Реализация клеточного автомата *WireWorld* с помощью инструментального средства *CAME&L* и его зональная оптимизация

Дмитрий Трофимов[1], Лев Наумов[2]

1 Санкт-Петербургский государственный университет информационных технологий, механики и оптики. Кафедра компьютерных технологий. trofimov@rain.ifmo.ru;
2 University of Amsterdam. Section Computational Sciences. levnaumov@mail.ru.

В настоящей статье приводится пример использования инструментального средства *CAME&L* для моделирования клеточных автоматов. Приводится описание известного клеточного автомата *WireWorld* и его реализация на языке *C++* с использованием библиотеки *CADLib*. Реализация включает в себя разработку компонента хранилища данных и компонента правил. Рассматривается способ ускорения вычислений с применением зональной оптимизации. Приводится пример реализации зональной оптимизации с использованием возможностей *CAME&L*.

Введение

Клеточные автоматы являются дискретными динамическими системами, поведение которых может быть полностью описано в терминах локальных зависимостей [1]. Спектр их применения чрезвычайно широк [2–4]. Отдельной отраслью их применения является проведение вычислительных экспериментов на их основе. Клеточные автоматы чрезвычайно удобны для этой цели, так как обладают естественным параллелизмом, который может быть выгодно использован на практике. Одним из наиболее развитых и современных средств для организации вычислительных экспериментов на основе клеточных автоматов является разработанный Л. Наумовым пакет *CAME&L* [5–8].

Упомянутое инструментальное средство обладает универсальностью, позволяя моделировать системы, принадлежащие даже более широкому классу, чем класс "клеточные автоматы". Моделируемая система составляется из так называемых компонентов [9], а именно – из решетки, хранилища данных, метрики и правил автомата. Разработка решения задачи в среде *CAME&L* представляет собой выбор или создание компонентов, необходимых для проведения того или иного эксперимента. С точки зрения разработчика компонент представляет собой класс, описанный на языке *C++* и помещенный в подключаемую к среде *DLL*-библиотеку. Для создания таких классов служит библиотека разработчика клеточных автоматов *CADLib*. Упомянутая библиотека является одной из трех основных частей пакета *CAME&L*, в который также входят графическая среда для проведения экспериментов и набор стандартных компонентов. Более подробно типы компонентов рассматриваются во втором разделе настоящей статьи.

В настоящей статье приводится пример реализации одного из известных клеточных автоматов, *WireWorld* [13] в среде *CAME&L*. Правила *WireWorld* были предложены Брайаном Сильверманом (Brian Silverman) в 1987 году. С помощью четырех состояний и несложной функции переходов можно моделировать работу широкого класса логических схем, соединенных с помощью виртуальных проводов (отсюда и название W*ire World*, в переводе с английского – *Мир проводов*).

Для автомата *WireWorld* существует значительное количество примеров схем разной сложности, вплоть до реализации компьютера, запрограммированного на последовательное перечисление простых чисел [10]. Теоретически, *WireWorld* обладает свойством вычислительной универсальности, позволяя производить любые вычисления (так, на сайте [14] описывается реализация машины Тьюринга для *WireWorld*), хотя такие построения с практической точки зрения крайне неэффективны.

Статья состоит из двух разделов. В первом разделе описываются правила автомата *WireWorld*, перечислены некоторые базовые конфигурации клеток ("функциональные элементы"), из которых строятся более сложные системы. Второй раздел посвящен реализации автомата с использованием $CAME_{\&}L$: в разд. 2.1 описаны типы выделяемых в $CAME_{\&}L$ компонентов, в разд. 2.2–2.4 приводится реализация необходимых для моделирования *WireWorld* компонентов с исходными текстами на языке *C++*. Также в разд. 2.4 рассмотрен вопрос "зональной" оптимизации вычислений.

1. Описание клеточного автомата *WireWorld*

Клеточный автомат *WireWorld* представляет собой синхронный автомат с двумерной решеткой из квадратов, каждая клетка которой может находиться в одном из четырех состояний. Названия состояний, их значения и цвета, используемые для их визуализации во многих источниках (в настоящей статье используются те же обозначения), представлены в таблице.

Таблица. Состояния автомата *WireWorld*

Название состояния	**Цвет**	**Значение**
Пустая клетка	Черный	Клетка, не занятая проводником
Проводник	Желтый	По проводнику могут распространяться электроны
Голова электрона	Красный	Проводник, занятый электроном: вместе с "хвостом электрона" задает направление движения электрона
Хвост электрона	Синий	Проводник, занятый электроном: вместе с "головой электрона" задает направление движения электрона

Так, на рис. 1 изображен проводник, по которому перемещается электрон (пара клеток "голова электрона"–"хвост электрона"). Хвост электрона, очевидно, следует за головой и, тем самым, определяет направление движения электрона (на рис.1 — направо).

Рис. 1. Электрон

На каждом шаге автомата ко всем клеткам применяются следующие правила.

1. Пустая клетка остается пустой.

2. Клетка, находящаяся в состоянии "голова электрона" переходит в состояние "хвост электрона".
3. Клетка, находящаяся в состоянии "хвост электрона" переходит в состояние "проводник".
4. Клетка, находящаяся в состоянии "проводник" переходит в состояние "голова электрона", в том случае, если среди соседних клеток ровно одна или две находятся в состоянии "голова электрона". Во всех остальных случаях "проводник" остается "проводником".

При применении данных правил используется окрестность Мура – считается, что с данной клеткой соседствуют все восемь ее непосредственных соседей.

Как правило, большинство клеток автомата *WireWorld* пусты, однако кое-где на решетке расположены функциональные элементы (сочетания непустых клеток), соединенные между собой проводами (прямыми линиями клеток, находящихся в состоянии "проводник", соединенными под прямым углом), по которым с определенной скоростью распространяются электроны (пары клеток "голова электрона" – "хвост электрона").

Приведем несколько общих закономерностей. Электрон передвигается со скоростью одна клетка за шаг. Если по проводу навстречу идут два электрона, при столкновении они исчезают. При достижении электроном разветвления проводов по каждому из направлений, кроме исходного, уходит по электрону. Если к разветвлению одновременно подходит 2 и более электронов, все они исчезают. При толщине провода в 2 клетки поведение электронов аналогично обычному, при большей толщине поведение становится хаотичным.

Приведем примеры базовых функциональных элементов.

На рис. 2. изображен "тактовый генератор". Он представляет собой "петлю" из клеток проводника, к которой подсоединен провод – выход генератора, и изначально содержит один электрон. С периодом, равным длине петли (у генератора на рис.2 период равен 6), этот электрон достигает точки соединения петли с выходом, и дальше разветвляется на два электрона, один из которых идет по выходу, второй – дальше по петле. Таким образом, этот элемент можно использовать для получения в проводе бесконечного количества электронов, следующих один за другим на расстоянии, регулируемом длиной петли.

Рис. 2. Тактовый генератор

На рис. 3 изображен "диод". Этот функциональный элемент имеет две точки подсоединения к проводам – вход и выход, и его действие состоит в том, что электроны, пришедшие на вход, передаются на выход, а электроны, пришедшие на выход – исчезают. Таким образом, электроны могут перемещаться по проводу, в который включен диод, лишь в одном направлении.

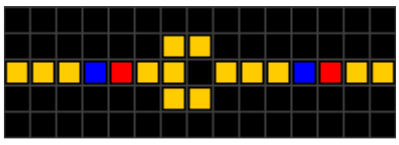

Рис. 3. Диод

На рис. 4 – 6 изображены логические элементы OR, XOR и NAND соответственно. Каждый из них имеет по 2 входа (на рис. – провода слева), и выход (на рис. – провод справа). Наличие электрона на входе соответствует логическому значению "единица", отсутствие – логическому значению "ноль". Электрон на выходе появляется согласно таблице истинности соответствующей логической операции. Так, для элемента OR электрон на любом из входов, или электроны на обоих входах одновременно дают электрон на выходе. Для элемента XOR электрон на любом из входов дает электрон на выходе, но при одновременной подаче электронов на оба входа они исчезают, и электрон на выходе не создается. Элемент NAND работает как тактовый генератор, и посылает электроны на выход во всех случаях, за исключением случая, когда на оба входа одновременно подаются электроны.

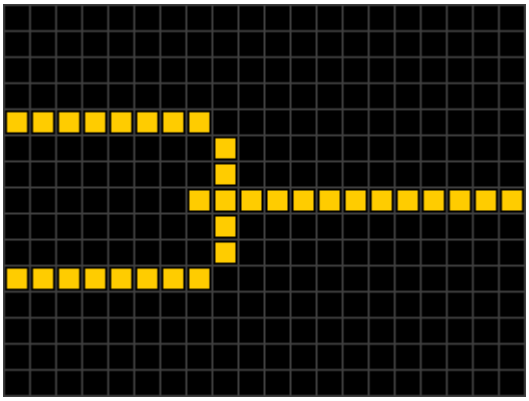

Рис. 4. Элемент OR

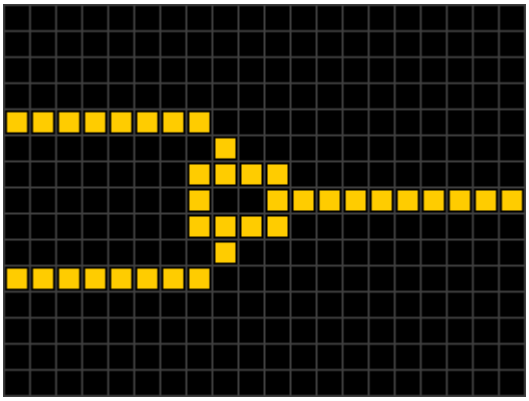

Рис. 5. Элемент XOR

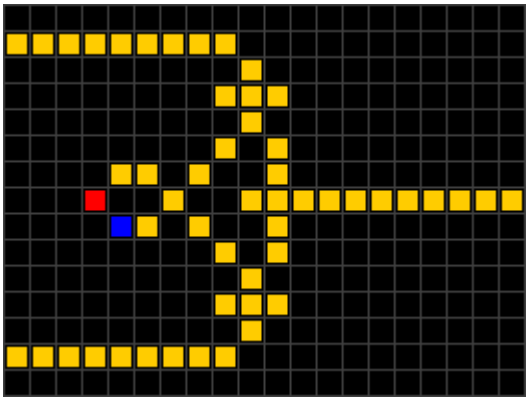

Рис. 6. Элемент NAND

На рис. 7 изображен пример более сложной структуры, состоящей из множества простых элементов – двоичный сумматор. Его функция заключается в том, что при подаче на два входа закодированных особым образом чисел, через фиксированное количество шагов (в изображенном примере – 48) на выходе появится закодированное таким же образом число – сумма чисел на входах. Числа кодируются в двоичном виде, от младших битов к старшим, каждый бит кодируется наличием или отсутствием электрона на определенной позиции. На рис.7 эти позиции отмечены точками и изображениями чисел (значений каждого бита) из клеток в состоянии "проводник" по краям входов и выхода. Сами по себе эти отметки не несут никакой функциональной нагрузки, а служат лишь в пояснительных целях.

Изображенный на рис. 7 сумматор имеет разрядность входов три бита, но можно получить сумматор с любой разрядностью, удлинив или укоротив провода на входах и выходе.

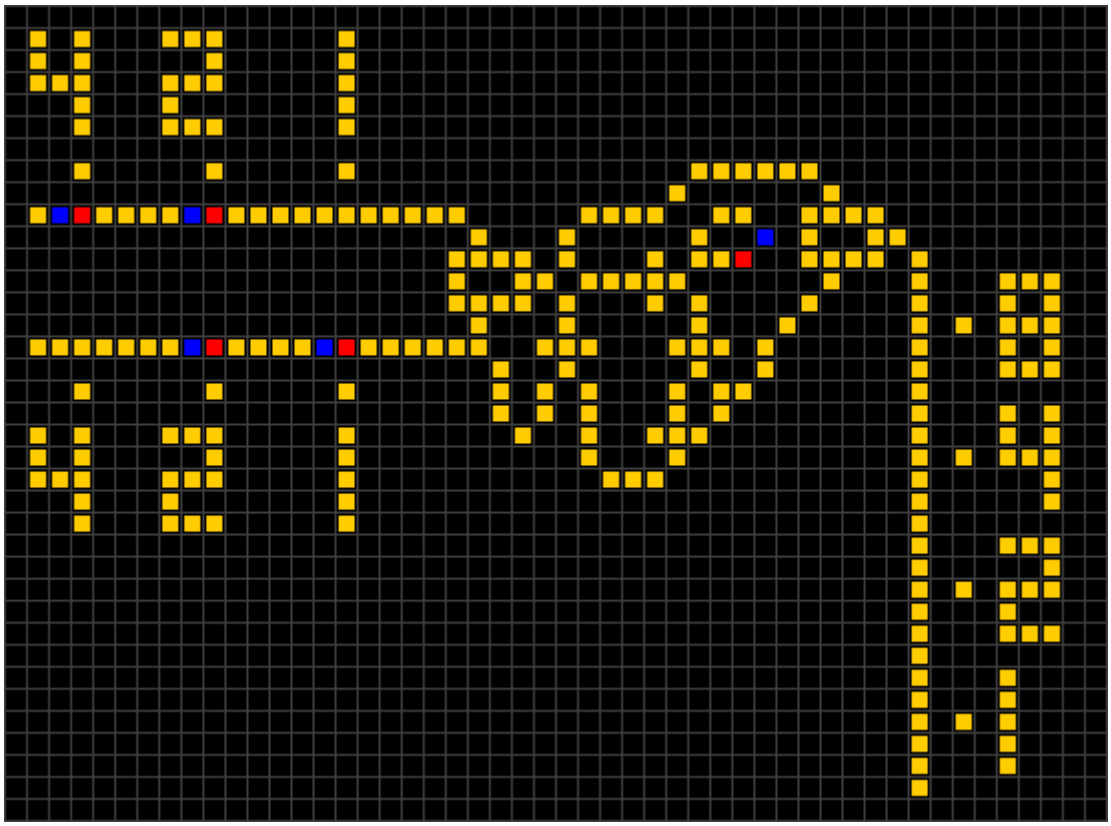

Рис. 7. Двоичный сумматор

Большие коллекции функциональных элементов имеются в пакетах *Mirek's Cellebration* [11] и *Zillions of Games* [12], а также на сайтах [10, 13]. Кроме того, на сайте [10] приводится пример построения в *WireWorld* компьютера с определенным набором инструкций и регистров, и реализация алгоритма перечисления простых чисел для этого компьютера.

2. Реализация клеточных автоматов в рассматриваемом инструментальном средстве

2.1. Компоненты

Реализация автомата с помощью средства *CAME&L* [9] заключается в разработке на языке *C++* так называемых "компонентов", библиотек, подключаемых к программе и реализующих ту или иную функциональность, необходимую для проведения эксперимента. Для создания компонентов в пакет входит библиотека *CADLib*, набор базовых классов и макроопределений, упрощающих разработку, насколько это возможно. Выделяют пять типов компонентов.

1. Решетка – отвечает за визуализацию и навигацию по решетке автомата. Базовый класс для создания компонентов этого типа называется `CAGrid`. В набор стандартных компонентов включены, например, двумерная решетка из квадратов (класс `CASquBase2DGrid`), двумерная решетка из треугольников (класс `CATriBase2DGrid`), двумерная решетка из шестиугольников (класс `CAHexBase2DGrid`), трехмерная решетка из кубов (класс `CACube3DGrid`).
2. Метрика – разрешает отношение соседства и реализует функции вычисления расстояния между клетками. Базовый класс – `CAMetrics`. В числе стандартных компонентов имеются декартова метрика для двумерных (класс `CACrts2DMetrics`) и трехмерных (класс `CACrts3DMetrics`) решеток, метрика для обобщенных координат [15, 16] для двумерных решеток из треугольников (класс `CATriGen2DMetrics`), квадратов (класс `CASquGen2DMetrics`) и шестиугольников (`CAHexGen2DMetrics`).
3. Хранилище данных – обеспечивает хранение состояний клеток, определяет множества возможных состояний и отвечает за некоторые аспекты визуализации (соответствие состояний цветам, используемым при отображении). Базовый класс – `CADatum`. Примеры: хранилище булевых (`CACrtsBool2DDatum`), целочисленных (`CACrtsInt2DDatum`) и вещественных (`CACrtsDouble2DDatum`) значений для декартовых двумерных и для обобщенных (`CAGenBoolDatum`, `CAGenIntDatum`, `CAGenDoubleDatum`) метрик, соответственно. Хранилище описаний биологических клеток для декартовых трехмерных координат (`CACrtsCell3DDatum`).
4. Правила – определяет функцию переходов, а также инициализации и финализации вычислений. Позволяет описать разнообразные способы оптимизации и разделения задачи на подзадачи для случая распределенных вычислений. Кроме того, компонент этого типа перечисляет анализируемые параметры, динамику изменения которых в процессе вычислений можно наблюдать с помощью анализаторов. Базовый класс – `CARules`. Примеры: реализация игры *Жизнь* (`CALife2DRules`) для двумерных и обобщенных координатах, в вариантах с использованием "зональной" оптимизации и распределенных вычислений, а также решение двумерного уравнения теплопроводности (`CATCE2DRules`).
5. Анализатор – служит для изучения динамики изменения тех или иных параметров в течение вычислительного эксперимента. Базовый класс – `CAAnalyzer`. Примеры: анализаторы для построения графиков изменения целочисленных (`CAGraphIntAnalyzer`) и вещественных (`CAGraphDoubleAnalyzer`) параметров, для сохранения в файл отчета значений булевых (`CAFileBoolAnalyzer`), целочисленных (`CAFileIntAnalyzer`) и вещественных (`CAFileDoubleAnalyzer`) параметров. Особняком стоит анализатор производительности системы, позволяющий оптимизировать реализацию компонента правил и настроить вычислительную установку для достижения максимальной производительности.

Для выполнения автомата необходимо задать комбинацию из четырех компонентов: решетки, метрики, хранилища данных и правил. Очевидно, что не всякие компоненты совместимы между собой. Условия совместимости определяются с помощью специальных макроопределений `COMPONENT_REALIZES` и `COMPONENT_REQUIRES`, предоставляемых библиотекой *CADLib*.

Каждый компонент обладает рядом параметров, позволяющих пользователю настраивать его для решения той или иной задачи. Например, для решеток можно изменять размеры ребра клетки, а для хранилищ данных – цветовые схемы представления состояний.

Для реализации автомата *WireWorld* в инструментальном средстве *CAME&L* было разработано три компонента:

1. Специализированное хранилище данных – класс *Integers for Cartesians 2D (WireWorld)*.
2. Простейшая реализация правил автомата – *WireWorld (plain)*.
3. Реализация правил автомата с использованием "зональной" оптимизации – *WireWorld (optimized)*.

В качестве решетки можно использовать стандартный компонент `Square Basic 2D Grid`, а в качестве метрики – стандартный компонент `Cartesians 2D`, поэтому разработка специализированных компонентов этих типов не потребовалась.

Состояния клеток кодируются целыми числами. В приводимых далее исходных кодах компонентов для обозначения состояний клеток автомата определены следующие макроопределения:

```
#define WW_CELL_BACK 0  // пустая клетка
#define WW_CELL_WIRE 1  // проводник
#define WW_CELL_TAIL 2  // хвост электрона
#define WW_CELL_HEAD 3  // голова электрона
```

2.2. Хранилище данных *Integers for Cartesians 2D (WireWorld)*

Набор стандартных компонентов не содержит хранилища данных, удобного для описания автомата с небольшим дискретным множеством состояний.

Хотя требуемая функциональность обеспечивается стандартным компонентом *Integers for Cartesians 2D*, для лучшей визуализации и удобства ввода данных целесообразно создать специализированное хранилище.

Его отличия от стандартного хранилища заключаются в следующем:

- значения параметров `p_iMin` и `p_iMax`, определяющих диапазон допустимых значений, зафиксированы, и равны нулю и трем, соответственно, по числу возможных состояний автомата;
- значение параметра `p_bDouble` зафиксировано в `true`, так как автомат *WireWorld* является синхронным;
- выбор состояния клеток решетки, а также состояния "по умолчанию" осуществляется с помощью списка возможных значений, содержащего четыре элемента, а не посредством ввода числа;

- предусмотрена возможность выбора цветов для каждого из четырех состояний, в то время как стандартный компонент обеспечивал соответствие между состояниями и цветами с помощью спектра. Значения цветов по умолчанию приведены выше.

Для реализации требуемой функциональности был создан класс `CACrtsIntWW2DDatum`, унаследованный от входящего в библиотеку *CADLib* шаблона класса `CABasicCrts2DDatum`.

Ввиду того, что в классе `CChoiceDlg` библиотеки *CADLib*, реализующем диалог с возможностью выбора варианта из списка, отсутствует возможность работы с клетками решетки, был создан класс `CIntChoiceDlg` с требуемой функциональностью.

Так как от стандартного класса новый отличается только настройкой параметров и использованием альтернативного диалога ввода значений состояния клетки, то его реализация не приводится.

2.3. Правила *WireWorld (plain)*

Был разработан класс `CAWireWorld2DRules`, реализующий правила перехода автомата *WireWorld*. Как и любой другой класс компонента данного типа, он был унаследован от класса `CARules`, входящего в библиотеку *CADLib*.

Класс реализует следующие возможности:

- Вычисление новых состояний клеток на основании правил автомата.
- Вычисление значений целочисленного анализируемого параметра `p_iElectrons`, представляющего собой количества клеток решетки, находящихся в состоянии "голова электрона" на данном шаге.

Настраиваемые пользователем параметры отсутствуют.

Компонент требует [9] для совместной работы хранилище целочисленных данных `Data.int.*` (под этот критерий подходит как рассмотренный в разд. 2.2 компонент *Integers for Cartesians 2D (WireWorld)*, так и стандартный компонент `Integers for Cartesians 2D`) и метрику для двумерных решеток `Metrics.2D.*` (подходит стандартный компонент `Cartesians 2D`). В случае, если в качестве хранилища данных используется стандартный компонент `Integers for Cartesians 2D`, для корректной работы следует вручную установить его параметр `Double` в значение `true`. В специализированном хранилище это сделано на уровне реализации.

Таким образом, описание класса имеет следующий вид:

```
class CAWireWorld2DRules:public CARules
{
public:
    // Конструктор, в котором происходит инициализация анализируемого параметра
    CAWireWorld2DRules();

    // Описание компонента с помощью макросов из библиотеки CADLib
    COMPONENT_NAME(WireWorld (plain))
    COMPONENT_INFO(WireWorld Rules (plain variant))
    COMPONENT_ICON(IDI_ICON)

    // Описание условий совместимости
    COMPONENT_REQUIRES(Data.int.*&Metrics.2D.*)
```

```
    // Переопределяемые функции из базового класса CARules
    virtual bool Initialize();
    virtual bool SubCompute(Zone& z);

public:
// Анализируемый параметр:
    ParamInt p_iElectrons;

// Задание карты параметров с помощью макросов из библиотеки CADLib:
public:
    // Нет настраиваемых пользователем параметров
    NO_PARAMETERS

    // Количество анализируемых параметров
    APARAMETERS_COUNT(1)

    // Карта анализируемых параметров
    BEGIN_APARAMETERS
        PARAMETER(0,&p_iElectrons)
    END_APARAMETERS

private:
    // Временная переменная для вычисления значения параметра p_iElectrons
    int iq;
};
```

В конструкторе создается единственный анализируемый параметр и устанавливается строковой комментарий:

```
CAWireWorld2DRules::CAWireWorld2DRules():
    p_iElectrons("Electrons", "Amount of Electron Head cells", 0)
{
    // Комментарий, отображаемый в строке состояния (член класса CARules)
    sComment="WireWorld";
}
```

Функция `Initialize()` переопределяется только для вычисления значения анализируемого параметра перед первым шагом:

```
bool CAWireWorld2DRules::Initialize()
{
    // Если параметр не анализируется, нет необходимости вычислять его значение
    if (!p_iElectrons.IsAnalyzed()) return true;

    // Макрос, определяющий переменную datum - указатель на хранилище данных
    DATUM(CACrtsInt2DDatum);

    CACell c; // Текущая клетка
    iq = 0;   // Счетчик "электронов"

    // Цикл по всем клеткам
    for (CACell i = datum->z.a1; i <= datum->z.b1; i++)
    {
        for (CACell j = datum->z.a2; j <= datum->z.b2; j++)
        {
            // Получение "текущей" клетки
            c=GetUnion()->Metrics->ToCell(i, j, 0);
            // Увеличение счетчика, если "электрон"
            if (datum->Get(c) == WW_CELL_HEAD) iq++;
        }
```

```
    }

    // Установка значения параметра
    p_iElectrons.Set(iq);

    return true;
}
```

Функция `SubCompute(Zone&)` является главной для класса, реализующего компонент правил и обеспечивает выполнение функции переходов в некоторой области решетки. В рассматриваемом случае для данной клетки эта функция будет подсчитывать количество ее соседей в каждом из четырех состояний, и, на основании этого, а также старого состояния клетки вычислять ее новое состояние. Кроме того, внутри этой функции необходимо вычислять новое значение анализируемого параметра.

```
bool CAWireWorld2DRules::SubCompute(Zone& z)
{
    // Макрос, определяющий переменную datum - указатель на хранилище данных
    DATUM(CACrtsInt2DDatum);

    CACell c;         // Текущая клетка
    CACell neig[8];   // Массив соседей

    // Получение количества соседей
    unsigned int ncount = GET_METRICS->GetNeighboursCount();

    unsigned char ntypes[4]; // Массив счетчиков соседей в каждом из состояний
    iq = 0;                  // Обнуление счетчика "электронов"

    // Цикл по всем клеткам
    for (CACell i = z.a1; i <= z.b1; i++)
    {
        for (CACell j = z.a2; j<=z.b2; j++)
        {
            // Инициализация счетчиков соседей
            for (unsigned int k = 0; k < 4; k++) ntypes[k] = 0;

            // Получение текущей клетки
            c = GetUnion()->Metrics->ToCell(i, j, 0);

            // Получение соседей
            pUnion->Metrics->GetNeighbours(c, neig);

            // Подсчет количества соседей в каждом из состояний
            for (k = 0; k < ncount; k++)
            {
                ntypes[datum->Get(neig[k])]++;
            }

            // Определение нового состояния по правилам автомата "WireWorld"
            int new_state;
            switch (datum->Get(c))
            {
                case WW_CELL_BACK:
                    // Пустая клетка всегда остается пустой
                    new_state = WW_CELL_BACK;
                    break;

                case WW_CELL_WIRE:
                    // Проводник становится головой электрона только
                    // если среди соседей есть ровно 1 или 2 головы электрона
```

```
                    new_state = (ntypes[WW_CELL_HEAD] == 1 ||
                                 ntypes[WW_CELL_HEAD] == 2)
                        ? WW_CELL_HEAD
                        : WW_CELL_WIRE;

                    // Увеличиваем счетчик электронов, если произошел переход
                    if (new_state == WW_CELL_HEAD) iq++;
                    break;

                case WW_CELL_HEAD:
                    // Голова электрона всегда становится хвостом электрона
                    new_state = WW_CELL_TAIL;
                    break;

                case WW_CELL_TAIL:
                    // Хвост электрона всегда становится проводником
                    new_state = WW_CELL_WIRE;
                    break;

                default:
                    new_state = WW_CELL_BACK;
            }

            // Установка нового состояния клетки
            datum->Set(c, new_state);

        }
    }
    // Обновление анализируемого параметра
    p_iElectrons.Set(iq);
    return true;
}
```

Следует отметить, что, поскольку используется двойная буферизация решетки автомата, все изменения состояний клеток происходят единовременно, по завершении итерации, поэтому результат вычислений не зависит от порядка обхода клеток.

2.4. Правила *WireWorld (optimized)*

По функциональности и требованиям данный компонент полностью повторяет описанный выше компонент *WireWorld (plain)*, но отличается тем, что для ускорения вычислений используется метод "зональной" оптимизации [9].

Говоря общо, этот метод основан на том, что на каждом шаге изменяется лишь незначительная часть решетки. Это истинно в силу того, что радиус окрестностей клеток фиксирован, а также в силу наличия состояний, в которых клетки будут находиться перманентно в отсутствие влияния соседей. При этом на данном шаге можно определить те клетки, в которых могут произойти изменения на следующем шаге. Назовем такие клетки "значимыми". В результате, при выполнении перехода нецелесообразно заново вычислять новые состояния для каждой клетки решетки, если это можно сделать только для значимых.

Обратим внимание на следующие особенности правил автомата *WireWorld*:

- пустая клетка никогда не изменяет своего состояния;
- проводник может изменить свое состояние, только если в смежных клетках есть головы электронов;

- голова и хвост электрона всегда изменяют свое состояние (на хвост электрона и проводник, соответственно);
- можно сказать, что клетки решетки делятся на два множества – на пустые и непустые. При этом клетки из одного множества не могут перейти в другое.

Заметим, что на текущем шаге могут измениться только клетки, которые изменялись на предыдущем шаге (переходы "голова электрона" → "хвост электрона" и "хвост электрона" → "проводник"), а также смежные с ними (переход "проводник" → "голова электрона"). Таким образом, только окрестности голов и хвостов электронов являются значимыми.

Сравним эффективность такого метода с рассмотренным в разд. 2.3 вариантом реализации. Время выполнения одного шага в "простом" варианте зависит только от размеров решетки, поскольку новое состояние каждой клетки вычисляется ровно один раз на каждом шаге вне зависимости от ее прежнего состояния. Время выполнения одного шага в варианте с зональной оптимизацией зависит от числа электронов на решетке (рядом с которыми и происходят все изменения), так как помечаются значимыми, и следовательно, обрабатываются на очередном шаге только их окрестности. Обычно число электронов на решетке во много раз меньше общего числа клеток. Как следствие, быстродействие оптимизированного таким методом компонента по сравнению с "простым" вариантом реализации увеличивается в несколько раз.

В библиотеке *CADLib* для осуществления зональной оптимизации реализован класс `OptZonal`, обеспечивающий группировку значимых клеток в зоны и поддержку обработки списка таких зон. Зоной называется прямоугольная область решетки, содержащая в себе помеченные значимыми клетки, а также некоторую их окрестность.

Работа с классом `OptZonal` происходит следующим образом. Экземпляр класса создается вместе с экземпляром класса правил в конструкторе последнего. При инициализации эксперимента следует вызвать метод `Reset` для инициализации оптимизатора, после чего добавить все клетки, состояние которых может измениться в ходе работы автомата, в список значимых. Для добавления значимой клетки служит метод `AddCell`, которому передаются координаты клетки. При выполнении шага автомата следует перебрать все зоны с одновременным удалением их из списка оптимизатора (поскольку на следующем шаге множество значимых клеток будет другим), для чего предусмотрен метод `ReleaseZone`. Для клеток каждой из зон вычисляется новое состояние, при этом клетки, в окрестности которых могут произойти изменения на следующем шаге, следует добавить в список значимых с помощью метода `AddCell`.

Конструктор `OptZonal` принимает два параметра. Первый из них – радиус окрестности помечаемых значимыми клеток, клетки которой также считаются значимыми. В рассматриваемой реализации этот параметр равен единице, что означает, что значимой считается клетка, для которой непосредственно вызван метод `AddCell`, а также ее соседи. Второй параметр задает минимальное расстояние между отдельными зонами. Если вновь образованная зона оказывается от какой-либо ранее созданной зоны на расстоянии, меньшем, чем значение этого параметра, то новая зона объединяется с имеющейся – прямоугольник существующей зоны расширяется таким образом, чтобы включить в себя новую зону. Это требуется для сокращения количества зон, и, как следствие, уменьшения накладных расходов при работе со списком зон (в частности, при их переборе). В рассматриваемой реализации этот параметр равен четырем.

Реализация с использованием зональной оптимизации будет отличаться от варианта рассмотренного в разд. 2.3 следующим образом:

- в класс `CAWireWorld2DRules` добавлен атрибут `OptZonal oOpt;`

- при инициализации все клетки, кроме пустых, помечаются, как значимые. Это необходимо для корректной инициализации оптимизатора в условиях двойной буферизации хранилища данных;
- на каждом шаге обрабатывается не вся решетка, а только зоны, о которых информирует оптимизатор;
- в процессе выполнения функции переходов, если клетка меняет свое состояние, то оптимизатору дается команда пометить ее, как значимую для следующего шага.

Исходный текст функций `Initialize()` и `SubCompute(Zone&)` с учетом изменений:

```
bool CAWireWorld2DRules::Initialize()
{
    // Подготовка оптимизатора
    oOpt.Reset();

    // Макрос, определяющий переменную datum - указатель на хранилище данных
    DATUM(CACrtsInt2DDatum);

    CACell c; // Текущая клетка
    iq = 0;   // Счетчик "электронов"

    // Цикл по всем клеткам
    for (CACell i = datum->z.a1; i <= datum->z.b1; i++)
    {
        for (CACell j = datum->z.a2; j <= datum->z.b2; j++)
        {
            // Получение текущей клетки
            c = GetUnion()->Metrics->ToCell(i, j, 0);

            // Подсчет числа электронов
            if (datum->Get(c) == WW_CELL_HEAD) iq++;

            // Инициализация оптимизатора:
            // все клетки, не являющиеся пустыми, помечаются, как значимые
            if (datum->Get(c) != WW_CELL_BACK)
            {
                oOpt.AddCell(i, j);
            }
        }
    }

    // Установка анализируемого параметра
    p_iElectrons.Set(iq);

    // Инициализация двойного хранилища для корректной работы оптимизатора
    if (datum->p_bDouble.Get())
    {
        for (CACell i = datum->z.a1; i <= datum->z.b1; i++)
        {
            for (CACell j = datum->z.a2; j <= datum->z.b2; j++)
            {
                c = GetUnion()->Metrics->ToCell(i, j, 0);
                datum->Set(c, WW_CELL_BACK);
            }
        }
    }

    return true;
}
```

```
bool CAWireWorld2DRules::SubCompute(Zone&)
{
    // Макрос, определяющий переменную datum - указатель на хранилище данных
    DATUM(CACrtsInt2DDatum);

    CACell c;        // Текущая клетка
    CACell neig[8];  // Массив соседей

    // Получение количества соседей
    unsigned int ncount = GET_METRICS->GetNeighboursCount();

    unsigned char ntypes[4]; // Массив счетчиков соседей в каждом из состояний
    iq = 0;                  // Обнуление счетчика "электронов"

    // Сброс оптимизатора перед новым шагом
    oOpt.NewStep();

    Zone z; // текущая зона

    // Проход по всем зонам, содержащим значимые клетки
    while (oOpt.ReleaseZone(z))
    {
        // Цикл по всем клеткам зоны
        for (CACell i = z.a1; i <= z.b1; i++)
        {
            for (CACell j = z.a2; j <= z.b2; j++)
            {
                // Инициализация счетчиков количества соседей
                for (unsigned int k = 0; k < 4; k++) ntypes[k] = 0;

                // Получение "текущей" клетки
                c = GetUnion()->Metrics->ToCell(i, j, 0);

                // Получение соседей
                pUnion->Metrics->GetNeighbours(c, neig);

                // Подсчет количества соседей в каждом из состояний
                for (k = 0; k < ncount; k++)
                {
                    ntypes[datum->Get(neig[k])]++;
                }

                // Определение нового состояния по правилам автомата "WireWorld"
                int new_state;
                switch (datum->Get(c))
                {
                    case WW_CELL_BACK:
                    // Пустая клетка всегда остается пустой
                    new_state = WW_CELL_BACK;
                    break;

                    case WW_CELL_WIRE:
                    // Проводник становится головой электрона только
                    // если среди соседей есть ровно 1 или 2 головы электрона
                    new_state = (ntypes[WW_CELL_HEAD] == 1 ||
                                 ntypes[WW_CELL_HEAD] == 2)
                        ? WW_CELL_HEAD
                        : WW_CELL_WIRE;

                    // Увеличиваем счетчик электронов, если произошел переход
                    if (new_state == WW_CELL_HEAD) iq++;
                    break;
```

```
                    case WW_CELL_HEAD:
                    // Голова электрона всегда становится хвостом электрона
                    new_state = WW_CELL_TAIL;
                    break;

                    case WW_CELL_TAIL:
                    // Хвост электрона всегда становится проводником
                    new_state = WW_CELL_WIRE;
                    break;

                    default:
                    new_state = WW_CELL_BACK;
                }

                // Если состояние изменилось, клетку следует пометить "значимой"
                if (new_state != datum->Get(c))
                {
                    oOpt.AddCell(i, j);
                }

                // Установка нового состояния клетки
                datum->Set(c, new_state);
            }
        }
    }

    // Обновление значения анализируемого параметра
    p_iElectrons.Set(iq);

    return true;
}
```

Заключение

Приведенный пример реализации автомата *WireWorld* иллюстрирует тот факт, что пакет *CAME&L* берет на себя реализацию огромного количества вспомогательной функциональности, в том числе весь пользовательский интерфейс. Для реализации правил автомата требуется владение языком *C++* на минимальном уровне.

На примере автомата *WireWorld*, где на каждом шаге изменяет свое состояние небольшая часть клеток решетки, показана эффективность метода зональной оптимизации. Разработка компонента правил, использующего зональную оптимизацию, значительно упрощается благодаря наличию в пакете *CAME&L* специальных классов для ее реализации.

Все иллюстрации для настоящей статьи были получены с помощью инструментального средства *CAME$_{\&}$L* и компонентов, которые обсуждались в ней. Исходные коды и скомпилированные версии компонентов, разработанных для реализации клеточного автомата *WireWorld* доступны с сайта [5].

Источники

1. *Тоффоли Т., Марголус Н.* Машины клеточных автоматов. М.: Мир. 1991.
2. *Wolfram S.* A New Kind of Science. Wolfram Media Inc. 2002.
3. *Cellular Automata*. Ed. P.M.A. Sloot, B. Chopard, A. Hoekstra / Sixth International Conference on Cellular Automata for Research and Industry, ACRI-2004. Springer–Verlag. 2004.
4. *Sarkar P.* A Brief History of Cellular Automata // ACM Computing Surveys. Vol. 32. 2000. № 1.

5. Сайт "CAMEL Laboratory" – http://camellab.spb.ru.
6. *Naumov L.* CAME&L – Cellular Automata Modeling Environment & Library / Cellular Automata. Sixth International Conference on Cellular Automata for Research and Industry, ACRI-2004. Springer–Verlag. 2004.
7. *Наумов Л.* CAME&L – среда моделирования и библиотека разработчика клеточных автоматов / Труды XI Всероссийской научно-методической конференции Телематика'2004. Том 1. СПб.: СПбГУ ИТМО. 2004.
8. *Наумов Л.* CAME&L – средство для осуществления параллельных и распределенных вычисления на основе клеточных автоматов / Технологии распределенных вычислений. 2005.
9. *Наумов Л.* Решение задач с помощью клеточных автоматов посредством программного обеспечения CAME&L // Информационно-управляющие системы. 2005. №№5, 6.
10. Сайт "The Wireworld computer" – http://www.quinapalus.com/wi-index.html.
11. Сайт "Mirek's Cellebration" – http://mirekw.com/ca/index.html.
12. Сайт "Zillions of Games" – http://zillionsofgames.com.
13. Сайт "WireWorld" – http://karl.kiwi.gen.nz/CA-Wireworld.html.
14. Сайт "Mathpuzzle" – http://www.mathpuzzle.com/
15. *Naumov L.* Generalized Coordinates for Cellular Automata Grids / Computational Science – ICCS 2003. Part 2. Springer-Verlag. 2003.
16. *Наумов Л.* Преимущества использования обобщенных координат для многомерных решеток клеточных автоматов // Труды XII Всероссийской научно-методической конференции "Телематика-2005". Том 1. СПбГУ ИТМО. 2005.