Оглавление

Перечень сокращений 3
Терминология 3
1. Введение 4
2. Предпроектное исследование 6
2.1. Основные положения языка РДО[2] 6
2.2. Процессный подход дискретного имитационного моделирования 6
2.3. Система имитационного моделирования Rao X 6
3. Формирование ТЗ 8
3.1. Введение 8
3.2. Общие сведения 8
3.3. Назначение разработки 8
3.4. Требования к программе или программному изделию 8
3.4.1. Требования к функциональным характеристикам 8
3.4.2. Требования к надежности 9
3.4.3. Условия эксплуатации 9
3.4.4. Требования к составу и параметрам технических средств 9
3.4.5. Требования к информационной и программной совместимости 9
3.4.6. Требования к маркировке и упаковке 10
3.4.7. Требования к транспортированию и хранению 10
3.5. Требования к программной документации 10
3.6. Стадии и этапы разработки 10
3.7. Порядок контроля и приемки 10
4. Концептуальный этап проектирования подсистемы 11
4.1. Процессно-ориентированный подход в Rao X 11
4.2. Блоки подсистемы 11
4.2.1. Создание и удаление транзактов 11
4.2.2. Очередь транзактов 11
4.2.3. Захват и освобождение ресурсов 12
4.2.4. Задержка транзактов 12
4.2.5. Ветвление модели 12
5. Технический этап проектирования подсистемы 13
5.1. Алгоритм работы подсистемы процессно-ориентированного подхода 13
5.2. Блоки подсистемы 14

5.2.1. Блок Generate 14
5.2.2. Блок Terminate 15
5.2.3. Блок Queue 16
5.2.4. Блок Seize 17
5.2.5. Блок Release 18
5.2.6. Блок Advance 19
5.2.7. Блок Test 20
6. Рабочий этап проектирования подсистемы 21
6.1. Реализация алгоритма подсистемы процессно-ориентированного подхода 21
6.2. Реализация блоков подсистемы 21
6.2.1. Блок Generate 21
6.2.2. Блок Terminate 22
6.2.3. Блок Queue 22
6.2.4. Блок Seize 23
6.2.5. Блок Release 23
6.2.6. Блок Advance 23
6.2.7. Блок Test 24
7. Апробирование разработанной подсистемы 25
8. Заключение 26
Список используемых источников 27
Список использованного программного обеспечения 27
Приложение 1 – Тест линейной модели 28
Приложение 2 – Тест модели с использованием блока очереди 29
Приложение 3 – Тест модели с использованием блока ветвления 30

Перечень сокращений

ИМ – Имитационное Моделирование

СДС – Сложная Дискретная Система

СМО – Система Массового Обслуживания

IDE – Integrated Development Environment (Интегрированная Среда Разработки)

Терминология

Плагин – независимо компилируемый программный модуль, динамически подключаемый к основной программе и предназначенный для расширения и/или использования её возможностей

Транзакт – объект, обслуживание которого производится в имитационной модели.

Блок – объект, который обрабатывает транзакты в имитационной модели.

Фреймворк – программная платформа, определяющая структуру программной системы.

Юнит-тестирование – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

1. Введение

Имитационное моделирование (ИМ)[1] на ЭВМ находит широкое применение при исследовании и управлении сложными дискретными системами (СДС) и процессами, в них протекающими. К таким системам можно отнести экономические и производственные объекты, морские порты, аэропорты, комплексы перекачки нефти и газа, ирригационные системы, программное обеспечение сложных систем управления, вычислительные сети и многие другие. Широкое использование ИМ объясняется тем, что размерность решаемых задач и неформализуемость сложных систем не позволяют использовать строгие методы оптимизации. Эти классы задач определяются тем, что при их решении необходимо одновременно учитывать факторы неопределенности, динамическую взаимную обусловленность текущих решений и последующих событий, комплексную взаимозависимость между управляемыми переменными исследуемой системы, а часто и строго дискретную и четко определенную последовательность интервалов времени. Указанные особенности свойственны всем сложным системам.

Проведение имитационного эксперимента позволяет:

1. Сделать выводы о поведении СДС и ее особенностях:
    - без ее построения, если это проектируемая система
    - без вмешательства в ее функционирование, если это действующая система, проведение экспериментов над которой или слишком дорого, или небезопасно
    - без ее разрушения, если цель эксперимента состоит в определении пределов воздействия на систему
2. Синтезировать и исследовать стратегии управления.
3. Прогнозировать и планировать функционирование системы в будущем.
4. Обучать и тренировать управленческий персонал и т.д.

ИМ является эффективным, но и не лишенным недостатков, методом. Трудности использования ИМ, связаны с обеспечением адекватности описания системы, интерпретацией результатов, обеспечением стохастической сходимости процесса моделирования, решением проблемы размерности и т.п. К проблемам применения ИМ следует отнести также и большую трудоемкость данного метода.

Интеллектуальное ИМ, характеризующиеся возможностью использования методов искусственного интеллекта и прежде всего знаний, при принятии решений в процессе имитации, при управлении имитационным экспериментом, при реализации интерфейса пользователя, создании информационных банков ИМ, использовании нечетких данных, снимает часть проблем использования ИМ.

2. Предпроектное исследование

2.1. Основные положения языка РДО[2]

Основные положения системы РДО могут быть сформулированы следующим образом[1]:

- Все элементы СДС представлены как ресурсы, описываемые некоторыми параметрами. Ресурсы могут быть разбиты на несколько типов; каждый ресурс определенного типа описывается одними и теми же параметрами.

- Состояние ресурса определяется вектором значений всех его параметров; состояние СДС - значением всех параметров всех ресурсов.

- Процесс, протекающий в СДС, описывается как последовательность целенаправленных действий и нерегулярных событий, изменяющих определенным образом состояние ресурсов; действия ограничены во времени двумя событиями: событиями начала и событиями конца.

- Нерегулярные события описывают изменения состояния СДС, непредсказуемые в рамках продукционной модели системы (влияние внешних по отношению к СДС факторов либо факторов, внутренних по отношению к ресурсам СДС). Моменты наступления нерегулярных событий случайны.

- Действия описываются операциями, которые представляют собой модифицированные продукционные правила, учитывающие временные связи. Операция описывает предусловия, которым должно удовлетворять состояние участвующих в операции ресурсов, и правила изменения состояния ресурсов в начале и в конце соответствующего действия.

- Множество ресурсов R и множество операций O образуют модель СДС.

2.2. Процессный подход дискретного имитационного моделирования

2.3. Система имитационного моделирования Rao X

Система имитационного моделирования Rao X представляет собой плагин для интегрированной среды разработки Eclipse, позволяющий вести

разработку имитационных моделей на языке РДО. Система написана на языке Java[3] и состоит из четырех основных компонентов:

- rao – компонент, производящий преобразование кода на языке РДО в код на языке Java
- rao.lib – библиотека системы. Этот компонент реализует ядро системы имитационного моделирования
- rao.ui – компонент, реализующий графический интерфейс системы с помощью библиотеки SWT[5]
- rao.tests – компонент, реализующий тестирование системы посредством Unit-тестов

3. Формирование ТЗ

3.1. Введение

Программный комплекс Rao X предназначен для разработки и отладки имитационных моделей на языке РДО. Основные цели данного комплекса - обеспечение пользователя легким в обращении, но достаточно мощным средством разработки текстов моделей на языке РДО, обладающим большинством функций по работе с текстами программ, характерных для сред программирования, а также средствами проведения и обработки результатов имитационных экспериментов.

Программный комплекс Rao X позволяет разрабатывать имитационные модели на основе двух подходов: событийного и подхода сканирования активностей.

Требуется реализовать возможность проведения имитационных экспериментов на основе процессно-ориентированного подхода дискретного имитационного моделирования.

3.2. Общие сведения

Основание для разработки: задание на курсовой проект.

Заказчик: Кафедра «Компьютерные системы автоматизации производства» МГТУ им. Н.Э. Баумана

Разработчик: студент кафедры «Компьютерные системы автоматизации производства» Зудина О.В.

Наименование темы разработки: «Реализация процессно-ориентированного подхода в системе имитационного моделирования Rao X»

3.3. Назначение разработки

Реализовать подсистему процессно-ориентированного подхода в системе имитационного моделирования Rao X.

3.4. Требования к программе или программному изделию

3.4.1. Требования к функциональным характеристикам

Подсистема процессно-ориентированного подхода должна удовлетворять следующим требованиям:

- Интеграция в Rao X

- Реализация следующих блоков:
  - создания транзактов
  - удаления транзактов
  - очереди
  - захвата ресурса
  - освобождения ресурса
  - задержки транзакта
  - ветвления модели

3.4.2. Требования к надежности

Основное требование к надежности направлено на поддержание в исправном и работоспособном состоянии ЭВМ, на которой происходит использование программного комплекса Rao X и подсистемы процессно-ориентированного подхода.

3.4.3. Условия эксплуатации

- Эксплуатация должна производиться на оборудовании, отвечающем требованиями к составу и параметрам технических средств, и с применением программных средств, отвечающим требованиям к программной совместимости
- Аппаратные средства должны эксплуатироваться в помещениях с выделенной розеточной электросетью 220В ±10%, 50 Гц с защитным заземлением

3.4.4. Требования к составу и параметрам технических средств

Программный продукт должен работать на компьютерах со следующими характеристиками:

- объем ОЗУ не менее 2 Гб
- объем жесткого диска не менее 50 Гб
- микропроцессор с тактовой частотой не менее 2ГГц
- монитор с разрешением от 1366*768 и выше

3.4.5. Требования к информационной и программной совместимости

Система должна работать под управлением следующих ОС: Windows 7, Windows 8, Ubuntu 15.10.

3.4.6. Требования к маркировке и упаковке

Требования к маркировке и упаковке не предъявляются.

3.4.7. Требования к транспортированию и хранению

Требования к транспортированию и хранению не предъявляются.

3.5. Требования к программной документации

Требования к программной документации не предъявляются.

3.6. Стадии и этапы разработки

Плановый срок начала разработки – 10 октября 2015г.

Плановый срок окончания разработки – 19 декабря 2015г.

Этапы разработки:

- Концептуальный этап проектирования подсистемы
- Технический этап проектирования подсистемы
- Рабочий этап проектирования подсистемы

3.7. Порядок контроля и приемки

Контроль и приемка работоспособности системы осуществляются посредством запуска последовательности JUnit-тестов подсистемы и сравнения полученных результатов с эталонными результатами моделирования.

4. Концептуальный этап проектирования подсистемы

На концептуальном этапе проектирования требовалось:

- определить взаимосвязь процессно-ориентированного подхода с реализованными подходами в системе имитационного моделирования Rao X
- разработать концепцию перемещения транзактов по блокам имитационной модели
- разработать набор блоков, необходимых для моделирования

4.1. Процессно-ориентированный подход в Rao X

Процессно-ориентированный подход должен быть способен работать в Rao X как независимо, так и взаимодействуя с другими подходами (событийным и сканирования активностей). Процессная часть имитационной модели должна состоять из последовательности стандартных блоков, каждый из которых выполняет определенное действие над транзактом. При этом процессная часть имитационной модели не должна завершать свою работу, если остался хотя бы один блок, все еще способный совершить действие над транзактом.

4.2. Блоки подсистемы

Процессный подход ориентирован на моделирование СМО. В соответствии с этим выявлены определенные требования к блокам.

4.2.1. Создание и удаление транзактов

Для обеспечения потока транзактов через систему необходимо наличие блока создания транзактов, который будет генерировать новые транзакты с заданной периодичностью. Периодичность транзакта должна быть конфигурируемой. Блок создания транзактов должен иметь один выход и не иметь входов.

Для обеспечения возможности удаления транзактов из системы должен существовать отдельный блок удаления транзактов, имеющий один вход и не имеющий выходов.

4.2.2. Очередь транзактов

Любая СМО включает в себя очередь.

Блок, реализующий очередь транзактов, должен иметь один вход и один выход.

4.2.3. Захват и освобождение ресурсов

Любая СМО имеет ограничение на количество доступных ресурсов. В связи с этим необходима реализация двух блоков: захвата и освобождения ресурса. Разработчик модели должен иметь возможность связывать ресурс с блоками захвата и освобождения. Каждый из блоков должен иметь один вход и один выход.

4.2.4. Задержка транзактов

Обслуживание транзактов в СМО – процесс, занимающий определенное время. Для моделирования этого процесса необходим блок, способный задерживать транзакт, пока не завершится его обслуживание. Длительность обслуживания должна быть конфигурируемой. Блок задержки транзактов должен иметь один вход и один выход.

4.2.5. Ветвление модели

Для реализации сложных СМО возникает потребность в блоке ветвления модели. Условие ветвления должно задаваться разработчиком модели. Блок ветвления должен иметь один вход и несколько выходов.

5. Технический этап проектирования подсистемы

5.1. Алгоритм работы подсистемы процессно-ориентированного подхода

Вызов подсистемы, отвечающий за реализацию процессно-ориентированного подхода, должен быть встроен в общий алгоритм прогона имитационных моделей симулятора Rao X. При старте модели или после того, как произошло очередное продвижение модельного времени, происходит сканирование всех активностей модели. Если ни одна из активностей не была выполнена, симулятор запускает процессную часть имитационной модели.

Функция работы процессной части имитационной модели возвращает один из трех возможных статусов:

- SUCCESS – сканирование процесса завершено успешно, модель изменила свое состояние. Выполняется оповещение о том, что модель изменила свое состояние, и цикл сканирования активностей запускается заново.
- NOTHING_TO_DO – сканирование процесса завершено успешно, модель не изменила свое состояние. Процессная часть имитационной модели завершила свою работу, не выполнив ни одного действия. Симулятор, если это возможно, продвигает модельное время, выполняет следующее событие и запускает заново цикл сканирования активностей.
- FAILURE – сканирование процесса завершено с ошибкой. Симулятор не может больше продолжать работу, и прогон модели завершается со статусом ошибки (RUNTIME_ERROR).

В процессной части имитационной модели происходит поочередная проверка всех блоков и выполнение их действий, если это представляется возможным. Функция проверки блоков может вернуть три вида статусов:

- SUCCESS – блок успешно выполнил свои действия. Процессная часть модели завершает свою работу со статусом SUCCESS.
- NOTHING_TO_DO - нет действий, которые блок мог бы выполнить. Происходит переход к следующему блоку.
- CHECK_AGAIN - блок не может выполнить требуемое действие. Если в процессе проверки остальных блоков, ни один из них не завершится со статусом SUCCESS, т.е. состояние модели больше не будет изменено, то вся функция выполнения процессной части имитационной модели должна вернуть статус FAILURE.

5.2. Блоки подсистемы

5.2.1. Блок Generate

Блок Generate имеет статус готовности создавать транзакт. Если статус готовности выставлен в значение *false*, то блок при его проверке не производит никаких действий и возвращает статус NOTHING_TO_DO. В противном случае блок проверяет свой выходной узел. Если в выходном узле блока уже содержится транзакт, то новый транзакт не может быть сгенерирован. В этом случае функция проверки блока возвращает статус CHECK_AGAIN. В противном случае блок создает новый транзакт и помещает его в выходной узел. Для обеспечения генерации следующего транзакта в нужный момент времени, блок Generate выставляет свой статус готовности в значение *false* и планирует событие, которое выставит блоку статус готовности *true*. После чего функция проверки блока возвращает статус SUCCESS.

Интервал времени, с которым блок Generate создает транзакты, задается при его создании и хранится в нем в виде функции (*Supplier<Integer>*), возвращающей некоторое значение при ее вызове.

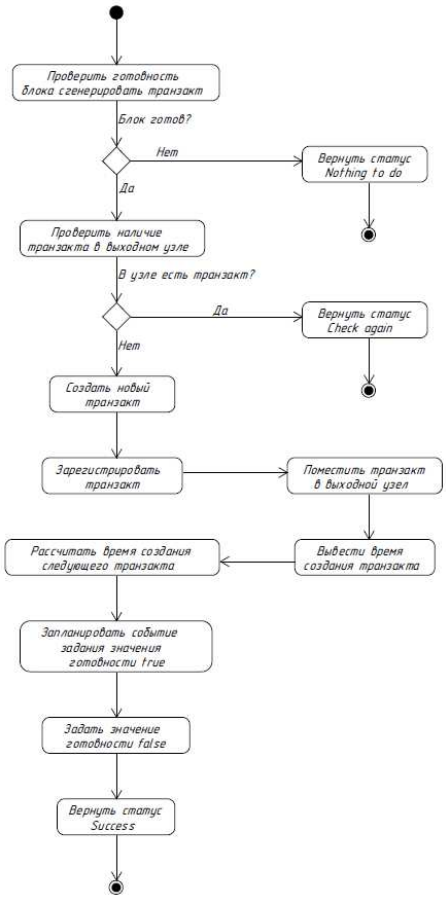

Рис. 1. Диаграмма активности блока Generate

5.2.2. Блок Terminate

В начале своей работы блок Terminate получает транзакт из входного узла. В случае, если транзакт не был получен, блок возвращает статус NOTHING_TO_DO. В противном случае полученный транзакт удаляется, возвращается статус SUCCESS.

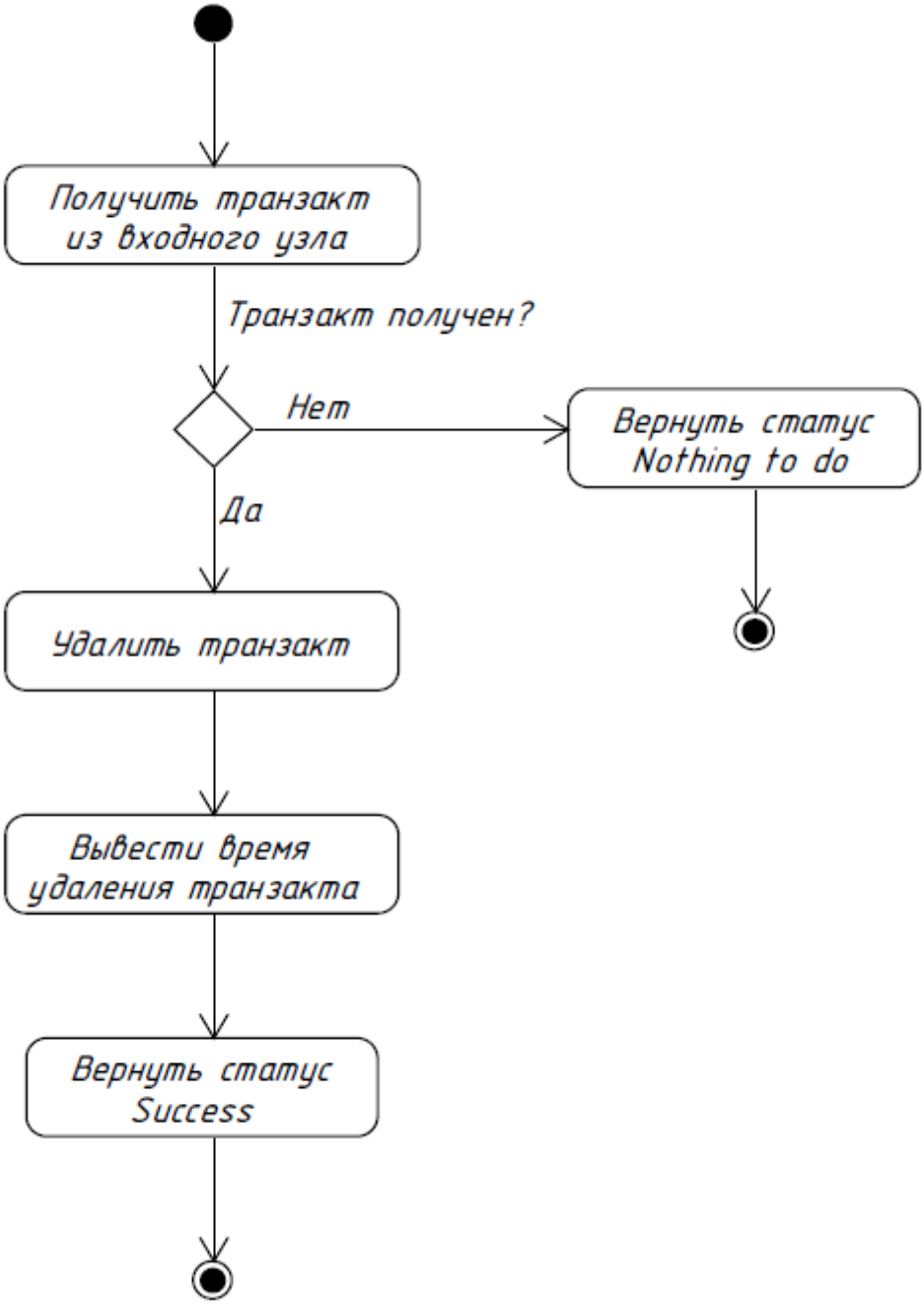

Рис. 2. Диаграмма активности блока Terminate

5.2.3. Блок Queue

В начале своей работы блок Terminate получает транзакт из входного узла. В случае, если транзакт был получен, он добавляется в очередь, а блок возвращает статус SUCCESS. В противном случае, необходимо получить первый транзакт из очереди. Если транзакт не был получен, блок возвращает статус NOTHING_TO_DO. Иначе, происходит проверка наличия транзакта в выходном узле. При занятом выходном узле возвращается статус NOTHING_TO_DO. При свободном – транзакт удаляется из очереди, возвращается статус SUCCESS.

Параметры, необходимые для создания очереди: дисциплина (FIFO, LIFO и др.) и емкость (максимально возможное количество транзактов в очереди).

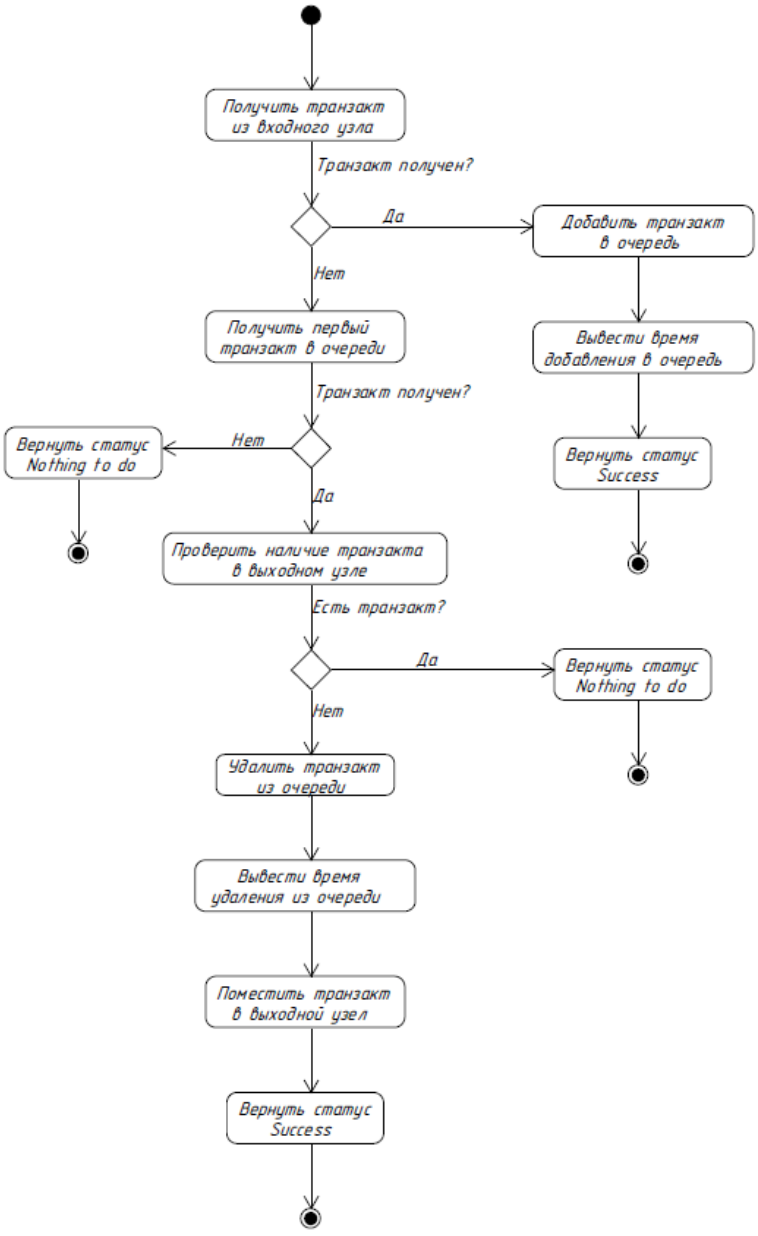

Рис. 3. Диаграмма активности блока Queue

5.2.4. Блок Seize

Работа блока Seize начинается с проверки занятости ресурса: если ресурс занят, блок возвращает статус NOTHING_TO_DO. В противном случае, проверяется наличие транзакта в выходном узле. При занятом выходном узле возвращается статус CHECK_AGAIN. При свободном – происходит попытка получения транзакта из входного узла. Если транзакт не получен, происходит возвращение статуса NOTHING_TO_DO. При получении транзакта состояние ресурса изменяется на "занят" и возвращается статус SUCCESS.

Необходимый параметр для блока: ресурс.

Рис. 4. Диаграмма активности блока Seize

5.2.5. Блок Release

Работа блока Release происходит по аналогии с блоком Seize. Отличия наблюдаются в последовательности проверок и в реакции на проверку занятости ресурса: если ресурс свободен, то следует сообщение об ошибке. Иначе, блок продолжает свою работу.

Необходимый параметр для блока: ресурс.

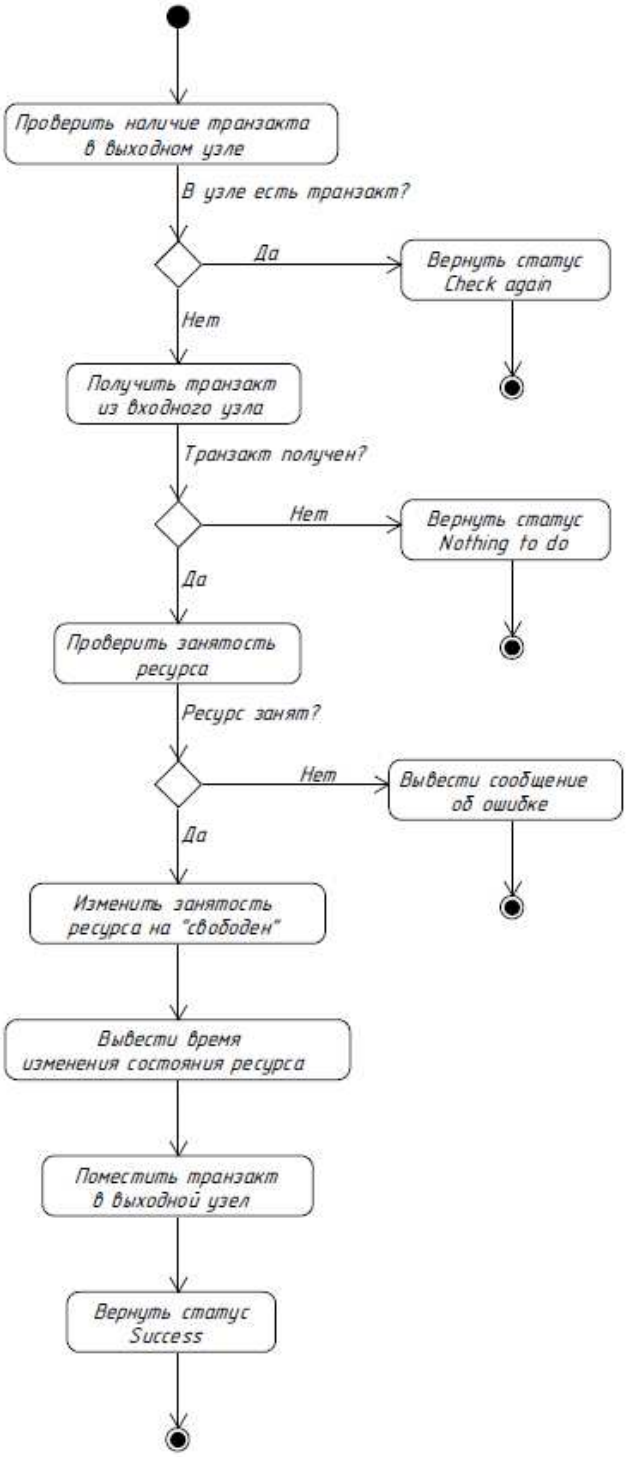

Рис. 5. Диаграмма активности блока Release

5.2.6. Блок Advance

Блок Advance служит для задержки транзакта на определенный отрезок модельного времени. В первую очередь блок проверяет, имеется ли у него транзакт, который закончил моделируемое обслуживание и готов продвигаться дальше по модели. Если такой блок имеется, то производится проверка наличия транзакта на выходном узле блока. В случае, если имеется транзакт, блок возвращает статус CHECK_AGAIN. В противном случае, временный транзакт перемещается в выходной узел и работа блока продолжается. Далее блок происходит проверка возможности получить транзакт из своего входного узла. Если такой возможности нет, блок возвращает статус NOTHING_TO_DO. Иначе, блок забирает транзакт и планирует событие окончания его обслуживания с заданной блоку длительностью. После чего блок возвращает статус SUCCESS.

Интервал времени, на который блок задерживает транзакты задается функцией, как и в случае с блоком Generate.

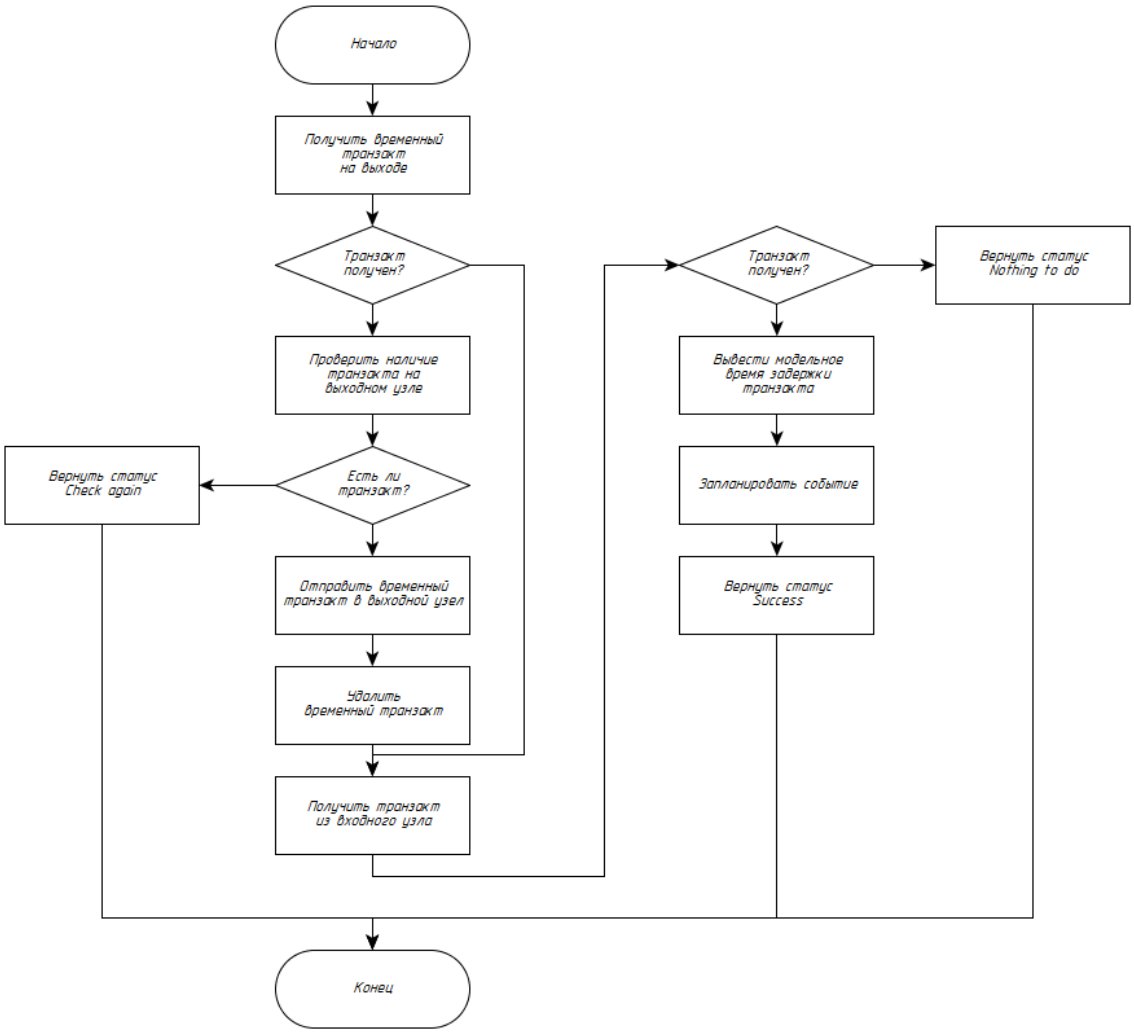

Рис. 6. Блок-схема алгоритма блока Advance

5.2.7. Блок Test

Алгоритм блока Test начинается с получения транзакта из входного узла. Если транзакт не получен, следует возврат статуса NOTHING_TO_DO. В противном случае, выполняется проверка условия, заданного в блоке Test, назначается соответствующий условию выходной узел, происходит проверка наличия транзакта в выходном узле. При наличии транзакта возвращается статус CHECK_AGAIN. Иначе, SUCCESS.

Параметр, необходимый для реализации блока Test: условие ветвления модели.

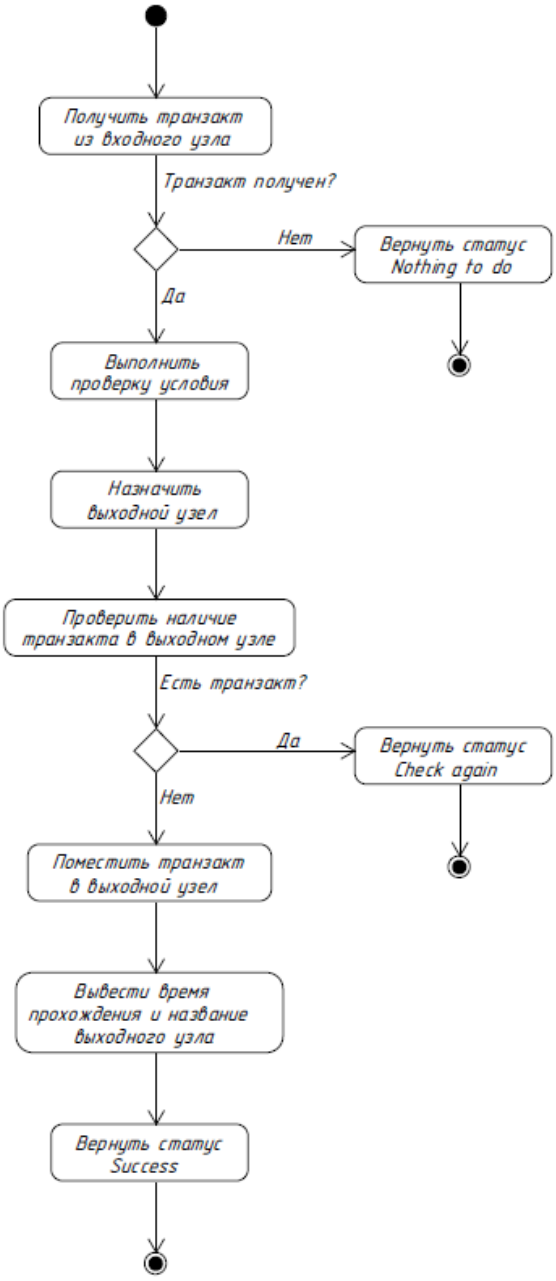

Рис. 7. Диаграмма активности блока Test

6. Рабочий этап проектирования подсистемы

На рабочем этапе проектирования подсистемы были реализованы разработанные на предыдущих этапах схемы и концепции.

6.1. Реализация алгоритма подсистемы процессно-ориентированного подхода

Для реализации работы процессного подхода в рамках общего алгоритма симулятора Rao X в основной цикл моделирования был добавлен код выполнения подсистемы процессного подхода:

```
ProcessStatus processStatus = INSTANCE.processManager.scan();
if (processStatus == ProcessStatus.SUCCESS) {
	notifyChange(ExecutionState.STATE_CHANGED);
	continue;
} else if (processStatus == ProcessStatus.FAILURE) {
	return stop(SimulationStopCode.RUNTIME_ERROR);
}
```

Реализация основного алгоритма подсистемы была выполнена в метода *scan()* класса *Process.java*:

```
public ProcessStatus scan() {
	boolean needCheckAgain = false;
	for (Block block : blocks) {
		BlockStatus blockStatus = block.check();
		if (blockStatus == BlockStatus.SUCCESS)
			return ProcessStatus.SUCCESS;
		if (blockStatus == BlockStatus.CHECK_AGAIN)
			needCheckAgain = true;
	}

	return needCheckAgain ? ProcessStatus.FAILURE
			: ProcessStatus.NOTHING_TO_DO;
}
```

6.2. Реализация блоков подсистемы

Все классы блоков подсистемы реализуют интерфейс *Block:*

```
public interface Block {
	public BlockStatus check();
}
```

В методе *check()* описывается алгоритм работы блока.

6.2.1. Блок Generate

При создании блока Generate в конструкторе требуется задать интервал создания транзактов и запланировать события создания первого транзакта:

```
public Generate(Supplier<Integer> interval) {
	this.interval = interval;
	Simulator.pushEvent(new GenerateEvent(interval.get()));
}
```

Для реализации алгоритма блока Generate вызывается метод *pushEvent()*, с помощью которого планируется событие типа *GenerateEvent*, реализующее интерфейс *Event:*

```
private class GenerateEvent implements Event {

	private double time;

	public GenerateEvent(double time) {
		this.time = time;
	}

	@Override
	public String getName() {
		return null;
	}

	@Override
	public double getTime() {
		return time;
	}

	@Override
	public void run() {
		ready = true;
	}
}
```

6.2.2. Блок Terminate

Метод *check()*:

```
@Override
public BlockStatus check() {
	Transact currentTransact = inputDock.pullTransact();
	if (currentTransact == null)
		return BlockStatus.NOTHING_TO_DO;
	System.out.println(Simulator.getTime() + ": terminate "
			+ currentTransact.getNumber());
	Transact.eraseTransact(currentTransact);
	return BlockStatus.SUCCESS;
}
```

6.2.3. Блок Queue

На момент завершения курсового проекта блок Queue был реализован без возможности задания параметров очереди. По умолчанию очередь имеет бесконечную емкость и дисциплину FIFO (First In First Out).

Метод *check()*:

```
@Override
public BlockStatus check() {
	Transact inputTransact = inputDock.pullTransact();
	if (inputTransact != null) {
		queue.offer(inputTransact);
		System.out.println(Simulator.getTime() + ": queue added "
				+ inputTransact.getNumber());
```

```
                return BlockStatus.SUCCESS;
        }

        Transact outputTransact = queue.peek();
        if (outputTransact != null) {
                if (!outputDock.hasTransact()) {
                        queue.remove();
                        System.out.println(Simulator.getTime() + ": queue removed "
                                        + outputTransact.getNumber());
                        outputDock.pushTransact(outputTransact);
                        return BlockStatus.SUCCESS;
                }
        }
        return BlockStatus.NOTHING_TO_DO;
}
```

6.2.4. Блок Seize

При создании блока Seize необходимо указывать ресурс, с которым должен взаимодействовать блок в процессе моделирования:

```
public Seize(Resource resource) {
        this.resource = resource;
}
```

Взаимодействие с ресурсом:

```
        if (resource.isLocked())
                return BlockStatus.NOTHING_TO_DO;

        resource.lock();
```

6.2.5. Блок Release

По аналогии с блоком Seize, при создании блока Release необходимо указывать ресурс:

```
public Release(Resource resource) {
        this.resource = resource;
}
```

При попытке освободить ресурс, имеющий статус "свободен", возникает сообщение об ошибке:

```
        if (!resource.isLocked()) {
                throw new ProcessException(
                                "Attempting to release unlocked resource");
        }
```

6.2.6. Блок Advance

При создании блока Advance необходимо задать длительность обработки транзакта:

```
public Advance(Supplier<Integer> duration) {
        this.duration = duration;
}
```

Для реализации алгоритма блока Advance необходимо создавать события типа *AdvanceEvent*:

```
	private class AdvanceEvent implements Event {

		private double time;
		private Transact transact;

		public AdvanceEvent(double time, Transact transact) {
			this.time = time;
			this.transact = transact;
		}

		@Override
		public String getName() {
			return null;
		}

		@Override
		public double getTime() {
			return time;
		}

		@Override
		public void run() {
			if (temporaryTransactOnOutput != null)
				throw new ProcessException(
						"Transact collision in Advance block");
			System.out.println(Simulator.getTime() + ": advance finish "
					+ transact.getNumber());
			temporaryTransactOnOutput = transact;
		}
	}
```

6.2.7. Блок Test

На момент завершения курсового проекта блок Test имеет условие по умолчанию: проверка четности номера транзакта:

```
		OutputDock outputDock;
		if (transact.getNumber() % 2 == 0) {
			System.out.println(Simulator.getTime() + ": test true " +
transact.getNumber());
			outputDock = trueOutputDock;
		} else {
			System.out.println(Simulator.getTime() + ": test false " +
transact.getNumber());
			outputDock = falseOutputDock;
		}
```

7. Апробирование разработанной подсистемы

Для обеспечения возможности тестирования работы подсистемы процессно-ориентированного подхода был разработан и внедрен в систему набор юнит-тестов. Запуск тестов и проверка корректности их работы осуществляется с помощью фреймворка JUnit. JUnit-тест представляет из себя отдельный класс на языке Java, содержащий методы, описывающие алгоритмы тестирования.

Тестирование алгоритмов работы процессного подхода осуществляется по следующей схеме:

- конфигурация процессной части имитационной модели, описание блоков и связей между ними
- инициализация симулятора Rao X, задание условия завершения модели
- запуск прогона модели
- проверка времени завершения работы модели и возвращаемого результата

Код разработанных тестов приведен в приложениях 1, 2, 3.

8. Заключение

По завершению работы над данным курсовым проектом были получены следующие результаты:

- Процессно-ориентированный подход реализован и внедрен в систему имитационного моделирования Rao X
- Реализованы требуемые функциональные возможности блоков:
  - создания (Generate) и удаления (Terminate) транзактов
  - очереди (Queue)
  - захвата (Seize) и освобождения (Release) ресурсов
  - задержки (Advance) транзактов
  - ветвления (Test) модели
- Разработаны и реализованы JUnit-тесты для проверки корректности работы подсистемы
- Проведено пошаговое сравнение результатов процессно-ориентированного подхода с эталонными

Список используемых источников

1. **Емельянов В.В., Ясиновский С.И.** Введение в интеллектуальное имитационное моделирование сложных дискретных систем и процессов. Язык РДО. - М.: "Анвик", 1998. - 427 с., ил. 136.
2. **Документация по языку РДО** [http://raox.ru/docs/reference/base_types_and_functions.html]
3. **Java™ Platform, Standard Edition 7. API Specification.** [http://docs.oracle.com/javase/7/docs/api/]

Список использованного программного обеспечения

1. Eclipse IDE for Java Developers Luna Service Release 1 (4.4.1)
2. openjdk version "1.8.0_40-internal"
3. UMLet v13.3
4. Inkscape v0.48.4
5. yEd Graph Editor v3.14.4
6. Microsoft® Office Word 2010
7. Microsoft® Office Excel 2010
8. Microsoft® Visio 2013

Приложение 1 – Тест линейной модели

```
package ru.bmstu.rk9.rao.tests;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

import org.junit.Test;

import ru.bmstu.rk9.rao.lib.process.Advance;
import ru.bmstu.rk9.rao.lib.process.Block;
import ru.bmstu.rk9.rao.lib.process.Generate;
import ru.bmstu.rk9.rao.lib.process.Link;
import ru.bmstu.rk9.rao.lib.process.Release;
import ru.bmstu.rk9.rao.lib.process.Resource;
import ru.bmstu.rk9.rao.lib.process.Seize;
import ru.bmstu.rk9.rao.lib.process.Terminate;
import ru.bmstu.rk9.rao.lib.simulator.Simulator;
import ru.bmstu.rk9.rao.lib.simulator.Simulator.SimulationStopCode;

public class LinearProcessTest {

	@Test
	public void test() {
		ProcessTestSuite.initEmptySimulation();
		Simulator.addTerminateCondition(() -> Simulator.getTime() > 1000);
		Simulator.getProcess().addBlocks(generateSituation());
		SimulationStopCode simulationStopCode = Simulator.run();
		System.err.println(simulationStopCode);
		assertEquals("linear_process_test", SimulationStopCode.RUNTIME_ERROR,
				simulationStopCode);
		assertTrue(Math.abs(Simulator.getTime() - 40) < 1e16);
	}

	private List<Block> generateSituation() {
		List<Block> blocks = new ArrayList<Block>();
		Generate generate = new Generate(() -> 10);
		Terminate terminate = new Terminate();
		Advance advance = new Advance(() -> 15);
		Resource resource = new Resource();
		Seize seize = new Seize(resource);
		Release release = new Release(resource);
		blocks.add(generate);
		blocks.add(seize);
		blocks.add(advance);
		blocks.add(release);
		blocks.add(terminate);
		Link.linkDocks(generate.getOutputDock(), seize.getInputDock());
		Link.linkDocks(seize.getOutputDock(), advance.getInputDock());
		Link.linkDocks(advance.getOutputDock(), release.getInputDock());
		Link.linkDocks(release.getOutputDock(), terminate.getInputDock());

		return blocks;
	}

}
```

Приложение 2 – Тест модели с использованием блока очереди

```
package ru.bmstu.rk9.rao.tests;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

import org.junit.Test;

import ru.bmstu.rk9.rao.lib.process.Advance;
import ru.bmstu.rk9.rao.lib.process.Block;
import ru.bmstu.rk9.rao.lib.process.Generate;
import ru.bmstu.rk9.rao.lib.process.Link;
import ru.bmstu.rk9.rao.lib.process.Queue;
import ru.bmstu.rk9.rao.lib.process.Release;
import ru.bmstu.rk9.rao.lib.process.Resource;
import ru.bmstu.rk9.rao.lib.process.Seize;
import ru.bmstu.rk9.rao.lib.process.Terminate;
import ru.bmstu.rk9.rao.lib.simulator.Simulator;
import ru.bmstu.rk9.rao.lib.simulator.Simulator.SimulationStopCode;

public class QueueProcessTest {

	@Test
	public void test() {
		ProcessTestSuite.initEmptySimulation();
		Simulator.addTerminateCondition(() -> Simulator.getTime() > 55);
		Simulator.getProcess().addBlocks(generateSituation());
		SimulationStopCode simulationStopCode = Simulator.run();
		System.err.println(simulationStopCode);
		assertEquals("linear_process_test",
SimulationStopCode.TERMINATE_CONDITION, simulationStopCode);
	}

	private List<Block> generateSituation() {
		List<Block> blocks = new ArrayList<Block>();
		Generate generate = new Generate(() -> 10);
		Terminate terminate = new Terminate();
		Advance advance = new Advance(() -> 15);
		Resource resource = new Resource();
		Seize seize = new Seize(resource);
		Release release = new Release(resource);
		Queue queue = new Queue();
		blocks.add(generate);
		blocks.add(queue);
		blocks.add(seize);
		blocks.add(advance);
		blocks.add(release);
		blocks.add(terminate);
		Link.linkDocks(generate.getOutputDock(), queue.getInputDock());
		Link.linkDocks(queue.getOutputDock(), seize.getInputDock());
		Link.linkDocks(seize.getOutputDock(), advance.getInputDock());
		Link.linkDocks(advance.getOutputDock(), release.getInputDock());
		Link.linkDocks(release.getOutputDock(), terminate.getInputDock());

		return blocks;
	}

}
```

Приложение 3 – Тест модели с использованием блока ветвления

```
package ru.bmstu.rk9.rao.tests;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;
import ru.bmstu.rk9.rao.lib.process.Advance;
import ru.bmstu.rk9.rao.lib.process.Block;
import ru.bmstu.rk9.rao.lib.process.Generate;
import ru.bmstu.rk9.rao.lib.process.Link;
import ru.bmstu.rk9.rao.lib.process.Queue;
import ru.bmstu.rk9.rao.lib.process.Release;
import ru.bmstu.rk9.rao.lib.process.Resource;
import ru.bmstu.rk9.rao.lib.process.Seize;
import ru.bmstu.rk9.rao.lib.process.Terminate;
import ru.bmstu.rk9.rao.lib.process.Test;
import ru.bmstu.rk9.rao.lib.simulator.Simulator;
import ru.bmstu.rk9.rao.lib.simulator.Simulator.SimulationStopCode;

public class BranchedProcessTest {

	@org.junit.Test
	public void test() {
		ProcessTestSuite.initEmptySimulation();
		Simulator.addTerminateCondition(() -> Simulator.getTime() > 45);
		Simulator.getProcess().addBlocks(generateSituation());
		SimulationStopCode simulationStopCode = Simulator.run();
		System.err.println(simulationStopCode);
		assertEquals("linear_process_test",
SimulationStopCode.TERMINATE_CONDITION, simulationStopCode);
	}

	private List<Block> generateSituation() {
		List<Block> blocks = new ArrayList<Block>();
		Generate generate = new Generate(() -> 10);
		Terminate terminate = new Terminate();
		Advance advance = new Advance(() -> 15);
		Resource resource = new Resource();
		Seize seize = new Seize(resource);
		Release release = new Release(resource);
		Queue queue = new Queue();
		Test test = new Test();
		blocks.add(generate);
		blocks.add(test);
		blocks.add(queue);
		blocks.add(seize);
		blocks.add(advance);
		blocks.add(release);
		blocks.add(terminate);
		Link.linkDocks(generate.getOutputDock(), test.getInputDock());
		Link.linkDocks(test.getTrueOutputDock(), queue.getInputDock());
		Link.linkDocks(queue.getOutputDock(), seize.getInputDock());
		Link.linkDocks(seize.getOutputDock(), advance.getInputDock());
		Link.linkDocks(advance.getOutputDock(), release.getInputDock());
		Link.linkDocks(release.getOutputDock(), terminate.getInputDock());
		Link.linkDocks(test.getFalseOutputDock(), terminate.getInputDock());

		return blocks;
	}
}
```