САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Автоматизированное построение клеточного автомата
на основе описания логики игры

Автор магистерской диссертации Столбов С.А.

Научный руководитель Шалыто А.А.

Санкт-Петербург
2007

Аннотация

Результаты позволяют упростить реализацию игр с преобладающими локальными взаимодействиями. Предлагается подход и инструментальное средство для автоматической генерации программного кода по описанию поведения игровых объектов. Вводится понятие клеточного автомата с распределенной функцией переходов, модель которого значительно облегчает процесс формализации логики игры. Предлагается язык формализации логики игры, с помощью которого можно описывать поведение игровой системы в рамках модели расширенного клеточного автомата или клеточного автомата с распределенной функцией переходов.

В работе доказано, что любой клеточный автомат с распределенной функцией переходов представим в виде классического клеточного автомата, и разработан алгоритм такого преобразования. На этой базе строится преобразователь-генератор, который по описанию игры в рамках модели клеточного автомата с распределенной функцией переходов, строит классический клеточный автомат, а затем генерирует код на языке *C++*, реализующий логику описываемой игры.

Приводится пример реализации упрощенной версии игры *Pac-Man* с использованием преобразователя-генератора для автоматизированного построения программного кода.

Оглавление

Введение

В работе [1] показано, что многие игры могут быть удобно реализованы на основе клеточных автоматов. Предложенный метод позволяет проектировать поведение клеток игрового поля, строя графы переходов, а затем по ним однозначно строить программный код. Несмотря на достоинства такого подхода, существует ряд факторов, которые делают использование предложенной схемы неудобным для разработчика.

Во-первых, нет автоматизированных средств генерации программного кода, реализующего клеточный автомат по графам переходов, хотя возможность такого построения была в [1] заявлена. В этой работе также не было представлено никаких соображений ни о способах построения таких средств, ни о возможных способах формализации алгоритмов клеточного автомата в текстовом виде.

Во-вторых, функция переходов для клетки игрового поля может быть очень сложной, что затрудняет ее построение. Более того, часто получается так, что строить эту функцию, в силу архитектуры клеточного автомата, приходится противоестественным для человека образом.

Настоящая работа призвана дополнить работу [1], устранив указанные недостатки.

1. Модель клеточного автомата

1.1. Клеточные автоматы

Клеточные автоматы — дискретные детерминированные системы, поведение которых полностью определяется в терминах локальных взаимодействий [2].

Клеточные автоматы были использованы Джоном фон Нейманом для исследования самовоспроизведения [3]. Однако они применяются и для совершенно других целей. Если универсальной моделью для последовательных вычислений считается машина Тьюринга, то клеточные автоматы являются такой моделью для параллельных вычислений [4].

Отличительной особенностью клеточных автоматов является то, что они имеют структуру, объединяющую вычислительную компоненту и данные, с которыми автомат оперирует. Эта особенность хорошо сочетается с принципами объектно-ориентированного программирования, поскольку класс обладает такими же свойствами.

Как правило, рассматривают одномерные либо двумерные клеточные автоматы. Хотя количество измерений ничем не ограничено, трехмерные и выше клеточные автоматы сами по себе практически не исследуются, либо имеют очень схожие с двумерными свойства. Отдельного внимания заслуживают одномерные клеточные автоматы. В частности, существует такой одномерный клеточный автомат, для которого доказана универсальность — с его помощью реализуется любой вычислительный процесс [5]. В данной работе будут рассматриваться только двумерные клеточные автоматы.

Клеточный автомат — это однородная «решетка», в каждой клетке которой находится конечный автомат. В двумерном автомате наиболее

часто используется тетрагональная решетка. Возможны также клеточные автоматы с треугольной или гексагональной решеткой. Отличительная особенность конечных автоматов, являющихся элементами клеточного автомата: в них не используются входные и выходные воздействия, а на каждом шаге новое состояние каждого конечного автомата определяется его собственным состоянием и состоянием его соседей (элементов окрестности).

Понятие окрестности клетки задается как свойство клеточного автомата, причем для каждой из клеток это понятие одинаково. Например, для двумерного клеточного автомата с тетрагональной решеткой чаще всего в качестве соседей рассматривают либо четыре клетки, соприкасающиеся с данной сторонами, либо восемь клеток, соприкасающиеся с данной хотя бы углами.

Клеточные автоматы обычно обладают следующими свойствами [6]:

- решетка однородна;
- взаимодействия локальны;
- множество состояний клетки конечно;
- изменения значений всех клеток происходят одновременно.

Иногда какие-то из этих свойств могут не выполненяться. Например, в реализации, рассмотренной в работе [7], решетка не является однородной, однако остальные свойства клеточных автоматов выполняются.

Формально клеточный автомат — четверка объектов:

$$A = < R, S, O, f >,$$

где R — решетка автомата (рабочее пространство);

S — конечное множество состояний клетки;

O — определение окрестности клетки (множество соседей);

$f : S \times S^{|O|} \to S$ — функция переходов.

В данной работе рассматриваются двумерные клеточные автоматы на **тетрагональной решетке**, а окрестностью считаются клетки, которые соприкасаются сторонами или углами. При практической реализации для удобства хранения такая решетка может быть представлена в виде матрицы — двумерного массива (рис. 1).

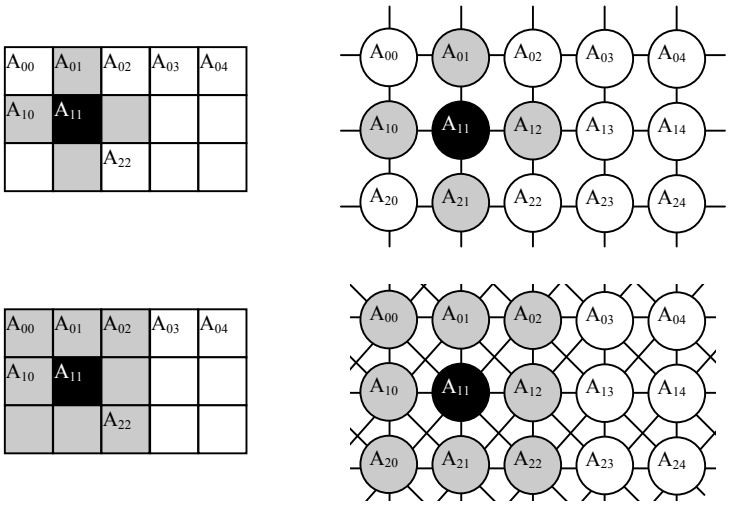

Рис. 1. Тетрагональная решетка с окрестностью из четырех клеток (сверху) и восьми клеток (снизу)

1.2. Расширенный клеточный автомат

Покажем, что для описания сложных систем, которыми являются некоторые компьютерные игры, оказывается недостаточно рассматривать только один автомат в каждой клетке.

Рассмотрим пример. Пусть имеется поле, каждая является «ящиком», в который может быть помещен предмет. «Ящик» может быть закрыт или открыт. Это его состояние. В «ящике» может храниться геометрическая фигура: куб, шар или тетраэдр. При этом поведение «ящика» зависит, как от того, открыт он или закрыт, так и от того, что в

нем содержится. Удобно использовать для хранения информации о «ящике» две ячейки памяти: для состояния ящика и для формы фигуры.

Докажем, что такую структуру в общем случае также можно представить в виде клеточного автомата, — поместить все свойства каждой клетки лишь в одну многозначную ячейку памяти, которая будет являться состоянием клетки. В дальнейшем все ячейки памяти будут рассматриваться как многозначные.

Доказательство. Пусть каждая клетка содержит k ячеек памяти, которые могут хранить значения из множеств S_i, $i = \overline{0,k-1}$. Если считать, что значение каждой ячейки — состояние некоторого автомата, то формально клеточный автомат, описывающий поведение поля, запишется в виде:

$$A' = < R,\ \{S_i\}_{i=0}^{k-1},\ O,\ \{f_i\}_{i=0}^{k-1} >,$$

где $f_i : (S_0 \times \cdots \times S_{k-1}) \times (S_0 \times \cdots \times S_{k-1})^{|O|} \to S_i$ — функция, которая ставит состояние номер «i» текущей клетки в зависимость от всех состояний соседей и ее самой.

Пусть $S := S_0 \times \cdots \times S_{k-1}$. Это множество можно назвать множеством «общих состояний» клетки. Оно содержит все возможные комбинации состояний из наборов S_i. Поскольку $\forall i \mid S_i \mid < \infty$, то и $\mid S \mid < \infty$.

Введем также «общую функцию переходов» $f : S \times S^{|N|} \to S$:

$$f(s^0, s^1, \ldots, s^{|O|}) = \left(f_0(s^0, s^1, \ldots, s^{|O|}), \ldots, f_{k-1}(s^0, s^1, \ldots, s^{|O|}) \right),$$

где $s^i \in S$.

В рассмотренном примере с «ящиками» $k = 2$. Пусть $S_0 = \{open, closed\}$, $S_1 = \{cube, sphere, tetrahedron\}$. Тогда

$$S = \{(open, cube), (open, sphere), (open, tetrahedron), \ldots, (closed, tetrahedron)\}.$$

При этом $|S| = |S_0||S_1| = 2 \cdot 3 = 6$.

Таким образом, с использованием введенных обозначений можно построить клеточный автомат

$$A =< R, S, O, f >,$$

поведение которого будет повторять поведение автомата A', причем автомат A будет являться классическим клеточным автоматом.

Назовем клеточный автомат, имеющий в каждой клетке несколько ячеек памяти, **расширенным клеточным автоматом**.

В данной работе будем использовать такие клеточные автоматы. Приведенное доказательство показывает, что такой автомат теоретически представим в виде классического, однако такое представление использоваться не будет.

Обозначим символом A_{ij} клетку автомата, находящуюся в ряду «*i*» и столбце «*j*»[1]. Для того, чтобы обозначить значение определенной ячейки памяти расширенного клеточного автомата, используется верхний индекс: A_{ij}^n — значение ячейки памяти с номером «*n*» клетки A_{ij}. Эта ячейка памяти может содержать **состояния** клетки (в рассмотренном примере это одно состояние: «ящик открыт» или «ящик закрыт») или **свойства** клетки (в примере — форма хранящейся фигуры). Для того, чтобы отличать состояния клетки от ее свойств, будем обозначать свойства символом D_{ij}^n. Обратим внимание на то, что между состояниями и свойствами клетки сохраняется сквозная нумерация.

[1] Избрана принятая в линейной алгебре индексация элементов матрицы. При этом координаты перечисляются в порядке «строка, столбец»: *i* — *y*-координата элемента, а *j* — *x*-координата, а не наоборот.

Таким образом, в рассмотренном примере A_{ij}^0 — состояние ящика (открыт или закрыт), D_{ij}^1 — форма фигуры.

1.3. Архитектура системы

Управляющие функции игры можно декомпозировать на две части: глобальные и локальные [1]. Так же делится и сама система:

- клеточный автомат;
- внешние супервизоры.

В качестве внешних супервизоров используются конечные автоматы. Использование конечных автоматов для внешнего управления позволяет соблюдать столь же строгую формализацию задачи описания управляющей логики, как и при описании клеточного автомата [8]. Структурная схема системы и структурная схема клетки поля изображены на рис. 2, 3.

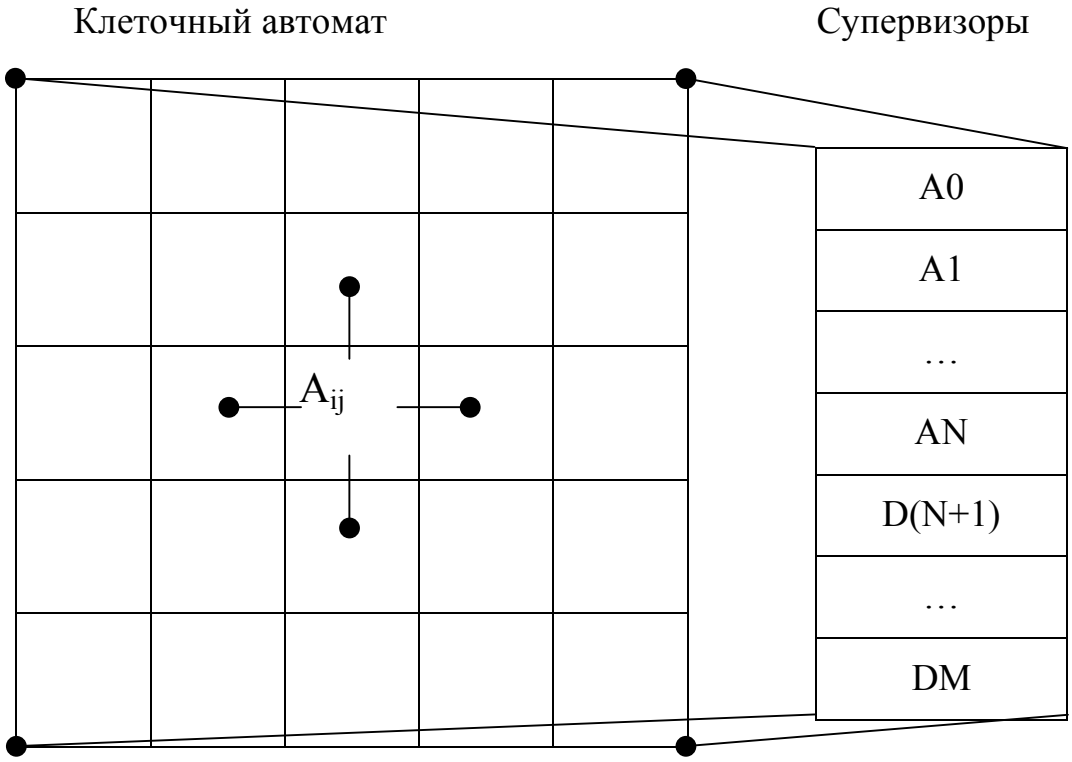

Рис. 2. Структурная схема системы

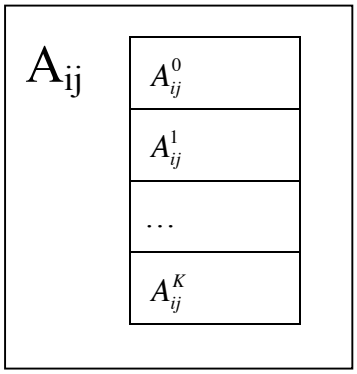

Рис. 3. Структурная схема клетки поля с *K* автоматами

Клеточный автомат:

- отвечает за локальные взаимодействия, включая перемещение всех объектов;
- каждая клетка получает информацию только о состоянии своих соседей и состояниях супервизоров;
- каждая клетка может модифицировать состояние некоторых супервизоров.

Супервизоры:

- отвечают за глобальные действия, взаимодействие с пользователем и искусственный интеллект;
- имеют полный доступ ко всем клеткам поля.

Помимо супервизоров понадобятся и вспомогательные «глобальные» ячейки данных, например, ячейка, в которой хранится текущий счет. Они схожи по роли с супервизорами. Поэтому удобно хранить их данные вместе с состояниями этих автоматов. Будем обозначать супервизоры символами *A0*, *A1*, …, *AN*, а глобальные ячейки данных — *D(N+1)*, *D(N+2)*, …, *DM*. Таким образом, для супервизоров и глобальных ячеек данных, как и для состояний и свойств клеток поля, используется сквозная нумерация.

Для расширенного клеточного автомата и супервизоров принята следующая система обозначений:

A_{ij} — клетка поля в столбце *j* строки *i*;

A_{ij}^{n} — состояние «*n*» клетки A_{ij};

AN — состояния супервизоров;

DN — состояния глобальных ячеек.

При создании игр важно следить за тем, чтобы одни клетки не могли получать значения состояний супервизоров и глобальных ячеек, установленные другими клетками, поскольку это нарушило бы свойство одновременности изменения состояний всех клеток и свойство локальности взаимодействий в клеточном автомате. Проще всего учесть это ограничение, разделив супервизоры и глобальные ячейки на два класса:

- супервизоры и ячейки, состояния которых клетки поля могут изменять, но не могут считывать для последующего использования;
- супервизоры и ячейки, состояния которых могут только считываться клетками.

Если клетка должна произвести с числом, хранящимся в глобальной ячейке, некоторую математическую операцию, например, увеличить счет на единицу, то это запись.

В любом случае при изменении состояния супервизоров и значений глобальных ячеек необходимо следить за тем, чтобы результат суперпозиции всех операций не зависел от того, в каком порядке они производятся.

Пример набора супервизоров и глобальных ячеек памяти:

- супервизор *A0* имеет два состояния: 0 — игра продолжается, 1 — игра окончена;
- ячейка *D1* содержит счет (целое число);
- ячейка *D2* содержит значение ноль, если призовой элемент не действует, или число большее нуля, что соответствует количеству шагов клеточного автомата до окончания действия призового элемента.

В данном случае состояния автомата *A0* и значение ячейки *D1* клетками изменяются, а значение ячейки *D2* только считывается.

1.4. Описание поведения клеточного автомата

Выше были описаны различные характеристики клеточных автоматов, но множества состояний и функции переходов не приводились. Предположим, что рассматривается не расширенный клеточный автомат, а клеточный автомат, каждая клетка которого содержит лишь одну ячейку памяти. Поскольку клеточные автоматы на тетрагональном поле легко могут быть представлены в виде матрицы, будем считать, что конечные автоматы клеток организованы именно так. Состояние каждого элемента структуры будем обозначать, как было оговорено в разд. 1.2, символом A_{ij}^0, где i и j — номера строки и столбца, в которых содержится данный автомат, а 0 — номер единственной ячейки памяти.

Соответственно, в случае тетрагонального поля с восемью соседями функция переходов f получает на вход девять чисел. При этом новое состояние клетки в позиции (i, j) вычисляется следующим образом:

$$A_{ij}^{0^{new}} = f(A_{ij}^0, A_{i,j+1}^0, A_{i-1,j+1}^0, A_{i-1,j}^0, A_{i-1,j-1}^0, A_{i,j-1}^0, A_{i+1,j-1}^0, A_{i+1,j}^0, A_{i+1,j+1}^0).$$

Такой порядок следования аргументов в этом соотношении выбран по порядку следования соседей против часовой стрелки, начиная с правой оси (как принято откладывать углы в математике). Новое значение обозначено иначе, нежели то, которое является аргументом функции. Это выполнено для обеспечения одновременности смены состояний при последовательном вычислении значений функции, о чем речь пойдет ниже.

Функцию переходов клетки поля, как и в обычных конечных автоматах, можно формировать в виде **графа переходов**. Его вершинами служат возможные значения состояния клетки (элементы множества *S*), а дуги показывают изменение состояний в зависимости от конфигурации состояний соседей и других элементов системы. Построение графа выполняется так же, как для конечных автоматов, с той лишь разницей, что вместо входных переменных в качестве условий переходов на дугах указываются условия, составленные из состояний соседей, супервизоров и значений глобальных ячеек памяти, а вместо выходных воздействий — действия над данными, которые следует произвести. В качестве языка для формализации условий удобно использовать синтаксис не математической логики, а языка программирования *C*.

1.5. Примеры

Пример 1. Рассмотрим клеточный автомат, реализующий «цепную реакцию» и ведущий подсчет «среагировавших» клеток в глобальной ячейке *D0* — счетчике. Пусть в начале процесса все клетки имеют состояние «0». Как только (например, как результат действий пользователя) какая-то клетка перейдет в состояние «1», при следующем шаге четыре ее соседа (слева, справа, сверху и снизу) также перейдут в это состояние («среагируют»). На следующем шаге все повторится, но

клеток в состоянии «1» будет уже больше. На рис. 4 проиллюстрирован этот процесс.

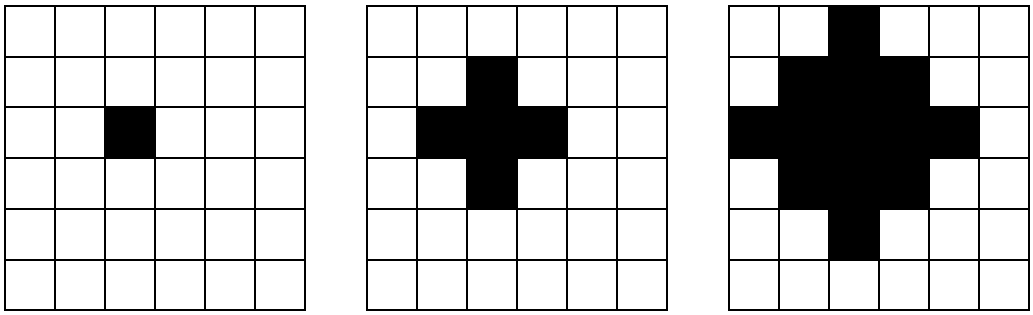

Рис. 4. Цепная реакция на первом, втором и третьем шагах

Граф переходов для клетки такого клеточного автомата изображен на рис. 5.

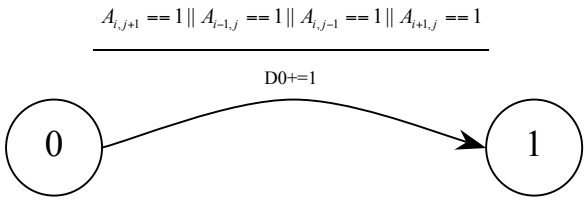

Рис. 5. Граф переходов клеточного автомата, реализующего цепную реакцию

В случае наличия сложных условий, которые с незначительными модификациями используются на нескольких дугах, можно использовать **макросы** — функции тех же аргументов, что и функция f, возвращающие элемент множества $S \cup \{\bot\}$ — состояние одного из соседей или значение «ложь».

Пример 2. Рассмотрим другой, более сложный вариант «цепной реакции» с двумя типами реакции и двумя типами «среагировавших» клеток. Если у «несреагировавшей» клетки есть один из четырех соседей по сторонам, который «среагировал» первым образом, она перейдет в первое состояние. Если есть сосед, среагировавший вторым образом, то она перейдет во второе состояние. Если клетка имеет несколько соседей, среагировавших различным образом, то установим следующий

приоритет: «справа», «сверху», «слева», «снизу». Подсчет клеток, «среагировавших» тем и другим образом, будет производиться в глобальных ячейках *D0* и *D1* — счетчиках. Процесс жизни такого клеточного автомата проиллюстрирован на рис. 6.

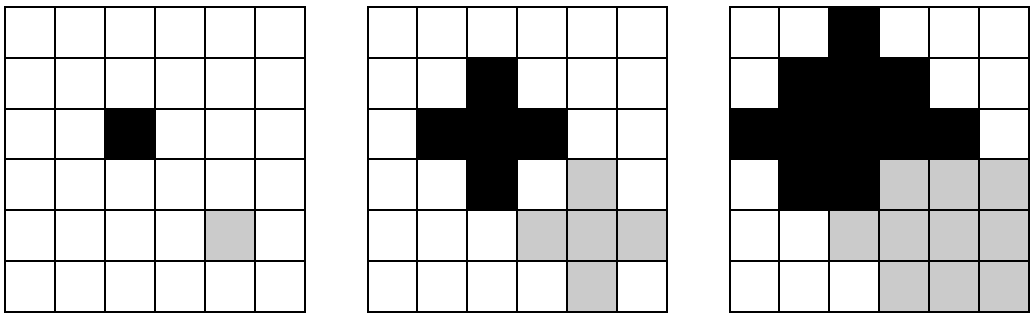

Рис. 6. «Двуцветная» цепная реакция на первом, втором и третьем шаге

На рис. 7 изображен граф переходов клетки клеточного автомата с тремя состояниями, реализующего такой процесс.

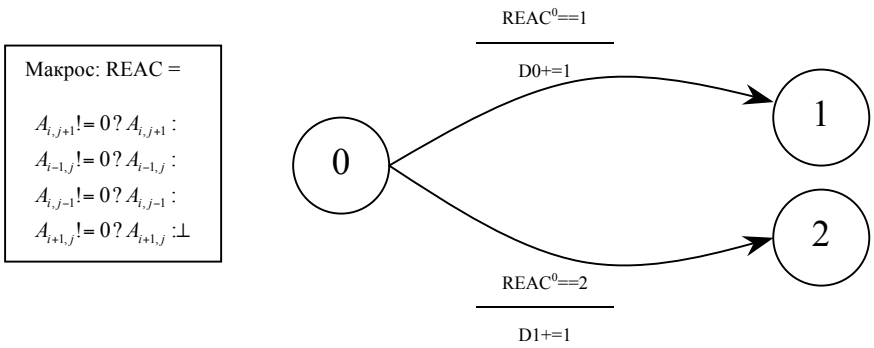

Рис. 7. Граф переходов клеточного автомата, реализующего «двуцветную» реакцию, использующий макрос

1.6. Выводы по главе 1

1. Введен расширенный клеточный автомат. Доказано, что его можно представить через классический автомат.

2. Предложена структура системы управления, состоящая из клеточного автомата и набора супервизоров.

3. Введены обозначения для элементов системы.

2. Формализация поведения клеточного автомата

Для реализации алгоритма построения программного кода необходимо формализовать запись алгоритма, реализуемого клеточным автоматом.

Описание лишь в виде графа переходов неудобно по ряду причин. Во-первых, такой подход не позволяет в явном виде задать все возможные значения и семантику для всех параметров клеток поля. Во-вторых, на дугах невозможно обозначать сложные действия, если они будут использоваться (даже такие традиционные как, например, подсчет количества соседей определенного типа). В-третьих, в случае большого количества состояний граф может стать слишком сложным или непланарным, что также приводит к неудобству его чтения.

В то же время, визуализационные достоинства графа переходов не следует и недооценивать, однако, в качестве единственного средства представления, полностью описывающего подавляющую часть аспектов поведения игровой системы, всегда его использовать не получится. Целесообразно использовать граф переходов на этапе проектирования для обозначения основных черт поведения клеток, но детальную формализацию производить **в текстовом виде**. В настоящей работе предлагается один из возможных способов описания поведения клеточного автомата, который позволяет генерировать код на языке *C++*, реализующий этот автомат.

В дальнейшем будем называть вводимый язык «форматом формализации логики игры», а тексты на этом языке — «кодом формализации».

2.1. Синтаксис описания условий и действий

Поскольку формат формализации логики игры будет применяться для генерации кода на языке *C++*, целесообразно использовать язык *C* или *C++* для записи выражений. Это позволит избавиться от необходимости семантического анализа.

Для обозначения клеток будем использовать обозначения, удобные в рамках модели. Они приведены в табл. 1.

Таблица 1. Соответствие между математическими обозначениями и обозначениями в коде формализации клеточного автомата

Математическое обозначение	Обозначение в формате
A_{ij}	`A[i,j]`
A_{ij}^{n}	`A[i,j]n`
AN	`GN`

Например, действие «присвоить состоянию 2 значение 5» будет записываться как:

```
A[i,j]2 = 5;
```

«Присвоить состоянию 3 состояние 3 соседа справа»:

```
A[i,j]3 = A[i,j+1]3;
```

Условие равенства состояния 3 состоянию 3 соседа сверху:

```
A[i,j]3 == A[i-1,j]3
```

2.2. Определение состояний

Каждая клетка может иметь конечное число состояний, допустимой областью каждого из которых является конечное множество.

Пусть, например, каждая клетка имеет два параметра: цвет (красный, желтый или синий) и возраст (целое число). Будем записывать это следующим образом:

```
A0 = Color (Red, Blue)
A1 = Age (int)
```

В качестве типа без перечисления значений в базовом варианте возможен только тип *int* (это обусловлено тем, что множество состояний клетки удобно хранить в виде массива, элементы которого должны иметь один и тот же тип данных).

В случае, если состояние определено как перечисляемый тип с явным указанием значений, в дальнейшем можно ссылаться на эти значения по их имени в том виде, в котом они были названы, например:

```
A[i,j]0 = 'Red';
```

2.3. Определение макросов

При формировании функции переходов на основе графа переходов удобно использовать макросы. Макросы, в свою очередь, удобно реализовывать в коде в виде макросов препроцессора *C*. Предоставим возможность указывать определять макросы. Макрос *REAC* из примера 2 можно записать в следующем виде:

```
macro REAC(i,j) = A[i,j+1]0 != 0 ? A[i,j+1]:
      A[i-1,j]0 != 0 ? A[i-1,j]:
      A[i,j-1]0 != 0 ? A[i,j-1]:
      A[i+1,j]0 != 0 ? A[i+1,j]: 0;
```

2.4. Определение правил перехода

Каждый переход клеточного автомата обозначается на графе переходов ребром между вершиной, соответствующей состоянию, из которого происходит переход, и вершиной, соответствующей состоянию, в которое переход производится. Если переход условный, то ребро

промаркировано условием перехода. Ребро также может быть промаркировано действием, которое производится в случае перехода.

Таким образом, основными параметрами каждого перехода являются:

- начальное состояние;
- конечное состояние;
- условие;
- дополнительное действие.

Соответственно этому разделению, секция объявления перехода будет состоять из трех разделов:

1. Начальное и конечное состояния.
2. Условие (`IF`).
3. Дополнительное действие (`ALSO`).

Важно обратить внимание на то, что переход не является условной управляющей конструкцией псевдоязыка. Это текстовая запись ребра графа функции переходов клеточного автомата.

Записывать переход будем так:

```
A[i,j]0: <нач. сост.> -> <кон. сост.>
      IF
            <условие>
      ALSO
            <дополнительные действия>
      END
```

Переходы из примера разд. 1.5 будут реализованы следующим образом:

```
A[i,j]0: 0 -> 'Red'
      IF
            REAC(i,j)0 == 'Red'
      ALSO
            G0 += 1;
      END

A[i,j]0: 0 -> 'Blue'
      IF
            REAC(i,j)0 == 'Blue'
```

```
ALSO
      G1 += 1;
END
```

2.5. Описание супервизоров

Описание набора супервизоров практически ничем не отличается от описания поведения отдельной клетки. Единственным существенным отличием является то, что в клетке управляющим автоматом является один, а среди супервизоров их может быть несколько (переходы в них производятся в порядке нумерации этих автоматов).

Описание супервизоров производится аналогичным образом с описанием структуры клетки клеточного автомата. Определение состояний:

```
G0 = RedCount (int)
G1 = BlueCount (int)
```

Определение переходов производится такой же конструкцией «ПЕРЕХОД—IF—ALSO», как и для клетки поля. В примере из разд. 1.5 управляющих супервизоров нет.

2.6. Синтаксис файла описания клеточного автомата в БНФ

В форме Бэкуса-Наура (БНФ) формат описания клеточного автомата можно записать в следующем виде:

```
<CA Program> ::=
%CELLSTATES
<CA State declarations>
%END
%GLOBALSTATES
<A State declarations>
%END
%MACROS
<Macro declarations>
%END
%CELLAUT
<CA Transitions>
%END
%GLOBALAUT
<A Transitions>
%END
```

```
<CA State declarations> ::= <CA State declaration> {<CA State
declaration>}
<CA State declaration> ::= A<number> = <identifier> (<Type>)

<A State declarations> ::= <A State declaration> {<A State declaration>}
<A State declaration> ::= G<number> = <identifier> (<Type>)

<Macro declarations> ::= <Macro declaration> {<Macro declaration>}
<Macro declaration> ::= macro <identifier>(i,j) = <C expression> ;

<CA Transitions> ::= <CA Transition> {<CA Transition>}
<CA Transition> ::= A[i,j]0: <state> -> <state> IF <C expression> ALSO <C
statements> END

<A Transitions> ::= <CA Transition> {<CA Transition>}
<A Transition> ::= G<number>: <state> -> <state> IF <C expression> ALSO <C
statements> END

<Type> ::= <Enum type> | <Base type>
<Enum type> ::= <identifier> {, <identifier>}
<Base type> ::= int

<C expression> ::= /выражение на языке C/
<C statements> ::= <C statement> {<C statement>}
<C statement> ::= /операция на языке C/

<state> ::= <number> | '<identifier>'

<digit> ::= 0|1|2|3|4|5|6|7|8|9
<number> ::= <digit>{<digit>}
<alpha> ::= a|b|c|...|y|z|A|B|C|...|Y|Z|_
<identifier> ::= <alpha>{<alpha>|<digit>} | '<alpha>{<alpha>|<digit>}'
```

Нетерминалы «C expression» и «C statement» не раскрыты из соображений экономии места (их раскрытие потребовало бы включения БНФ всего языка *C*). Первый соответствует выражению на языке программирования *C* с допущением обозначений вида «*A[i,j]n*» и «*Gn*», а второй — операции на языке *C* с допущением этих же обозначений.

2.7. Выводы по главе 2

Введен формат формализации логики игры, позволяющий генерировать на его основе код на языке *C++*, реализующий описанный клеточный автомат.

3. Клеточный автомат с распределенной функцией переходов

Клеточный автомат (обычный или расширенный) предусматривает изменение состояния каждой клетки согласно функции переходов (функции состояний этой клетки и клеток окрестности). Этого достаточно для того, чтобы описать многие возможные варианты поведения объектов [1], но это не всегда удобно с практической точки зрения.

Предположим, что персонаж представляется одной клеткой с состоянием «персонаж», и его перемещение вверх происходит при условии «нажата клавиша „вверх“ (глобальная ячейка памяти *D0* содержит значение «2»), клетка сверху имеет состояние „пусто“ или „горошина“»[2]. В таком случае, правило функции переходов будет выглядеть следующим образом:

1. Переход клетки из состояния «персонаж» в состояние «пусто» при условии, что сосед сверху имеет состояние «пусто» или «горошина», а D0 == 2.

2. Переход клетки из состояния «пусто» в состояние «персонаж» при условии, что сосед снизу имеет состояние «персонаж», а D0 == 2.

3. Переход клетки из состояния «горошина» в состояние «персонаж» при условии, что сосед снизу имеет состояние «персонаж», а D0 == 2.

Таким образом, одно условие переводится в три правила перехода, каждое из которых, на первый взгляд, не имеет ничего общего с перемещением персонажа.

---

[2] Ситуация в игре *Pac-Man*, разработанной в 1980 году японской компанией *Namco*.

Предлагается модифицировать клеточный автомат так, чтобы при изменении состояния клетки, вместе с ней модифицироваться могли и соседние клетки. При этом на каждом шаге для каждой клетки вычисляется не только ее функция переходов, но и связанные с ней функции переходов для всех ее соседей. При этом получается, что для каждой клетки вычисляется не одно новое состояние, а сразу несколько (при окрестности в виде квадрата 3×3 новое состояние будет вычислено девять раз).

Обозначим функции переходов символом f_{kl}, где k и l — смещение клетки, от которой вычислена функция, относительно той, для которой эта функция вычислена. Например, функция переходов для соседа слева будет обозначаться как $f_{0,-1}$. Значения функций связанные с заданной клеткой A_{ij}, будем обозначать символом f_{kl}^{ij}. В обобщенном виде значение, вычисленное для клетки a относительно клетки b — f_b^a.

Поскольку для каждой клетки вычисляется некоторое множество N новых состояний, следует построить функцию, которая переводила бы такое множество в одно состояние, которое примет клетка на следующем шаге. Наложим на возможные значения функций следующее ограничение: они должны либо соответствовать старому состоянию клетки, либо другому, но одному для всех функций, значение которых не соответствует старому. Это единственное состояние и будет являться новым состоянием клетки. В случае же, если все функции возвращают старое состояние, переход не производится.

Другими словами, функции, связанные с данной клеткой могут возвращать только два возможных состояния: старое или новое.

3.1. Определение

Формализуя изложенные выше соображения, можно сформулировать определение такого клеточного автомата. **Клеточным автоматом с распределенной функцией переходов** будем называть:

$$A^{dis} =< R,S,O,F >,$$

где R — решетка автомата;

S — множество состояний клетки;

O — окрестность клетки;

$F = \{f\} \cup \{f^o\}_{o \in O}$ — набор функций переходов, где каждая функция $f : S \times S^{|O|} \to S$.

На множество F накладываются следующие ограничения. Пусть $N_a := \{f^a(a,O)\} \cup \{f^a_{a'}(a',O')\}_{a':a \in O'}$[3] — множество значений функций переходов для заданной клетки a. Тогда первое ограничение имеет следующий вид:

$$\forall a \in R : N_a = \{a,a'\} \vee N_a = \{a\}.$$

При этом переход производится в новое состояние a' в случае выполнения левой части дизъюнкции, либо не изменяется в случае выполнения правой.

Второе ограничение выглядит следующим образом:

$$f^{a'}(a',O) = f^{a'}(a',O \cap (O' \cup \{a\}),\tilde{O}) \quad \forall \tilde{O}$$

Из этого ограничения следует, что функции переходов, связанные с данной клеткой, не могут зависеть от клеток, которые не входят в ее собственную окрестность и не являются ею самой.

---

[3] В данной записи *f(O)* не является функцией множества *O*, а является функцией набора аргументов, обозначаемых буквой *O* для компактности записи.

3.2. Доказательство эквивалентности классическому клеточному автомату

Как и при доказательстве эквивалентности расширенного клеточного автомата классическому (разд. 1.2), будем проводить доказательство построением классического клеточного автомата по заданному клеточному автомату с распределенной функцией переходов.

Пусть имеется клеточный автомат с распределенной функцией переходов:

$$A^{dis} = < R,S,O,F > .$$

Как следует из этого определения, единственное отличие такого автомата от классического клеточного автомата заключается в том, что вместо функции переходов f определено множество F. Таким образом, если удастся построить функцию f по F, то эквивалентность будет доказана.

Используя введенное ранее обозначение N_a, определим функцию переходов:

$$f(a,O) = \begin{bmatrix} a, N_a = \{a\}, \\ a', N_a = \{a,a'\}. \end{bmatrix}$$

Остается показать, что такое определение функции корректно: N_a является лишь функцией a и O и ничего более.

Множество N_a было определено как множество значений функций переходов, ассоциированных с клеткой a и зависит от:

1. Состояния клетки a.
2. Состояния окрестности этой клетки.
3. Состояния окрестности каждого из соседей.

С первыми двумя пунктами вопросов не возникает: это и есть a и O. Что касается третьего пункта, то действительно, множество состояний

соседей, за исключением вырожденных случаев, шире множества соседей самой клетки (объединенного с самой клеткой).

В то же время, второе ограничение, наложенное на функции переходов в определении клеточного автомата, говорит о том, что ни одна из этих функций переходов, связанных с a, не может зависеть от состояния клеток, не входящих в $\{a\} \cup O$, а, следовательно, и N_a может зависеть только от состояния этого множества клеток.

Таким образом, функция f действительно является функцией переходов клеточного автомата, и поэтому автомат

$$A = < R, S, O, f >$$

и будет искомым клеточным автоматом, эквивалентным исходному.

3.3. Расширенный клеточный автомат с распределенной функцией переходов

Аналогично тому, как на основе клеточного автомата строился расширенный клеточный автомат, можно сконструировать расширенный клеточный автомат с распределенной функцией переходов.

Доказательство эквивалентности такого клеточного автомата клеточному автомату с распределенной функцией переходов производится аналогично тому, как производилось доказательство эквивалентности расширенного клеточного автомата и клеточного автомата. При этом строится множество возможных состояний для клетки, равное произведению множеств состояний расширенного клеточного автомата.

Таким образом, *клеточный автомат*, *расширенный клеточный автомат*, *клеточный автомат с распределенной функцией переходов* и *расширенный клеточный автомат с распределенной функцией переходов*

являются эквивалентными классами объектов и могут быть реализованы друг через друга.

3.4. Выводы по главе 3

1. Введено понятие клеточного автомата с распределенной функцией переходов.

2. Доказано, что любой клеточный автомат с распределенной функцией переходов может быть реализован через классический клеточный автомат.

4. Преобразование клеточных автоматов

Способ построения функции переходов, используемый в доказательстве, не представляет практической ценности, поскольку он не указывает, как именно должна выглядеть функция переходов в конечном итоге. Утверждается лишь то, что такая функция может быть найдена. На самом же деле, учитывая, что набор функций переходов для всех клеток одинаков, из такого набора можно построить единую функцию переходов.

Будем считать, что функции переходов для соседей заданы на графе переходов текущей клетки в качестве дополнительных действий. Например, правило из примера, приведенного в разд. 3, о том, что персонаж перемещается на одну клетку вверх, если «нажата клавиша „вверх“ (глобальная ячейка памяти *D0* содержит значение 2 и клетка сверху имеет состояние „пусто“ или „горошина“», будет словесно выглядеть следующим образом:

если состояние текущей клетки «персонаж», состояние соседа сверху «пусто» или «горошина», а D0==2, то переход клетки в состояние пусто и переход соседа сверху в состояние «персонаж».

В данном случае первый переход определяет функцию переходов для текущей клетки, а второй — для соседа сверху. Оба перехода имеют общее условие, зависящее только от состояния пересечения окрестностей этих клеток и состояний самих клеток.

Как было указано ранее, такое правило эквивалентно следующему набору правил перехода обычного расширенного клеточного автомата:

1. Переход клетки из состояния «персонаж» в состояние «пусто» при условии, что сосед сверху имеет состояние «пусто» или «горошина», а D0 == 2.

2. Переход клетки из состояния «пусто» в состояние «персонаж» при условии, что сосед снизу имеет состояние «персонаж», а D0 == 2.

3. Переход клетки из состояния «горошина» в состояние «персонаж» при условии, что сосед снизу имеет состояние «персонаж», а D0 == 2.

Стоит задача получить алгоритм такого преобразования. Это позволит по формальному описанию поведения клеточного автомата с распределенной функцией переходов строить обычный клеточный автомат, который в свою очередь может быть легко преобразован в программный код.

Алгоритм будем строить для преобразования расширенного клеточного автомата с распределенной функцией переходов в расширенный клеточный автомат. Окрестность — все клетки, соприкасающиеся углами (восемь соседей). Обработку границ поля учитывать не будем (отсутствие необходимости обрабатывать края можно обеспечить, расставив по краю поля клетки с состоянием «стена»). Управляющим автоматом клеток является автомат с индексом 0. Остальные состояния клетки предназначены только для хранения данных.

4.1. Исходные данные

Исходный набор функций переходов задан в виде формализованного в текстовом виде графа переходов — набора конструкций «ПЕРЕХОД—IF—ALSO»:

```
A[i,j]0: A -> B
      IF
            A[i,j+1] == 2 && D0 == 2 && A[i-1,j] == 4
      ALSO
            A[i,j+1] = 2;
            A[i-1,j] = 4;
            D1 += 2;
      END
```

Для того, чтобы построенный таким образом клеточный автомат соответствовал определению клеточного автомата с распределенной функцией переходов, необходимо следить за тем, чтобы все клетки, относительно которых записаны условия секции `IF`, содержались в окрестности каждой из клеток, значения которых изменяются.

Будем считать, что присутствуют только переходы вида:

```
A[i,j]0: A -> B
      IF
            A[i,j+1]0 == Q1_0 && … && A[i+1,j+1]0 == Q8_0 && cond(A[i,j],
…, A[i+1,j+1], G0, G1, …)
      ALSO
            A[i,j] = FA(A[i,j], A[i,j+1], …, A[i+1,j+1], G0, …);
            A[i,j+1]0 = P1_0;
            A[i,j+1]1 = P1_1;
            …
            A[i-1,j+1]0 = P2_0;
            …
            A[i+1,j+1] = P8;
            G0 = F0(A[i,j], A[i,j+1], …, A[i+1,j+1], G0, …);
            …
      END
```

Здесь условие перехода является конъюнкцией условий относительно состояний управляющего автомата соседей и некоего дополнительного условия *cond* относительно соседей, текущей клетки и супервизоров. Некоторые условия (на практике, большинство из них) могут отсутствовать.

Действие при переходе состоит из:

- изменения состояния текущей клетки (как функции состояния самой клетки, клеток окрестности и супервизоров, обозначенной *FA*);
- изменения состояний соседей (состояния соседей изменяются на фиксированные значения, определенные правилом);
- изменения состояний супервизоров (как функций состояния текущей клетки, клеток окрестности и супервизоров, обозначенных *F0*, *F1*, …, *FN*).

Поведение любого расширенного клеточного автомата с распределенной функцией переходов может быть записано с помощью набора правил такого вида.

4.2. Верификация распределенной функции переходов

При составлении распределенной функции переходов встает задача верификации: соответствует ли клеточный автомат с такими функциями переходов определению клеточного автомата с распределенной функцией переходов. Проверке должны подвергнуться два свойства:

- значение, присваиваемое соседу, зависит только от множества клеток, которое содержится в его окрестности;
- функции переходов не противоречат друг другу.

Первое свойство можно проверить, рассмотрев условия переходов, в которых производится изменение соседей. Условие перехода содержится в блоке `IF`, а также условием является начальное состояние перехода. Поскольку начальное состояние относится к текущей клетке, которая заведомо является соседом для всех своих соседей, проверке следует подвергать только блок `IF`. Таким образом, можно сформулировать **необходимое и достаточное** условие соблюдения первого свойства:

Для каждого из соседей: в каждом переходе, где происходит изменения данного соседа, блок IF должен содержать условия только относительно клеток окрестности данного соседа.

Проверка второго свойства требует проверки семантики условий. Поскольку условие может содержать функцию *cond*, семантический анализ которой может быть затруднен, сформулируем лишь **достаточное** условие отсутствия противоречий. Для каждого из соседей, изменение которого производится разными правилами на разные значения, нужно

удостовериться в том, что разные правила не произведут его изменения в разные состояния на одном и том же шаге. Для того, чтобы изменение не было произведено, достаточно несовместности условий. Если не брать в расчет функцию *cond*, благодаря тому, что условия задаются в виде условий равенства относительно состояний соседей, проверка совместности не составляет труда. Таким образом, **достаточное** условие отсутствия противоречий будет выглядеть следующим образом:

Для каждого из соседей: если существует два правила перехода, изменяющие состояние этого соседа на разные значения, то условия этих переходов относительно соседей должны быть несовместны.

Это правило не является необходимым, поскольку функция *cond*, фигурирующая как член конъюнкции может принимать ложное значение, тем самым делая проверку остальных условий ненужной. Например, если в каком-то переходе функция *cond* тождественно ложна, остальные элементы конъюнкции не играют никакой роли вообще.

4.3. Дробление правил переходов

Каждый набор строк вида (для одного и того же *n*)

```
A[i+k,j+l]m = Pn_m
```

породит правило перехода вида:

```
A[i,j]0: Qn -> Pn
      IF
            A[i-k,j-l]0 == A && A[i-k,j+1-l]0 == Q1_0 && … && A[i+1-k,j+1-
l]0 == Q8_0 && cond(A[i-k,j-l], …, A[i+1-k,j+1-l], G0, G1, …)
      ALSO
            A[i,j]1 = Pn_1;
            A[i,j]2 = Pn_2;
            …
      END
```

где *n* — номер, соответствующий паре чисел *(k,l)*, а условий относительно клеток вида $A_{i+c,\,j+d} : (|c| > 1) \vee (|d| > 1)$, нет (их не должно получиться в связи с ограничениями, наложенными на условие).

Если в условии исходного правила нет критерия относительно состояния управляющего автомата данной клетки «n», переход получится безусловным. Говоря формально, генерируется набор правил по числу возможных состояний управляющего автомата. С точки зрения программирования, эти преобразования будут включены в каждый блок *case*, а также появятся в блоке *default*, т.к. передача управления к ним происходит в любом случае.

После выделения всех правил переходов, относящихся к соседям, исходное правило приобретает вид:

```
A[i,j]0: A -> B
      IF
            A[i,j+1]0 == Q1_0 && … && A[i+1,j+1]0 == Q8_0 && cond(A[i,j],
…, A[i+1,j+1], G0, G1, …)
      ALSO
            A[i,j] = FA(A[i,j], A[i,j+1], …, A[i+1,j+1], G0, …);
            G0 = F0(A[i,j], A[i,j+1], …, A[i+1,j+1], G0, …);
            …
      END
```

Это правило перехода обычного расширенного клеточного автомата, как и все правила, которые были созданы в процессе дробления.

Таким образом, каждое правило перехода может быть преобразовано в набор других правил, число которых может достигать девяти.

4.4. Слияние правил переходов

При дроблении правил переходов может получиться, что найдется пара вершин, между которыми будет больше одного ребра:

```
A[i,j]0: A1 -> B1
      IF
            <condition1>
      ALSO
            <actions1>
      END

A[i,j]0: A1 -> B1
      IF
            <condition2>
```

```
            ALSO
                  <actions2>
            END
```

Такой набор правил можно преобразовать в одно:

```
A[i,j]0: A1 -> B1
      IF
            <condition1> || <condition2>
      ALSO
            if (<condition1>) {
                  <actions1>
            }

            if (<condition2>) {
                  <actions2>
            }
      END
```

4.5. Общий алгоритм приведения

Общий алгоритм приведения расширенного клеточного автомата с распределенной функцией переходов к расширенному клеточному этапу состоит из следующих этапов:

1. Анализ правил на предмет их соответствия модели.
2. Дробление правил.
3. Слияние правил.

Важно отметить, что для такого преобразования не требуется полноценный семантический анализ кода условий или действий. Благодаря тому, что все условия, которые необходимо специально распознавать, являются либо условиями равенства, либо операциями присваивания, их можно выделить на этапе частичного синтаксического анализа.

4.6. Выводы по главе 4

Представлен алгоритм для преобразования правил переходов клеточного автомата с распределенной функцией переходов в правила переходов классического клеточного автомата.

5. Построение преобразователя и генератора кода

5.1. Постановка задачи и выбор средств реализации

Задача интерпретатора — получить на вход описание игры в формате формализации логики игры, представленном в разд. 2, и выдать реализацию логики игры на языке *C++*. Логика игры будет реализована в виде трех классов:

- *GameCell* — клетка автомата;
- *GameCellAut* — система из клеточного автомата и супервизоров;
- *GameGame* — класс игры.

Каждый класс описывается двумя файлами: заголовочным файлом *.h* и файлом реализации класса *.cpp*. В общей сложности должно получиться шесть файлов:

- *Cell.h*
- *Cell.cpp*
- *Cellaut.h*
- *Cellaut.cpp*
- *Game.h*
- *Game.cpp*

Для простоты реализации, преобразователь-генератор не будет производить проверок соответствия модели и синтаксической корректности.

Программа преобразователя и генератора кода будет написана на языке *Perl*. Выбор этого языка программирования обусловлен тем, что он включает в себя мощные средства работы с текстовыми файлами,

встроенную поддержку регулярных выражений и лаконичный синтаксис, а также является платформенно-независимым.

Интерпретатор языка *Perl* (программа *perl*) включен в стандартную поставку большинства современных операционных систем (подавляющее большинство дистрибуций *Linux*, *BSD*, *Mac OS X* и др.). Язык *Perl* платформенно-независим, однако зависит от реализации интерпретатора. Тестирование и отладка программы проводились с использованием интерпретатора версии `v5.8.6 built for darwin-thread-multi-2level`, включенного в поставку операционной системы *Mac OS X 10.4 Tiger*.

5.2. Разбор входного файла

Входной файл имеет следующую структуру:

```
%CELLSTATES
…
%END

%GLOBALSTATES
…
%END

%MACROS
…
%END

%CELLAUT
…
%END

%GLOBALAUT
…
%END
```

Благодаря возможностям языка *Perl*, разбор входного файла описания алгоритма на блоки деклараций производится одной простой командой:

```
$in_file =~
/%CELLSTATES\n(.*?)%END.*?%GLOBALSTATES\n(.*?)%END.*?%MACROS\n(.*?)%
END.*?%CELLAUT\n(.*?)%END.*?%GLOBALAUT\n(.*?)%END/s;
```

В результате такого разбора секции файла хранятся в отдельных переменных[4]:

`$cellstates` — декларация возможных состояний клетки автомата;

`$globalstates` — декларация возможных состояний супервизоров;

`$macros` — описание макросов;

`$cellaut` — описание переходов клетки;

`$globalaut` — описание переходов супервизоров.

Далее, каждый из блоков подвергается индивидуальному разбору и обработке.

5.3. Анализ деклараций

Допустимые состояния клеток автомата и супервизоров

Строки

```
A0 = Color (Red, Blue)
```

распознаются регулярным выражением вида:

```
$cellstate =~ /A(\d)\s*=\s*(\w[\w\s]*?)\((.*?)\)/;
```

В случае состояний с перечисляемым типом каждому символьному имени состояния сопоставляется численный индекс, который будет использоваться в программе. Каждому из допустимых значений также сопоставляется свой численный индекс. Строятся три ассоциативных массива:

---

[4] В языке *Perl* названия всех *скалярных переменных* (содержащих текст или число) должны начинаться с символа «$». Аналогично, название всех *списков* начинается со знака «@», а *ассоциативных массивов* (словарей) — со знака «%».

`%CA_TYPE_VALUES` — содержит численное значение, соответствующее символьному имени значения ячейки;

`%CA_TYPE_ORIGIN` — содержит номер ячейки, к которой относится это символьное имя значения;

`%CA_TYPES` — содержит символьное имя, соответствующее состоянию заданного индекса, относящегося к заданной ячейке.

Аналогично происходит с декларацией состояний супервизоров. Регулярное выражение имеет вид:

```
$globalstate =~ /G(\d)\s*=\s*(\w[\w\s]*?)\((.*?)\)/;
```

Формируются три аналогичных ассоциативных массива, имеющих префикс «A» вместо «CA».

Макросы

Макрос определяет функцию клетки поля, возвращающую один из элементов окрестности либо значение «ложь». Поскольку язык *C* поддерживает макросы препроцессора, макросы можно переводить напрямую в них, что избавляет от необходимости их синтаксического и семантического анализа.

Базовый синтаксический разбор описаний макросов можно задать регулярным выражением:

```
$macro =~ /(\w+)\(i,j\)\s+?=(.*)/;
```

Данные, полученные от анализа макросов, не требуется обрабатывать: они необходимы только для построения кода инструкций препроцессора.

5.4. Анализ правил перехода

Каждое правило перехода клеточного автомата состоит из фиксированных блоков, каждый из которых является обязательным (хотя

и может быть пустым). Благодаря этому, разбор будет выглядеть следующим образом:

```
$cellaut =~ s/A\[([ij,+-1]+)\](\d)?:\s*(\d+|\'\w+\')\s*=\s*
(\d+|\'\w+\')\s+IF\s(.*?)ALSO\s(.*?)END//;
```

На этапе анализа правила не подвергаются обработке, а содержимое блоков правила лишь сохраняются в структурах данных. Формируется пять списков, соответствующих отдельным частям блока перехода:

`@CA_TRANS_ij` — индекс клетки, для которой записан переход (должен всегда быть равным «i,j»);

`@CA_TRANS_oldstate` — состояние, из которого производится переход;

`@CA_TRANS_newstate` — состояние, в которое производится переход;

`@CA_TRANS_if` — условие перехода (блок `IF`);

`@CA_TRANS_also` — действия при переходе (блок `ALSO`).

Полностью аналогичная процедура производится с правилами переходов супервизоров. В отличие от правил переходов клеточного автомата, правила переходов супервизоров не будут подвергнуты синтезу, а будут использованы напрямую для генерации кода программы. Формируется пять списков:

`@GA_TRANS_n` — индекс супервизора, которому соответствует это правило;

`@GA_TRANS_oldstate` — состояние, из которого производится переход;

`@GA_TRANS_newstate` — состояние, в которое производится переход;

`@GA_TRANS_if` — условие перехода (блок `IF`);

`@GA_TRANS_also` — действия при переходе (блок `ALSO`).

5.5. Синтез правил

Поскольку, как было оговорено ранее, проверка на соответствие модели не производится, синтез правил состоит из двух этапов:

1. Дробление правил.
2. Слияние правил.

С помощью регулярного выражения

```
$also =~ s/(A\[([^\]1]*1[^\]]*)\](\d+)\s*=\s*(\d+|\'\w+\')\s*;)//
```

можно вычленить все изменения соседей (наличие символа «1» в индексе изменяемой клетки гарантирует это) из секции ALSO каждого перехода и по каждому из таких изменений синтезировать правило согласно описанному в разд. 3 алгоритму.

Для каждого оператора изменения соседней ячейки:

- производится анализ изменения;
- вычисляется смещение изменяемой клетки относительно текущей;
- из блока ALSO рассматриваемого перехода удаляется рассматриваемое изменение соседа;
- анализируется блок IF рассматриваемого перехода на предмет условий относительно изменяемого соседа;
- создается копия блока IF и изменяется в соответствии со смещением изменяемого соседа относительно текущей клетки;
- создается новое правило перехода, для которого старое состояние — состояние, полученное в результате анализа блока IF, новое состояние — то, на которое производилось изменение (либо состояние не меняется, в случае если меняется ячейка не управляющего автомата), блок IF —

модифицированный с учетом смещения, блок ALSO — пустой, если изменяется значение управляющей ячейки, либо соответствующий изменению рассматриваемой ячейки, в противном случае;

- в случае, если в блоке IF исходного перехода не было записано условий относительно изменяемой клетки, переход получается безусловным.

Безусловные переходы будем обозначать как переходы «*any* → *P*» (*P* — некое определенное состояние). Безусловные переходы, которые не изменяют состояния управляющего автомата клетки, будем обозначать как переходы «*any* → *same*». Эти обозначения будут расшифрованы на этапе генерации кода.

После того как новые правила перехода созданы, преобразователь переходит ко второму шагу — слиянию.

Для каждой группы правил, исходное и конечное состояние которых совпадает между собой, производятся следующие действия:

- формируется условие общего перехода, являющееся дизъюнкцией всех условий переходов;
- формируется действие при переходе, являющееся последовательностью условных операторов, связанных с каждым правилом, в каждом из которых условие — блок IF соответствующего правила, а действие — его блок ALSO;
- создается правило перехода для рассматриваемых исходного и конечного состояний, с созданными блоками IF и ALSO;
- исходная группа правил удаляется.

При слиянии в блоках ALSO могут образовываться конструкции вида:

```
if (true) {
      …
}
```

(безусловное выполнение) или

```
if (…) {
}
```

(условное выполнение отсутствия действий). Они приводятся к виду обычных операторов, как в первом случае, либо подавляются, как во втором.

Правила, получающиеся в результате синтеза, хранятся в тех же структурах данных, как и правила, извлеченные напрямую из исходного файла.

5.6. Генерация кода

Генерация кода происходит на основе шаблонов. В каталоге «*templates*» содержатся шаблоны файлов, которые будут созданы. Часть текста, которая не изменяется в зависимости от логики игры, уже включена в эти файлы. Зависимая часть (код, который будет сгенерирован исходя из заданного описания) заменена на специальные метки вида `!!T_CA_SWITCH!!`.

Преобразователь подготавливает код, который будет вставлен на место каждой из этих меток, а затем, сканируя файлы шаблонов и находя эти метки, заменяет их на этот код.

Предусмотрены следующие метки:

`!!T_CA_STATES!!` — блок макросов препроцессора *C*, задающих константы, используемые для обозначения значений состояний клеток клеточного автомата.

`!!T_CA_SWITCH!!` — блок *switch*, реализующий функцию переходов клетки автомата.

`!!T_GA_STATES!!` — блок макросов препроцессора *C*, задающих константы, используемые для обозначения значений состояний супервизоров.

`!!T_GA_SWITCH!!` — один или несколько (по количеству управляющих супервизоров) блоков *switch*, реализующих работу автоматов-супервизоров.

`!!T_MACRO!!` — блок макросов препроцессора *C*, описывающий макросы.

Блоки *switch* реализуются однозначным образом по принципу, предложенному в [1], на основе правил перехода, полученных в результате синтеза. Переход из условного состояния «*any*» добавляется в каждый из описываемых блоков *case*, а также в блок *default*.

Блоки макросов составляются на основе декларации перечисляемых состояний в исходном файле.

Полный исходный текст программы и файлы шаблонов приведены в приложении 1.

5.7. Выводы по главе 5

Построено инструментальное средство, которое по описанию игры в формате формализации логики игры строит код на языке *C++*, который реализует эту логику.

6. Пример работы преобразователя и генератора кода. Реализация упрощенного варианта игры *Pac-Man*

Проиллюстрируем работу преобразователя на упрощенной реализации игры *Pac-Man*.

6.1. Правила игры Pac-Man

В игре *Pac-Man* игрок управляет персонажем («колобком»), который должен съедать «горошины», разбросанные по лабиринту, образованному стенами [10]. Персонаж может перемещаться по коридорам лабиринта, но не может проходить сквозь стены. За «съедание» горошины игроку начисляется одно очко.

В лабиринте также находятся враги, которые перемещаются по его коридорам, и с которыми персонажу нельзя сталкиваться. Враги являются сложными объектами с искусственным интеллектом, которыми в рамках рассматриваемой системы должны были бы управлять супервизоры, поэтому они реализованы не будут.

Такой упрощенный вариант игры, в котором присутствует только персонаж, лабиринт и горошины, хорошо укладывается в идеологию реализации игры на базе клеточных автоматов, так как все взаимодействия в нем только локальны. Глобальные взаимодействия отсутствуют.

Для демонстрации работы преобразователя с супервизорами введем уровни игры. Как только игрок набирает 100 очков, он переходит на следующий уровень, а количество очков обнуляется.

6.2. Структура клетки поля

В таком простом варианте игры для клетки достаточно одного состояния — оно же и будет являться управляющим.

A^0 (*A0* в обозначениях языка формализации) — тип клетки. Может принимать значения «Пусто» («Empty»), «Стена» («Wall»), «Горошина» («Pea») или «Пак-ман» («Pacman»).

На языке формализации логики декларация состояний клетки поля будет выглядеть следующим образом:

```
A0 = Type (Empty, Wall, Pea, Pacman)
```

6.3. Структура супервизоров

Супервизоры требуются для управления персонажем с клавиатуры (его состояние устанавливается в зависимости от того, какая клавиша нажата), подсчета очков и учета текущего уровня игры:

A0 (`G0` в обозначениях языка формализации) — управление с клавиатуры. Может принимать значения «нет» («none»), «право» («right»), «верх» («up»), «лево» («left») или «низ» («down»).

A1 (`G1` в обозначениях языка формализации) — подсчет очков. Содержит текущее количество очков.

A2 (`G2` в обозначениях языка формализации) — номер текущего уровня.

На языке формализации логики игры декларация супервизоров будет выглядеть следующим образом:

```
G0 = Keyboard (none, right, up, left, down)
G1 = Score (int)
G2 = Level (int)
```

6.4. Правила переходов клетки поля

В игре будет четыре группы правил, по количеству возможных направлений движения персонажа: вправо, вверх, влево или вниз. Они полностью повторяют друг друга с точностью до условий относительно первого супервизора и ссылок на соседей в одном или другом

направлении. Будем рассматривать только одну группу — связанную с движением вправо. Другие три группы строятся полностью аналогично.

Если нажата клавиша «вправо» (супервизор *A0* имеет состояние «право»), а справа от персонажа находится горошина, то персонаж перемещается в клетку справа, оставляя на своем месте пустоту, а счет увеличивается на единицу. Таким образом, меняется состояние текущей клетки («Пак-ман» → «Пусто»), и меняется состояние соседа справа («Горошина» → «Пак-ман»).

На языке формализации логики игры такой переход записывается в следующем виде:

```
A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'right' && A[i,j+1]0 == 'Pea'
      ALSO
            G1 ++;
            A[i,j+1]0 = 'Pacman';
      END
```

Аналогично, если нажата клавиша «вправо», а справа от персонажа находится пустота, то персонаж просто перемещается в клетку справа. Меняется состояние текущей клетки («Пак-ман» → «Пусто») и правого соседа («Пусто» → «Пак-ман»).

Такой переход будет записан следующим образом:

```
A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'right' && A[i,j+1]0 == 'Empty'
      ALSO
            A[i,j+1]0 = 'Pacman';
      END
```

В приложении 2 приведены все восемь правил перехода клеток поля.

6.5. Правила переходов супервизоров

Управляющим супервизором является только супервизор *A1* — «счет». При достижении состояния (значения) «100», он переходит в

состояние «0», изменяя состояние супервизора *A2* — «уровень». Такой переход будет записан следующим образом:

```
G1: 100 -> 0
      IF
      ALSO
            G2 ++;
      END
```

6.6. Анализ правил перехода и генерация кода

Если посмотреть на правила перехода клетки поля, то несложно заметить, что они не соответствуют ни одной возможной функции переходов клеточного автомата. Действительно, такие правила соответствуют только модели клеточного автомата с распределенной функцией перехода: на каждом шаге вычисляются новые состояния не только для самой клетки, но и для четырех ее соседей.

Преобразователю-генератору предстоит провести дробление и слияние этих правил, для того чтобы привести правила к правилам перехода классического клеточного автомата (в данном случае, когда в каждой клетке находится только один автомат, понятие расширенного клеточного автомата не требуется), а затем записать эти правила в виде кода языка *C++*.

6.7. Анализ результата

Результатом работы преобразователя служит автомат с тремя переходами: «Пак-ман» → «Пусто», «Горошина» → «Пак-ман» и «Пусто» → «Пак-ман». Следует отметить, что на этапе формализации логики такие переходы в явном виде не проектировались: были переходы только одного вида: «Пак-ман» → «Пусто». Переходы между другими состояниями образовались в результате дробления заданных правил.

Хотя итоговый автомат и имеет всего три правила перехода вместо восьми (правда, между одними и теми же состояниями), условия этих переходов очень сложны:

```
if ((super[0] == A_0_right && east[0] == CA_0_Pea ) || (super[0] ==
A_0_right && east[0] == CA_0_Empty ) || (super[0] == A_0_up && north[0] ==
CA_0_Pea ) || (super[0] == A_0_up && north[0] == CA_0_Empty ) || (super[0]
== A_0_left && west[0] == CA_0_Pea ) || (super[0] == A_0_left && west[0]
== CA_0_Empty ) || (super[0] == A_0_down && south[0] == CA_0_Pea ) ||
(super[0] == A_0_down && south[0] == CA_0_Empty ))
```

Это результат слияния правил.

Исходный файл описания логики имеет размер 1193 байта. Общий объем результата, состоящего из шести файлов (трех файлов *.h* и трех файлов *.cpp*) — 4822 байта.

6.8. Выводы по главе 6

Из файла размером около 1Кб, описывающего логику упрощенного варианта игры *Pac-Man*, автоматически сгенерировано около 5Кб кода на *C++*, реализующего эту логику.

Выводы

1. По сравнению с работой [1] упрощена реализация игр с помощью клеточных автоматов.
2. Предложен подход и инструментальное средство для автоматической генерации программного кода исходя из описания поведения игровых объектов.
3. Клеточные автоматы, правила переходов которых изменяют состояния соседних клеток («автоматы с распределенной функцией переходов»), могут быть реализованы через классические клеточные автоматы.
4. Предъявлен алгоритм преобразования таких автоматов в классические.
5. Предложен язык для формализации поведения игровых объектов в терминах локальных взаимодействий, пригодный для автоматической генерации программного кода на языке *C++*, реализующего косвенно описанный им клеточный автомат.
6. На примере реализации упрощенной версии игры *Pac-Man* показано, что в результате автоматической генерации до 75% программного кода может быть построено автоматически.

Источники

1. *Столбов С.А.* Разработка метода реализации игр на тетрагональном поле, в которых преобладают локальные взаимодействия //http://is.ifmo.ru/papers/gamecell/
2. *Тоффоли Т., Марголус Н.* Машины клеточных автоматов. М.: Мир, 1991.
3. *Фон Нейман Дж.* Теория самовоспроизводящихся автоматов. М.: Мир, 1971.
4. *Шалыто А.*, *Туккель Н.* От тьюрингова программирования к автоматному //Мир ПК. 2002, № 2, с. 144—149. http://is.ifmo.ru/works/turing/
5. *Wolfram S.* A New Kind of Science. Wolfram Media, Inc., 2002.
6. *Наумов Л.А.*, *Шалыто А.А.* Клеточные автоматы. Реализация и эксперименты //Мир ПК. 2003. № 8, с. 64—71. http://is.ifmo.ru/works/klet/
7. *Esser J.*, *Schrechenberg M.* Microscopic simulation of urban traffic based on cellular automata //International Journal of Modern Physics, Vol. 8, No. 5 (1997) p. 1025—1036.
8. *Шалыто А.А.* Технология автоматного программирования //Мир ПК 2003 № 10, с. 74—78. http://is.ifmo.ru/works/tech_aut_prog/
9. *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
10. *Wikipedia.* Pac-Man //http://en.wikipedia.org/wiki/Pac-Man

Приложение 1. Исходные тексты преобразователя автоматов и генератора кода

build_ca.pl

```
#!/usr/bin/perl

$debug = 0;

###
# CONSTANTS AND FUNCTIONS
@gen_files = ('Cell.cpp', 'Cell.h', 'Cellaut.cpp', 'Cellaut.h',
'Game.cpp', 'Game.h');

###  offset: A function to adjust cell coordinates
# offset('i-1,j', 1, 0) == 'i,j'
# offset('i-1,j-1', 0, 1) == 'i-1, j'
sub offset {
      my ($ij, $di, $dj) = @_;
      $ij =~ s/[ij]//g;

      my ($i, $j) = split /,/, $ij;

      $i += $di;
      $j += $dj;

      if ($i > 0) {
            $i = '+'.$i;
      } elsif ($i == 0) {
            $i = '';
      }

      if ($j > 0) {
            $j = '+'.$j;
      } elsif ($j == 0) {
            $j = '';
      }

      return "i$i,j$j";
}
```

```
### dx: A function to extract the X cell coordinate
# dx('i-1,j') == -1
sub dx {
      my ($ij) = @_;

      if ($ij =~ /^i([+-\d]+),/)
            { return $1; }
      else
            { return 0; }
}

### dx: A function to extract the Y cell coordinate
# dy('i-1,j-1') == -1
sub dy {
      my ($ij) = @_;

      if ($ij =~ /,j([+-\d]+)/)
            { return $1; }
      else
            { return 0; }
}



###
# READ COMMAND-LINE PARAMETERS
{
$in = "in.txt";
$in = $ARGV[0] if $ARGV[0];
}

###
# READ TEMPLATE FILES
for (my $i = 0; $i < scalar @gen_files; $i++) {
      open F, 'templates/_template_'.$gen_files[$i] or die 'Unable to
locate template file for '.$gen_files[$i].'!';
      $templates{$gen_files[$i]} = join '', <F>;
      close F;
}

###
# READ INPUT FILE
```

```
{
open F, $in;
$in_file = join '', <F>;
close F;

# delete windows line breaks
$in_file =~ s/\r//g;

# delete comments
$in_file =~ s/\#.*?\n//g;



# Parsing the basic file structure
$in_file =~
/%CELLSTATES\s*\n(.*?)%END.*?%GLOBALSTATES\s*\n(.*?)%END.*?%MACROS\s*\n(.*
?)%END.*?%CELLAUT\s*\n(.*?)%END.*?%GLOBALAUT\s*\n(.*?)%END/s or die
"Unable to parse file structure";
# storing the matched parts
$cellstates = $1;
$globalstates = $2;
$macros = $3;
$cellaut = $4;
$globalaut = $5;
}

# Cellstates
{
@cellstates = split /\n/, $cellstates;
$CA_N = scalar @cellstates;
# A0 = Parameter name (A, B)
foreach $cellstate (@cellstates) {
      $cellstate =~ /A(\d)\s*=\s*(\w[\w\s]*?)\((.*?)\)/;
      my $n = $1;
      my $name = $2;
      my $raw_type = $3;
      my $type;

      if ($raw_type =~ /^(int)$/i) {
            $type = $raw_type
      } else {
            # enum type
            my @types = split /,\s?/, $raw_type;
```

```
            $type = 'int';
            $CA_TYPES{"$n|N"} = scalar @types;
            for (my $i = 0; $i< scalar @types; $i++) {
                  $CA_TYPES{"$n|$i"} = $types[$i];
                  $CA_TYPE_VALUES{$types[$i]} = $i;
                  $CA_TYPE_ORIGIN{$types[$i]} = $n;
                  $CA_TYPE_NAMES .= $types[$i]."|";
            }
      } #type

      $CA_NAMES[$n] = $name;
}

}

# Globalstates
{
@globalstates = split /\n/, $globalstates;
$A_N = scalar @globalstates;

# G0 = Parameter name (A, B)
foreach $globalstate (@globalstates) {
      $globalstate =~ /G(\d)\s*=\s*(\w[\w\s]*?)\((.*?)\)/;
      my $n = $1;
      my $name = $2;
      my $raw_type = $3;
      my $type;

      if ($raw_type =~ /^(int)$/i) {
            $type = $raw_type
      } else {
            # enum type
            my @types = split /,\s?/, $raw_type;
            $type = 'int';
            $A_TYPES{"$n|N"} = scalar @types;
            for (my $i = 0; $i< scalar @types; $i++) {
                  $A_TYPES{"$n|$i"} = $types[$i];
                  $A_TYPE_VALUES{$types[$i]} = $i;
                  $A_TYPE_ORIGIN{$types[$i]} = $n;
                  $A_TYPE_NAMES .= $types[$i]."|";
            }
```

```
	} #type

	$A_NAMES[$n] = $name;
}
}

# Macros
{
$macros =~ s/\n/ /gs;
$macros =~ s/\s\s+/ /g;
@macros = split /;/, $macros;


foreach $macro (@macros) {
	if ($macro =~ /macro\s+(\w+)\(i,j\)\s+?=(.*)/) {
		my $name = $1;
		my $macro_text = $2;

		$MACRO{$name} = $macro_text;
		$is_macro .= "$name|"
	}
}
$macro =~ s/\|$//;

}

# Read Cellaut rules
{
$cellaut =~ s/\n/ /gs;
$cellaut =~ s/\s\s+/ /g;

@CA_TRANS = ();

sub addcarule {
	my ($ij, $oldstate, $newstate, $if, $also) = @_;

	$CA_TRANS[scalar @CA_TRANS_ij] = 1;

	$ij =~ s/\s//g;

	push @CA_TRANS_ij, $ij;
```

```
	push @CA_TRANS_oldstate, $oldstate;
	push @CA_TRANS_newstate, $newstate;
	push @CA_TRANS_if, $if;
	push @CA_TRANS_also, $also;
}

while ($cellaut =~ s/A\[([ij,+-1]+)\](\d*):\s*(\d+|\'\w+\')\s*-
>\s*(\d+|\'\w+\')\s+IF\s(.*?)ALSO\s(.*?)END//) {
	my $ij = $1;
	my $n = $2;
	my $oldstate = $3;
	my $newstate = $4;
	my $if = $5;
	my $also = $6;

	addcarule($ij, $oldstate, $newstate, $if, $also);
}

}

# Read Supervisor rules
{
$globalaut =~ s/\n/ /gs;
$globalaut =~ s/\s\s+/ /g;

%GA_TRANS = ();
@GA_TRANS = ();

sub addglobalrule {
	my ($n, $oldstate, $newstate, $if, $also) = @_;

	if ($GA_TRANS{$n} != 1) {
		push @GA_TRANS, $n;
	}

	$GA_TRANS{$n} = 1;



	$n =~ s/\s//g;

	push @GA_TRANS_n, $n;
```

```
      push @GA_TRANS_oldstate, $oldstate;
      push @GA_TRANS_newstate, $newstate;
      push @GA_TRANS_if, $if;
      push @GA_TRANS_also, $also;

      $GA_n{$n} = 1;
}
while ($globalaut =~ s/G(\d+):\s*(\d+|\'\w+\')\s*-
>\s*(\d+|\'\w+\')\s+IF\s(.*?)ALSO\s(.*?)END//) {
      my $n = $1;
      my $oldstate = $2;
      my $newstate = $3;
      my $if = $4;
      my $also = $5;

      if ($newstate =~ /'(.*)'/) {
            $newstate = "A_".$n."_".$1;
      }

      addglobalrule($n, $oldstate, $newstate, $if, $also);
}

}

# Process (centralize) Cellaut rules (2 steps)
{
      # outer fold

# step 1: splitting rules

$CA_TRANS_cnt = scalar @CA_TRANS_ij;
for (my $i = 0; $i < $CA_TRANS_cnt; $i++) {
      my ($ij, $oldstate, $newstate, $if, $also) =
            ($CA_TRANS_ij[$i], $CA_TRANS_oldstate[$i],
$CA_TRANS_newstate[$i], $CA_TRANS_if[$i], $CA_TRANS_also[$i]);

      while ($also =~
s/(A\[([^\]1]*1[^\]]*)\](\d+)\s*=\s*(\d+|\'\w+\')\s*;)//) {
            my $p_act = $1;
            my $p_ij = $2;
            my $p_n = $3;
            my $p_newstate = $4;
```

```
            my $p_dx = dx($p_ij);
            my $p_dy = dy($p_ij);

            print $p_ij." -> ".$p_dx.";".$p_dy."\n\n" if $debug;

            $qm_p_act = quotemeta $p_act;
            $also =~ s/$qm_p_act//;
            $CA_TRANS_also[$i] = $also;

            print "converting: $p_act\n" if $debug;

            my $p_oldstate;
            my $p_ij_qm = quotemeta $p_ij;
            if ($if =~ /A\[$p_ij_qm\]$p_n\s*==\s*(\d+|'\w+')/) {
                  $p_oldstate = $1;
            } else {
                  $p_oldstate = 'any';
            }

            my $p_also;
            if ($p_n == 0) {
                  $p_also = '';
            } else {
                  $p_also = 'A[i,j]'.$p_n.' = '.$p_newstate.';';
                  $p_newstate = 'same';
            }

            my $p_if = $if;
            $p_if =~ s/([\[\(])([ij,+-1]+)([\]\)])/$1.offset($2,-$p_dx,-
$p_dy).$3/ge;
            $p_if = "( $p_if ) && ( A[".offset("i,j",-$p_dx,-$p_dy)."]0 ==
$oldstate )";

            addcarule("i,j", $p_oldstate, $p_newstate, $p_if, $p_also);
            print "i,j *** $p_oldstate *** $p_newstate *** $p_if ***
$p_also\n" if $debug;

      }


}
```

```
# step 2: merging rules

$CA_TRANS_cnt = scalar @CA_TRANS_ij;

my %dup_STATES;
my %dup_STATES_count;

# Collect all keys to count duplicates
for (my $i = 0; $i < $CA_TRANS_cnt; $i++) {
      my ($ij, $oldstate, $newstate, $if, $also) =
            ($CA_TRANS_ij[$i], $CA_TRANS_oldstate[$i],
$CA_TRANS_newstate[$i], $CA_TRANS_if[$i], $CA_TRANS_also[$i]);

      if ($CA_TRANS[$i] ne '') {
            $dup_STATES{$ij.'|'.$oldstate.'|'.$newstate} .= "$i|";
            $dup_STATES_count{$ij.'|'.$oldstate.'|'.$newstate} ++;
      }
}

# Now check all collected keys for duplicates
foreach my $key (keys %dup_STATES_count) {
      if ($dup_STATES_count{$key} > 1) {
            # duplicate found
            my @d_ids = split /\|/, $dup_STATES{$key};
            my ($d_ij, $d_oldstate, $d_newstate) = split /\|/, $key;


            # compose new IF block
            my @d_IFs;
            foreach my $d_id (@d_ids) {
                  push @d_IFs, $CA_TRANS_if[$d_id];
            }
            my $new_if = '(' . (join ') || (', @d_IFs) . ')';

            # compose new ALSO block
            my $new_also = '';
            foreach my $d_id (@d_ids) {
                  my $d_if = $CA_TRANS_if[$d_id];
                  my $d_also = $CA_TRANS_also[$d_id];
```

```
                  if ($d_also !~ /^\s*$/) {
                        if ($d_if =~ /^\s*$/) {
                              $new_also .= "
                              $d_also";
                        } else {
                              $new_also .= "
                              if ($d_if) {
                                    $d_also
                              }";
                        }
                  }

            }

            # push the new rule
            addcarule($d_ij, $d_oldstate, $d_newstate, $new_if,
$new_also);
            print "i,j @@@ $d_oldstate @@@ $d_newstate @@@ $new_if @@@
$new_also\n" if $debug;

            # disable the old merged rules
            foreach my $d_id (@d_ids) {
                  $CA_TRANS[$d_id] = '';
            }

      }
}

} # outer fold

###
# GENERATING CODE
{

sub coderep_CA {
      my $source = shift;

      $source =~ s/A\[i,j\](\d+)/state[$1]/gs;
      $source =~ s/A\[i,j\]/state/gs;

      $source =~ s/A\[i,j\+1\](\d+)/east[$1]/gs;
      $source =~ s/A\[i,j\+1\]/east/gs;
```

```
	$source =~ s/A\[i-1,j\+1\](\d+)/se[$1]/gs;
	$source =~ s/A\[i-1,j\+1\]/se/gs;

	$source =~ s/A\[i-1,j\](\d+)/north[$1]/gs;
	$source =~ s/A\[i-1,j\]/north/gs;

	$source =~ s/A\[i-1,j-1\](\d+)/nw[$1]/gs;
	$source =~ s/A\[i-1,j-1\]/nw/gs;

	$source =~ s/A\[i,j-1\](\d+)/west[$1]/gs;
	$source =~ s/A\[i,j-1\]/west/gs;

	$source =~ s/A\[i\+1,j-1\](\d+)/sw[$1]/gs;
	$source =~ s/A\[i\+1,j-1\]/sw/gs;

	$source =~ s/A\[i\+1,j\](\d+)/south[$1]/gs;
	$source =~ s/A\[i\+1,j\]/south/gs;

	$source =~ s/A\[i\+1,j\+1\](\d+)/se[$1]/gs;
	$source =~ s/A\[i\+1,j\+1\]/se/gs;

	$source =~ s/G(\d+)/super[$1]/gs;

	$source =~
s/'($CA_TYPE_NAMES)'/'CA_'.$CA_TYPE_ORIGIN{$1}.'_'.$1/gse;
	$source =~ s/'($A_TYPE_NAMES)'/'A_'.$A_TYPE_ORIGIN{$1}.'_'.$1/gse;

	$source =~ s/($is_macro)\([ij,+-1]+\)(\d+)/$1\[$2\]/gs;
	$source =~ s/($is_macro)\([ij,+-1]+\)/$1/gs;

	return $source;
}

sub coderep_GA {
	my $source = shift;

	$source =~ s/G(\d+)/state[$1]/gs;

	$source =~
s/'($CA_TYPE_NAMES)'/'CA_'.$CA_TYPE_ORIGIN{$1}.'_'.$1/gse;
	$source =~ s/'($A_TYPE_NAMES)'/'A_'.$A_TYPE_ORIGIN{$1}.'_'.$1/gse;
```

```
	return $source;
}

print "\n\n\n" if $debug;

# Generating macros
foreach my $macro_name (keys %MACRO) {
	$T_MACRO .= "#define $macro_name(i,j) ($MACRO{$macro_name})\n";
}
$T_MACRO = coderep_CA($T_MACRO);
print "!!MACRO!! $T_MACRO\n\n" if $debug;

# Generating CA state list
foreach my $catype (keys %CA_TYPES) {
	my ($n, $i) = split /\|/, $catype;
	if ($i ne "N") {
		$T_CA_STATES .= "#define CA_${n}_$CA_TYPES{$catype}
".$CA_TYPE_VALUES{$CA_TYPES{$catype}}."\n";
	}
}
$T_CA_STATES .= "\n#define CA_N $CA_N\n";
print "!!CA_STATES!! $T_CA_STATES\n\n" if $debug;

# Generating GA state list
foreach my $gatype (keys %A_TYPES) {
	my ($n, $i) = split /\|/, $gatype;
	if ($i ne "N") {
		$T_GA_STATES .= "#define A_${n}_$A_TYPES{$gatype}
".$A_TYPE_VALUES{$A_TYPES{$gatype}}."\n";
	}
}
$T_GA_STATES .= "\n#define A_N $A_N\n";
print "!!GA_STATES!! $T_GA_STATES\n\n" if $debug;

# Generating CA SWITCH block
$CA_TRANS_cnt = scalar @CA_TRANS_ij;
my %VARIANTS = ();
for (my $i = 0; $i < $CA_TRANS_cnt; $i++) {
	if ($CA_TRANS[$i] == 1) {
		my ($ij, $oldstate, $newstate, $if, $also) =
```

```
            ($CA_TRANS_ij[$i], $CA_TRANS_oldstate[$i],
$CA_TRANS_newstate[$i], $CA_TRANS_if[$i], $CA_TRANS_also[$i]);

            my $var;

            if ($newstate ne "same") {
                  $var = "
                        if ($if) {
                              $also
                              state[0] = $newstate;
                        }
                  ";
            } else {
                  if ($also !~ /^\s*$/) {
                  $var = "
                        if ($if) {
                              $also
                        }
                  ";
                  }
            }

            $VARIANTS{coderep_CA($oldstate)} .= coderep_CA($var);
      }
}

foreach my $var (keys %VARIANTS) {
      if ($var ne "any") {
            $T_CA_SWITCH .= "
                  case $var:
                        $VARIANTS{$var}
                        $VARIANTS{any}
                        break;";
      }
}

if ($VARIANTS{"any"} ne "") {
      $T_CA_SWITCH .= "
                  default:
                        $VARIANTS{any}
                        break;";
```

```
}

$T_CA_SWITCH = "switch (state) {".$T_CA_SWITCH."\n\t}";
print "!!CA_SWITCH!! $T_CA_SWITCH\n\n" if $debug;

# Generating GA switch block
for (my $ga_i = 0; $ga_i < scalar @GA_TRANS; $ga_i++) {
      my $ga = $GA_TRANS[$ga_i];

      my $T_GA_SWITCH_curr = "";
      print "$ga\n\n";

      my $GA_TRANS_cnt = scalar @GA_TRANS_n;
      my %VARIANTS = ();
      for (my $i = 0; $i < $GA_TRANS_cnt; $i++) {
            if ($GA_TRANS_n[$i] == $ga) {
                  my ($ij, $oldstate, $newstate, $if, $also) =
                  ($GA_TRANS_ij[$i], $GA_TRANS_oldstate[$i],
$GA_TRANS_newstate[$i], $GA_TRANS_if[$i], $GA_TRANS_also[$i]);

                  my $var;

                  if ($if ne "") {
                        $var = "
                              if ($if) {
                                    $also
                                    state[0] = $newstate;
                              }";
                  } else {
                        $var = "
                              $also
                              state[0] = $newstate;";
                  }

                  $VARIANTS{coderep_GA($oldstate)} .= coderep_GA($var);
            }
      }

      foreach my $var (keys %VARIANTS) {
                  $T_GA_SWITCH_curr .= "
                        case $var:
```

```
                              $VARIANTS{$var}
                              break;";
      }

      if ($T_GA_SWITCH_curr ne "") {
            $T_GA_SWITCH .= "switch (state[$ga])
{".$T_GA_SWITCH_curr."\n\t}\n\n";
      }
}

print "!!GA_SWITCH!! $T_GA_SWITCH\n\n" if $debug;

}



###
# OUTPUT SOURCE FILES
{

for (my $i = 0; $i < scalar @gen_files; $i++) {
      $templates{$gen_files[$i]} =~ s/!!(\w+)!!/$$1/ge;
      open F, ">output/$gen_files[$i]";
      print F $templates{$gen_files[$i]};
      close F;
}
}
```

templates/_template_Cell.h

```
!!T_CA_STATES!!


class GameCell {

public:

	int state[CA_N];



	static int border_state[CA_N];

	static int default_state[CA_N];



	GameCell();

	void Step(const int* east, const int* se, const int* north, const
int* nw, const int* west, const int* sw, const int* south, const int* se,
const int* super);

};
```

templates/_template_Cell.cpp

```
#include "Prefix.h"


GameCell::GameCell()
{
	// TODO: Insert CA initialization code here
}


!!T_MACRO!!


void GameCell::Step(const int* east, const int* se, const int* north,
const int* nw, const int* west, const int* sw, const int* south, const
int* se, const int* super)
{
	!!T_CA_SWITCH!!
}
```

templates/_template_Cellaut.h

```
#include "Cell.h"

!!T_GA_STATES!!

// TODO: Change field size to fit your needs
#define FIELD_W   200
#define FIELD_H 200

typedef GameCell GameField[FIELD_H+2][FIELD_H+2];

class GameCellAut {
private:
      // TODO: Insert private methods if needed
public:
      GameField field;
      GameField snapshot;

      int state[A_N];
      static int default_state[A_N];

      GameCellAut();
};
```

templates/_template_Cellaut.cpp

```
#include "Prefix.h"

GameCellAut::GameCellAut()
{
      int i, j;

      // Initializing the CA
      for (i = 0; i <= FIELD_H; i++)
            for (j = 1; j <= FIELD_W; j++)
            {
                  for (int k = 0; k < CA_N; k++)
                        field[i][j].state[k] = 0;
            }

      // Initializing supervisors
      for (i = 0; i < A_N; i++)
```

```
            state[i] = 0;

      // TODO: Set the initial field layout and/or supervisor states
}

// System step
void GameCellAut::Step()
{
      int i, j;
      // Creating a snapshot
      memcpy(&snapshot, &field, sizeof(field));

      // Every cell makes a step
      for (i = 1; i <= FIELD_H; i++)
            for (j = 1; j <= FIELD_W; j++)
                  field[i][j].Step(
                                          snapshot[i][j+1].state,
                                          snapshot[i-1][j+1].state,
                                          snapshot[i-1][j].state,
                                          snapshot[i-1][j-1].state,
                                          snapshot[i][j-1].state,
                                          snapshot[i+1][j-1].state,
                                          snapshot[i+1][j].state,
                                          snapshot[i+1][j+1].state,
                                          state);

      // Suprevisor steps

      !!T_GA_SWITCH!!
}
```

templates/_template_Game.h

```
#include "Prefix.h"

// TODO: Adjust step length as needed
#define STEP_LENGTH     0.3

class GameGame {
private:

      double t_since_last_step;
```

```
	GameCellAut logic;

	void LoadResources();

public:
	int quitting;

	GameGame(HGE* n_hge);
	~GameGame();

	void Step(double dt);
	void Draw();
};
```

templates/_template_Game.cpp

```
#include "Prefix.h"

void GameGame::LoadResources()
{
	// TODO: Insert resource initialization code
}

GameGame::GameGame(HGE *n_hge)
{
	// TODO: Insert game initialization code

	LoadResources();
	quitting = 0;

	t_since_last_step = 0;
}

GameGame::~GameGame()
{
	// TODO: Insert game termination code
}

void GameGame::Draw()
{
	// TODO: Insert game drawing code
```

```
}


void GameGame::Step(double dt)
{
	t_since_last_step += dt;

	if (t_since_last_step > STEP_LENGTH)
	{
		t_since_last_step -= STEP_LENGTH;

		logic.Step();
	}

	// TODO: Insert keyboard handling code if needed
}
```

Приложение 2. Исходный текст описания игры *Pac-Man*

pacman.txt

```
%CELLSTATES
# A0 = Parameter name (A, B)
# A1 = Parameter name (int)
A0 = Type (Empty, Wall, Pea, Pacman)
%END

%GLOBALSTATES
G0 = Keyboard (none, right, up, left, down)
G1 = Score (int)
G2 = Level (int)
%END

%MACROS

%END

%CELLAUT
A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'right' && A[i,j+1]0 == 'Pea'
      ALSO
            G1 ++;
            A[i,j+1]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'right' && A[i,j+1]0 == 'Empty'
      ALSO
            A[i,j+1]0 = 'Pacman';
      END



A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'up' && A[i-1,j]0 == 'Pea'
```

```
      ALSO
            G1 ++;
            A[i-1,j]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'up' && A[i-1,j]0 == 'Empty'
      ALSO
            A[i-1,j]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'left' && A[i,j-1]0 == 'Pea'
      ALSO
            G1 ++;
            A[i,j-1]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'left' && A[i,j-1]0 == 'Empty'
      ALSO
            A[i,j-1]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'down' && A[i+1,j]0 == 'Pea'
      ALSO
            G1 ++;
            A[i+1,j]0 = 'Pacman';
      END

A[i,j]0: 'Pacman' -> 'Empty'
      IF
            G0 == 'down' && A[i+1,j]0 == 'Empty'
      ALSO
            A[i+1,j]0 = 'Pacman';
      END
```

```
%END

%GLOBALAUT


G1: 100 -> 0
      IF
      ALSO
            G2 ++;
      END

%END
```

Приложение 3. Сгенерированные исходные тексты для игры *Pac-Man*

Cell.h

```
#define CA_0_Pea 2
#define CA_0_Pacman 3
#define CA_0_Wall 1
#define CA_0_Empty 0

#define CA_N 1



class GameCell {
public:
	//
	int state[CA_N];

	static int border_state[CA_N];
	static int default_state[CA_N];

	GameCell();
	void Step(const int* east, const int* se, const int* north, const
int* nw, const int* west, const int* sw, const int* south, const int* se,
const int* super);
};
```

Cell.cpp

```
#include "Prefix.h"


GameCell::GameCell()
{
	// TODO: Insert CA initialization code here
}


void GameCell::Step(const int* east, const int* se, const int* north,
const int* nw, const int* west, const int* sw, const int* south, const
int* se, const int* super)
{
	switch (state) {
	case CA_0_Pacman:
```

```
            if ((super[0] == A_0_right && east[0] == CA_0_Pea ) ||
(super[0] == A_0_right && east[0] == CA_0_Empty ) || (super[0] == A_0_up
&& north[0] == CA_0_Pea ) || (super[0] == A_0_up && north[0] == CA_0_Empty
) || (super[0] == A_0_left && west[0] == CA_0_Pea ) || (super[0] ==
A_0_left && west[0] == CA_0_Empty ) || (super[0] == A_0_down && south[0]
== CA_0_Pea ) || (super[0] == A_0_down && south[0] == CA_0_Empty )) {


                  if (super[0] == A_0_right && east[0] == CA_0_Pea ) {
                        super[1] ++;
                  }
                  if (super[0] == A_0_up && north[0] == CA_0_Pea ) {
                        super[1] ++;
                  }
                  if (super[0] == A_0_left && west[0] == CA_0_Pea ) {
                        super[1] ++;
                  }
                  if (super[0] == A_0_down && south[0] == CA_0_Pea ) {
                        super[1] ++;
                  }
                  state[0] = CA_0_Empty;
            }



            break;
      case CA_0_Pea:


            if ((( super[0] == A_0_right && state[0] == CA_0_Pea  ) && (
west[0] == CA_0_Pacman )) || (( super[0] == A_0_up && state[0] == CA_0_Pea
) && ( south[0] == CA_0_Pacman )) || (( super[0] == A_0_left && state[0]
== CA_0_Pea  ) && ( east[0] == CA_0_Pacman )) || (( super[0] == A_0_down
&& state[0] == CA_0_Pea  ) && ( north[0] == CA_0_Pacman ))) {


                  state[0] = CA_0_Pacman;
            }



            break;
      case CA_0_Empty:


            if ((( super[0] == A_0_right && state[0] == CA_0_Empty  ) && (
west[0] == CA_0_Pacman )) || (( super[0] == A_0_up && state[0] ==
CA_0_Empty  ) && ( south[0] == CA_0_Pacman )) || (( super[0] == A_0_left
&& state[0] == CA_0_Empty  ) && ( east[0] == CA_0_Pacman )) || (( super[0]
== A_0_down && state[0] == CA_0_Empty  ) && ( north[0] == CA_0_Pacman )))
{
```

```
                  state[0] = CA_0_Pacman;
            }


            break;
      }
}
```

Cellaut.h

```
#include "Cell.h"

#define A_0_up 2
#define A_0_left 3
#define A_0_right 1
#define A_0_none 0
#define A_0_down 4

#define A_N 3



// TODO: Change field size to fit your needs
#define FIELD_W   200
#define FIELD_H 200

typedef GameCell GameField[FIELD_H+2][FIELD_H+2];

class GameCellAut {
private:
      // TODO: Insert private methods if needed
public:
      GameField field;
      GameField snapshot;

      int state[A_N];
      static int default_state[A_N];

      GameCellAut();
};
```

Cellaut.cpp

```
#include "Prefix.h"
```

```
GameCellAut::GameCellAut()
{
	int i, j;

	// Initializing the CA
	for (i = 0; i <= FIELD_H; i++)
		for (j = 1; j <= FIELD_W; j++)
		{
			for (int k = 0; k < CA_N; k++)
				field[i][j].state[k] = 0;
		}

	// Initializing supervisors
	for (i = 0; i < A_N; i++)
		state[i] = 0;

	// TODO: Set the initial field layout and/or supervisor states
}

// System step
void GameCellAut::Step()
{
	int i, j;
	// Creating a snapshot
	memcpy(&snapshot, &field, sizeof(field));

	// Every cell makes a step
	for (i = 1; i <= FIELD_H; i++)
		for (j = 1; j <= FIELD_W; j++)
			field[i][j].Step(
					snapshot[i][j+1].state,
					snapshot[i-1][j+1].state,
					snapshot[i-1][j].state,
					snapshot[i-1][j-1].state,
					snapshot[i][j-1].state,
					snapshot[i+1][j-1].state,
					snapshot[i+1][j].state,
					snapshot[i+1][j+1].state,
					state);
```

```
	// Suprevisor steps

	switch (state[1]) {
			case 100:

				state[2] ++;
				state[0] = 0;
				break;
	}

}
```

Game.h

```
#include "Prefix.h"

// TODO: Adjust step length as needed
#define STEP_LENGTH	0.3

class GameGame {
private:

	double t_since_last_step;

	GameCellAut logic;

	void LoadResources();

public:
	int quitting;

	GameGame(HGE* n_hge);
	~GameGame();

	void Step(double dt);
	void Draw();
};
```

Game.cpp

```
#include "Prefix.h"

void GameGame::LoadResources()
```

```
{
	// TODO: Insert resource initialization code
}

GameGame::GameGame(HGE *n_hge)
{
	// TODO: Insert game initialization code

	LoadResources();
	quitting = 0;

	t_since_last_step = 0;
}

GameGame::~GameGame()
{
	// TODO: Insert game termination code
}

void GameGame::Draw()
{
	// TODO: Insert game drawing code
}


void GameGame::Step(double dt)
{
	t_since_last_step += dt;

	if (t_since_last_step > STEP_LENGTH)
	{
		t_since_last_step -= STEP_LENGTH;

		logic.Step();
	}

	// TODO: Insert keyboard handling code if needed
}
```