The DigiHive environment.

Rafał Sienkiewicz

March 4, 2011

Contents

Chapter 1

The description of the DigiHive environment

This chapter presents a new artificial world model in which various self-organization and self-modification processes could be simulated. The model is a two dimensional space in which there are stacks of hexagonal tiles. The system operates on two levels. On the first level objects of the system move, collide, rebound, making bonds between them and randomly change their structures. On the second level, some structures of the objects are capable of inducing changes in other objects. The types of changes are encoded in the structures of objects in specially defined Prolog-like language.

The existence of objects capable of inducing changes in other objects creates a possibility of mutual change of structures of objects and thus the functions performed by them. These enable the possibility of simulation of complex, global behaviour of systems and the emergent phenomena as a result of simple, local interactions. Especially the various self-reproduction strategies or a number of open problems in the field of artificial life [1] can be investigated e.g. simulations of spontaneous generation of life-like systems, novel living organization or open-ended evolution of life. The proposed DigiHive environment could be seen as an artificial life or an artificial chemistry (with implicit reaction laws) system or a model of an autonomous multi-agent system.

In Sect. 1.1 the first level of interaction (simplified physics) has been described. Sect. 1.2 contains a description of the embedded language. The last section 1.3 contains a discussion of environmental assumptions. The documentation and the results of experiments are included in the following PhD dissertation: [15]. Other results were published in the following papers: [5, 6, 7, 8, 13, 14, 17, 16].

1.1 First level: "Thermodynamics"

The environment is defined over two-dimensional continuous space with a toroidal boundary condition.

1.1.1 Particles and complexes

The basic universe constituent objects are called *particles*. The particles are persistent – they can not be created nor annihilated during the simulation. The particles are modelled as hexagons inside circles with a radius $R \in \mathbb{R}^+$ (typically $R = 1$). Particles have its velocity ($\boldsymbol{v} = \boldsymbol{i}v_x + \boldsymbol{j}v_y$, $\boldsymbol{v} \in \mathbb{R} \times \mathbb{R}$, position ($\boldsymbol{s} = \boldsymbol{i}s_x + \boldsymbol{j}s_y$, $\boldsymbol{s} \in \mathbb{R} \times \mathbb{R}$),

internal energy ($E_i \in \mathbb{R}^+$), and are of 255 types ($t \in \mathsf{N} \cap < 0, 255 >$). Each type is connected with set of attributes[1] :

1. mass $m(t) \in \mathbb{R}^+ \setminus \{0\}$ – mass of particle of type t,
2. bond energy: $E_{pb}(t, t') \in \mathbb{R}$ – energy needed to disrupt the bond between particle of type t and t',
3. activation energy: $E_{ac} \in \mathbb{R}^+$ – energy needed to initiate any transformation of bonds
4. bond mask – specifies possible bond directions (described later)

The particles can bond together forming a *complex* of particles. The permanence of the complex and its ability to react, depends on its constituent particles bond energies. Particles can bind horizontally in the following directions : N, NE, SE, S, SW, NW. Besides, in order to reduce structure surface, it is also possible to create vertical bonds: in U (up), and D (down) directions, normal to horizontal ones. However, without any limitations, such a possibility will end in creating very complex, hard to examine and modify 3D structures (compare 1.2). The following restriction is then implemented:

Rule 1 (Bond limitation) *The bond between particle P1 and particle P2 on P1's direction* D *can exists if and only if particle P1 has no horizontal bonds* [2]

As the result, complexes are build from a set of stacks of particles , different stacks are bound together only via its bottom particles. Examples of complexes and possible bond directions are shown in Fig. 1.1.

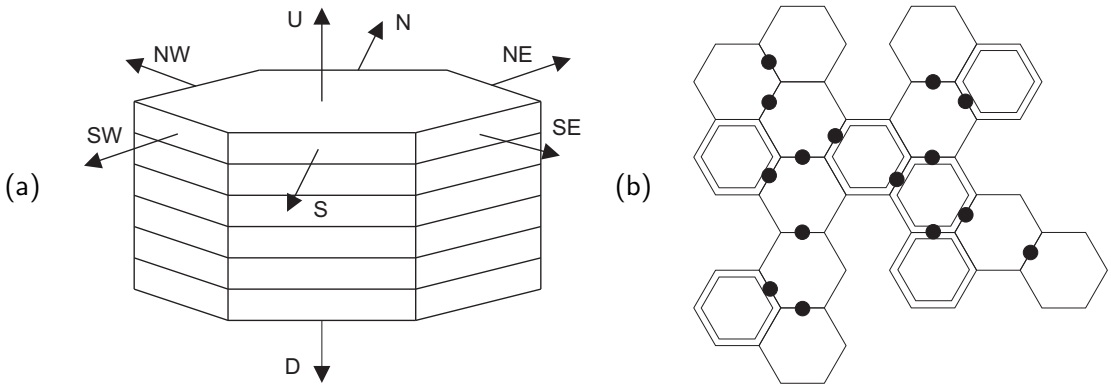

Figure 1.1: Examples of complexes: (a) the horizontal view of single stack of particles with directions shown, and (b) the vertical view of complex formed by horizontal bonds, where hexagons drawn by single lines represent single particles, and by double lines represent stacks of particles and black dots mark horizontal bonds between particles

The distance between the particles should always obey the following rule:

Rule 2 (Minimal distance) [3] *Particles cannot occupy the same place, unless they form a stack of particles. The minimal distance between them (i.e. distance between centres of particles:* $d = |\boldsymbol{s_1} - \boldsymbol{s_2}|$*) should be not less than* $R\sqrt{3}$.

[1] The possible setting are presented in Chapter 2

[2] Note that bond between particle P1 and P2 on P1's direction D implicates bond between P2 and P1 on P2's direction U.

[3] The rule describes desirable state at the beginning of each time step. In fact it may be temporary violated e.g. during particle movements.

All processes are synchronised by a discrete clock. At the end of the time step each particle's position is updated according to:

$$\boldsymbol{s}(i+1) = \boldsymbol{s}(i) + \boldsymbol{v}T \tag{1.1}$$

where i denotes the time cycle, T denotes the length of the time step (usually $T = 1$). During movements for every pair of particles, with possible collision, calculation of the minimal distance is performed. If the minimal distance is less than $1.4R$ (value chosen empirically – see also note to rule 2) the collision procedure is resolved. According to the preset probabilities, one of the following types of collisions is selected: perfectly elastic or perfectly inelastic.

During the perfectly elastic collision, the assumption of conservation of the momentum, as well as the conservation of kinetic energy, makes it possible to calculate final velocities:

$$v_1(i+1) = \frac{v_1(i)(M_1 - M_2) + 2M_2 v_2(i)}{M_1 + M_2} \tag{1.2}$$

$$v_2(i+1) = \frac{v_2(i)(M_2 - M_1) + 2M_1 v_1(i)}{M_1 + M_2} \tag{1.3}$$

where v_1, v_2 denotes the collision-causing velocity components, i denotes the time cycle, M_1 and M_2 denotes objects masses i.e. mass of complexes to which the colliding particles belong.

After the perfectly inelastic collision only the momentum is preserved. Final velocities are calculated according to:

$$\boldsymbol{v}_1(i+1) = \boldsymbol{v}_2(i+1) = \frac{M_1\boldsymbol{v}_1(i) + M_2\boldsymbol{v}_2(i)}{M_1 + M_2} \tag{1.4}$$

where $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ denotes velocities, i denotes the time cycle, M_1 and M_2 denotes masses. To compensate the energy loss, a new photon (see 1.1.2) is generated with the following energy:

$$E_{ph}(i+1) = \frac{M_1 M_2}{2(M_1 + M_2)} |\boldsymbol{v}_1(i) - \boldsymbol{v}_2(i)|^2 \tag{1.5}$$

There is no transitional kind of collision nor any type of rotation movements.

1.1.2 Photons

In addition to permanent particles, the universe contains temporary entities called *photons*, which transport energy and are created by reactions which dissipate energy. The photons may also be emitted spontaneously by particles – after movement every particle may (with a preset probability) convert a part (the maximal quantity of such energy is bounded by the preset constant) of its internal energy into a new photon.

The photons have no mass or momentum and always move with a constant velocity. The photons are characterised by the following attributes: energy ($E_{ph} \in \mathbb{R}^+ \setminus \{0\}$), position ($\boldsymbol{s}_{ph} = \boldsymbol{i}s_{phx} + \boldsymbol{j}s_{phy}$, $\boldsymbol{s}_{ph} \in \mathbb{R} \times \mathbb{R}$) and direction angle ($\alpha \in \mathbb{R} \cap < -\pi, \pi)$).

At the end of each time cycle, each photon's position is updated according to:

$$\boldsymbol{s}_{ph}(i+1) = \boldsymbol{s}_{ph}(i) + c(\boldsymbol{i}\cos\alpha + \boldsymbol{j}\sin\alpha) \tag{1.6}$$

where i denotes the time cycle, $c \in \mathbb{R}$ denotes photon velocity (the predefined value, usually $c = 5$).

Photons may collide with particles. Like in particle-particle collisions, there are two types of collisions: elastic and inelastic. Elastic collisions change photon direction only (the direction angle has a new random value). After an inelastic collision one of the following reactions is randomly selected (each reaction has its own preset probability):

1. Rebounding of the particle hit by the photon from an adjoining particle. Requires:

   - Distance between particles: $d \leq 2R \cdot 1.1$
   - Particles cannot belong to the same complex

   The photon is absorbed by the hit particle – its energy is transformed into the kinetic energy of rebounded particles. New velocities are calculated according to:

$$v_{1x}(i+1) = \frac{p_x \pm M_2\sqrt{2\beta M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \tag{1.7}$$

$$v_{1y}(i+1) = \frac{p_y \mp M_2\sqrt{2(1-\beta) M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \tag{1.8}$$

$$v_{2x}(i+1) = \frac{p_x \mp M_1\sqrt{2\beta M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \tag{1.9}$$

$$v_{2y}(i+1) = \frac{p_y \pm M_1\sqrt{2(1-\beta) M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \tag{1.10}$$

   where: v_1, v_2 denote the objects velocities, i denotes the time cycle, M_1 and M_2 denote complex masses (to which particles belong), $p_x = M_1 v_{1x}(i) + M_2 v_{2x}(i)$, $p_y = M_1 v_{1y}(i) + M_2 v_{2y}(i)$, $M_r = (M_1 + M_2)/(M_1 M_2)$. The parameter β is chosen randomly from range $0 \leq \beta \leq 1$ and describes photon energy distribution for the kinetic energy increases in x and y directions.

2. Creating a horizontal bond between the hit particle (of type t) and the adjoining particles (of type t'). Requires:

   - Distance between particles: $d \leq 2R \cdot 1.1$
   - Particles cannot be bound together
   - $E_{ph} \geq E_{ac}$
   - Photon energy after reaction $E_{ph} > 0$

   New velocities are calculated according to the equation (1.4). Besides, new photon energy is:

$$E_{ph}(i+1) = E_{ph}(i) - E_{pb}(t, t') + \frac{M_1 M_2}{2(M_1 + M_2)} |\boldsymbol{v}_1(i) - \boldsymbol{v}_2(i)|^2 \tag{1.11}$$

   where: v_1, v_2 denote the objects' velocities, i denotes the time cycle, M_1 and M_2 denote complex masses (into which particle belongs) and s_1, s_2 denote particles' positions. If the calculated photon energy is positive, particles' position is fixed (the distance between any of the horizontally bound particles should be equal to: $d = R\sqrt{3}$) photon randomly changes its direction and finally a new bond is created. New particles' positions are:

$$\boldsymbol{s_1}(i+1) = \boldsymbol{s_1}(i) \tag{1.12}$$

$$\boldsymbol{s_2}(i+1) = \begin{cases} \boldsymbol{i}s_{1x}(i) + \boldsymbol{j}(s_{1y}(i) + R\sqrt{3}) & \text{if } dir = \mathsf{N} \\ \boldsymbol{i}(s_{1x}(i) + 1.5R) + \boldsymbol{j}(s_{1y}(i) + 0.5R\sqrt{3}) & \text{if } dir = \mathsf{NE} \\ \boldsymbol{i}(s_{1x}(i) + 1.5R) + \boldsymbol{j}(s_{1y}(i) - 0.5R\sqrt{3}) & \text{if } dir = \mathsf{SE} \\ \boldsymbol{i}s_{1x}(i) + \boldsymbol{j}(s_{1y}(i) - R\sqrt{3}) & \text{if } dir = \mathsf{S} \\ \boldsymbol{i}(s_{1x}(i) - 1.5R) + \boldsymbol{j}(s_{1y}(i) - 0.5R\sqrt{3}) & \text{if } dir = \mathsf{SW} \\ \boldsymbol{i}(s_{1x}(i) - 1.5R) + \boldsymbol{j}(s_{1y}(i) + 0.5R\sqrt{3}) & \text{if } dir = \mathsf{NW} \end{cases} \tag{1.13}$$

where dir is calculated according to:

$$dir = \begin{cases} \mathsf{N} & \text{if } \varphi \geq -\frac{\pi}{6} \text{ and } \varphi < \frac{\pi}{6} \\ \mathsf{NE} & \text{if } \varphi \geq \frac{\pi}{6} \text{ and } \varphi < \frac{\pi}{2} \\ \mathsf{SE} & \text{if } \varphi \geq \frac{\pi}{2} \text{ and } \varphi < \frac{5\pi}{6} \\ \mathsf{S} & \text{if } \varphi \geq \frac{5\pi}{6} \text{ or } \varphi < -\frac{5\pi}{6} \\ \mathsf{SW} & \text{if } \varphi \geq -\frac{5\pi}{6} \text{ and } \varphi < -\frac{\pi}{2} \\ \mathsf{NW} & \text{if } \varphi \geq -\frac{\pi}{2} \text{ and } \varphi < -\frac{\pi}{6} \end{cases} \qquad \varphi = \arctan \frac{s_{2x} - s_{1x}}{s_{2y} - s_{1y}} \tag{1.14}$$

3. Creating a vertical bond between the hit particle (of type t) and the adjoining particles (of type t') [4].

   Requires:

   - Distance between particles: $d \leq 2R \cdot 1.1$
   - Particles cannot be bound together
   - $E_{ph} \geq E_{ac}$
   - Photon energy after reaction $E_{ph} > 0$

   Reaction is very similar to previous one, the only difference is, that the new particles' positions are calculated according to:

$$\boldsymbol{s_1}(i+1) = \boldsymbol{s_2}(i+1) = \boldsymbol{s_1}(i) \tag{1.15}$$

4. Removing the horizontal bond between the hit particle (of type t) and the bound particles (of type t'). Requires:

   - $E_{ph} \geq E_{ac}$
   - Photon energy after reaction $E_{ph} > 0$
   - Particles should be bound together horizontally

   After the reaction, photon energy is updated:

$$E_{ph}(i+1) = E_{ph}(i) + E_{pb}(t, t') \tag{1.16}$$

   Positions and velocities of participating particles are invariable.

5. Removing the vertical bond between the hit particle (of type t) and the bound particles (of type t'). Requires:

[4] Counterpart of concatenating particles from previous version.

- $E_{ph} \geq E_{ac}$
- Photon energy after reaction $E_{ph} > 0$
- Particles should be bond together vertically

The reaction is very similar to the previous one. The only difference is, that the new particles' positions are calculated according to the equation (1.13), where *dir* is randomly chosen, s_1 stands for the position of the bottom particle and s_2 stand for the position of the top particle. If the top particle has some particles bound on its `U` direction, the whole particle stack is relocated (the top particle becomes the bottom particle of a new separate particle stack). After moving the top particle to its new position, it should not overlap the position of any other particle (the minimal distance must be preserved – rule 2) – if this requirement cannot be fulfilled, another direction in equation (1.13) is tried.

6. Photon absorption. The photon is absorbed by the hit particle, and is converted into its internal energy:
$$E_i(i+1) = E_i(i) + E_{ph} \tag{1.17}$$

If the selected reaction cannot be completed, e.g. because of insufficient photon energy, no other action is performed (the photon moves with the same direction in next time cycle).

1.2 Second level: "Biochemistry"

Another class of interactions results from the assumption that the particles forming complexes are capable of inducing reactions in their surroundings. The possible reactions include moving the particles or creating and removing the bonds between them. The description of the reaction is contained in the types and locations of the particles in a complex, which is interpreted as a program written in a language described below language. Programs can recognize and manipulate particular structures.

1.2.1 Syntax

Program syntax is similar to Prolog with the following predicates (also called commands): `program`, `search`, `action`, `structure`, `exists`, `bind`, `unbind`, `move`, `not`. Specific syntax of the above predicates (especially `exists`) prevent, however, from easy transformation of the program into the valid Prolog. Similarity to the pure Prolog is clearly visible in the overall structure and the execution algorithm (more details are described in 1.2.2).

Language syntax in BNF notation is presented in Fig. 1.2 (see also: 2.3). An example of a program is presented in List. 1.1 – the program recognizes the structure presented in Fig. 1.3.

Program structure forms a tree with a root element `program()`. The tree representation of the example from List. 1.1 is presented in Fig. 1.4.

1.2.2 Semantics

Program execution is based on the Prolog backtracking algorithm (e.g. [2, 4]).

```
program ::= search action definitions
search ::= search() :- header .
action ::= action() :- row_action {, row_action } .
definitions ::= row_definition {row_definition }
row_definition ::= header :- body.
header ::= structure( integer )
body ::= exists( exists ) {,exists( exists )} {,not( header )}
     | not( header ) {,not( header )}
short ::= 0|1|2|3|4|5|6|7|8|9
integer ::= short {short}
exists ::= [not] c [[[not] bound [to f] [in d]] | [adjacent [to f] [in d] ]],
     [mark f]
row_action ::= bind( action_spec )
     | unbind( unbind_spec )
     | move( action_spec )
action_spec ::= f to f in d
unbind_spec ::= f [from f] [in d]
c ::= [0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×]
f ::= V short
d ::= N|NE|SE|S|SW|NW|U|D
```

Figure 1.2: BNF syntax of the DigiHive language.

Each predicate returns one of the following status: *OK** (this means that unification succeed but not every possibility has been yet checked), *OK* (this means that unification succeed and every possibility has been already checked) and *FAIL* (this means that unification failed).

The predicates can be divided into the following groups:

1. Program structure maintaining: `program`, `search`, `action`, `structure`

   Each program consists of a single, main predicate `program`. This predicate always consists of the two following predicates: `search` and `action`. The first one groups searching predicates, while the second one groups execution predicates that can affect the neighbouring particles (by moving the particles or creating and removing the bonds between them). "Actions" are performed only if searching succeeded, i.e. if particular structures were recognized.

   Searching commands are grouped in the `structure` predicates, which describe some particular structures. These predicate groups both `exists` (see 2) and other `structure`. The predicate `structure` embedded in another `structure` is always in its negative form (`not(structure())` – see 4), i.e. structure description is formed by the sequence of `exists` predicates and the sequence of some negative conditions. The information flow (via `Vv`ariables) in the embedded structures is possible only downwards, i.e. the embedded structures can check some additional condition, but they cannot send any information upwards (except a simple answer – structure exists or structure does not exists).

   Each predicate of the group returns (note: see also 4):

```
program():-
    search(),
    action().

search():-
    structure(0).

structure(0):-
    exists(000000xx mark V1),
    exists(11111111 bound to V1 in N mark V2),
    exists(00000000 mark V5),
    not(structure(1)),
    not(structure(2)).

structure(1):-
    exists(11110000 bound to V2 in NW mark V3),
    exists(11110000 bound to V3 in SW mark V4),
    not(structure(3)).

structure(3):-
    exists(00001111 bound to V4 in S).

structure(2):-
    exists(10101010).

action():-
    bind(V2 to V5 in SW).
```

Listing 1.1: Example of a program recognising the structure shown in Fig. 1.3.

- *OK* if every called predicate returns *OK*
- *OK** if no one of the called predicates returns *FAIL* and at least one returns *OK**
- *FAIL* if at least one of the called predicates returns *FAIL*

2. Searching: `exists`

   The predicate `exists` is the basic command for structure searching. It recognizes the particle (or empty place – see 2j) of a particular type, with particular bond structure. It also is checks the basic states and relations between two objects.

   In predicate definition:

   `not c [not] bound to f in d, mark f`

   square brackets denote an optional part of the predicate. All parts are hierarchically ordered – the nonexistence of top level elements implicates the nonexistence of lower level parts. The hierarchy is shown in Fig. 1.5.

   Depending on its final syntax, the predicate is able to (for more details see 1.2.3):

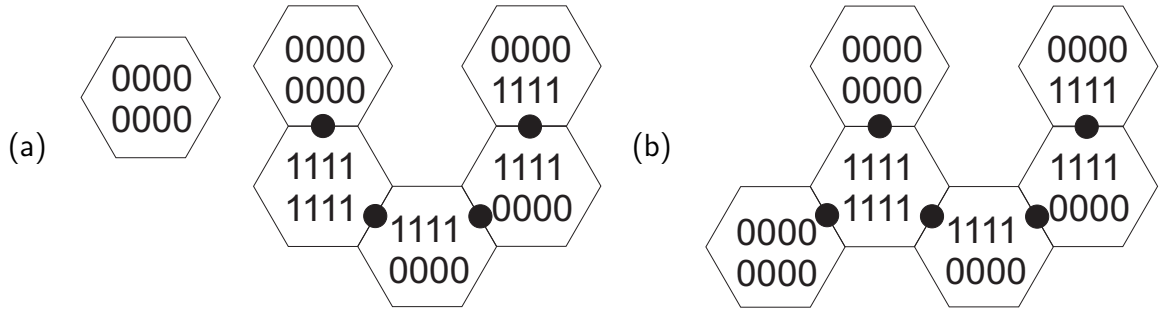

Figure 1.3: Single particle and a complex of particles recognized by the program presented in Fig. 1.1 (a), and the structure after action of the program (b).

(a) find the particle of a specified type in near space: `exists [not] c`. The number of different types describes by the predicate depends on the number of fixed bits (i.e. 0 or 1) used in `c`. For the given number of fixed bits $b \in < 0, 8 >$, `exists c` describes $2^{(8-b)}$ different types and `exists not c` describes $256 - 2^{(8-b)}$ different types. Note that for $b = 0$, `exists not c` (i.e. `exists not xxxxxxxx`) cannot find any particle – for more information see 2j. E.g.:

   i. `exists 11110000` – find a particle of type 11110000
   ii. `exists not 11110000` – find a particle of any type but not 11110000
   iii. `exists 0000xxxx` – find a particle of type from 00000000 to 00001111
   iv. `exists not 0000xxxx` – find a particle of type from 00010000 to 11111111
   v. `exists xxxxxxxx` – find a particle of any type (see also 2j for `exists not xxxxxxxx`)

(b) check if the particle found in 2a is (or not is) bound to any other particle: `exists [not] c [not] bound`, e.g.:

   i. `exists 11110000 not bound` – find a particle of type 11110000 which is not bound.
   ii. `exists 11110000 bound` – find a particle of type 11110000 which is bound.
   iii. `exists not 11110000 not bound` – find a particle of any type but not 11110000 which is not bound.

(c) check if particle found in 2a is bound to a previously found particle: `exists [not] c bound to v`, where `v` denotes a variable which should be earlier (by some previous `exists`) set by a `mark` part (see 2k). If `v` stands for an empty place (see 2j), the predicate fails. E.g.:

   i. `exists 11110000 bound to V1` – find a particle of type 11110000 which is bound to the particle identified by `V1`.
   ii. `exists not 11110000 bound to V1` – find a particle of any type but not 11110000 which is bound to the particle identified by `V1`.

(d) check if the particle found in 2a is bound in (or not in) specified direction: `exists [not] c bound in d`, e.g.:

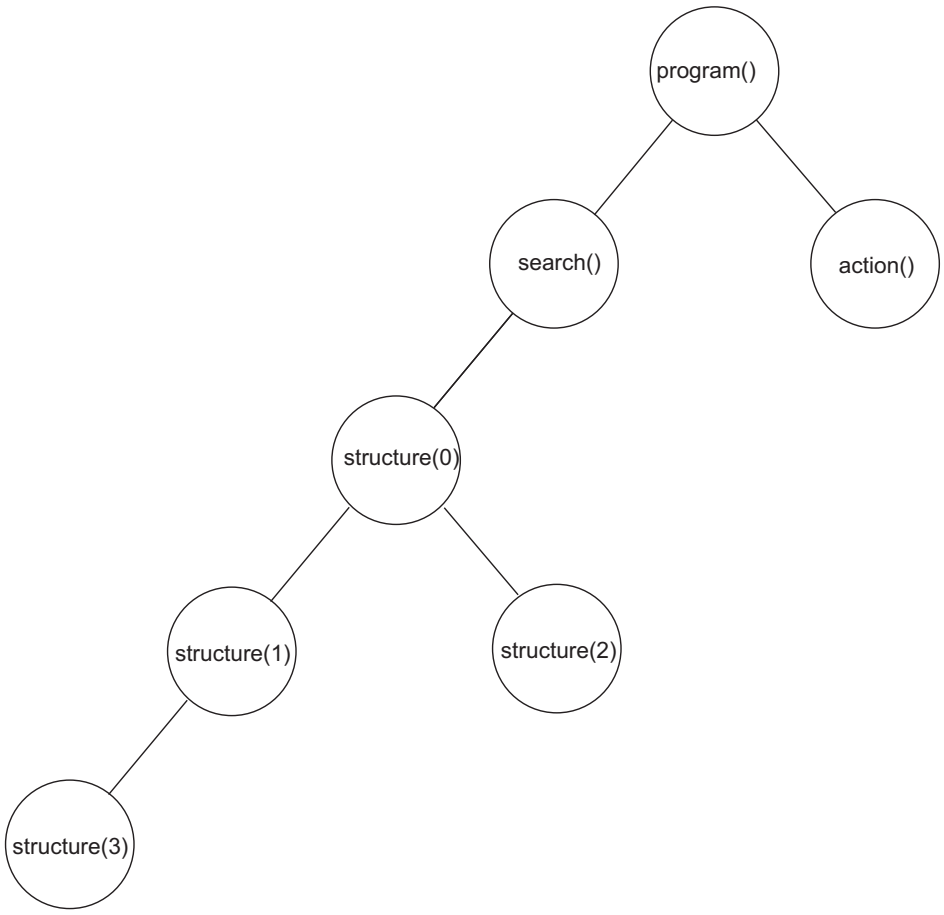

Figure 1.4: Example of a tree structure. The predicates: `exists` and `bind` are omitted.

   i. `exists 11110000 bound in N` – find a particle of type 11110000 which is bound in direction `N` (note: see also 1.2.4 on page 20).
   ii. `exists not 11110000 bound in N` – find a particle of any type but not 11110000, which is bound in direction `N` (note: see also 1.2.4 on page 20).

(e) check if the particle found in 2a is bound to (or not to) any other particle in (or not in) specified direction: `exists [not] c bound to v in d`, where `v` as described in 2c and `d` as in 2d, e.g.:
   i. `exists 11110000 bound to V1 in N` – find a particle of type 11110000 which is bound in direction `N` to the particle identified by `V1`.
   ii. `exists not 11110000 bound to V2 in SE` – find a particle of any type but not 11110000, which is bound in direction `SE` to the particle identified by `V2`. to the particle identified by `V1`.

(f) check if the particle found in 2a or the empty place found in 2j is adjacent to any other particle or empty place: `exists [not] c adjacent`. This predicate always succeeds as every object is adjacent. E.g.:
   i. `exists 11110000 adjacent` – find a particle of type 11110000
   ii. `exists not 11110000 adjacent` – find a particle of any type but not 11110000 identified by `V1`.

(g) check if the particle found in 2a or the empty place found in 2j is adjacent to any other particle (or empty place) in the specified direction:

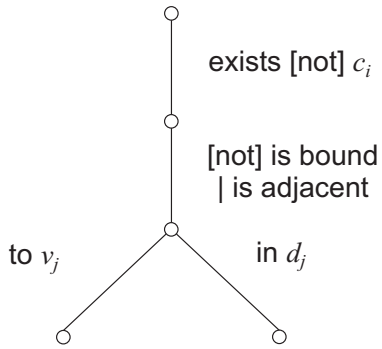

Figure 1.5: Hierarchy

`exists [not] c adjacent in d`. This predicate always succeeds as every object is adjacent in every direction. E.g.:

i. `exists 11110000 adjacent in N` – find a particle of type 11110000.

ii. `exists not 11110000 adjacent in S` – find a particle of any type but not 11110000.

(h) check if the particle found in 2a or the empty place found in 2j is adjacent to the specified particle (or empty place): `exists [not] c adjacent to v`.

The particle is adjacent to another particle if they belong to the same complex and the distance between the particles is not greater than $R\sqrt{3}$.

The empty place is adjacent to another particle (or empty place) if the distance between the objects is not greater than $R\sqrt{3}$.

(i) check if the particle found in 2a or the empty place found in 2j is adjacent to the specified particle (or empty place) in the specified direction: `exists [not] c adjacent to v in d`.

The particle is adjacent to another particle if they belong to the same complex and the position of the particle specified by `v` is one of $\boldsymbol{s_2}$ values calculated by the equation (1.13) where $\boldsymbol{s_1}$ is the checked object position and *dir* is the specified direction.

The empty place is adjacent to another particle (or empty place) if the position of the object specified by `v` is one of $\boldsymbol{s_2}$ values calculated by the equation (1.13) where $\boldsymbol{s_1}$ is the checked object position and *dir* is the specified direction.

i. `exists 11110000 adjacent to V1 in N` – find a particle of type 11110000 which is adjacent in direction `N` to the particle identified by `V1`.

ii. `exists not 11110000 adjacent to V2 in S` – find a particle of any type but not 11110000 which is adjacent in direction `S` to the particle identified by `V2`.

(j) find an empty place: `exists not xxxxxxxx adjacent to v in d`, where `v` is as described in 2c and `d` as in 2d. Since `exists not xxxxxxxx` cannot describe any particle (see also 2a) it describes an empty place. The empty place must be adjacent to another particle or an empty place, i.e. it is always related to other objects. Searching succeeds if at the described position (by `v` and `d` – see 2e) there does not exists any particle belonging to

the same complex as the particle (or the empty place) identified by `v` (note that if `d` describes one of horizontal direction – i.e. `U` or `D` – searching will always fail). E.g.

i. `exists not xxxxxxx adjacent to V1 in N` – find an empty place which is adjacent to the particle identified by `V1` in direction `N`.

(k) mark found particle or empty place in order for future use in 2c, 2d and 2e: `exists [not] c [[not] bound [to v] [in d]], mark f`. As the result, the particle (or empty place) found by the predicate is bound to the variable `f`. If the variable is already bound the to particle, the predicate is interpreted as the conjunction of the current predicate and the previous one. E.g.:

i. `exists 11110000 bound to V1 in N, mark V2` – find a particle of type 11110000 which is bound in direction `N` (note: see also 1.2.4 on page 20) to the particle identified by `V1` and store it in the `V2` variable.

ii. `exists not 11110000, mark V3` – find a particle of any type but not 11110000 and store it in the `V3` variable.

iii. `exists 11110000, mark V3` – find a particle of type 11110000, store it in the `V3` variable
`exists xxxxxxxx bound in N, mark V3` – then check if the particle identified by `V3` has any bonds in its `N` direction.

Predicate returns:

- *OK* if the particle has been found (unifying succeeded) and every particle in near space has been checked
- *OK** if the particle has been found (unifying succeeded) but not every particle in near space has been checked
- *FAIL* if particle has not been found (unifying failed)

3. Execution: `bind`, `unbind`, `move`. The predicates of the group affect the environment space. If any of the predicate fails, the whole program fails (backtracking is not performed) In case of program failure, any partial changes in space are rolled back – "everything or nothing").

(a) `move v1 to v2 in d` Moves the particle identified by `v1` to a place adjacent to the particle (or the empty place) identified by `v2` in direction `d`. At least `v1` or `v2` must identify the particle, if both identify empty places or one of them is not unified with any value, execution fails. If `v1` identifies an empty place, the predicate is changed into: `move v2 to v1 in d'` where `d'` is the opposite direction of `d` (the opposite direction of `N` is `S`, etc.). If both `v1` and `v2` identify the particles which belongs to the same complex, the predicate checks if the position of `v1` is equal to the one calculated in 3(a)ii, 3(a)iii or 3(a)iv respectively – if so, it returns *OK* otherwise, it returns *FAIL*.

The predicate executes the following algorithm:

i. calculate the position of the mass centre of collection: a complex encoding program, a complex where `v1` belongs and a complex into

where `v2` belongs (if the object identified by `v2` is not an empty place). The position of the centre of mass is calculated according to the equation:

$$\boldsymbol{s}_{CM} = \frac{\sum_{i=1}^{\mathsf{N}} \boldsymbol{s}_i m_i}{\sum_{i=1}^{\mathsf{N}} m_i} \tag{1.18}$$

where `N` denotes the number of collection constituent particles, $\boldsymbol{s}_i$ denotes particle's position and m_i denotes particle's mass.

ii. if movement direction is horizontal, it calculates the new `v1` position according to the equation 1.13 (page 6) where $\boldsymbol{s}_1$ is the `v2` position.

iii. if direction `d` is equal to U, it checks if the particle identified by `v1` has no horizontal bonds (see rule 1 on page 3). The new `v1` position is simply equal to the `v2` position.

iv. if direction `d` is equal to D, it checks, if the particle identified by `v2` has no horizontal bonds (see rule 1 on page 3). The new `v1` position is simply equal to the `v2` position.

v. update particle's `v1` position and calculate a new position of mass centre: $\boldsymbol{s}'_{CM}$ (according to the equation 1.18).

vi. if value of $d = |\boldsymbol{s}'_{CM} - \boldsymbol{s}_{CM}| > 0.25R$ update all positions of the particles from the collection (see 3(a)i) according to: $\boldsymbol{s}' = \boldsymbol{s} + \boldsymbol{s}_{CM} - \boldsymbol{s}'_{CM}$

vii. check if no particle from collection 3(a)i overlaps with any other particle (i.e. the minimal distance between any particles should not be less than $R\sqrt{3}$ – see rule 2 on page 3). The only exception is when the predicate moves in U or D direction – in that case it is possible that particles from different complexes may temporarily occupy the same place (in fact rule 2 is never violated – see 1.2.4).

If all of the above steps are successful, predicate returns *OK*; otherwise, returns *FAIL*

(b) `bind v1 to v2 in d` Moves the particle identified by `v1` to the place adjacent to the particle (or the empty place) identified by `v2` in direction `d` and then creates a bond between them in `d` direction.

(c) `unbind v1 from v2` Removes the bond between particles `v1` and `v2`.

4. Helper: `not`

Wrapper for single `structure` predicate. Returns:

- *OK* if `structure` returns *FAIL*
- *FAIL* if `structure` returns *OK* or *OK**

1.2.3 Internal representation

The programs written in the previously described language, are translate into the standard Prolog and run by a built-in Prolog interpreter. The translation rules are described later in this section. This section contains the technical details, it can be omitted during the first reading.

The list of the low-level predicates

The built-in interpreter contains a library of predicates which are able to search the fact list. The fact list is a database, which contains information about nearby particles and empty places. It is built from the following predicates:

1. `particle(id, c1, ..., c8, b1, ..., b8, px, py, vx, vy, ei)`, where:

   - `id` – particle's identity,
   - `c1` ... `c8` $\in \{0, 1\}$ – denotes particle type,
   - `b1`, ..., `b8` – contains the identities of particles bound to the current particle in the directions: N, NE, ... NW, U, D respectively (or 0 if the current particle is not bound).
   - `px`, `py` – the position of the particle
   - `vx`, `vy` – the velocity of the particle
   - `ei` – the value of the particle's internal energy

   The predicate embodies the whole information about the particle in the environment (1.1.1).

2. `empty(id, px, py)`, where:

   - `id` – the place's identity
   - `px`, `py` – the position of the place

   The predicate embodies information about the empty place. This fact is added to the list by a predicate of the same name (i.e. `empty()`)– see 6

The interpreter supports up to 16 different variables, named: `V0` to `V15`. Each variable can be unified by the identity of the particle or the empty place from the fact list, or remain unbound to any value. The variable `V0` is reserved for internal implementation, and the other ones are freely available to the program.

The built-in library contains the following predicates:

1. `hastype(V, c1, ..., c8)` – looks up the facts list and unifies `V` with the particle identity of the specified type (see also: 2a on page 10). Returns:

   (a) *OK* if the particle has been found (unifying succeeded) and every particle in the fact list has been checked

   (b) *OK** if the particle has been found (unifying succeeded) but not every particle in the fact list has been checked

   (c) *FAIL* if the particle has not been found (unifying failed)

   E.g.:

   ```
   hastype(V1, 1, 1, _, _, 0, 0, 0, 1)
   ```

2. `nhastype(V, c1, ..., c8)` – looks up the fact list and unifies `V` with the particle identity of the type other than the specified one (see also: 2a on page 10). Returns:

(a) *OK* if the particle has been found (unifying succeeded) and every particle in the fact list has been checked

(b) *OK** if the particle has been found (unifying succeeded) but not every particle in the fact list has been checked

(c) *FAIL* if the particle has not been found (unifying failed)

E.g.:

```
nhastype(V1, 1, 1, _, _, 0, 0, 0, 1)
```

3. `any(V)` – looks up facts list and unifies `V` with any fact (both the particle and the empty place). Returns:

   (a) *OK* if the fact has been found (unifying succeeded) and every fact in the list has been checked

   (b) *OK** if the fact has beene found (unifying succeeded) but not every fact in the list has been checked

   (c) *FAIL* if the fact has not been found (unifying failed)

   E.g.:

   ```
   any(V1)
   ```

4. `isbound(V1, V2, d)` – checks if `V1` is bound to `V2` in direction `d`. Both `V1` and `d` may be empty, but the predicate never unifies any variables. Returns:

   (a) *OK* if the particle `V1` is bound to `V2` in direction `d`

   (b) *FAIL* if `V1` is not unified with the particle or is not bound to a specified particle in a specified direction

   E.g.:

   `isbound(V1, _, _)` – `V1` is bound
   `isbound(V1, _, N)` – `V1` is bound in direction `N`
   `isbound(V1, V2, _)` – `V1` is bound to `V2`
   `isbound(V1, V2, N)` – `V1` is bound to `V2` in direction `N`

5. `isadjacent(V1, V2, d)` – checks if `V1` is adjacent to `V2` in direction `d`. Both `V1` and `d` may be empty, but the predicate never unifies any variables. Returns:

   (a) *OK* if the object `V1` is adjacent to `V2` in direction `d`. If `V2` is not unified, also returns *OK*.

   (b) *FAIL* if `V1` is not adjacent to `V2` in direction `d`

   E.g.:

   `isadjacent(V1, V2, _)` – `V1` is adjacent to `V2`
   `isadjacent(V1, V2, S)` – `V1` is adjacent to `V2` in direction `N`

Command	Call	Implementation
`program()`	`program()`	`program()`
`search()`	`search(V1,...,V15)`	`search(V1,...,V15)`
`action()`	`action(V1,...,V15)`	`action(V1,...,V15)`
`structure()`	expanded	expanded
`not(structure())`	`not(structure(V1,...,V15))`	`structure(V1,...,V15)`
`exists [...]`	expanded subprogram	ordered list of the following predicates: `hastype`, `nhastype`, `any isbound`, `isadjacent`, `not`, `isunique`, `unref`. More details can be found in App. 3 on page 41
`bind V1 to V2 in d`	`bind(V1, V2, d)`	built in
`unbind V1 from V2`	`unbind(V1, V2, d)`	built in
`move V1 to V2 in d`	`move(V1, V2, d)`	built in

Table 1.1: Rules of translation

6. `empty(V1, V2, d)` – finds an empty place adjacent to `V2` in `N` and, if found, unifies it with `V1`. Returns:

   (a) *OK* if the place has been found (unifying succeeded)

   (b) *FAIL* if the place has not been found (unifying failed)

   E.g.:

   `empty(V1, V2, N)` – finds a place adjacent to `V2` in direction `N`

7. `not(P)` – returns:

   - *OK* if `P` returns *FAIL*
   - *FAIL* if `P` returns *OK* or *OK**

8. `isunique(V1, V2, ..., V15)` – returns:

   - *OK* if `V1` is unified with the value different than every other argument
   - *FAIL* if `V1` is unified with the same value as any other argument

9. `unref(V)` – removes any value unified with `V`. Always returns *OK*.

Translation

Commands from the first group (1.2.2) i.e.: `program()`, `search()`, `action()`, `not(structure()))` are directly mapped into the corresponding low-level commands and command `structure()` is expanded in the place of call. The predicates: `search()`, `action()` and `structure()` operate on the whole set of variables (i.e. `V1` to `V15`). The set of variables passed to `not(structure())` is, however, restricted to the variables used in the top level structure (i.e. used in the `mark` part of the commands). Every unifying made by `not(structure())` has then a scope limited to the implementation of the predicate.

```
program() :-
  search(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  action(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15).

search(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15) :-
  // exists 000000XX, mark V1
  hastype(V1,0,0,0,0,0,0,_,_),
  isunique(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  // exists 11111111, bound to V1, in N, mark V2
  hastype(V2,1,1,1,1,1,1,1,1),
  isunique(V2,V1,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  any(V1),
  isbound(V2,V1,N),
  // exists 00000000, mark V5
  hastype(V5,0,0,0,0,0,0,0,0),
  isunique(V5,V1,V2,V3,V4,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  not(structure[1](V1,V2,_,_,V5,_,_,_,_,_,_,_,_,_,_)),
  not(structure[3](V1,V2,_,_,V5,_,_,_,_,_,_,_,_,_,_)).

structure[1](V1,V2,_,_,V5,_,_,_,_,_,_,_,_,_,_) :-
  // exists 11110000, bound to V2, in NW, mark V3
  hastype(V3,1,1,1,1,0,0,0,0),
  isunique(V3,V1,V2,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  any(V2),
  isbound(V3, V2, NW),
  // exists 11110000, bound to V3, in SW, mark V4
  hastype(V4,1,1,1,1,0,0,0,0),
  isunique(V4,V1,V2,V3,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15),
  any(V3),
  isbound(V4, V3, SW).
  not(structure[2](V1,V2,V3,V4,V5,_,_,_,_,_,_,_,_,_,_)),

structure[3](V1,V2,_,_,V5,_,_,_,_,_,_,_,_,_,_) :-
  // exists 00001111, bound to V4, in S
  hastype(V0,0,0,0,0,1,1,1,1),
  any(V4),
  isbound(V0, V4, S),
  unref(V0).

structure[2](V1,V2,V3,V4,V5,_,_,_,_,_,_,_,_,_,_) :-
  // exists 10101010
  hastype(V0,1,0,1,0,1,0,1,0),
  unref(V0).

action(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15) :-
  bind(V2,V5,SW).

// fact list
particle(101,0,0,0,0,0,0,0,0,0,0,0,102,0,0,0,0,28.80,20.36,0.4,0.2,1).
particle(105,0,0,0,0,1,1,1,1,0,0,0,102,0,0,0,0,31.80,20.36,0.4,0.2,1).
particle(100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,25.97,17.10,-0.2,-1.1,1).
particle(103,1,1,1,1,0,0,0,0,0,104,0,0,0,102,0,0,30.30,22.96,0.4,0.2,1).
particle(102,1,1,1,1,1,1,1,1,1,101,0,103,0,0,0,0,0,28.80,22.09,0.4,0.2,1).
particle(104,1,1,1,1,0,0,0,0,105,0,0,0,103,0,0,0,31.80,22.09,0.4,0.2,1).
```

Listing 1.2: Example of a low-level program

Similarly – action commands are directly implemented (obviously, with a slightly different syntax). The searching command i.e. `exists` is, however, too complex to be implemented in such a way. Each `exists` command is then expanded into the ordered list of basic predicates: `hastype`, `nhastype`, `any`, `isbound`, `isadjacent`, `not`, `isunique`, `unref`. The complete list of commands and corresponding subprograms can be found in App. 3.

Tab. 1.1 contains a summary of translation rules. An example of a translated program from List. 1.1 is presented in List. 1.2.

Validation notes

Not every program can be executed successfully. A quick glance at program examples identifies the following sources of problems with execution:

1. Empty action part of the program (the program only performs searching)
2. Wrong order of commands, e.g.: `exists xxxxxxxx bound to V2 mark V1` and then `exists xxxxxxxx mark V2`
3. The action command refers to variables not used in the search part of the program (i.e. always unbound to any value) – also due to the lack of any commands in the search part of the program

In case 1, the program is simply invalid and the interpreter should not performed execution (as the program cannot affect the environment space, its execution is only a waste of time). Case 2 is resolved by initialisation of the referenced variables by the predicate `any`, i.e. the referenced variable is always bound to the identity of the variable or the empty place. In fact, commands: `exists [not] c [not] bound to v1 [in d] [mark v2]` and `exists [not] c [not] adjacent to v1 [in d], [mark v2]` binds both `v1` and `v2` with some identities. The result of executing the referencing command is then a set of values, satisfying the relation contained in this command.

Case 3 is resolved in the similar way – the search part of the program is supplemented with the predicate `any` for each variable used in the action part which is not marked by commands in the search part of the program. Note, that even a program with a single command in the action part, and no commands in the search part is valid and is able to be successfully executed, e.g.:

```
program():-
  search(), action().

search().

action():-
    bind(V1 to V2).
```

The sample program works in the following way: unifies variables `V1` and `V2` with randomly chosen particles and tries to bind them together.

The above described solutions have hardly any impact on human-designed programs, but should significantly help in executing spontaneously emerging programs.

1.2.4 Interpreter

After the photons movements (see 1.1.2), every complex in the environment is processed. If the complex encodes one or more programmes (see 1.2.5), it is passed to the interpreter then translated into an internal representation and run. Just before execution, the interpreter creates a *fact list* from all the nearby particles, which should be visible to program. The nearby space around the program (i.e. the program header particle – see 1.2.5) is called Ω. The maximal distance between the visible particle and the program header is one of the predefined environment settings.

After database creation, the execution is performed. If both the search and action parts of the program succeeded, the interpreter tries to apply the changes into the environment (note that until that moment, the program only affects the fact list and has no influence on space). The following conditions are checked:

1. Energy balance must be positive and not less than the activation energy (1.1.1). If this is not fulfilled, the interpreter gains some energy by lowering the participating particles' internal energy.

2. After applying the changes the rule 2 must not be violated

If the above conditions are fulfilled, the interpreter applies the changes to the environment. Otherwise, the program is rotated clockwise and executed again, i.e. before another execution, every direction-related argument in predicates is changed according to rules: N → NE, NE → SE, SE→ S, S→ SW, SW → NW, NW → N. Only if all possibilities have been tried and failed, execution of program is cancelled. The summary of the execution algorithm is presented in Fig. 1.6.

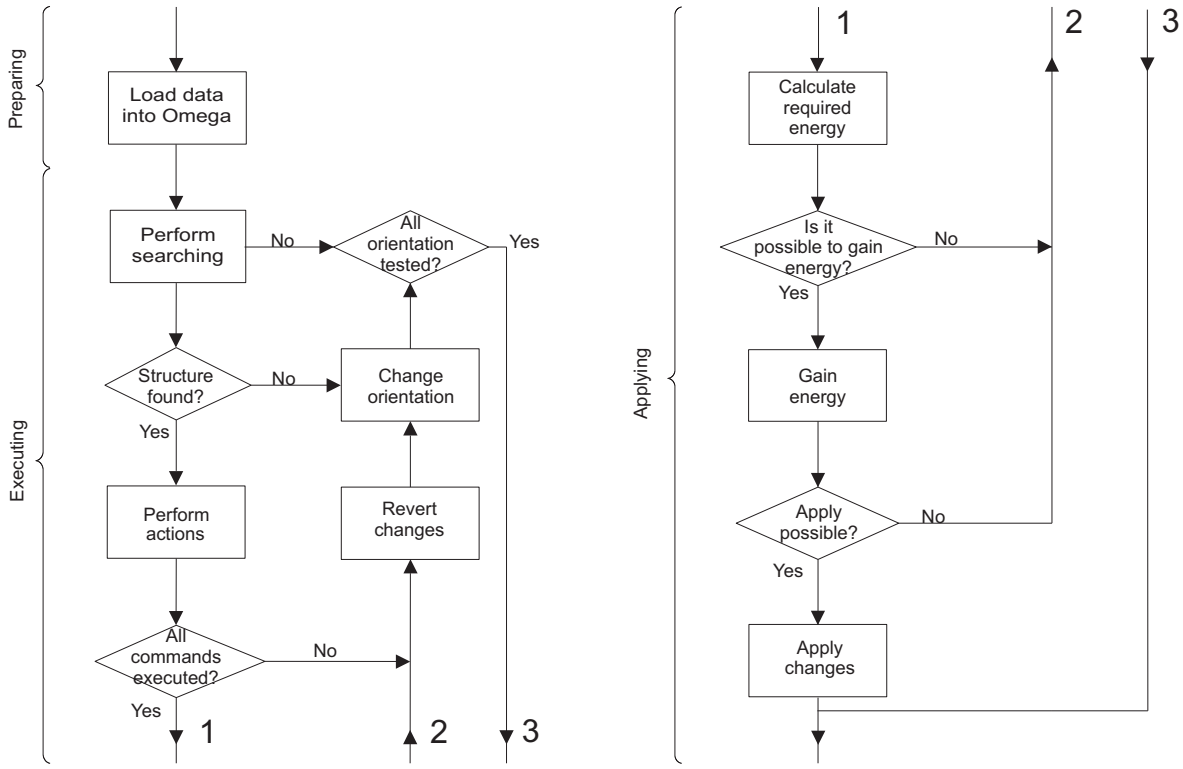

Figure 1.6: Execution algorithm

1.2.5 Encoding

As mentioned earlier, complexes may encode one or more programs. The spatial structure of the complex is mapped into the tree representation of the program (Fig. 1.4). Each node `structure` of the tree is represented by the single stack of particles – the nodes: `program`, `search`, `action` exist in each program and there is no need to encode them. Every stack encodes the list of predicates `exists`. Stack which encodes the single "positive" `structure` forms the main stack of the program and also encodes the predicates `bind`, `unbind` and/or `move`. The area of particles visible to program (Ω – see 1.2.4) is formed as a circle of a given radius from the bottom particle of the main stack. The program introduced in Fig. 1.1 is represented by the structure of the particles as shown in Fig. 1.7.

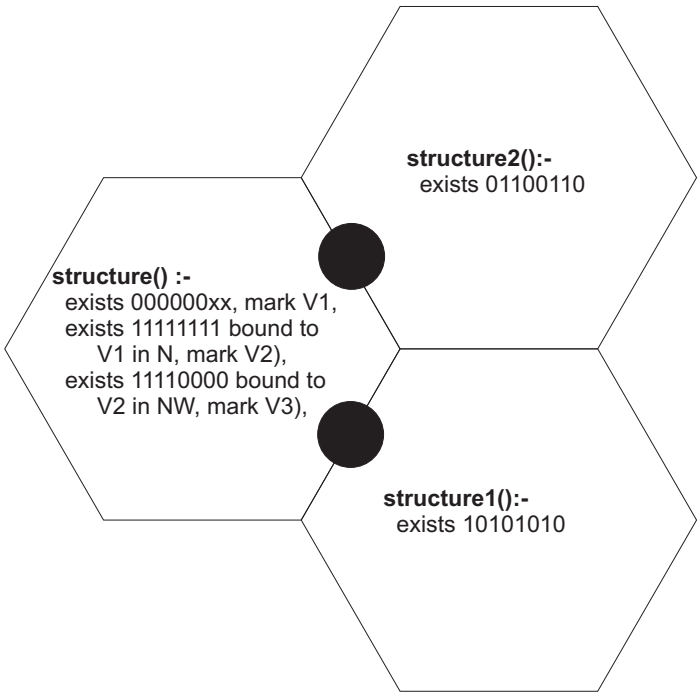

Figure 1.7: An example of a program contained in a complex. Action commands are omitted.

The predicate `not(structure())` is called from within the parent node if there exists a horizontal bond between the stack representing the parent and the stack representing the children nodes. If there exists more than one bond between the stacks, only one is considered when forming a program tree. The general rule is, that the chosen bond should be on the shortest path to the main stack. If more than one bond forms the shortest path, the decision is make randomly – Fig. 1.8.

Fig. 1.9 shows an interpretation of the stack of particles encoding "positive" `structure`, which directly defines predicate `search`. The stack also contains the definition of some action commands, embodies in `action`. The stack encoding another `structure` is similar, except for a different structure header, i.e. 1,1,0,0,×,×,×,× and no interpretation of any action command.

The type of particle encoding specification is interpreted according to Fig. 1.2.5 for `exists` command and according to Fig. 1.2.5 for action commands. Bytes taken from types of particles called "type" and "type mask" in the figure, encode first part of `exists` command which contains mask of the searched particle type. The second

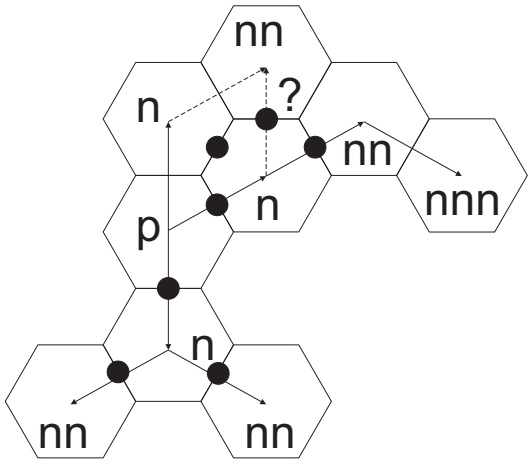

Figure 1.8: Example of encoding of complex program.

byte contains information about "x" chars in mask, i.e. if the bit at the given position is set to 0, mask contains "x" at the same position. Otherwise mask contains "1" or "0" from the first byte, e.g.: bytes 10101010, 11100111 encode mask: 101xx010. The pointers are encoded by a single particle – the first and the last 4 bits of particle type form numbers from 1 to 15 that contain the number of the program variable (from `V1` to `V15`). Direction is encoded by calculating value:

$$d = 1 + type \mod 8 \tag{1.19}$$

where *type* is the type of particle. The value d has the following meaning: $1 \leftrightarrow$ N, $2 \leftrightarrow$ NE, $3 \leftrightarrow$ SE, $4 \leftrightarrow$ S, $5 \leftrightarrow$ SW, $6 \leftrightarrow$ NW, $7 \leftrightarrow$ U, $8 \leftrightarrow$ D.

1.3 Summary

This chapter describes the concepts of the DigiHive simulation environment. The DigiHive environment takes the general concept of programs embedded in the complexes of particles from the Universum environment [3]. The DigiHive environment was, however, designed from scratch to avoid some drawbacks of the previous system.

The important application of the artificial world environments is the modelling of spontaneous evolution of the system. To perform this effectively, small changes in the structure of the constituent objects of the system should result in small changes in their behaviour. The proposed declarative language has the desired feature. The removing or changing of a part of a program may lead to small changes in the program behaviour. E.g. removing the whole predicate `structure(2)` from the program presented in List. 1.1 should affect only the effectiveness of the program, as the presence of the particle of type 10101010 would not inhibit the reaction. Changing the single bit in the "type" part of the `exists` predicate should lead to create of the same spatial structure but from the slightly different building material etc. Such a property of the program is the result of a tree-like structure of the Prolog language [5] (Fig. 1.4).

It is reasonably to assume that the random assembly of the DigiHive language predicates gives a better chance of creating useful programs, i.e. programs able to selectively change its neighbourhood.

[5]Similar idea was the reason for the choosing of the LISP representation of trees in genetic programming invented by Koza [9, 10].

top

⋮
×,×,×,×,×,×,×,×
direction
pointer 1 and 2 (1 byte)
specification
action command header (1,1,0,0,×,×,×,×)
×,×,×,×,×,×,×,×
⋮
×,×,×,×,×,×,×,×
direction
pointers (2) – 1 byte
type mask
type
specification
exists header (0,0,1,1,×,×,×,×)
×,×,×,×,×,×,×,×
⋮
×,×,×,×,×,×,×,×
program header (1,1,1,1,×,×,×,×)
×,×,×,×,×,×,×,×
⋮
×,×,×,×,×,×,×,×

bottom

Figure 1.9: A stack of particles forming a program. Bytes taken from the type of the particle.

1 – type 0 – not type	01 – adjacent 10 – bound 00 – not bound	11 – to f oth. - to any	11 – in dir. d oth. – in any dir.	1 – mark ff 0 – do not mark
1 bit	2 bits	2 bits	2 bits	1 bit

Figure 1.10: Specification of the `exists` predicate.

unused	00 – bind 10 – move 01 – unbind	1 – to f 0 – to any	1 – in dir. d 0 – in any dir.
4 bits	2 bits	1 bit	1 bit

Figure 1.11: Specification of the action predicates.

It is interesting to note, that the idea of the randomly created or the randomly evolving Prolog programs was present in the concept of Random Prolog Processing in the field of collective intelligence [18].

In the DigiHive environment, each particle type, have some unique set of attributes which defines low level (physical) rules that govern the environment dynamics. It opens various possibilities, like modelling complexes of particles acting like "food" where regrouping in them the order of particles such that the low energy bonds are replaced by the high ones provides the energy for the other energy consuming reactions [6]. Note, that the environment can be also configured with drastically simplified energy usage model, i.e. by using the internal energy only.

Another important feature is the relative spatial orientation of the program, i.e. the program rotates itself clockwise after failure. Let's consider the following example: building a hexagonal cell wall. Without rotation, it would be necessary to construct 6 different version of programs for each wall direction (N, NE, SE, SW, S, NW). Rotation allows to finish this task with just one version of program.

[6]Note, that when energy of bonds is a negative value, programs can gain energy from fission.

Chapter 2

User's manual

This chapter contains an instruction for use of the DigiHive application. Sect. 2.1 describes program installation. Sect. 2.2 contains executable file specifications and an interface guide. Sect. 2.3 contains a detailed description of state files, which are basic tools for preparing simulations. Sect. 2.4 describes experiment files, that automatise user activities during long-term experiments.

2.1 Installation

Downloable zip files can be found on the project website [12]. After downloading, the files should be simply unpacked into the preferred location. Before the program can run, it is required to install the GTK Runtime library (also available on the project website[1]).

The environment was developed in C++, using *Microsoft Visual Studio 2005 Express Edition* IDE. The program requires an MS Windows 2000, XP or Vista operating systems (tested also on 64bit versions). The memory requirements depend on running the simulation, on the biggest performed [14] the application allocates about 40MB of RAM. The executable files needs about 2.5MB of disc space, the GTK Runtimes needs additionally over 27MB of space.

The DigiHive doesn't use any hardware or system specific features, it is possible to prepare versions on other operating systems with GTK Runtime support. This possibility was successfully tested on various Linux versions, however versions other than for MS Windows aren't fully maintained.

2.1.1 File structure

After unpacking, the DigiHive environment has the following file structure:

`$DIGIHIVE_FOLDER$` – the folder where files are unpacked
- `bin` – executable files
- `dtd` – dtd files for each environment's xml
- `settings` – settings files (2.3.3)
- `save` – default folder for experiment definition files (2.4)
  - `bin` – default folder for binary state files (2.3.1)
  - `xml` – default folder for xml state files (2.3.2)

[1] latest version is available on the GIMP website: `http://www.gimp.org/~tml/gimp/win32/downloads.html`

+- `src` – environment sources

2.2 Running

The environment starts after running the `digihive.exe` in MS Windows (`./digihive` in Unix based systems). The file can be found in `$DIGIHIVE_FOLDER$/bin/` folder. When no options are given, the program runs in default GTK mode (displays the interface – 2.2.1) with no data loaded. The file specification:

```
digihive.exe [[-t file] | [[-lb | -lx | -le] file [-r]]]
             [-lgd] [-lgt] [-version] [-help]
```

- `-t file`: runs the environment in text mode (no user interface is displayed), loads and immediately runs the specified experiment file (2.4),
- `-lb file`: runs DigiHive in GTK mode, loads the specified binary file (2.3.1),
- `-lx file`: runs DigiHive in GTK mode, loads the specified xml file (2.3.2),
- `-le file`: runs DigiHive in GTK mode, loads the specified experiment file (2.4),
- `-r`: runs the simulation immediately after load,
- `-lgd`: prints a detail debug log during environment running. Note: the output, stream may contain a large amount of data. It is strongly recommended to write logs to the output log file. This option should be used with care, only while some bugs in the environment are suspected.
- `-lgt`: prints detailed information about the program running during the simulation. Note: the output stream may contain a large amount of data. This option should be used with care, only while debugging the simulation programs.
- `-version`: prints the environment version and the last build date,
- `-help`: prints a list of options.

2.2.1 Interface

User interface is displayed only when the environment is running in GTK mode (2.2). The screenshot from the main window is presented in Fig. 2.1.

Menu

Program menu contains the following options:

1. File
   (a) Open Binary: loads the environment state from a binary file (see 2.3.1). Option accessible also from the toolbar.
   (b) Open XML: loads the environment state from an XML file (see 2.3.2).
   (c) Save as Binary: saves the current state as a binary file (see 2.3.1). Option accessible also from the toolbar.
   (d) Save as XML: saves the current state as an XML file (see 2.3.2).
   (e) Save as JPEG: saves the current screen as JPEG file.

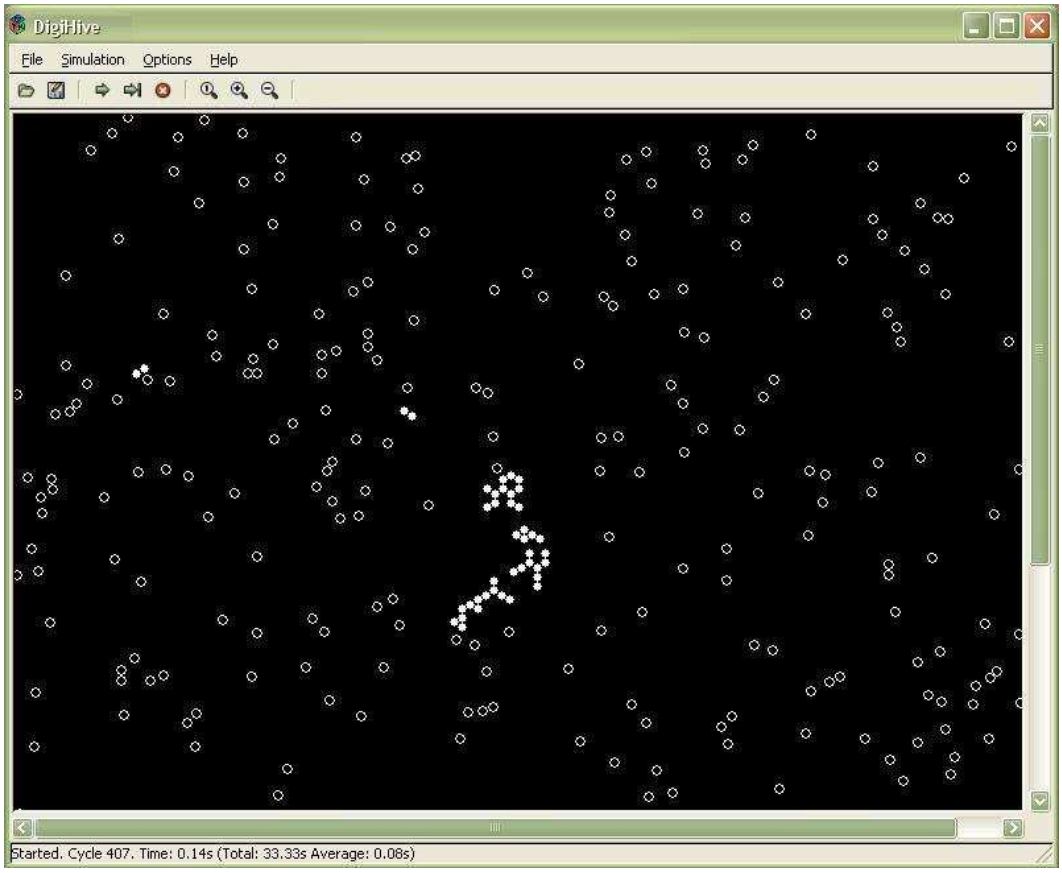

Figure 2.1: Main window. Simulation is running – state after 407 time cycles. Last step took 0.14s, the whole simulation took 33.33s, an average time step took 0.08s.

(f) Save as EPS: saves the current screen as EPS file.

(g) Open Experiment: opens the experiment definition file (see 2.4). Name of the experiment is displayed on the application's title bar.

(h) Exit: closes the environment

2. Simulation

(a) Start: starts the simulation. The environment state should be previously loaded via one of the "Open" options. The option is also accessible from the toolbar. If the state was loaded via the Experiment definition file (2.4) each step may be related with saving additional information into experiment files.

(b) Stop: stops the simulation, this option is available if the simulation is started. The option is also accessible also from the toolbar.

(c) Step: executes one step of the simulation. The environment state should be previously loaded with one of the "Open" options. The option is also accessible from the toolbar.

(d) Statistics: displays basic statistics of the environment – the same as stored during the experiment (see 2.4), with the exception of program listings.

3. Options

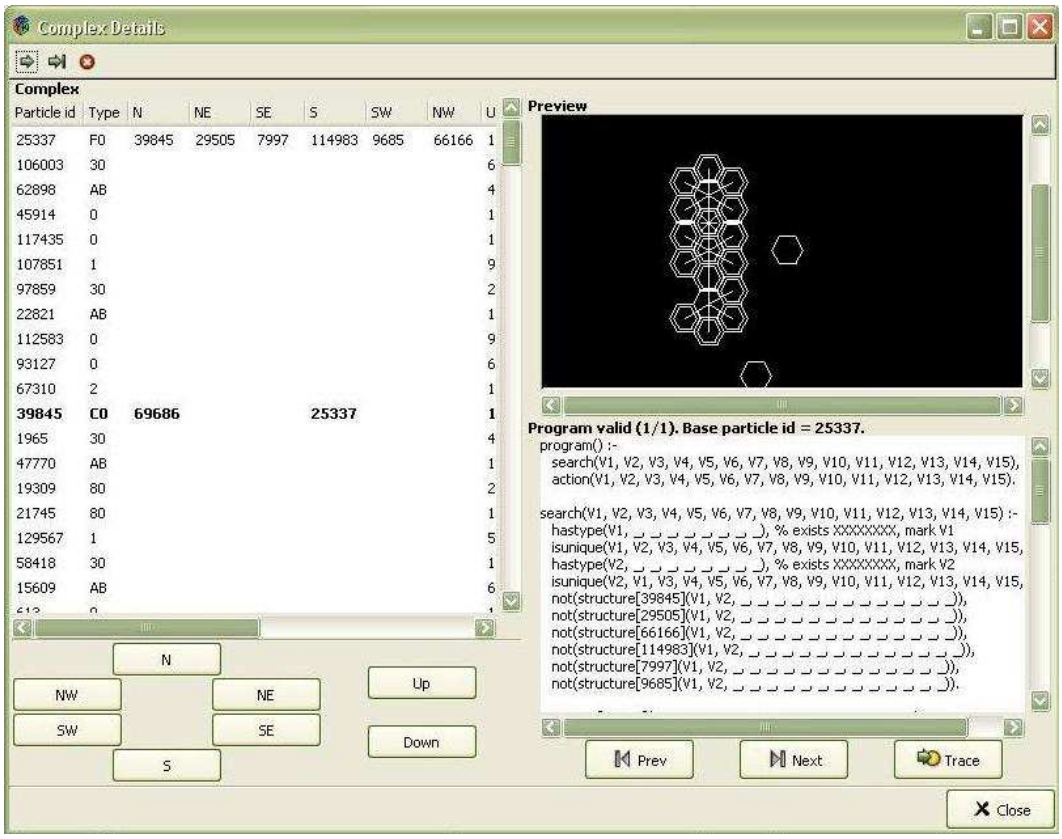

Figure 2.2: Complex details

(a) Settings: displays the dialog window that allows to manually change the environment settings – the same as contained in the setting part of the XML file (see 2.3.2).

(b) Element Table: displays the table with the chemical properties of each particle type: mass, bond mask, maximal bond count, and bond energy between any particle types.

(c) Show Photons: shows or hides the photons in the environment screen.

4. Help

(a) About: displays the version and the build date.

Complex details

Click the left mouse button on the particle showed in the main window, this displays its details as shown in Fig. 2.2. The window is divided into 3 section: Preview, Complex and Programs.

The Preview section contains a magnified view of the selected particle and its neighbourhood. The horizontal bonds are drawn using lines, particles with vertical bonds are drawn using double lines. The particle selected in the main window is marked with an asterisk.

The Complex section contains a list of all the particles forming the complex to which the selected particle belongs. The following attributes are shown: id, type, bonds in all directions, position, velocity, mass, kinetic energy, inner energy. The attributes belonging to the selected particle are printed in bold. The navigation buttons in the

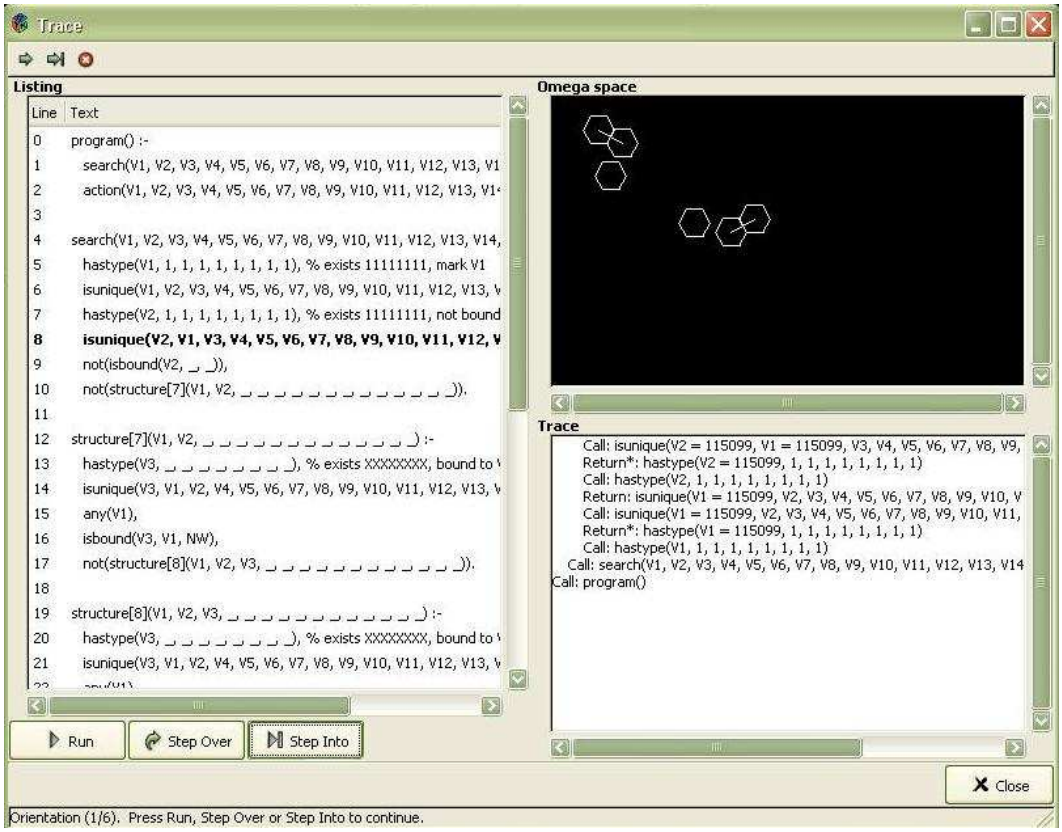

Figure 2.3: Program trace

lower part of the section change the selected particle (moves the asterisk in the Preview section in the direction according to the label on the button).

The Programs section lists all the programs contained in the complex to which the selected particle belongs. The buttons Prev and Next navigate through the list. The button Trace opens the simple debugger, described below.

Debugger

The Debugger window as shown in Fig. 2.3 consists of 3 sections: Listing, Omega Space and Trace. When the window is open, the environment stops before trying to execute the currently displayed program.

The Listing section shows a listing of the programs being debugged. The currently executed command is printed with the bold font. The lower part of the section contains the following buttons:

- Run: runs the currently displayed program, the debugger will eventually stop later during another attempt to execute the program.
- Step Into: runs exactly one command of the currently displayed program (i.e. if the command is related to the subprogram e.g. the predicate `search` is related to a set of other commands, the debugger will jump into the subprogram),
- Step Over: runs one command of the currently displayed program (i.e. if the command is related with subprogram e.g. predicate `search` is related to the set of other commands, the debugger will skip the subprogram).

The Omega Space section displays a view of the omega space at each step of the program. The section is especially helpful in an examination of the effects of the action commands.

The Trace section presents the exact trace of the program. Each predicate call and each result is logged. The section is especially helpful in an examination of the search part of the program

2.3 DigiHive state

The environment state can be saved and read in binary files (2.3.1) or *xml* definition files (2.3.2). Sets of environment settings are stored in settings files (2.3.3).

2.3.1 Binary files

Binary files (of type *.unv*) contain an exact snapshot of some DigiHive states i.e. – position and attributes of particles, complexes and photons. Contrary to the xml definition files (2.3.2), after reading the binary file it is always guaranteed that the environment state will be fully restored – for that reason, it is strongly recommended that the temporary simulation files (2.4) are stored as binary files only.

2.3.2 XML definition files

The XML[2] files are designed to prepare some initial state of the environment. It is possible to store an exact state of the environment, but it is also possible to describe some general state with partial information only (e.g. without position and velocity of every possible particle etc.). The XML file contains a list of sequentially processed commands being able to put particles, complexes and photons in the environment. The XML syntax is described in the universum.dtd file (available on the DigiHive website [12][3]). A detailed description of creating the XML and DTD files can be found in [11].

The document element is formed by the tag `<universum>`. For this tag it is required to set the following attributes: `width` means the horizontal size of the environment, `height` means the vertical size of the environment and `ccn` means the current simulation cycle. The tag is a container containing the following tags:

1. `<settings>` – environment settings,
2. `<particles>` – particle definitions,
3. `<programs>` – program definitions,
4. `<complexes>` – complex definitions,
5. `<photons>` – photon definitions,
6. `<multiply>` – multiplication of previous definitions.

The following example of a definition file describes the simulation with no particles, complexes or photons in a space of size $100 \cdot 100$

[2]Due to large size of XML file it is recommended to store them in compressed folder

[3]`http://www.digihive.pl/dtd/universum.dtd`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE universum SYSTEM
      "http://www.digihive.pl/dtd/universum.dtd">
<universum ccn="1" width="100" height="100">
   <settings/>
   <particles/>
   <programs/>
   <complexes/>
</universum>
```

Settings

The settings are loaded form an external settings file (2.3.3), according to its name in the `template` attribute (if the attribute is absent, the default settings are loaded). It is also possible to quickly change the physical settings according to the nested tags: `settings-common`, `settings-particles`, `settings-photons`, `settings-programs` which are counterparts of the tags nested in the `physics` tag of the setting file. An example of the settings:

```
<settings template="self-organization">
   <settings-particles prob-col-elastic="0.8" />
</settings>
```

Particle defition

A definition of particles is contained in the tag `<particles>`. The single particle is described by the tag `<particle>` with the following attributes:

1. `id` – the unique identifier of a particle: positive integer value between 0 and 65535, e.g. `id="1"`. If the attribute is absent, it takes some random value. An exact definition of the identifier allows future reference in the program or the multiply block,
2. `ei` – internal energy (1.1.1): positive real value, e.g. `ei="0.25"`. If absent, the the internal energy is set to 0,
3. `type` – particle type (1.1.1): positive real value, e.g. `type="15"`. This attribute is obligatory.

The particle position in the environment space can be determined by the embedded tag `<position>`. Its attributes `x` and `y` describe respectively horizontal and vertical position of particle. If the tag or its attributes are absent, the position is randomly chosen. The particle velocity is determined in the similar way, via the `velocity` tag and its attributes `x` i `y`, describing the horizontal and vertical velocity of the particle.

An example of the particle definition block:

```
<particles>
  <particle type="1" ei="0"/> <!-- particle of type 1 -->
  <particle type="4" ei="0">  <!-- particle of type 4   -->
     <position x="4" y="1"/>  <!-- position of particle -->
  </particle>
```

```
    <particle type="2" id="1">
                          <!-- particle of type 2 and id 1 -->
       <position x="2" y="4"/>  <!-- position of particle -->
       <velocity x="1" y="1"/>  <!-- velocity of particle -->
    </particle>
  </particles>
```

Program definitions

Program definitions are contained in the tag `<programs>`. It is possible to define the whole program with the tag `<program>` and the negated structure with `<not-structure>`. Each of the tags forms a unique stack of particles (separated complex), in fact it is one of the methods of creating complexes (see 2.3.2). For the tag `<program>`, the stack always begins with a particle of type 11110000, for the tag `<not-structure>` with the particle of type 11000000 (see 1.2.5).

A list of attributes of the `<program>` and `<not-structure>` tag:

1. `id` – the unique identifier of the stack (complex). As due to the next commands in the xml file, the complex may be changed, it is not guaranteed that the id will remain constant,
2. `particleid` – the unique identifier of the bottom particle of the stack. It is the counterpart of the `id` attribute in particle definitions. This identifier will remain constant as particles cannot be changed irrespective of other processed commands.

The tag `<program>` contains the following tags:

1. `<position>` – the position of the first bottom particle of the complex (the same as in the particle definition block),
2. `<velocity>` – the velocity of the complex (the same as in the particle definition block),
3. `<search>` – the searching block,
4. `<actions>` – the activity block.

The searching block consists of exactly one tag `<structure>` and at most 6 tags `<not-structure>` (the limitation is due to the assumed method of encoding the program structure). The `<not-structure>` tag, when contained in a program (not as a separated stack) may also have the following attribute:

1. `dir` – the direction in which, the stack encoding the negated structure is bound to the main stack of the program. Possible values are: `N`, `NE`, `NW`, `SW`, `S`, `SE`.

An example of the program block:

```
<programs>
  <not-structure />
  <program id="1" particleId="10">
    <position x="2" y="4"/>
    <velocity x="1" y="1"/>
```

```
    <search>
      <structure>
      ...
      </structure>
      <not-structure particleId="1" dir="NE">
      ...
      </not-structure>
    </search>
    <action/>
  </program>
</programs>
```

Both tags: `<structure>` and `<not-structure>` contain the sequence of commands encoded via the `<exists>` tags or directly by `<particle>` tags.

The `<exists>` tag encodes the predicate `exists`. The nested tags encode the following parts of the predicate:

1. `<type>` –checks the type of the particle. Possible value is a sequence of 8 chars: 0, 1 or X, e.g. `<type>00000001</type>` means: `exists 00000001`,
2. `<not-type>` –checks if the type of the particle is not as given. Possible value is a sequence of 8 chars: 0, 1 or X, e.g. `<not-type>XX000001</not-type>` means: `exists not XX000001`,
3. `<is-bound>` – checks if the particle is bound to any other particle. The tag may nest the following additional conditions:
   (a) `<in>` – sets the direction in which a bond should be checked. Possible values are: `N`, `NE`, `NW`, `SW`, `S`, `SE`, e.g. `<is-bound><in>N</in></is-bound>`,
   (b) `<to>` – the variable with the stored particle to which the current particle is bound. Possible values are: `V1`, `V2`, . . ., `V15`, e.g. `<is-bound><to>V5</to> </is-bound>`,
4. `<not-is-bound>` – checks if the particle is not bound to any other particle, the tag may nest additional conditions same as `<is-bound>`,
5. `<is-adjacent>` –checks if the particle is adjacent to any other particle, the tag may nest additional conditions same as `<is-bound>`,
6. `<mark>` – the variable in which the particle will be stored. Possible values are: `V1`, `V2`, . . ., `V15`, e.g. `<mark>V1</mark>`.

An example of the structure:

```
<structure>
  <exists>
    <type>XXXXXXXX</type>
    <is-bound/>
    <mark>V1</mark>
  </exits>
  <exists>
    <type>11110000</type>
```

```
    <not-is-bound>
      <to>V1</to>
      <in>N</in>
    </not-is-bound>
  </exits>
  <particle type="234"/>
</structure>
```

The action part encoded in the `<action>` tag consists of a sequence of `<action>` tags. The tag has the following attributes:

1. `type`: one of the following values: `bind`, `unbind`, `move` which determine the type of the command: binding particles, unbinding particles and moving particles. This attribute is obligatory.

2. `par1`: the variable with the stored first particle participating in the command. This attribute is obligatory.

3. `par2`: the variable with the stored second particle participating in the command. This attribute is obligatory if `type` is equal to `unbind` and the `dir` attribute is absent.

4. `dir`: the direction in which the command operates. This attribute is obligatory if `type` is equal to `unbind` and the `par2` attribute is absent.

An example of the action part:

```
<actions>
  <action type="bind" par1="V1" par2="V2" dir="N">
  <action type="move" par1="V1" par2="V2" dir="N">
  <particle type="099"/>
  <action type="unbind" par1="V1" par2="V2">
</actions>
```

Complex definitions

The `<complexes>` tag contains a set of complex definitions contained in a `<complex>` tag. Each `<complex>` tag contains a sequentially processed set of commands that creates bonds between the particles. It has only one optional attribute:

1. `id` – the unique identifier of the complex. As due to the next commands in the xml file, the complex may be changed, it is not guaranteed that the id will remain constant

The tag `<complex>` consists of a sequence of `<bond>` tags with the following obligatory attributes:

1. `par1` – the id of the first particle participating in the bond. The particle should be defined in the `particles` or `programs` part of the file.

2. `par2` – the id of the second particle participating in the bond. The particle should be defined in the `particles` or `programs` part of the file.

3. `dir` – the direction in which two particles are bound together

An example of the complex part:

```
<complexes>
  <complex id="123">
    <bond par1="1" par2="3" dir="N"/>
    <bond par1="1" par2="2" dir="S"/>
  </complex>
  <complex>
    <bond par1="4" par2="5" dir="U"/>
  </complex>
</complexes>
```

Photons definitions

The `<photons>` tag contains a set of tags: `<photon>`. Each tag has the following attributes:

1. `id` – the unique identifier of the photon. If absent, it is chosen randomly.
2. `alpha` – the angle at which the photon moves in the environment space. This attribute is obligatory.
3. `energy` – the energy of the photon. This attribute is obligatory.
4. `x` – the horizontal coordinate of the photon. This attribute is obligatory.
5. `y` – the vertical coordinate of the photon. This attribute is obligatory.

An example of the photon part:

```
<photons>
  <photon id="313" alpha="3.443" energy="10"
                               x="100" y="200"/>
  <photon alpha="443" energy="4" x="4" y="2"/>
</photons>
```

Multiplying definitions

The multiplying part of the file contained in the `<multiply>` tag consists of commands that allow to randomly put an exact copy of the previously defined complex or particle in the environment space. There are two tags: `<multiply-complex>` and `<multiply-particle>` that create a copy of the complex and of the particle respectively. Both tags have the same list of attributes and may nest the same optional two tags.

List of attributes:

1. `id` – the identifier of the particle (complex) to be copied
2. `multiply` – the number of additional copies to be created

List of optional nested tags:

1. `<multiply-position>` – sets boundaries of the space into which the copy will be put. The boundaries are determined by the following obligatory attributes: `minx` (minimal horizontal coordinate), `maxx` (maximal horizontal coordinate), `miny` (minimal vertical coordinate), `maxy` (maximal vertical coordinate). If the tag is absent the copy will be put anywhere in the environment space.

2. `<multiply-velocity>` – sets the range of the velocity of the created copies. The range is determined by the following obligatory attributes: `minx` (the minimal horizontal velocity), `maxx` (the maximal horizontal velocity), `miny` (the minimal vertical velocity), `maxy` (the maximal vertical velocity). If the tag is absent, velocity is set to 0.

An example of the multiplication part:

```
<multiply>
  <multiply-complex id="313" multiply="50">
     <multiply-velocity minx="-5" maxx="5" miny="0" maxy="0"/>
  </multipy-complex>
  <multiply-particle id="1024" multiply="34">
     <multiply-position minx="10" maxx="20" miny="5" maxy="6"/>
  </multiply-particle>
</multiply>
```

2.3.3 Settings file

Due to their possible large size and obvious redundancy, the environment settings are not a part of the state files, but are stored in separate ones. Different settings are distinguishes by file name. The settings folder (2.1.1) should always contain the following file: *default.xml* with the default environment's settings (tags are described below):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE settings SYSTEM "../dtd/settings.dtd">
<settings>
  <physics>
    <common time="1" eac="0.01"/>
    <particles prob-col-elastic="1" prob-emit-pho="0"
               emaxpho="1"/>
    <photons  enabled="true" prob-add-to-inner="1"
               prob-bind="0" prob-concatenate="0"
               prob-rebound="0" prob-reflect="0"
               prob-reqroup="0" prob-split="0"
               prob-unbind="0"/>
    <programs enable-program-rotation="true"
               enable-self-mutable-programs="false"
               omega="10"/>
  </physics>
  <chemistry>
    <default-element mass="1" max-bond-count="8"
        bond-mask="11111111" bond-energy="-1"/>
  </chemistry>
```

`</settings>`

The file is formed by the `<settings>` tag. The settings are divided into two parts: the physical and the chemical.

Physical settings

The physical settings are encoded in the `<physics>` tag. The physical setting are divided into four parts encoded in the following tags:

1. `<common>`, with the following attributes:
   (a) `time` – length of time step (usually 1) – see definition on p. 4
   (b) `eac` – activation energy – see definition on p. 3
2. `<particles>`, with the following attributes:
   (a) `prob-col-elastic` – the probability of an elastic collision (real value from 0 to 1) – see definition on p. 4
   (b) `prob-emit-pho` – the probability of a spontaneous photon emission – see description on p. 4
   (c) `emaxpho` – maximal photon energy after a spontaneous emission – see description on p. 4
3. `<photons>`, with the following attributes:
   (a) `enabled` – possible values are: `true` (default) or `false`. If set to `false` the photons are not calculated during the simulation – thus, the total energy of the environment decreases.
   (b) `prob-reflect` – the probability of an elastic collision between the particle and the photon – see description on p. 5
   (c) `prob-rebound` – the probability of rebounding of the particle hit by the photon from an adjoining particle – see description on p. 5
   (d) `prob-bind` – the probability of creating a horizontal bond between the hit particle and the adjoining particles – see description on p. 5
   (e) `prob-unbind` – the probability of removing the horizontal bond between the hit particle and the bound particles – see description on p. 6
   (f) `prob-concatenate` – the probability of creating a vertical bond between the hit particle and the adjoining particles – see description on p. 6
   (g) `prob-split` – the probability of removing the vertical bond between the hit particle and the bound particles – see description on p. 6
   (h) `prob-add-to-inner` – the probability of photon absorption – see description on p. 7
4. `<programs>`, with the following attributes:
   (a) `enable-program-rotation` – possible values are: `true` (default) or `false`. If set to `true` the program rotates after failure – see description on p. 20.

(b) `enable-self-mutable-programs`– possible values are: `true` (default) or `false`. If set to `true` the program may change itself during execution (i.e. may change its own complex).

Chemical settings

The chemical settings are encoded in the `<chemistry>` tag. The tag consists of one `<default-element>` tag which sets attributes common to all particle types, and a sequence of `<element>` tags which sets the particle type specific properties.

The `<default-element>` tag has the following attributes:

1. `mass` – sets the particles mass
2. `max-bond-count` – sets the maximal possible number of bonds that particle can form. Possible values are from 0 to 8.
3. `bond-mask` – binary value from 00000000 to 11111111. Each bit in mask encodes the following direction: `N`, `NE`, `SE`, `S`, `SW`, `NW`, `U`, `D`respectively. Value 1 at each position depicts the possibility of creating a bond in the specified direction. E.g. 00110011 means that particles can form bonds in directions: SE, S, U, D.
4. `bond-energy` – the energy of the bond between the particles. The real number is usually negative.

The `<element>` tag, has the same attributes with additional `type` that indicates the type of particle related to it. Definitions for specified particle type have higher priority than default ones (tag overrides default definition). Through the nested `<bond-energy>` tags it is also possible to define the energy of the bonds between the specified particles. The `<bond-energy>` tag has two attributes:

1. `to` – type of a particle to which the current one creates a bond
2. `energy` – the energy of the bond. The real number is usually negative.

2.4 Preparing experiment

The experiment definition file can be viewed as a macro which automatises some activities usually performed by the environment operator, during long-term experiments. After the file is loaded, the environment finds and loads its state according to the information encoded in the main tag of the file (see below). Each step of the simulation is connected with saving additional information in specified files.

The main tag: `<experiment>`, has the following attributes:

1. `name` – the name of the experiment, the name is displayed in the application's title bar,
2. `start-file` – the localisation of the file with an initial state of the experiment,
3. `find-last-save` – one of two values: `true` and `false`. If set to `false`, the experiment always starts with the state specified in the `start-file` attribute. If set to `true` (default), the environment tries to find the most accurate version of the simulation state in the experiment work directory (see below), and only in case of failure, loads the file specified in the `start-file` attribute.

The nested attributes allow to set the information that will be saved after each step of the simulation:

1. `<basedir>` – sets the experiment base directory where additional data will be saved. E.g. `<basedir>c:\experiment\</basedir>`
2. `<bin>` – an optional tag, connected with storing the simulation state as a binary file (2.3.1). The tag has the following attributes:
   (a) `dir` – sets the directory location (relative to the base directory) where the environment state will be saved.
   (b) `period` – sets the frequency of storing data.

   E.g.: `<bin dir="bin" period="10"/>` means that after each 10 time steps, the simulation state will be stored in a binary file. The name of the file always consists of the number describing the current time step (e.g. 0000010.unv etc.).
3. `<xml>` – same as the `<bin>` tag, but connected with an xml files (2.3.2)
4. `<gfx>` – similar to the `<bin>` tag, but connected with storing screenshots in JPG format. Besides the attributes described in the `<bin>` tag, it has two additional attributes:
   (a) `quality` – an integer number from 0 to 99, sets the quality of the graphics – greater values means better quality but also larger files.
   (b) `zoom` – an integer number from 1 to 8, which sets the size of a particle in the saved picture. Value 1 means, that each particle will be drawn as a one pixel dot, values from 2 to 4 mean, that each particle will be drawn as a circle, and values above 5 mean, that each particle will be drawn as a heaxagon.
5. `<log>` – similar to `<bin>`, but connected with storing detailed statistics during the experiment. The tag has the same attributes as `<bin>`, but also nests the set of `<detail>` tags for each type of log to be stored. The `<detail>` tag has the following attributes:
   (a) `file` – the name of the file in which the current statistics will be stored
   (b) `type` – the type of the statistics. Possible values are:
      i. `cmpl-count` – the number of complexes,
      ii. `cmpl-len-min` – the minimal size of complex,
      iii. `cmpl-len-max` – the maximal size of complex,
      iv. `cmpl-len-av` – the average size of complex,
      v. `cmpl-len-dev` – the standard deviation of the average size of complex,
      vi. `en-ecb` – overall complex bond energy,
      vii. `ek` – overall kinetic energy,
      viii. `ei` – overall internal energy,
      ix. `en-pho` – the overall energy of photons,
      x. `et` – total energy (the value should remain constant, any change indicates a serious error in the environment),

xi. `par-count` – the number of particles (the value should remain constant, any change indicates serious error in the environment),

xii. `pho-count` – the number of photons,

xiii. `prog-executed-count` – the number of programs that were executed by the environment,

xiv. `prog-completed-count` – the number of programs that were executed by the environment and succeeded,

xv. `prog-completed-lstlisting` – the full lstlistings of all programs that succeeded,

xvi. `prog-failed-lstlisting` – the full lstlistings of all programs that failed.

The logs are stored in a pseudo xml file (without the root tag) which consists of a sequence of `<entry>` tags. Each `<entry>` consists of two other tags:

(a) `cycle` – the number of cycle

(b) `value` – value to be stored. Integer/real number or program listing (i.e. `<program>` tag – see 2.3.2).

An example of the experiment definition file:

```
<?xml version="1.0"?>
<!DOCTYPE experiment PUBLIC
          "-//DIGIHIVE//DTD EXPERIMENTS 1.0//EN"
          "experiments.dtd">
<experiment name="Universal constructor"
  start-file="c:\exp\constr\cnstr.xml"
  find-last-save="true">
  <basedir>c:\exp\constr\</basedir>
  <xml dir="xml" period="1"/>
  <gfx dir="pic" period="1" quality="75" zoom="8"/>
  <work dir="work" period="1"/>
  <log dir="log" period="1">
    <detail type="cmpl-len-max" file="cmpl-len-max"/>
    <detail type="cmpl-len-min" file="cmpl-len-min"/>
    <detail type="en-ei" file="en-ei"/>
    <detail type="en-et" file="en-et"/>
    <detail type="prog-executed-count"
            file="prog-executed-count"/>
    <detail type="prog-completed-lstlisting"
            file="prog-completed-lstlisting"/>
    <detail type="pho-count" file="pho-count"/>
  </log>
</experiment>
```

Chapter 3

List of commands

This chapter contains a full list of the DigiHive environment commands. In the list, the following notation is used:

- "any" stands for _____ (eight "_")
- t stands for t1t2t3t4t5t6t7t8
- t1, . . ., t8 ∈ {0, 1, _} denote particle type
- v ∈ {V1, . . ., V15 } and vv ∈ {V0, . . ., V16 } denote variables
- d ∈ {N, NE, SE, S, SW, NW, U, D} denotes direction

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark
1	exists t bound to v in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isbound(VV,V,d).	1	10	11	11	1
2	exists t bound to v in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isbound(V0,V,d), unref(V0).	1	10	11	11	0
3	exists t bound to v, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isbound(VV,V,_).	1	10	11	00 01 10	1
4	exists t bound to v	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isbound(V0,V,_), unref(V0).	1	10	11	00 01 10	0
5	exists t bound in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), isbound(VV,_,d).	1	10	00 01 10	11	1
6	exists t bound in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), isbound(V0,_,d), unref(V0).	1	10	00 01 10	11	0
7	exists t bound, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), isbound(VV,_,_).	1	10	00 01 10	00 01 10	1
8	exists t bound	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), isbound(V0,_,_), unref(V0).	1	10	00 01 10	00 01 10	0
9	exists t not bound to v in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), not(isbound(VV,V,d)).	1	00	11	11	1
10	exists t not bound to v in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), not(isbound(V0,V,d)), unref(V0).	1	00	11	11	0

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark
11	exists t not bound to v, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), not(isbound(VV,V,_)).	1	00	11	00 01 10	1
12	exists t not bound to v	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), not(isbound(V0,V,_)), unref(V0).	1	00	11	00 01 10	0
13	exists t not bound in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), not(isbound(VV,_,d)).	1	00	00 01 10	11	1
14	exists t not bound in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), not(isbound(V0,_,d)), unref(V0).	1	00	00 01 10	11	0
15	exists t not bound, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), not(isbound(VV,_,_)).	1	00	00 01 10	00 01 10	1
16	exists t not bound	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), not(isbound(V0,_,_)), unref(V0).	1	00	00 01 10	00 01 10	0
17	exists t adjacent to v in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isadjacent(VV,V,d).	1	01	11	11	1
18	exists t adjacent to v in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isadjacent(V0,V,d), unref(V0).	1	01	11	11	0
19	exists t adjacent to v, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isadjacent(VV,V,_).	1	01	11	00 01 10	1
20	exists t adjacent to v	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isadjacent(V0,V,_), unref(V0).	1	01	11	00 01 10	0

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark
21	exists t adjacent in d, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), isadjacent(VV,_,d).	1	01	00 01 10	11	1
22	exists t adjacent in d	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), isadjacent(V0,_,d), unref(V0).	1	01	00 01 10	11	0
23	exists t, mark vv	hastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV),	1	11	00 01 10 11	00 01 10 11	1
24	exists t	hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), unref(V0).	1	11	00 01 10 11	00 01 10 11	1
25	exists not t bound to v in d, mark vv	if t $\neq$ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isbound(VV,V,d).	0	10	11	11	1
26	exists not t bound to v in d	if t $\neq$ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isbound(V0,V,d), unref(V0).	0	10	11	11	0
27	exists not t bound to v, mark vv	if t $\neq$ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isbound(VV,V,_).	0	10	11	00 01 10	1
28	exists not t bound to v	if t $\neq$ any: hastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isbound(V0,V,_), unref(V0).	0	10	11	00 01 10	0
29	exists not t bound in d, mark vv	if t $\neq$ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), isbound(VV,_,d).	0	10	00 01 10	11	1

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark
30	exists not t bound in d	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), isbound(V0,_,d), unref(V0).	0	10	00 01 10	11	0
31	exists not t bound, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), isbound(VV,_,_).	0	10	00 01 10	00 01 10	1
32	exists not t bound	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), isbound(V0,_,_), unref(V0).	0	10	00 01 10	00 01 10	0
33	exists not t not bound to v in d, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), not(isbound(VV,V,d)).	0	00	11	11	1
34	exists not t not bound to v in d	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), not(isbound(V0,V,d)), unref(V0).	0	00	11	11	0
35	exists not t not bound to v, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), not(isbound(VV,V,_)).	0	00	11	00 01 10	1
36	exists not t not bound to v	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), not(isbound(V0,V,_)), unref(V0).	0	00	11	00 01 10	0
37	exists not t not bound in d, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), not(isbound(VV,_,d)).	0	00	00 01 10	11	1

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark
38	exists not t not bound in d	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), not(isbound(V0,_,d)), unref(V0).	0	00	00 01 10	11	0
39	exists not t not bound, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), not(isbound(VV,_,_)).	0	00	00 01 10	00 01 10	1
40	exists not t not bound	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), not(isbound(V0,_,_)), unref(V0).	0	00	00 01 10	00 01 10	0

41	exists not t adjacent to v in d, mark vv	if t = any:
		empty(VV,V,d),
		isunique(VV).
		if t ≠ any:
		nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8),
		isunique(VV),
		any(V),
		isadjacent(VV,V,d).
			0	01	11	11	1

42	exists not t adjacent to v in d	if t = any:
		empty(V0,V,d),
		unref(V0).
		if t ≠ any:
		nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8),
		any(V),
		isadjacent(V0,V,d),
		unref(V0).
			0	01	11	11	0

43	exists not t adjacent to v, mark vv	if t ≠ any: nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8), isunique(VV), any(V), isadjacent(VV,V,_).	0	01	11	00 01 10	1
44	exists not t adjacent to v	if t ≠ any: nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8), any(V), isadjacent(V0,V,_), unref(V0).	0	01	11	00 01 10	0

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark

45	`exists not t adjacent in d, mark vv`	if t ≠ any:
		`nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8),`
		`isunique(VV),`
		`isadjacent(VV,_,d).`
			0	01	00
					01
					10
						11	1

46	`exists not t adjacent in d`	if t ≠ any:
		`nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8),`
		`isadjacent(V0,_,d),`
		`unref(V0).`
			0	01	00
					01
					10
						11	0

47	`exists not t, mark vv`	if t ≠ any:
		`nhastype(VV,t1,t2,t3,t4,t5,t6,t7,t8),`
		`isunique(VV),`
			0	11	00
					01
					10
					11
						00
						01
						10
						11
							1

48	`exists not t`	if t ≠ any:
		`nhastype(V0,t1,t2,t3,t4,t5,t6,t7,t8),`
		`unref(V0).`
			1	11	00
					01
					10
					11
						00
						01
						10
						11
							1

No	Name	Prolog	Specification encoding			
			unused	type	to	in
49	`bind v to vv`	`bind(V,VV,_).`	0000 ... 1111	00	1	0
50	`bind v to vv in d`	`bind(V,VV,d).`	0000 ... 1111	00	1	1
51	`move v to vv`	`move(V,VV,_).`	0000 ... 1111	10	1	0
52	`move v to vv in d`	`move(V,VV,d).`	0000 ... 1111	10	1	1
53	`unbind v from vv`	`unbind(V,VV,_).`	0000 ... 1111	01	1	0
54	`unbind v in d`	`unbind(_,_,d).`	0000 ... 1111	01	0	1
55	`unbind v from vv in d`	`unbind(V,VV,d).`	0000 ... 1111	01	1	1

Bibliography

[1] M.A. Bedau, J.S. McCaskill, N.H. Packard, S. Rasmussen, C. Adami, D.G. Green, T. Ikegami, K. Kaneko, and T.S. Ray. Open problems in artificial life. *Artificial life*, 6(4):363–376, 2000.

[2] W.F. Clocksin and C.S. Mellish. *Programming in PROLOG*. Springer, 2003.

[3] Wojciech Jędruch. *Programming environment for molecular modelling of complex systems*. Wydawnictwo Politechniki Gdańskiej, Gdańsk, Poland, 1997. (in Polish).

[4] Wojciech Jędruch. *Turbo Prolog*. Wydawnictwo Politechniki Gdańskiej, Gdańsk, Poland, 1997.

[5] Wojciech Jędruch and Rafał Sienkiewicz. Modelowanie indywiduowe. In *Aplikacje rozproszone i systemy internetowe*, Kask Book, pages 241–252. Gdańsk University of Technology, Gdańsk, Poland, 2006.

[6] Wojciech Jędruch and Rafał Sienkiewicz. Inteligencja zespołowa. In Z. Kowalczuk, W. Malina, and B. Wiszniewski, editors, *Inteligentne wydobywanie informacji w celach diagnostycznych*, volume 2 of *Automatyka i Informatyka*, pages 413–432. PWNT, Gdańsk, Poland, 2007.

[7] Wojciech Jędruch and Rafał Sienkiewicz. Modelowanie systemów samoreprodukujących się. *Metody informatyki stosowanej*, 16(3):135–147, 2008.

[8] Szymon Knitter. Badanie dynamiki systemów samoreprodukujących się w sztucznym środowisku. Master's thesis, Gdańsk University of Technology, Faculty of Electronics, Telecommunication and Informatics, Gdańsk, Poland, 2006.

[9] J.R. Koza, F.H. Bennett III, F. Bennett, D. Andre, and M. Keane. *Genetic Programming III: Automatic programming and automatic circuit synthesis*. Morgan Kaufmann, 1999.

[10] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic programming IV*. Kluwer Academic Publishers, 2003.

[11] Erik Ray. *Learning Xml*. O'Reilly, Sebastopol, CA, USA, 2001.

[12] Rafał Sienkiewicz. The digihive website. `http://www.digihive.pl/`.

[13] Rafał Sienkiewicz. A new language in environment of artificial life modeling. In Danuta Rutkowska, editor, *PD FCCS'2007: 3rd Polish and International PD Forum-Conference on Computer Science*, Lódź, Poland, 2007. (in Polish, available on-line at: `http://www.fccs.wshe.lodz.pl/fccs2007/artykuly/sienkiewicz.pdf`).

[14] Rafał Sienkiewicz. Experiments with the universal constructor in the digihive environment. In Kevin B. Korb, Marcus Randall, and Tim Hendtlass, editors, *ACAL*, volume 5865 of *Lecture Notes in Computer Science*, pages 106–115. Springer, 2009.

[15] Rafał Sienkiewicz. *The particle methods for simulation of emergent phenomena*. PhD thesis, Gdańsk University of Technology, Faculty of Electronics, Telecommunication and Informatics, Gdańsk, Poland, 2010.

[16] Rafał Sienkiewicz and Wojciech Jędruch. The universal constructor in the digihive environment. In *Advances in Artificial Life, 10th European Conference on Artificial Life, ECAL 2009, Budapest, Hungary, September 13-16, 2009*, volume 5778 of *LNCS*, pages 178–186.

[17] Rafał Sienkiewicz and Wojciech Jędruch. The universe for individual based modeling. Technical Report 11/2006/ETI, Gdańsk University of Technology, Gdańsk, Poland, 2006.

[18] Tadeusz M. Szuba. *Computational Collective Intelligence*. John Wiley And Sons, 2001.