CDL — A Language for Cellular Processing

Christian Hochberger Rolf Hoffmann

Fachgebiet Mikroprogrammierung
Institut für Systemarchitektur
Technische Hochschule Darmstadt
Alexanderstraße 10, D-64283 Darmstadt, Germany
E-mail: {hochberg,hoffmann}@informatik.th-darmstadt.de

Abstract

Cellular Processing is a simple, yet attractive massive parallel processing model. To increase its general acceptance and usability it must be supported by an efficient simulator and a software environment. For this purpose the cellular description language CDL was defined and implemented. With CDL complex cellular algorithms can be described in a concise and readable form. Compilers were developed which generate C–code for the software simulator, a logical design for field programmable gate arrays in the CEPRA–8L machine or machine code for digital signal processors in the CEPRA–8D machine. The special features of CDL are explained in detail, e.g. group constants and the loop construct for testing/describing complex conditions. We present a description of the Belousov–Zhabotinsky reaction as an example for CDL.

1. Introduction

1.1. Cellular Processing

Cellular Processing is based on the processing model of Cellular Automata. In this model the cells are only connected to their neighbours. In the two dimensional case 4 neighbours (von Neumann neighbourhood) or 8 neighbours (Moore neighbourhood) are considered. All processors of the array obey in parallel to the same, simple local rule, which results in a global transformation of the whole generation.

Cellular Automata with a few bits per state have been studied extensively from the theoretic point of view and are of special interest in physics. Large numbers of papers on cellular automata and their application in physics are periodically published in the PHYSICA D [2] and COMPLEX SYSTEMS[1]. We use the term Cellular Processing instead of Cellular Automata because we want to stress that the Cellular Processing model together with a software environment is one of the clearest and most efficient models of parallel computation which can be applied to a large number of applications. Also we do not limit the the state to a few bits but allow to use application specific records of information.

Typical applications are: crystal growth, biological growth, simulation of digital logic, neuronal switching, electrodynamic fields, diffusion, temperature distributions, movement and collision of particles, lattice gas models, liquid flow, wave optics, Ising systems, image processing, and pattern recognition.

With the current software simulator each cellular state may be a record of bits, bytes, integers and reals. Thus almost any application may be described. With the new designed language CDL the cellular algorithms are presented in a concise and readable form. CDL has been proved to be very useful for the description of complex cellular algorithms. One version of the compiler generates C code for the software simulator.

We have also developed two hardware supported simulators, CEPRA-8L and CEPRA-8D. CEPRA-8L is based on field programmable gate arrrays (Xilinx LCA), which are loaded with a special logical design for the cellular rules. The design can be described in the hardware description language 'LOGiC' as input to a design tool or in the language CDL which will be described in this paper. CDL can also be used for the programming of the machine CEPRA-8D, which consists of 8 digital signal processors (DSP). A special backend of the CDL compiler generates machine code for the DSPs.

1.2. Why a new language

Simple rules can be described with look up tables or case statements in conventional languages. For complex rules (e.g. traffic simulation, multi layer layout), which we have in mind, these descriptions become unreadable, long and difficult to change. Describing cellular algorithms with conventional languages like C or Pascal leads to long programs with deeply nested case- and if- statements. Also the cellular states and substates cannot easily be managed. Some other researchers in the field of cellular processing have also defined special celluar languages [3] [6] or extensions of conventional languages which emphazises the need for such languages.

"CDL" (Cellular automaton programming and Description Language) is a language for describing the behaviour of cellular automata for two different purposes. The first is to explain the behavior of a cellular automaton, therefore CDL is easy to read for human readers. Second, CDL is a simulator independent programming language for cellular algorithms.

1.3. Comparison with other Languages

One of the most powerful features of CDL is the possibility of describing the cell states by a user defined type. Types are nearly as flexible as in C or Pascal. CELLANG [3] also has structured cell types, but its subtypes are not as flexible.

CDL uses relative cell addressing as in CELLANG, whereas CDL makes a difference between the address and the contents of a cell. This allows to calculate addresses or store them in variables or in the cell state.

Cellular algorithms often use operations on the *set of neigbours*, like "is there a neigbour in state ...". To describe this behaviour in a program, the approach of shifting in CAL [6] may be useful, but is not sufficient. For the description of complex algortihms CDL offers more powerful operators and language constructs.

The visualization of cellular arrays is often done by assigning colours to cell states or to a set of cell states. To support this assignment a colour definition part is included in CDL. This makes it possible to use all CDL features in these definitions. Firstly the set of states to which a colour expression applies can be specified using statenames, constants and groups that are already defined for this particular cellular automaton. Secondly the colour expression can be built with components of the cell defining a spread of colours in a single expression.

To handle the borders of the cellular field, a cell should be able to find out its position relative to the border during the calculation of its state transition.

2. The Language CDL

CDL assumes that the cells form an n–dimensional lattice. Each dimension is wrapped around. E.g. a two–dimensional lattice forms a torus. The border function can be used to disable the wrap–around.

It is part of the definition of CDL that the C preprocessor is called before the CDL compiler. Therefore the syntax of comments corresponds to C. The user may use `#include` and `#define` to improve the CDL description.

2.1. Special Features of CDL

In this section we describe the special features of CDL. Not every syntactical detail will be explained because CDL is very much like Pascal or C. To make CDL programs short and easy to read we included some powerful and often needed features.

2.1.1 Data Types.

Predefined data types are `boolean`, `integer`, and `float`. User defined types are enumerations, subranges, records and unions. The set of cell states is described by the user defined data type `celltype`.

```
type
   /* an enum type */
   Direction = (up,down,left,right);
   celltype  = record
                 myDir   : Direction;
                 mySpeed : 0..99;
               end;
```

Most simulators (and languages) define the cell state as an integer. It is the users burden to code the states with numbers. The calculation of these numbers will make the program longer and difficult to read or change. In CDL the compiler codes the states. It frees the user of this tedious task.

2.1.2 Constants.

The user can define constants like in ANSI–C or Pascal for any type including records and unions.

To build a constant for a record type, the constant components are written in square brackets and are separated by commas.

The two special integer constants `distance` and `dimension` have to be defined. From this two constants the compiler automatically generates the type

`celladdress`. This type describes the format of addresses that can be used to address the cell array. It is a record with number of `dimension` components, which all have the integer subrange type `-distance` to `distance`.

```
const
   myDirection   = up;
   dimension     = 2;
   distance      = 1;
   /* Address of center cell */
   c             = [0,0];
```

In this example the cell array is two dimensional. Access is restricted to the eight direct neigbours.

2.1.3 Groups.

Groups are lists of constants which must be of the same data type. Integer subranges may be groups, other groups are enclosed in braces. Records and groups are combinable. With the group declaration, a name is introduced (as for constants).

```
group
   vertical  = { down,up };
   smallNum  =   1..4;
   myRecords = [ {1,2,4},true ];
   /*{[1,true],[2,true],[4,true]}*/
```

If the `in`-operator is used, the group will be treated as a set. The order of the elements makes it possible to generate loops over a group.

2.1.4 Colours.

To support the visualization, the user can assign colours to cell states within CDL. This opens up the possibility to access symbolic information on the cell state including the cell structure, constants and groups.

The colour on the left side of the tilde is associated with a cell state by the boolean expression on the right. To evaluate the colour of a cell state the expressions are processed top down. The first expression that matches the condition (that is the right part of the expression evaluates to `true`) defines the colour for this state.

Colours are described with their three components (RGB) in a record. Each component may be an expression. The expressions in the colour definition part must not contain any variables. Only constants and the state of the actual cell (`*[0,0]=*c`) are allowed.

```
colour
    /* red */
   [255,  0,  0]          ~ *c.Dir=left;
   [*c.Speed, 0, 0] ~ *c.Dir in vertical;
   /* white */
   [51*5,255,255]         ~ *c.Speed > 30;
```

2.1.5 Variables.

Variables can be declared for any type. During variable declaration, a new type may implicitly be declared. The scope of variables is limited to the calculation of the next cell state. It is not possible to transfer information to other cells or generations through variables.

```
var
   i,j : integer;
   d   : Direction;
   r   : record num : -5 .. 5;
                 dir : Direction;
         end;
```

2.1.6 Rules.

The behaviour of the cells is described in statements and expressions which are similar to those in C or Pascal. In CDL only one main cell procedure can be described beginning with the keyword `rule`. Descriptions are usually short, so procedures or functions are not necessary.

The reference operator `*` is used to access the contents of a cell. The relative address of the cell must be of type `celladdress`. Because this is a record, it is written in square brackets. For the center cell all `dimension` components of the address have to be zero. To access the next neigbour in direction to the origin of the cell field (in the two dimensional case defined as the lower left corner), decrement the index of the corresponding dimension. Note that the absolute value of an address component must not exceed `distance`.

CDL has the usual conditional statements `if`, `if–else`, and `case` with optional `otherwise` branch.

There are no conditional loops, but there are often needed powerful expressions instead. CDL has a `for` loop statement and the looping expressions `all`, `one`, and `num`. The loops are controlled by a *loopvariable–list*. The `for` loop's syntax is:

`for` *loopvariable–list* `do` *statement* ;

The statement is executed for each instantiation of the *loopvariable–list*. A *loopvariable–list* is a list of variables seperated by commas, followed by the keyword `in` and a group. For the second instanciation, the first constant of the group is assigned to the first variable, the second constant to the second variable, and so on. If there are more variables than constants in the group, the end of the group is continued with its beginning.

Example: `for  i,j in {3,4,5}  do` *statement*;

For the second instantiation, the second constant is assigned to the first variable, the third constant to the second variable, and so on. The loop is executed as many times as there are number of constants in the group. The above example will lead to three executions of the *statement* with the variables `i` and `j` set to the following values:

instance	i	j
1	3	4
2	4	5
3	5	3

There may be more than one of this variables group pair, separated by the character `&`. In this case, the number of instances is defined by the size of the largest group. The statement
`for d in vertical & i,j in {3,4,5} do ...;`
will also create three instances with the variables set to the following values:

instance	d	i	j
1	down	3	4
2	up	4	5
3	down	5	3

The expressions `num`, `one`, and `all` do also have a *loopvariable–list*. Their syntax is:

`one(` *loopvariable–list* `:` *expression* `)`

The *expression* must be of type boolean. It is evaluated for each instance with the variables set according to the rules explained above.

The `num` expression returns the number of instances in which the *expression* evaluates to `true`. The `one` expression returns `true`, if *expression* evaluates `true` at least once. The `all` expression returns `true`, if *expression* evaluates to `true` for each instance, and `false` otherwise.

`all` and `one` will "stop" execution if the result is already determined. For `all` this will be the first time *expression* evaluates to `false`, for `one` it will be the first *expression* that evaluates to `true`. The loop variables will be set according to the last evaluated *expression*.

The rules of CDL are deterministic except for the usage of the `random(`<*n*>`)` function.

The function `border` is used to control the border of the cell array. It returns a value of type `celladdress`. For inner cells (those, where the border is farther away than `distance` cells) all components of the returned record are zero. For the cell nearest to the origin all components are `-distance`. The farther away the cell is from the origin the more the component becomes `+distance`. E.g. in the two dimensional case: upper right corner = `[1,1]`, left border (without corners) = `[-1,0]`. Access to cells over the border is never forbidden. It is in the users control to call the border function and to use different rules on different return values.

2.2. Limitations of CDL

CDL was designed to be a portable, simulator independent programming language for cellular automata. However, some of its properties can impose rewstrictions on us.

Because there are no conditional loops in CDL it is possible to unroll the program. Therefore the calculation of the next cell state will always terminate. Some other languages, like CELLANG or CAL include this limitation. This is important for hardware supported simulators (see Sect. 3).

For the same reason and taking implementation costs into consideration, there are no functions or procedures. Recursions would break the possibility of unrolling.

The relative cell addressing with euclidean coordinates requires a rectangular lattice of cells. But other neighbourhood models (like hexagonal) may be simulated.

Assigning colours to states is a very simple visualisation concept. It may be useful to assign icons to cellstates or to blocks of cellstates.

3. Implementation of Compilers

At the moment we have implemented two CDL compilers. The first one translates the CDL program into an experiment for a X11/Motif based simulator. For this reason the rule part of the CDL program is translated into C. Additionally, files for controlling visualization and editing are generated[5].

The second compiler[7] translates the rule into a set of boolean equations and function tables that describe the automaton. The synthesized circuit uses the actual states of the cell and its neighbours as inputs and generates the next state as its output. The generated description is then automatically simplified and mapped onto field programmable gate arrays used in our hardware simulator CEPRA–8L [4]. Through this compiler we provide a complete mapping from CDL down to hardware. All CDL language features can be used. The complexity of the cell must not exceed eight bits. This restriction is due to the hardware of our simulator. The compiler is able to build the crossproduct of all the components of a cell to increase the possible complexity. A short example will illustrate the advantage of this. Assume the following CDL definitions:

```
type
   state = { hi, ho, good, bad, foo};
   range = 1..50;

   celltype = record  i1:state;
                      12:range;
              end;
```

The usual method of giving every component a number of bits in the state vector would lead to a complexity of nine bits. Three bits for the component `i1` (five elements) and six bits for the component `i2` (50 elements). The crossproduct gives us a number of 250 combinations which we simply enumerate. This enumeration will then fit into eight bits.

A third compiler is currently under development. It will generate DSP assembler code for another hardware supported simulator (CEPRA–8D). It has only a small amount of program memory but offers some additional data memory. The task of the compiler will be to put as much information about the rule into tables as possible.

In the future we plan to develop CDL compilers for massive parallel systems. This will basically result in a transformation of CDL into C, since this language will generally be supported by vendors of parallel systems.

The language itself will be extended to support moving objects and more complex visualization algorithms.

4. Examples

As an example of a CDL program we present the Belousov–Zhabotinsky reaction[8]. It does not show all the special features of CDL, but gives a good impression of CDL in general.

```
(01) cellular automaton Belousov_Zhabotinsky;
(02)
(03) const dimension = 2 ;
(04)       distance  = 1 ;
(05)       maxtimer  = 7 ;
(06)       c         = [0,0];
(07)
(08) type celltype = record
(09)        active : boolean;
(10)        alarm  : boolean;
(11)        timer  : 0..maxtimer;
(12)      end;
(13)
(14) group
(15)  neighbours={[-1,0],[1,0],[0,1],[0,-1],
(16)            [1,1],[-1,1],[1,-1],[-1,-1]};
(17) color
(18)  [0,255,0] ~ *c.active and     *c.alarm;
(19)  [255,0,0] ~ *c.active and not *c.alarm;
(20)  [*c.timer * 255 div maxtimer,0,0]
(21)                        ~ not *c.active;
(22)
(23) var
(24)   n : celladdress;
(25)
(26) rule
(27) begin
(28)  *c.active := *c.timer=0 ;
(29)  *c.alarm  := num(n in neighbours:
(30)                  *n.active) in {2,4..8};
(31)  if *c.active and *c.alarm
(32)               and (*c.timer=0)
(32)    then *c.timer:=maxtimer
(33)    else
(34)      if *c.timer!=0 then
(35)              *c.timer:=*c.timer-1;
(36) end;
```

At first the name of the automaton is declared. In lines `(03)`–`(06)` the two required constants `dimension` and `distance` are defined. The constants `maxtimer` and `cell` are defined only for convenience. The structure of the cell state is declared in lines `(08)`–`(12)`. The set of neighbours is declared in lines `(15)`–`(16)`. In this case all eight neighbours of the Moore neighbourhood are considered. Three colour expressions define the visualization of the cells. The first expression gives bright green in case the cell is active and alarmed. The cell looks bright red if the cell is active and not alarmed. In case the cell is inactive the colour is based on the timer value. The declaration of the local variable `n` in line `(24)` is necessary for the `num` expression in line `(30)`–l31. In line `(28)` the cell is activated if the timer is 0. In line `(30)`–`(31)` the number of active neighbours is evaluated. If this number is 2 or in the range of 4 to 8 the cell gets alarmed. Lines `(31)`–`(35)` control the timer. If the cell is not alarmed and not active the timer is decremented. Otherwise the timer is initialized to the `maxtimer` value.

Fig. 1 shows the state of the cellular array after approximately 2000 iterations, starting from a random configuration.

It is important to note again, that the assignments to the cell state are not performed immediately. All assignments are executed when the processing of the cell rule is finished.

5. Conclusion

CDL is an implemented language for the concise and readable description of cellular algorithms. There are three compiler versions, one which generates C–code

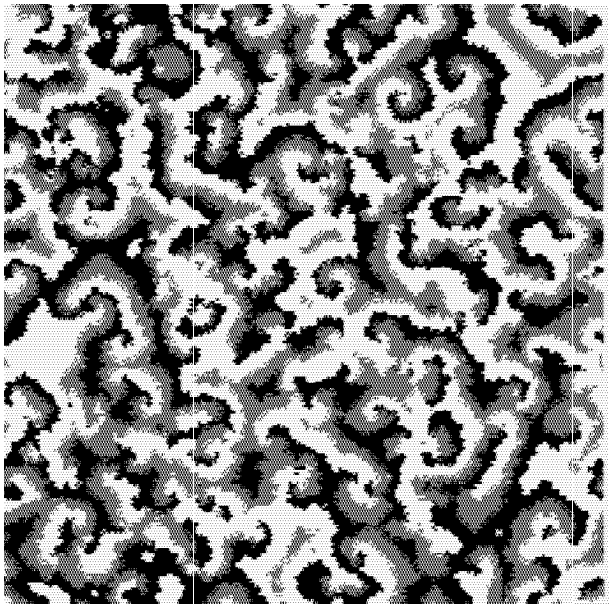

Figure 1. Example picture of the presented BZ–reaction

for the software simulator, one which generates logic designs for field programmable gate arrays, and one which generates machine code for digital signal processors. Main features of the language are records, unions, groups and the loop construct for testing/describing complex conditions. The language can be used to describe complex cellular algorithms of practical relevance like the Belousov–Zhabotinsky reaction. More complex systems like traffic simulation or routing algorithms have prooved the utility of CDL. Based on the experience with CDL we plan to extend the language with features like phase algorithms, moving cells, initialisation of the processing field and complex visualizations.

References

[1] *Complex Systems*, 1-7.

[2] *Physica D*, 1984, 1989, 1990.

[3] J. D. Eckart. A cellular automata simulation system. *SIGPLAN Notices*, (26):80–85, August 1991.

[4] R. Hoffmann, K.-P. Völkmann, and M. Sobolewski. The cellular processing machine cepra-8l. *Mathematical Research*, 81:179–188, 1994.

[5] H. Hussain. *Integration eines Compilers für die Zellularsprache CDL in das XCellsim–System.* Technische Hochschule Darmstadt, 1994.

[6] I. J. Palmer. *Scamper.* available by anonymous ftp from ftp.uu.net.

[7] S. Waldschmidt and C. Hochberger. Fpga synthesis for cellular processing. *Proceedings of the 1995 IEEE/ACM International Workshop on Logic Synthesis*, pages 9–55 – 9–63, May 1995.

[8] A. Zaikin and A. Zhabotinsky. *Nature*, (225):535–, 1970.