
- `z6_1` – Выбрать соответствующее кнопке изображение пульта.
- `z6_2` – Выбрать соответствующее кнопке изображение пульта с нажатой кнопкой.
- `z7` – Окончить передачу.

Схема связей

Схема связей автомата `RemoteControl` изображена на рис. 85.

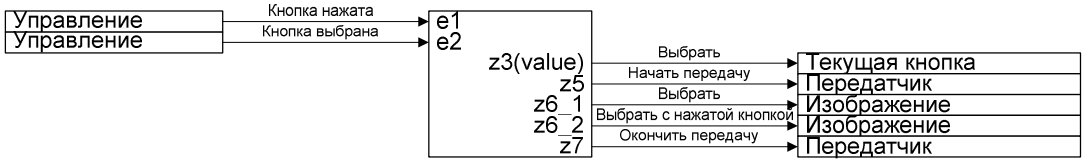

Рис. 85. Схема связей автомата `RemoteControl`

Граф переходов

Граф переходов автомата `RemoteControl` изображен на рис. 86.

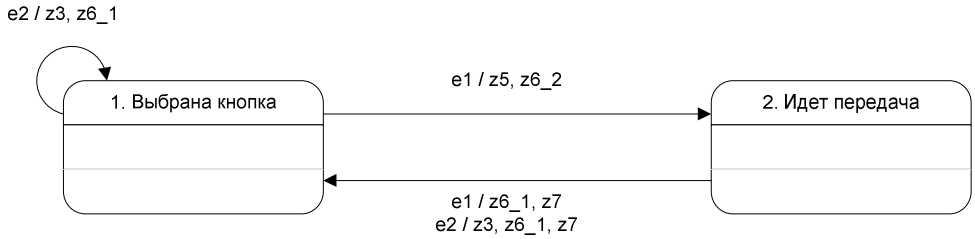

Рис. 86. Граф переходов автомата `RemoteControl`

5.1.14. Описание автоматов среды взаимодействия

Среда взаимодействия реализуется при помощи двух автоматом `BattleField` и `InfraredField`, оба из которых реализуют интерфейс `CalcObject`.

5.1.14.1. Общие события

- `e100` – Обновить координаты. Параметр `delta` (типа `double`) – интервал времени.

5.1.14.2. Автомат *BattleField*

Данный автомат реализует интерфейс `CalcObject` и контролирует процесс движения роботов по полю, а также загрузку новых данных поля. При помощи данного класса также выполняется процесс управления эмуляцией – запуск, остановка, перезапуск, пауза.

Список событий

- `e1` – Начать эмуляцию.
- `e2` – Остановить эмуляцию и вернуть все параметры к исходным.
- `e3` – Сделать паузу в эмуляции.
- `e4`– Загрузить новые данные поля. Параметр `objData` (типа `BattleFieldData`) – данные поля.

Список выходных воздействий

- `z0` – Вернуть роботы на их исходные позиции.
- `z1(fieldData)` – Загрузить новые данные поля. Параметр `fieldData` (типа `BattleFieldData`) – Данные поля.
- `z2` – Перезапустить управляющие системы роботов.
- `z4(delta)` – Обновить параметры поля. Параметр `delta` (типа `double`) – временной интервал.

Схема связей

Схема связей автомата `BattleField` изображена на рис. 87.

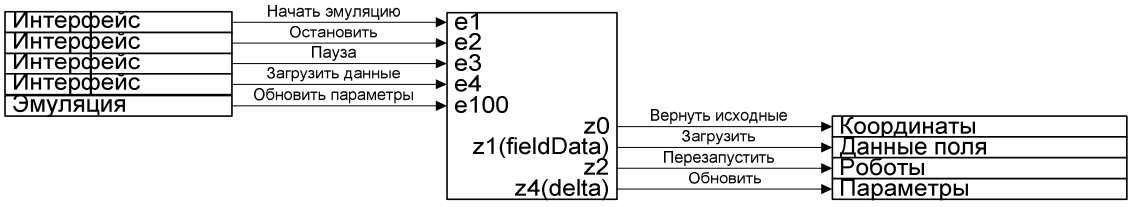

Рис. 87. Схема связей автомата `BattleField`

Граф переходов

Граф переходов автомата `BattleField` изображен на рис. 88.

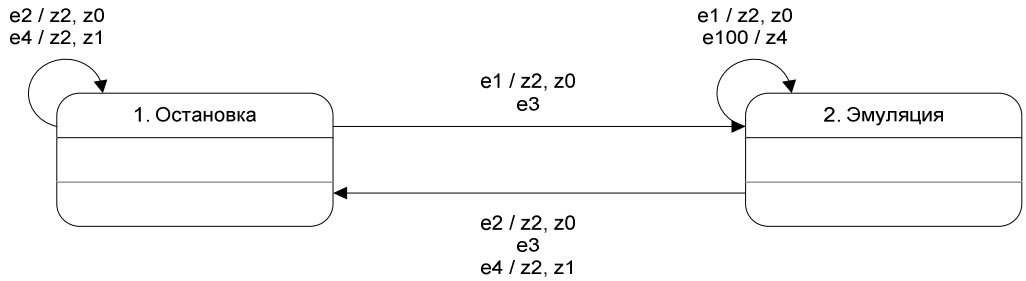

Рис. 88. Граф переходов автомата `BattleField`

5.1.14.3. Автомат *InfraredField*

Данный автомат реализует интерфейс `CalcObject` и управляет процессом передачи сообщений по инфракрасному порту.

Список выходных воздействий

- `z1` – Передать сообщения от передатчиков приемникам.
- `z3(delta)` – Обновить параметры всех инфракрасных устройств.

Схема связей

Схема связей автомата `InfraredField` изображена на рис. 89.

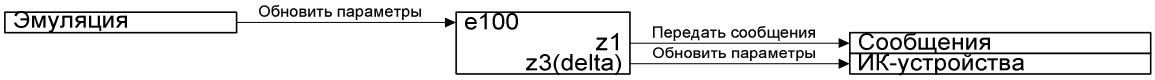

Рис. 89. Схема связей автомата `InfraredField`

Граф переходов

Граф переходов автомата `InfraredField` изображен на рис. 90.

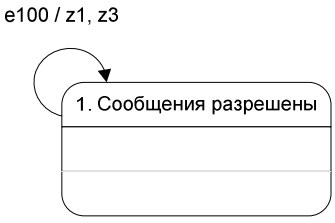

Рис. 90. Граф переходов автомата `InfraredField`

5.1.15. Описание автоматов управляющих систем

Приведем автоматы, которые были использованы в проекте *Isenguard*.

На рис. 91 приведен граф переходов автомата *A1* транспортного робота, который реализует следующие основные функции:

- прием и передача сообщений по инфракрасному порту;
- разворот после старта.

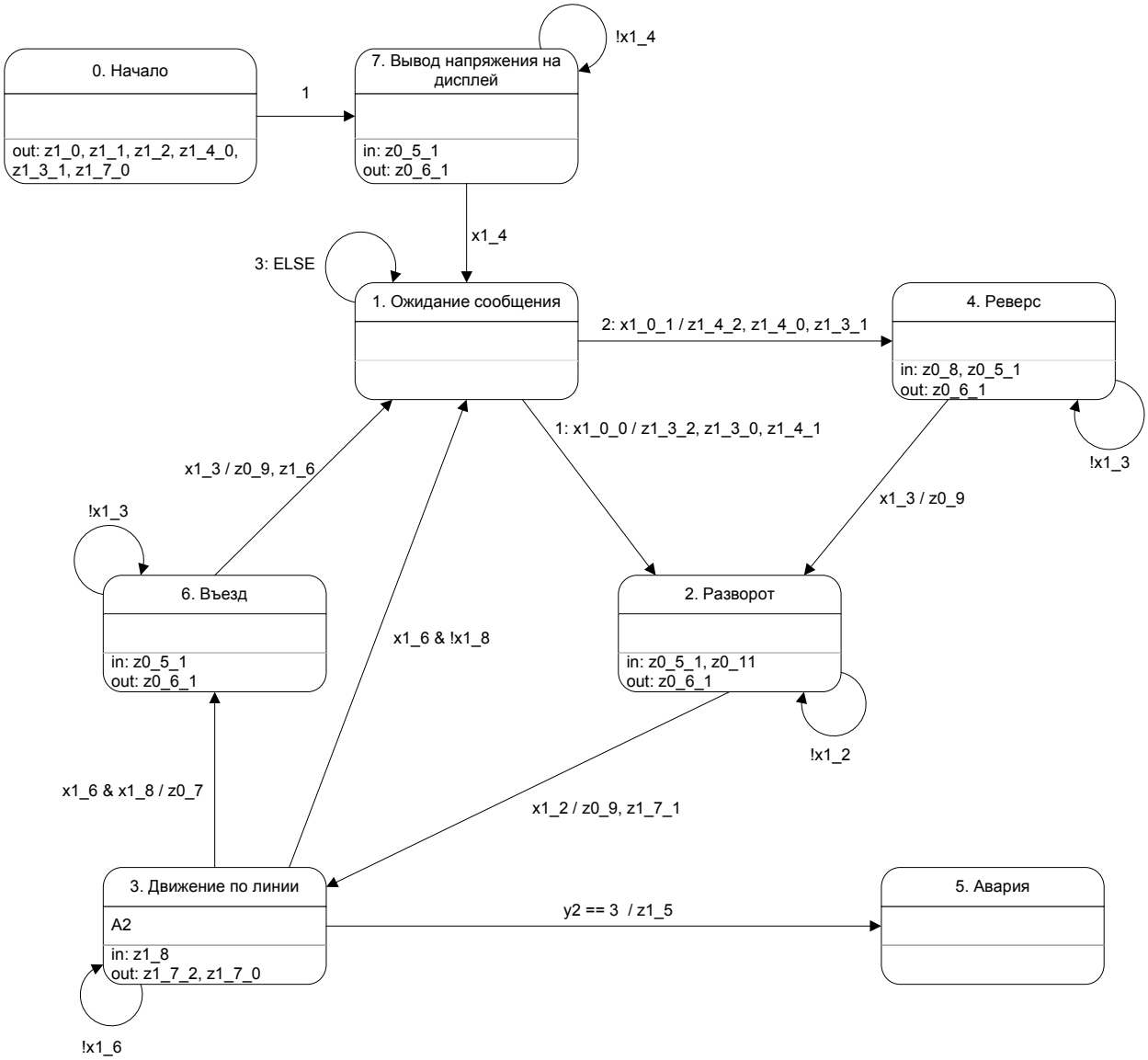

Рис. 91. Граф переходов автомата *A1* транспортного робота

На рис. 92 приведен граф переходов автомата *A2* транспортного робота, который обеспечивает движение по пути после начального поворота до места остановки.

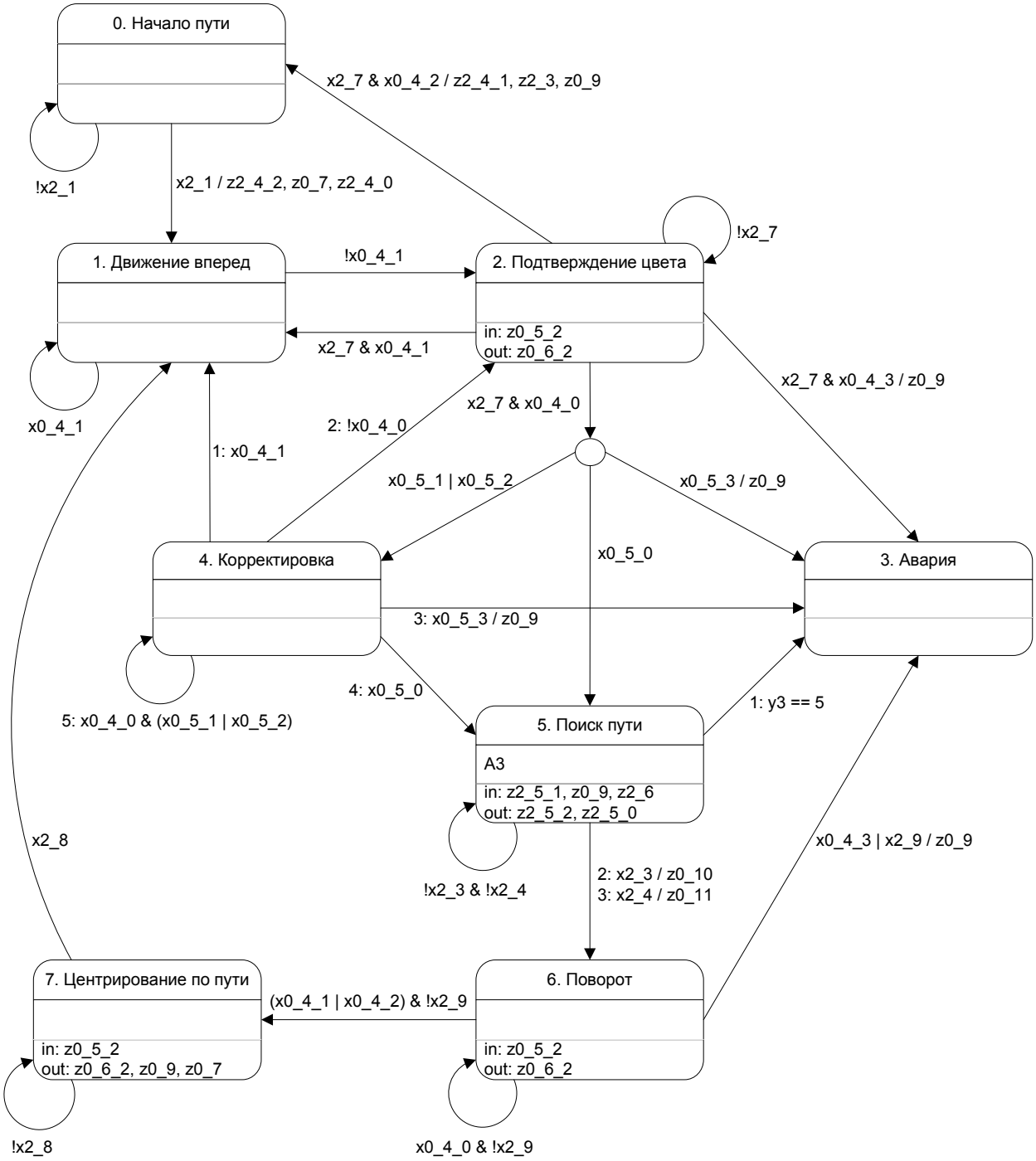

Рис. 92. Граф переходов автомата *A2* транспортного робота

На рис. 93 изображен граф переходов автомата *A3* транспортного робота, который обеспечивает поиск пути при помощи датчика, расположенного на штанге.

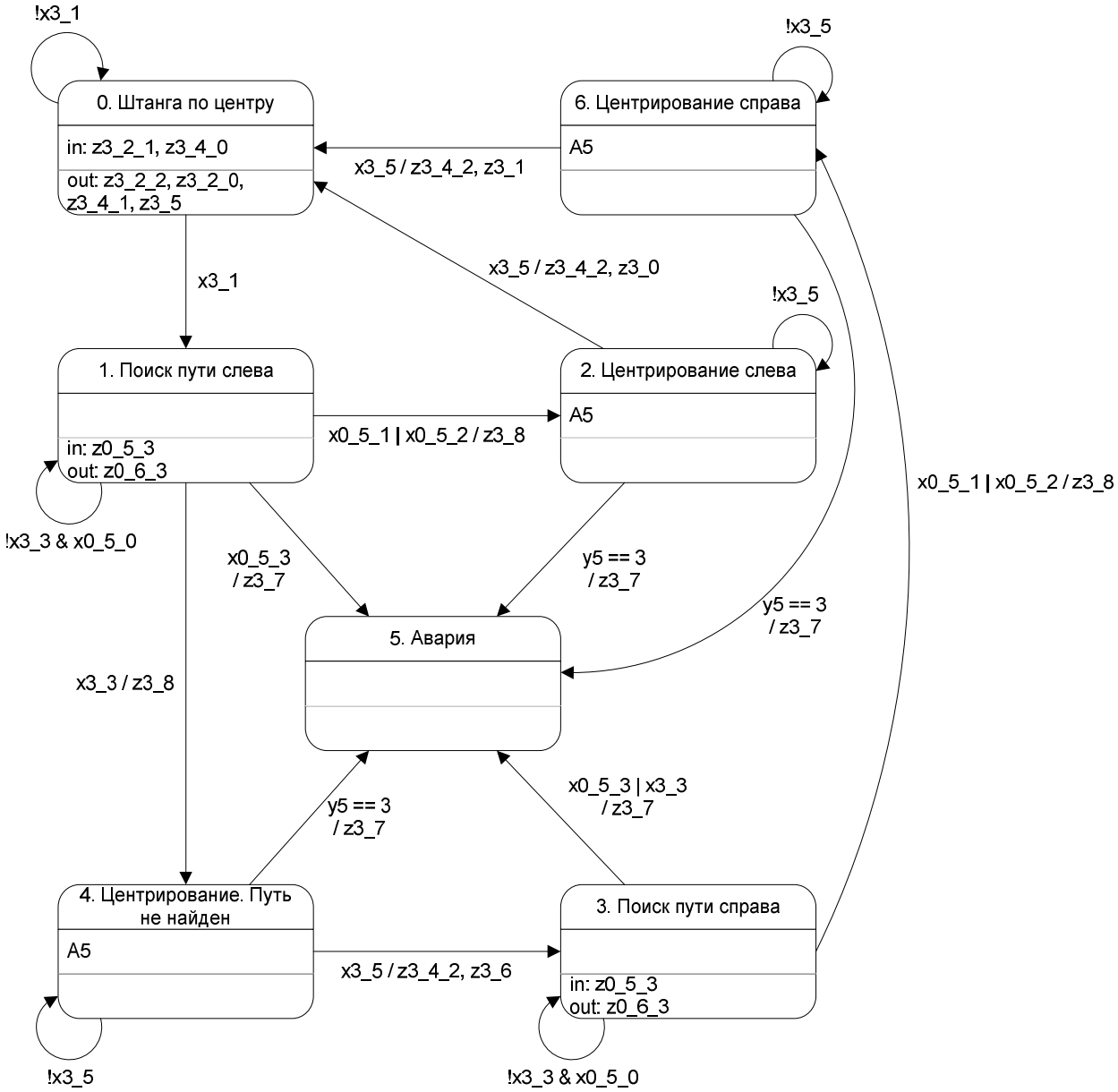

Рис. 93. Граф переходов автомата *A3* транспортного робота

На рис. 94 изображен граф переходов автомата *A5* транспортного робота, который обеспечивает возвращение штанги в центральное положение после поиска.

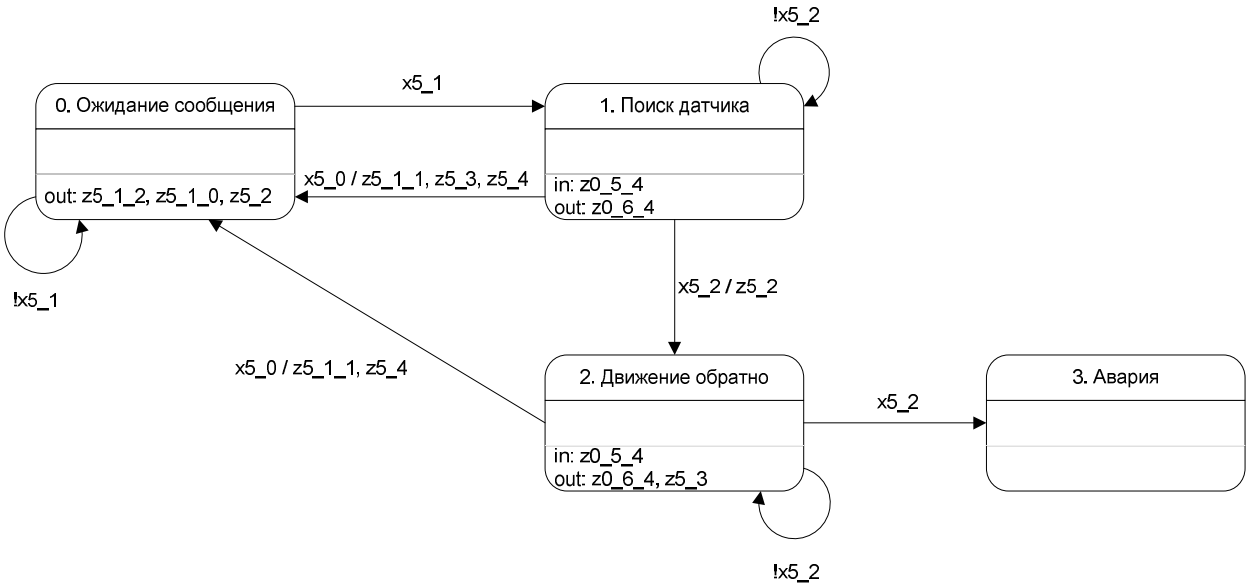

Рис. 94. Граф переходов автомата *A5* транспортного робота

Отметим, что по техническим причинам в нумерации автоматов транспортного робота цифра четыре не используется.

На рис. 95 изображен граф переходов автомата *A0* робота-поставщика, который обеспечивает выдачу заданного количества предметов после получения сообщения от транспортного робота.

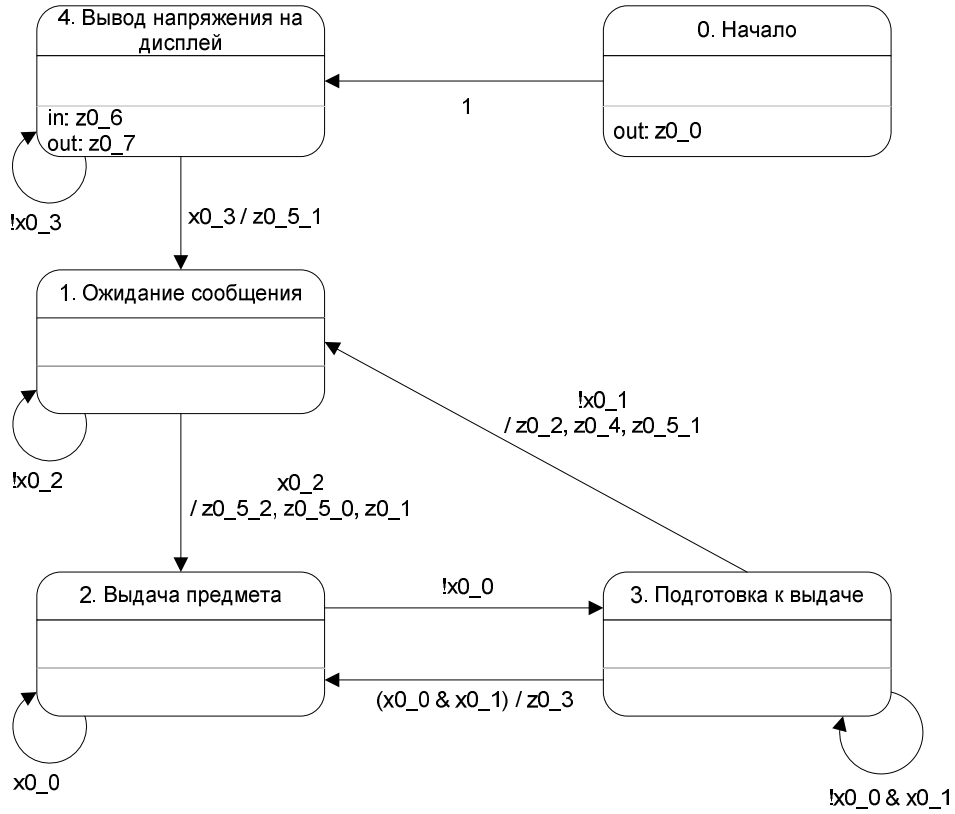

Рис. 95. Граф переходов автомата *A0* робота-поставщика

Каждый граф переходов был реализован в соответствующем классе, который наследовался от класса `ControlAutomaton`. При этом автомат *A1* был реализован в классе `A1Auto`, автомат *A2* – в классе `A2Auto`, автомат *A3* – в классе `A3Auto`, автомат *A5* – в классе `A5Auto`. Эти классы являются внутренними классами класса `LineTraceControlSystem`.

Автомат *A0* был реализован в классе `A0Auto`. Этот класс является внутренним классом класса `DispencerControlSystem`.

После проведенных испытаний при помощи эмулятора система подтвердила свою работоспособность, но выяснился недостаток проектирования логики работы автомата *A1* транспортного робота с точки зрения автоматного программирования. Этот недостаток состоял в том, что в этом автомате использовалась входная переменная `x8` (флаг), которая возвращала положительное значение, если транспортный робот имел пустой поддон. Проблема заключалась в том, что на поддоне не было никаких датчиков, которые могли бы диагностировать загруженные предметы, и эта переменная эмулировалась программно.

Было решено отказаться от флага в программе, заменив его соответствующими состояниями. Это привело к перепроектированию автомата *A1*, без изменения его интерфейса (схемы связей, которая приведена в проекте *Isenguard*). При этом проверка правильности работы нового автомата проводилась при помощи разрабатываемого эмулятора.

Переменная `x8` была предназначена для того, чтобы определить в каком направлении двигался транспортный робот – от пользователя (места доставки) к роботу-поставщику или же от робота-поставщика к пользователю (месту доставки). Поэтому в новом графе переходов были продублированы состояния "Движение по линии", "Ожидание сообщения" и "Поворот" для того, чтобы различать в какую сторону робот движется не по флагу, а по проходимым состояниям. Например, если транспортный робот делает поворот после получения сообщения от пульта, то, следовательно, управляющий им автомат *A1* находится при этом в состоянии "Поворот". Если же робот делает поворот после погрузки предметов, то указанный автомат *A1* находится при этом в состоянии "Поворот после реверса".

В результате граф переходов автомата *A1* стал выглядеть так, как изображено на рис. 96.

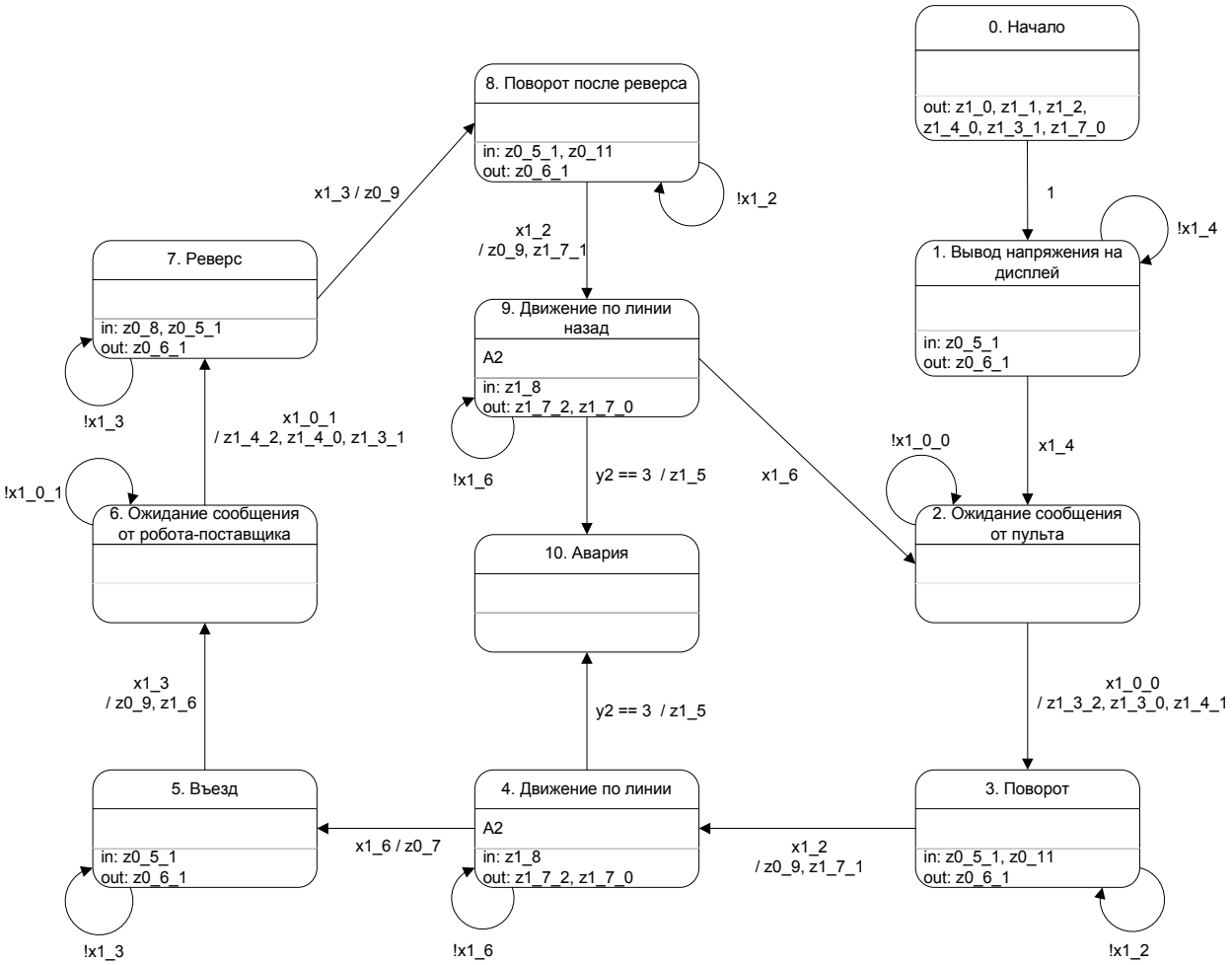

Рис. 96. Новый граф переходов автомата *A1* транспортного робота

Также при эмуляции использовалась еще одна модификация автомата *A1*. Она обусловлена следующим фактом. Когда проводилась работа с проектом *Isenguard*, предметы с поддона транспортного робота снимал пользователь. Однако при эмуляции пользователь не был реализован ☺. Поэтому в новый граф переходов автомата *A1* было добавлено еще одно выходное воздействие `z9`. Оно убирает предметы с поддона транспортного робота, как только пользователь дает ему очередное сообщение.

Модифицированный граф переходов автомата *A1* изображен на рис. 97. На переходе между состоянием "2" и состоянием "3" добавлено выходное воздействие `z9`.

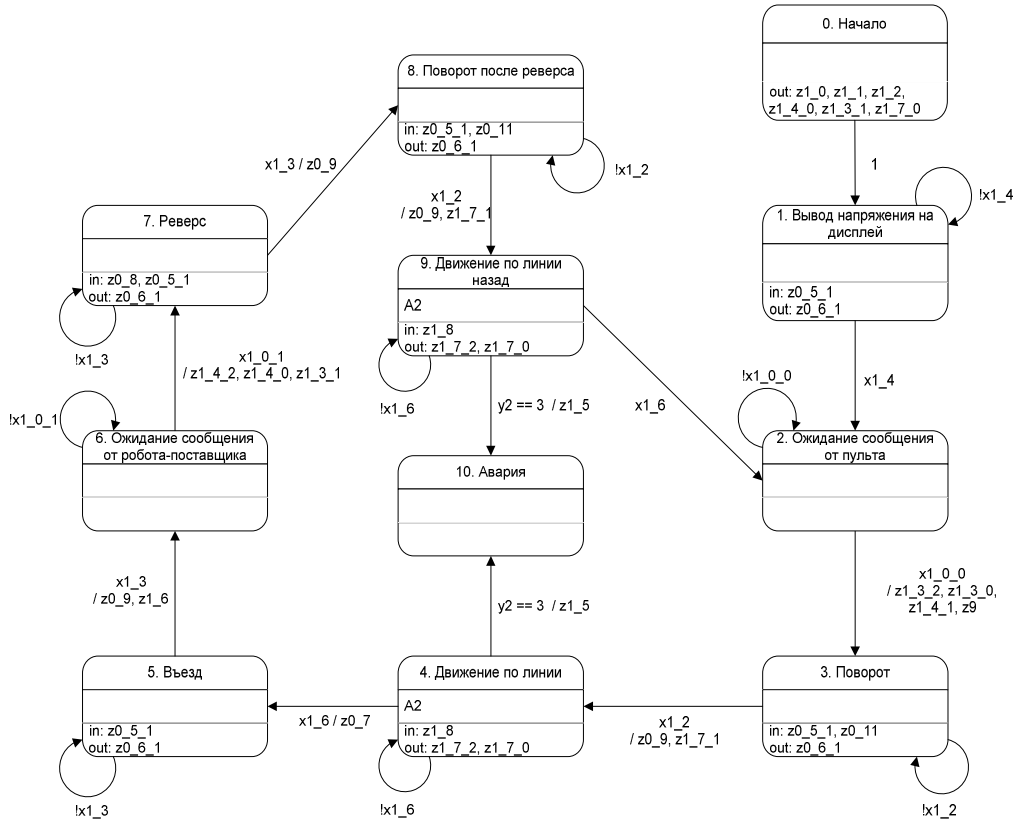

Рис. 97. Модифицированный граф переходов автомата *A1*

5.1.16. Описание автоматов пользовательского интерфейса

Пользовательский интерфейс описывается при помощи четырех основных классов, реализующих интерфейс `Automaton`: `BattleFieldView`, `BattleFieldEditor` (на графах переходов сокращенно BFE), `ProcessControlPanel` (на графах переходов сокращенно PCP), `EditPanel`. Так как у этих четырех автоматов много общих событий, то все константы, соответствующие данным события были размещены в коде класса `BattleFieldView`.

5.1.16.1. Список событий

- `e1` – Инициализировать автомат `BattleFieldView`.
- `e2` – Нажата кнопка "Редактировать" ("Edit").
- `e3` – Нажата кнопка "Старт"\"Пауза" ("Start"\"Pause").
- `e4` – Нажата кнопка "Стоп" ("Stop").
- `e5` – Нажата кнопка однократного увеличения ("Zoom 1x").
- `e6` – Нажата кнопка двукратного увеличения ("Zoom 2x").
- `e7` – Нажата кнопка "Влево" ("Left").

- `e8` – Нажата кнопка "Вправо" ("Right").
- `e9` – Нажата кнопка "Вниз" ("Down").
- `e10` – Нажата кнопка "Вверх" ("Up").
- `e11` – Нажата кнопка "Центр" ("Center").
- `e12` – Нажата кнопка слежения за транспортом ("Track Transport").
- `e13` – Нажата кнопка доставки одного предмета ("One Item").
- `e14` – Нажата кнопка доставки двух предметов ("Two Item").
- `e15` – Нажата кнопка доставки трех предметов ("Three Item").
- `e16` – Мышка сдвинута. Параметр `data` (типа `int`) – старший байт координата `x`, младший – координата `y`.
- `e17` – Мышка сдвинута при нажатой левой кнопке. Параметр `data` (типа `int`) – старший байт координата `x`, младший – координата `y`.
- `e18` – Мышка сдвинута при нажатой правой кнопке. Параметр `data` (типа `int`) – старший байт координата `x`, младший – координата `y`.
- `e19` – Нажата левая кнопка мыши. Параметр `data` (типа `int`) – старший байт координата `x`, младший – координата `y`.
- `e20` – Произведен щелчок левой кнопкой мыши. Параметр `data` (типа `int`) – старший байт координата `x`, младший – координата `y`.
- `e21` – Нажата кнопка "Установить транспорт" ("Place Transport").
- `e22` – Нажата кнопка "Вращать транспорт" ("Rotate Transport").
- `e23` – Нажата кнопка "Установить поставщик" ("Place Dispencer").
- `e24` – Нажата кнопка "Вращать поставщик" ("Rotate Dispencer").
- `e25` – Нажата кнопка "Рисовать путь" ("Path").
- `e26` – Нажата кнопка "Рисовать место остановки" ("Foil").
- `e27` – Нажата кнопка "Рисовать места ошибки" ("Error").
- `e28` – Нажата кнопка "Стереть" ("Clear").
- `e29` – Нажата кнопка "Стереть все" ("Clear All").
- `e30` – Нажата кнопка "Да" ("OK").

- `e31` – Нажата кнопка "Нет" ("Cancel").
- `e32` – Нажата кнопка "Справка" ("Help").
- `e42` – Слежение завершено.
- `e43` – Симуляция остановлена.

5.1.16.2. Автомат *BattleFieldView*

Данный автомат реализует интерфейс `Automaton` и определяет текущий режим работы эмулятора – редактирование, эмуляция, слежение за транспортным роботом и, в зависимости от выбранного режима, передает соответствующие события другим автоматам.

Выходные воздействия

- `z1` – Сделать панель редактирования активной и видимой.
- `z2` – Сделать панель управления активной и видимой.
- `z4` – Создать изображение для отрисовывания поля и роботов.
- `z6_1` – Загрузить новые параметры поля и роботов из редактора.
- `z6_2` – Восстановить параметры поля и роботов, которые были до редактирования.
- `z7` – Загрузить текущие параметры поля и роботов в редактор.
- `z8` – Загрузить изображение для отрисовывания в редактор.
- `z9` – Загрузить указатель мыши для рисования пути.
- `z10` – Загрузить указатель мыши для рисования мест остановки (фольга).
- `z11` – Загрузить указатель мыши для рисования мест ошибки.
- `z12` – Загрузить стандартный указатель мыши.
- `z13` – Загрузить указатель мыши для стирания.
- `z20 (x, y)` – Выставить координаты пульта управления. Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.
- `z21 (x, y)` – Выставить направление пульта управления по вектору (`x`, `y`). Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.
- `z22` – Нажать выбранную кнопку на пульте управления.

- z23_1 – Выбрать кнопку "1" на пульте управления.
- z23_2 – Выбрать кнопку "2" на пульте управления.
- z23_3 – Выбрать кнопку "3" на пульте управления.
- z30_1 – Начать эмуляцию.
- z30_2 – Остановить и вернуть симуляцию в начало.
- z30_3 – Пауза в эмуляции.
- z31_1 – Однократное увеличение.
- z31_2 – Двукратное увеличение.
- z32_1 – Сохранить текущее значение увеличения.
- z32_2 – Восстановить сохраненное значение увеличения.
- z32_3 – Сохранить текущее значение смещения.
- z32_4 – Восстановить текущее значение смещения.
- z33_1 – Сдвинуть поле влево.
- z33_2 – Сдвинуть поле вправо.
- z33_3 – Сдвинуть поле вверх.
- z33_4 – Сдвинуть поле вниз.
- z33_5 – Центрировать поле.

Схема связей

Схема связи автомата BattleFieldView представлена на рис. 98.

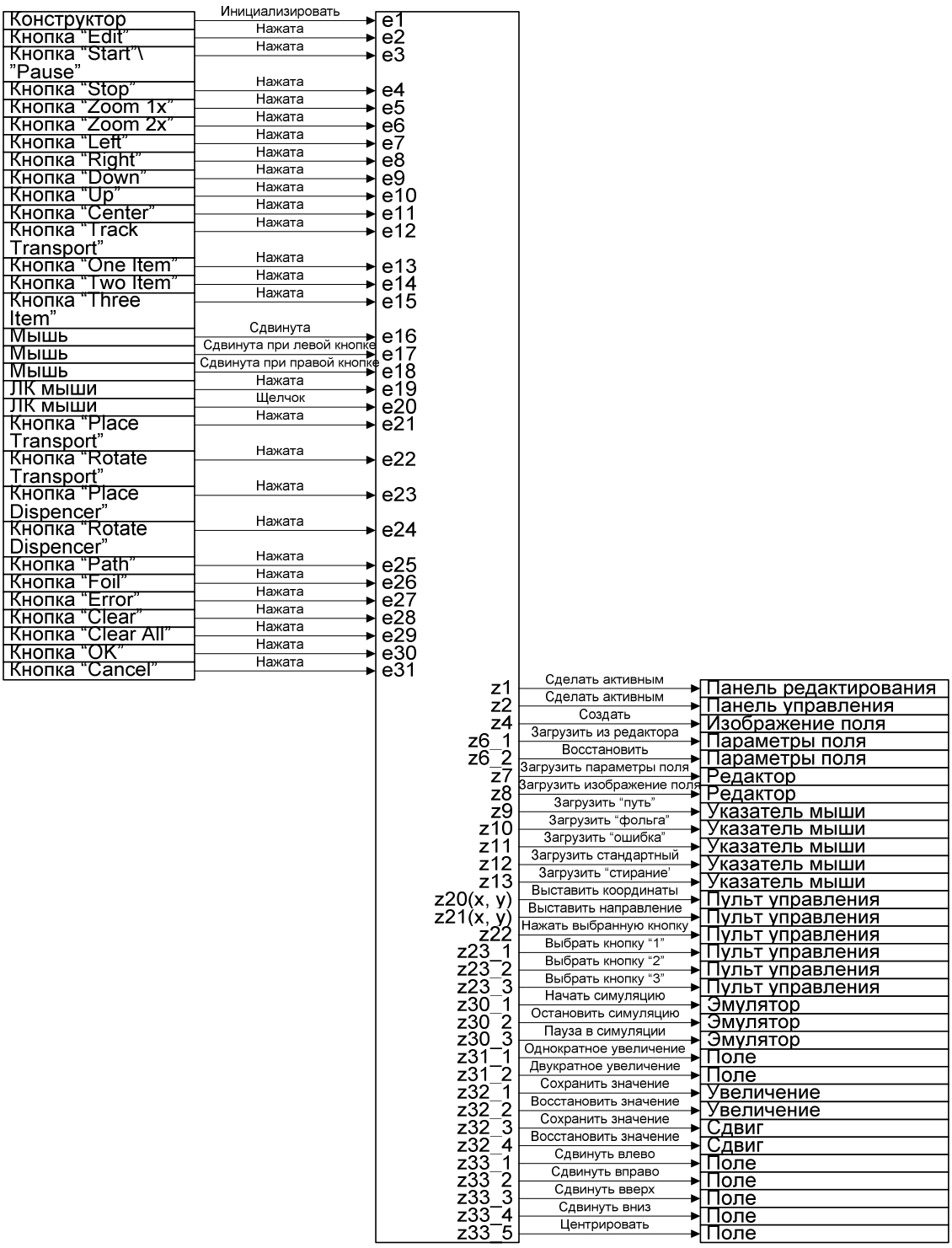

Рис. 98. Схема связей автомата `BattleFieldView`

Граф переходов

Граф переходов автомата `BattleFieldView` представлен на рис. 99.

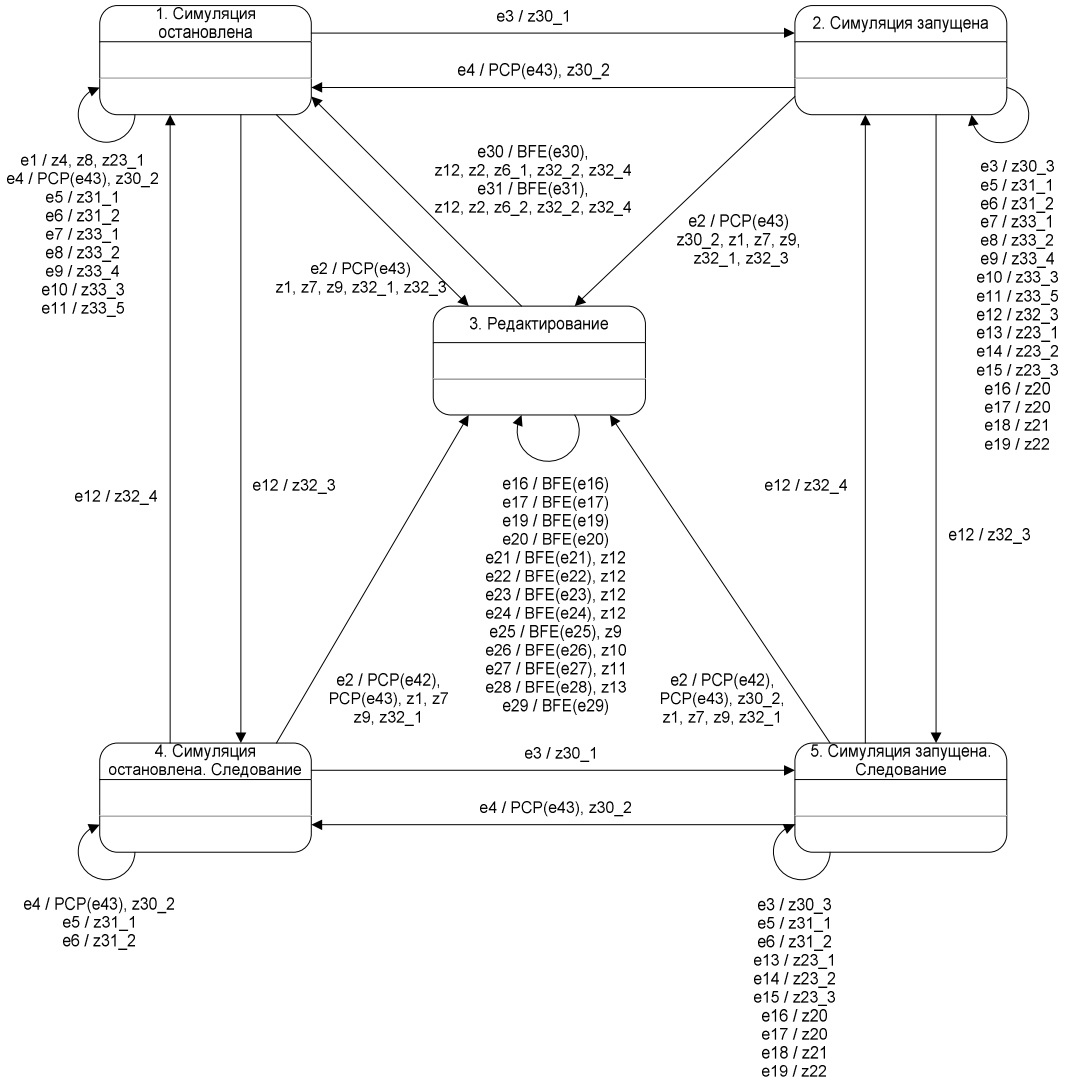

Рис. 99. Граф переходов автомата `BattleFieldView`

5.1.16.3. Автомат *BattleFieldEditor*

Данный автомат реализует интерфейс `Automaton` и управляет процессом редактирования поля и начальных положений роботов. Объект этого класса является членом класса `BattleFieldView`.

Выходные воздействия

- `z0_1` – Выбрать транспортный робот в качестве текущего робота.
- `z0_2` – Выбрать робот-поставщик в качестве текущего робота.
- `z1 (x, y)` – Нарисовать линию между сохраненными координатами и текущими координатами указателя. Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.
- `z2 (x, y)` – Сохранить текущие координаты указателя. Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.

- `z3 (x, y)` – Выставить координаты текущего робота. Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.

- `z4` – Сохранить координаты текущего робота.

- `z5 (x, y)` – Выставить направление текущего робота по вектору (`x`, `y`). Координата `x` – старший байт параметра `data`. Координата `y` – младший байт параметра `data`.

- `z6` – Сохранить направление текущего робота.

- `z7` – Приготовить кисть для рисования мест остановки (фольги).

- `z8` – Приготовить кисть для рисования мест ошибки.

- `z9` – Приготовить ластик для стирания нарисованных пути, мест остановки и мест ошибки.

- `z10` – Приготовить кисть для рисования пути.

- `z12` – Восстановить сохраненное поле, которое было до редактирования.

- `z13` – Сохранить отредактированное поле.

- `z14` – Очистить поле от всего нарисованного на нем.

Схема связей

Схема связей автомата `BattleFieldEditor` представлена на рис. 100.

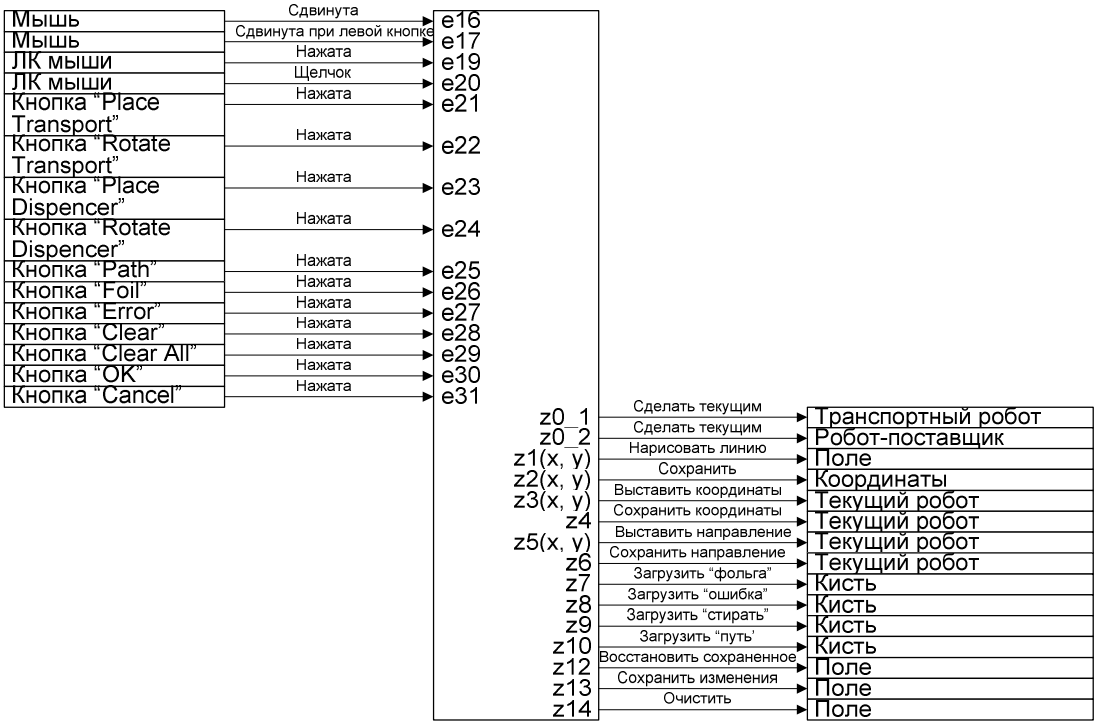

Рис. 100. Схема связей автомата `BattleFieldEditor`

Граф переходов

Граф переходов автомата `BattleFieldEditor` представлен на рис. 101.

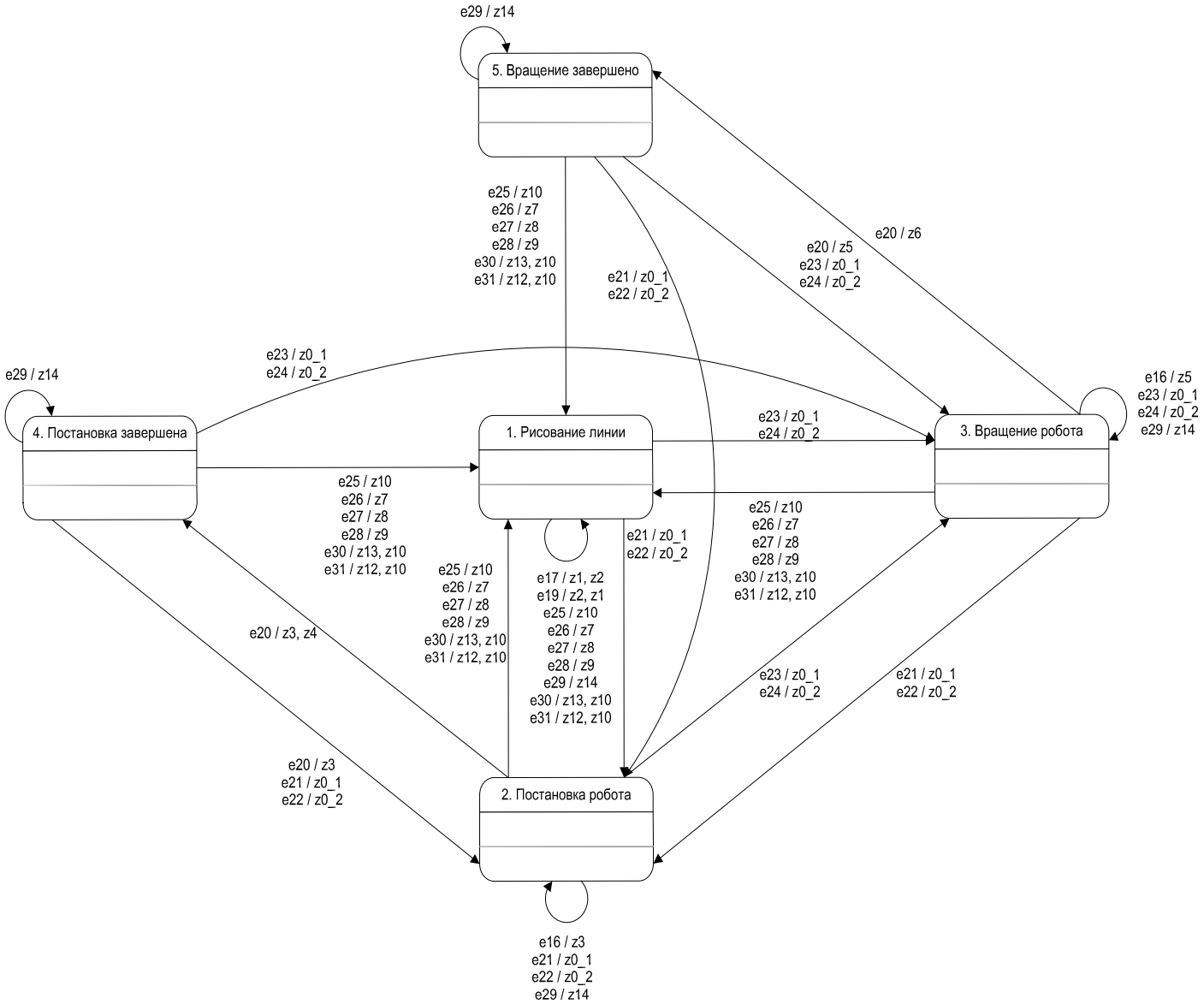

Рис. 101. Граф переходов автомата BattleFieldEditor

5.1.16.4. Автомат *ProcessControlPanel*

Данный автомат реализует интерфейс `Automaton` и управляет панелью управления эмуляцией. Объект этого класса является членом класса `BattleFieldView`.

Выходные воздействия

- `z1` – Деактивировать кнопки "Влево" ("Left"), "Вправо" ("Right"), "Вверх" ("Up"), "Вниз" ("Down") и "Центр" ("Center"). Выделить кнопку слежения ("Track Transport").

- `z2` – Активировать кнопки "Влево" ("Left"), "Вправо" ("Right"), "Вверх" ("Up"), "Вниз" ("Down") и "Центр" ("Center"). Снять выделение с кнопки слежения ("Track Transport").

- `z3` – Деактивировать кнопку однократного увеличения ("Zoom 1x"), активировать кнопку двукратного увеличения ("Zoom 2x").

- `z4` – Активировать кнопку однократного увеличения ("Zoom 1x"), деактивировать кнопку двукратного увеличения ("Zoom 2x").

- `z5` – Загрузить иконку "Пауза" на кнопку "Старт"\"Пауза" ("Start"\"Pause").

- `z6` – Загрузить иконку "Старт" на кнопку "Старт"\"Пауза" ("Start"\"Pause").

- `z7` – Деактивировать кнопки выбора количества предметов для доставки ("One Item", "Two Item", "Three Item").

- `z8` – Активировать кнопки выбора количества предметов для доставки ("One Item", "Two Item", "Three Item").

Схема связей

Схема связей автомата `ProcessControlPanel` представлена на рис. 102.

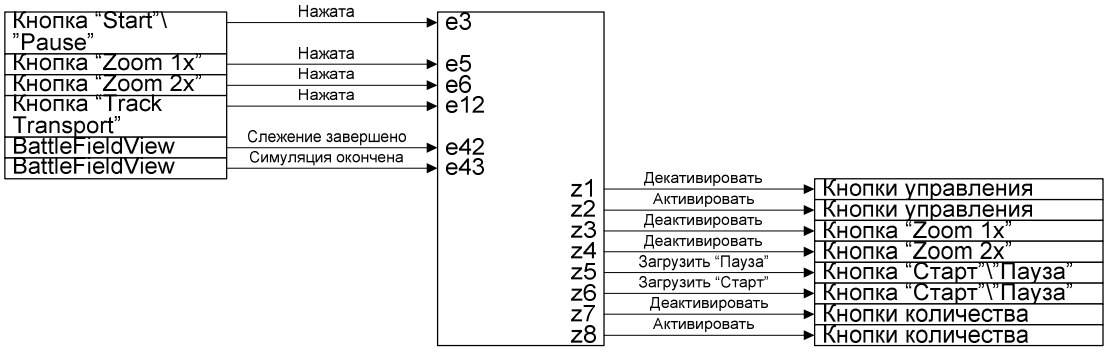

Рис. 102. Схема связей автомата `ProcessControlPanel`

Граф переходов

Граф переходов автомата `ProcessControlPanel` представлен на рис. 103.

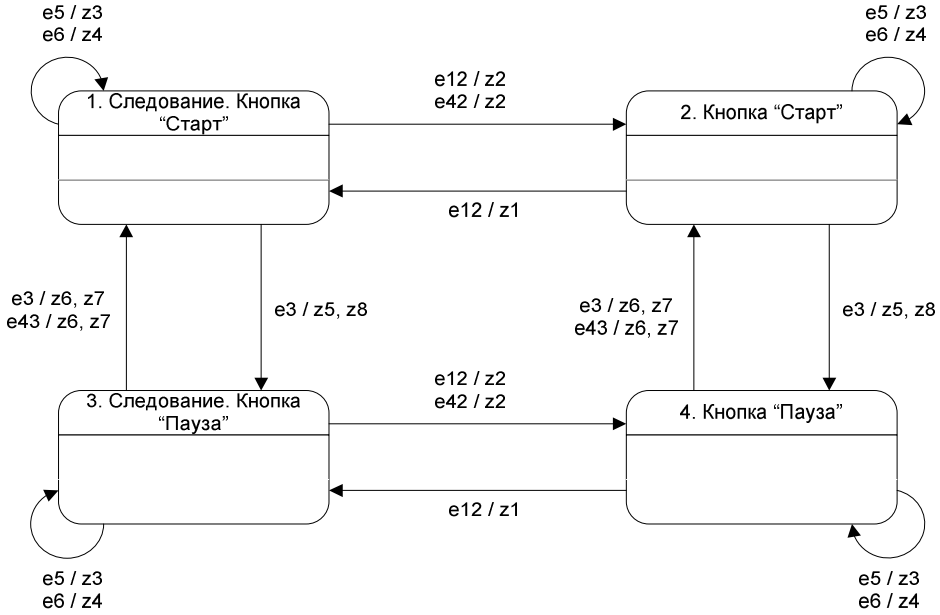

Рис. 103. Граф переходов автомата `ProcessControlPanel`

5.1.16.5. Автомат *EditPanel*

Данный автомат реализует интерфейс `Automaton` и управляет панелью управления при редактировании. Объект этого класса является членом класса `BattleFieldView`.

Выходные воздействия

- `z1` – Выделить кнопку "Рисовать путь" ("Path").
- `z2` – Выделить кнопку "Рисовать места остановки" ("Foil").
- `z3` – Выделить кнопку "Рисовать места ошибки" ("Error").
- `z4` – Выделить кнопку "Стереть" ("Clear").
- `z5` – Выделить кнопку "Установить транспорт" ("Place Transport").
- `z6` – Выделить кнопку "Вращать транспорт" ("Rotate Transport").
- `z8` – Выделить кнопку "Установить поставщик" ("Place Dispencer").
- `z9` – Выделить кнопку "Вращать поставщик" ("Rotate Dispencer").
- `z10` – Показать диалог со справкой о рисовании пути.

Схема связей

Схема связей автомата `EditPanel` представлена на рис. 104.

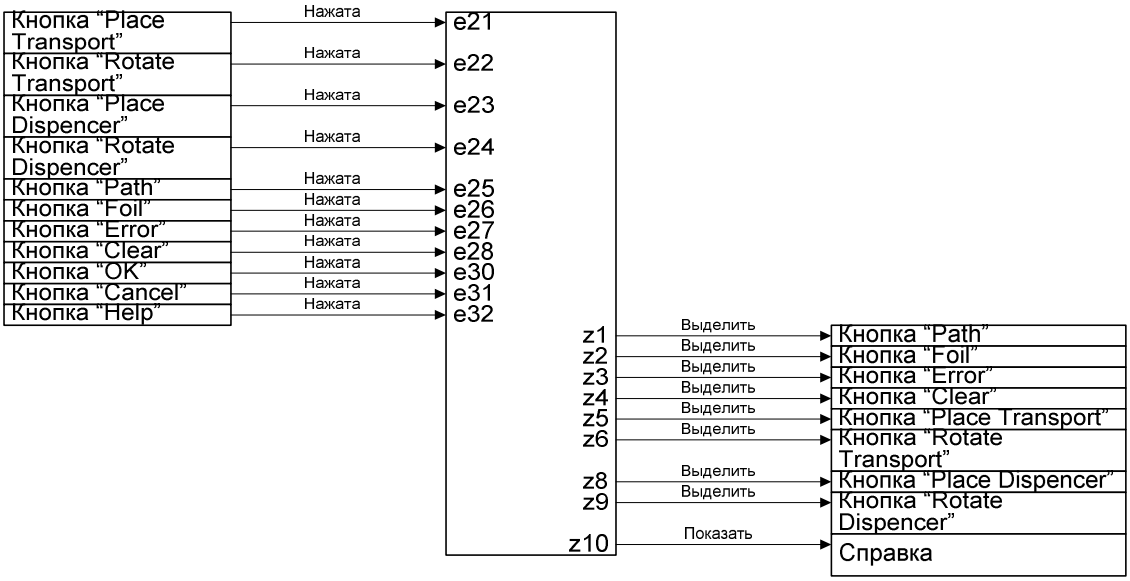

Рис. 104. Схема связей автомата `EditPanel`

Граф переходов

Граф переходов автомата `EditPanel` представлен на рис. 105.

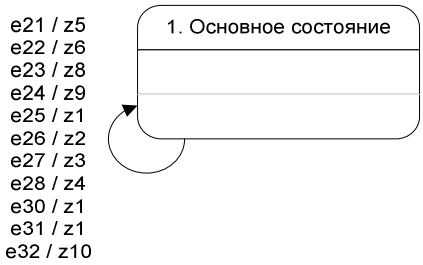

Рис. 105. Граф переходов автомата `EditPanel`

6. Аналогичные системы

Если рассматривать аналогичные системы взаимодействующих агентов, то можно выделить известный проект *Robocode* [10], который был очень моден в программистских кругах несколько лет назад. В этом проекте была реализована возможность проектировать управляющие программы для танков, а потом наблюдать, как они будут вести себя в бою. В рамках проекта *Robocode* была впервые создана управляющая система для такого танка на основе автоматного подхода и открытая проектная документация для него [8, 9].

Также стоит отметить проект *Terrarium* [11]. Этот проект был разработан корпорацией *Microsoft*. Он представляет собой оболочку для игры. Смысл этой игры состоит в разработке системы управления травоядным или плотоядным существом, а так же растением. Игра предоставляет конкурентоспособную среду для испытания различных вариантов существ. Среда является весьма реалистической эволюционной моделью искусственного интеллекта, в которой можно оценить роль различных черт поведения и свойств животных в процессе борьбы за выживание. Существа в данной игре являются агентами, а сама игра – средой их взаимодействия. В рамках проекта *Terrarium* была создана управляющая система на основе автоматного подхода для травоядного существа [12].

7. Планы на будущее

В ходе работы над проектом *Isenguardemu* родилось несколько идей, по поводу того, как можно дальше развивать мультиагентные системы в целом и данный проект в частности. В данном разделе эти идеи будут вкратце изложены.

7.1. Система доставки

Пусть имеется некоторое количество терминалов, через которые производится отгрузка контейнеров. Помимо этих терминалов есть некоторое количество мест, куда требуется эти контейнеры доставить. Между терминалами и местами для доставки нарисованы пути, по которым доставка может производиться. По этим путям двигаются несколько транспортных роботов. Они выступают в качестве агентов. Этим транспортным роботам приходят команды вида "Доставить контейнер с терминала N на место доставки K". После получения такой команды, транспортные роботы должны решить какой их них поедет выполнять доставку контейнера. Причем данная система взаимодействующих транспортных роботов должна продолжать функционировать по мере возможности, если один или несколько роботов выйдут из строя. Также необходимо реализовать возможность добавления в систему новых роботов "на ходу".

7.2. Новые системы управлением движения по пути

Система движения по пути, реализованная в транспортном роботе является далеко не самой эффективной. Ее можно модифицировать путем использования нейронных сетей, генетических алгоритмов, нечеткой логики, а также более эффективной установки и использования световых датчиков. Так как в эмуляторе нет ограничений на максимальное количество датчиков, можно спокойно установить на транспортный робот любое разумное датчиков.

7.3. Новые методы реализации автоматов

В данном проекте все методы, реализующие автоматные функции были объявлены при помощи спецификатора `synchronized`. Так что, при попытке вызова автоматной функции, когда она уже выполняется в другом потоке, приходится ждать, пока этот поток ее не освободит. Идея заключается в том, чтобы сделать для каждой из автоматных функций очередь из событий и параметров. К примеру, совершается вызов автоматной функции, когда она выполняется в другом потоке – в этом случае событие и параметры, передаваемые в автоматную функцию, добавляются в очередь. Когда выполнение функции прекратится, то в зависимости от того, есть события и параметры в очереди, или нет, автоматная функция либо продолжит свое выполнение с новым событием и параметрами, либо окончательно прекратит свое выполнение. Таким образом, можно сделать реализацию автоматных функций более эффективной.

Выводы

Автоматное программирование подтвердило свою эффективность, как при разработке систем эмуляции, так и пользовательского интерфейса.

Стало возможно демонстрировать функциональность проекта *Isenguard* на персональном компьютере.

Эмулятор обеспечил возможность проверки работоспособности системы управления транспортным роботом после переработки одного из автоматов в отсутствие конструктора *Lego Mindstorms*.

Благодарности

Алфавиту за предоставленные буквы.

Андрею Курочкину за дизайн робота-поставщика.

Маме за предложение "А не написать ли тебе, сынок, бакалаврскую?".

Папе за мотивацию методом пряника.

Товарищу Сталину за наше счастливое детство.

Маме Миши Ботвинника как лишнему человеку в прозе Юза Алешковского.

Густаву Климту за художественное оформление моей комнаты.

Казимиру Малевичу за предоставленную гамму цветов в левой части рис. 2.

Василию Кандинскому за крест на рис. 11.

Американскому доллару за похожесть на рис. 24 и рис. 25.

Всеобъемлющему Дао за факт существования.

Змию-искусителю за то, что сделал нашу жизнь интересной.

Лайаму Хоулетту за музыкальное сопровождение.

Оксане П. за … в общем, за все.

Даше Г. за прекрасное исполнение Шостаковича на фоно.

ММЗ и "Танцующему Дракону" за стабильное состояние сознания в ходе написания сего проекта.

Арвиду Яновичу Пельше за имя, фамилию и отчество.

Верке Сердючке за "гоп гоп гоп".

.Ru за новый метод сдавания зачетов путем длительного выдерживания и затем взятия на испуг.

Военной кафедре как колодезю мудрости.

Ваньке Бегунку за высокий градус и сильный эффект.

43 отделению милиции Петроградского района за то, что я туда не попадал в процессе написания данной работы.

Зинаиде Марковне за блинчики и борщики по особым малороссийским рецептам.

Список литературы

1. *Ярцев Б.М., Шалыто А.А.* Разработка программного обеспечения *Lego Mindstorms* на основе автоматного подхода (проект *Isenguard*). http://is.ifmo.ru/projects/lego/

2. *Baum D.* Definitive Guide to *Lego Mindstroms*, NY: Appress, 2000.

3. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

4. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. http://is.ifmo.ru/works/switch/1/

5. *Шалыто А.А., Туккель Н.И.* Система управления дизель-генератором. Проектная документация. http://is.ifmo.ru/projects/dg/

6. OS *leJOS*. http://lejos.sourceforge.net/

7. *Шалыто А.* Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК. 2003. № 9. http://is.ifmo.ru/works/open_doc/

8. *Шалыто А.А., Туккель Н. И.* Танки и автоматы // Byte/Россия. 2003. № 2. http://is.ifmo.ru/download/tanks_new.pdf

9. *Туккель Н.И., Шалыто А.А.* Система управления танком для игры “Robocode”. Версия 1. Проектная документация. http://is.ifmo.ru/projects/tanks/

10. *Robocode.* http://alphaworks.ibm.com/robocode/

11. *Terrarium Home.* http://www.windowsforms.net/Terrarium/

12. *Марков С.М.*, *Шалыто А.А.* Система управления травоядным существом для игры *Terrarium*. http://is.ifmo.ru/projects/terrarium/