The particle method for simulation of self-organization phenomena

PhD Thesis

AUTHOR: Rafał Sienkiewicz

Supervisor:
dr hab. inż. Wojciech Jędruch, prof. PG

Gdańsk, 2010

Contents

Acknowledgements

I would like to express my deep gratitude to my supervisor Prof. Wojciech Jędruch, for his interest, support, encouragement, and numerous suggestions during my PhD study.

This dissertation was supported by The Polish Ministry of Science and Higher Education, grant no. N 516 367636.

Extended abstract in Polish

Poniżej przedstawiono rozszerzone streszczenie rozdziałów rozprawy doktorskiej „Modelowanie procesów samoorganizacji metodą cząstek”.

1 Wprowadzenie

W ostatnich latach, badania w wielu dziedzinach nauki i inżynierii są coraz bardziej ukierunkowane na wyjaśnianie lub wykorzystanie ogólnych lub specyficznych zjawisk powstających w rezultacie oddziaływań pomiędzy wieloma jednorodnymi lub niejednorodnymi elementami systemu. Te oddziaływania mogą być neutralne, kooperatywne lub antagonistyczne i zwykle prowadzą do złożonego zachowania systemu jako całości. Tymi dziedzinami są biologia, socjologia, informatyka, automatyka, robotyka i ogólnie te dziedziny gdzie badane systemy składają się z wielu oddziałujących obiektów.

Powstaje pytanie jak modelować zachowanie systemu składających się z wielu oddziałujących komponentów. Klasyczny aparat matematyczny ma ograniczoną użyteczność ponieważ modele matematyczne są zbyt złożone lub nieadekwatne. Podstawowym narzędziem używanym w badaniach są tzw. modele indywiduowe lub agentowe lub metody cząstek. Podstawową ideą leżącą u podstaw tych podejść jest specyfikacja lokalnych reguł zachowania obiektów, włączając w to reguły opisujące interakcje z innymi obiektami i następnie symulacja komputerowa ewolucji systemu składającego się z wielu takich obiektów. Podczas symulacji powstaje złożone, globalne zachowanie modelu, jako rezultat lokalnych oddziaływań elementów.

Zaprojektowanie i implementacja środowiska symulacyjnego jest głównym celem niniejszej rozprawy.

Środowisko symulacyjne, nazwane DigiHive ukierunkowane jest na modelowanie systemów, których zachowanie się jest w istotny sposób związane z ruchem i wzajemnym oddziaływaniem swoich elementów, a w szczególności systemów złożonych, przejawiających się występowaniem w nich procesów wzrostu, samoreprodukcji, samoorganizacji i samomodyfikacji. DigiHive jest przeznaczone do modelowania i wykrywania podstawowych ogólnych własności systemów złożonych z wielu elementów, a w mniejszym stopniu na modelowanie konkretnych procesów fizycznych, chemicznych czy biologicznych.

W środowisku DigiHive zdefiniowanym w przestrzeni dwuwymiarowej poruszają się i zderzają cząsteczki. Na skutek zderzeń cząsteczki mogą losowo łączyć się w kompleksy cząsteczek lub kompleksy cząsteczek – rozpadać. Na wyższym poziomie oddziaływań, kompleksy cząsteczek mogą selektywnie oddziaływać na inne cząsteczki, łącząc je lub rozrywając wiązania. Rodzaj oddziaływania zapisany jest w strukturach kompleksów w specjalnie zdefiniowanym języku. Ta cecha otwiera nieograniczone możliwości modelowania zachowań systemów złożonych - systemów, elementy których mogą wzajemnie się modyfikować, zmieniając swoje struktury a zatem swoje funkcje.

To z kolei może prowadzić do wyłaniania się nowych funkcji i interakcji, co stanowi zasadniczą nowość zaprojektowanego środowiska. Własności środowiska stanowią model „sztucznego świata”, w którym mogą być badane za pomocą symulacji komputerowych: podstawowe mechanizmy procesów biologicznych, interakcji żywych organizmów lub agentów softwarowych.

System umożliwia symulację ewolucji różnych struktur, w szczególności samoreprodukujących się „organizmów” bez początkowo określonych kryteriów dopasowani (co jest wymaganiem w standardowych algorytmach ewolucyjnych). Kryteria są niejawnie zdefiniowane w regułach („fizyce”) działania środowiska.

Jednym z paradygmatów współczesnej sztucznej inteligencji jest działanie agentów sterujących swoim zachowaniem przy pomocy abstrakcyjnych „odcieleśnionych” manipulacji symbolami. Jest to przeciwieństwem modeli cybernetycznych rozwijanych w latach pięćdziesiątych XX wieku. W ostatnich latach pojawiają się przekonania, że w pełni inteligentne systemy powinny posiadać inteligencję „ucieleśnioną” w swojej własnej budowie, analogicznie jak w żywych organizmach. Podejmowane są próby budowy takich systemów. DigiHive może być widziane jako próba realizacji „ucieleśnienionej” inteligencji – struktura kompleksów cząsteczek jest interpretowana jako program zmieniający inne kompleksy.

Funkcje zakodowane w strukturach kompleksów są wyrażane w specjalnie zaprojektowanym języku podobnym do Prologu. Ważną cechą języka jest własność, że małe zmiany w kodzie programu zwykle powinny prowadzić do względnie małych zmian w zachowaniu programu. Taka własność języka jest kluczowa podczas symulacji spontanicznego powstawania złożonych struktur.

Stosując środowisko DigiHive zaprojektowano i przeprowadzono eksperymenty w dwu głównych kierunkach:

1. Symulowanie zjawisk emergentnych – samoorganizacja cząsteczek rozpoczynająca się od losowego stanu początkowego

2. Symulowanie uniwersalnego konstruktora – co stanowi pierwszy krok do badania dynamiki różnorodnych strategii samoreprodukcji

1.1 Tezy pracy

Przyjęto następujące tezy pracy:

- Zaprojektowane i zaimplementowane środowisko DigiHive jest oryginalnym narzędziem służącym do symulowania procesów złożonych. DigiHive umożliwia symulowanie różnych systemów samoreprodukujących się w losowym środowisku

- Język zakodowany w strukturach cząsteczek o właściwości, że niewielkie zmiany w kodzie programu prowadzą do niewielkich zmian w zachowaniu programu jest istotnym czynnikiem podczas symulacji spontanicznego wyłaniania się struktur złożonych

2 Metody cząstek

Rozdział 2 zawiera krótki przegląd metod cząstek, ze szczególnym uwzględnieniem metod abstrakcyjnych. Metody cząstek, używane podczas modelowania procesów fizycznych to m.in.: dynamika molekularna, SPH, gaz siatkowy itd. Jako metody abstrakcyjne przedstawiono: automaty komórkowe, sztuczną chemię oraz modelowanie agentowe.

3 Modelowanie systemów złożonych

Rozdział 3 zawiera przegląd zagadnień związanych z modelowaniem procesów złożonych rozpatrywanych w ramach dziedziny sztucznego życia.

3.1 Sztuczne życie

Rozdział 3.1 rozprawy zawiera przegląd zagadnień rozpatrywanych w ramach sztucznego życia. Sztuczne życie jest dziedziną symulowania procesów podobnych do występujących w systemach biologicznych, w szczególności symulowania procesów samoreprodukcji, samoorganizacji i samomodyfikacji czyli tego co składa się na podstawowe procesy życiowe i ich ewolucję. Mówiąc obrazowo do dziedziny sztucznego życia nie zalicza się np. opisanie równaniem różniczkowym procesu wymiany gazów w płucach, natomiast do tej dziedziny zaliczyć można symulację automatu, który korzystając z dostępnych elementów potrafi skonstruować swoją kopie. Do zastosowań związanych z symulacjami z dziedziny sztucznego życia szczególnie przydatne okazują się abstrakcyjne metody cząstek.

W artykule programowym pierwszej konferencji poświęconej zagadnieniom sztucznego życia w roku 1987, Langton sformułował następujące postulaty dotyczące środowiska modelowania systemów biologicznych:

- powinno składać się z populacji prostych programów lub specyfikacji;
- nie powinien w nim występować program sterujący pozostałymi programami;
- każdy program powinien zawierać opis reakcji jednostki na zaistniałe lokalne sytuacje w swoim otoczeniu włączając w to spotkania z innymi jednostkami;
- w systemie nie powinno być reguł sterujących jego globalnym zachowaniem się.

3.2 Modelowanie procesów samoreprodukcji

Badania nad systemami samoreprodukującymi się zapoczątkowane zostały w latach 50. XX wieku pionierskimi pracami von Neumanna (opisane w rozdziale 3.2.1). Miały one głównie charakter poznawczy i teoretyczny, ale ostatnio w związku z gwałtownym rozwojem nanotechnologii, dostrzega się, chociaż jeszcze bardzo odległe, możliwości realizacji fizycznych takich systemów.

W modelu von Neumanna kluczową rolę odgrywa tzw. uniwersalny konstruktor. Uniwersalny konstruktor, oznaczony przez A, jest maszyną tworzącą dowolny obiekt X na podstawie jego opisu $\phi(X)$. Latwo zauważyć, że dostarczając konstruktorowi opis jego samego tj. przyjmując że $X = A$ konstruktor utworzy kopię samego siebie. W celu uzyskania pełnego systemu samoreprodukującego, konieczne jest również powielenie opisu konstruktora $\phi(A)$. W tym celu wprowadza się maszynę B umożliwiającą skopiowanie opisu. Lącząc maszynę A i B oraz dodatkową maszynę C sterującą kolejnością wykonania uzyskujemy nową maszynę $A + B + C$. Maszyna $A + B + C$ uzupełniona o swój opis $\phi(A + B + C)$ stanowi pełny system samoreprodukujący się.

Szczegółowy przegląd zagadnień związanych z modelowaniem procesów samoreprodukcji zawiera rozdział 3.2.

3.3 Modelowanie procesów samoorganizacji

Poprzez modelowanie procesów samoorganizacji w ramach rozprawy rozumiane są symulacje polegające na obserwacji ewolucji systemu z zadanego stanu początkowego. W szczególności rozpatrywane jest zagadnienie spontanicznego powstawania systemu samoreprodukującego się oraz ewolucji systemu zawierającego początkowy zespół systemów samoreprodukujących się przy niezerowym prawdopodobieństwie zmian w modelowanych strukturach na skutek zdarzeń losowych.

4 Przegląd istniejących środowisk

W rozdziale 4 zawarto przegląd środowisk służących do modelowania procesów opisanych w rozdziale 3. Środowiska podzielono na następujące grupy:

- Klasyczne systemy agentowe: środowiska do szeroko rozumianego modelowania zgodnego z paradygmatem modelowania agentowego. Wydzielono podgrupę sztucznych ekosystemów tj. środowisk wzorowanych na modelach ekologii, gdzie pojedynczy agent reprezentuje organizm biologiczny.
- Core worlds: środowiska wywodzące się od programu Tierra, w których obserwuje się interakcje ewoluujących programów komputerowych.
- Sztuczna fizyka: środowiska niskopoziomowe, wykazujące podobieństwo do metod dynamiki molekularnej. Cechą charakterystyczną jest wbudowanie reguł uproszczonej fizyki oraz modelowanie na poziomie pojedynczych cząstek.

5 Opis środowiska

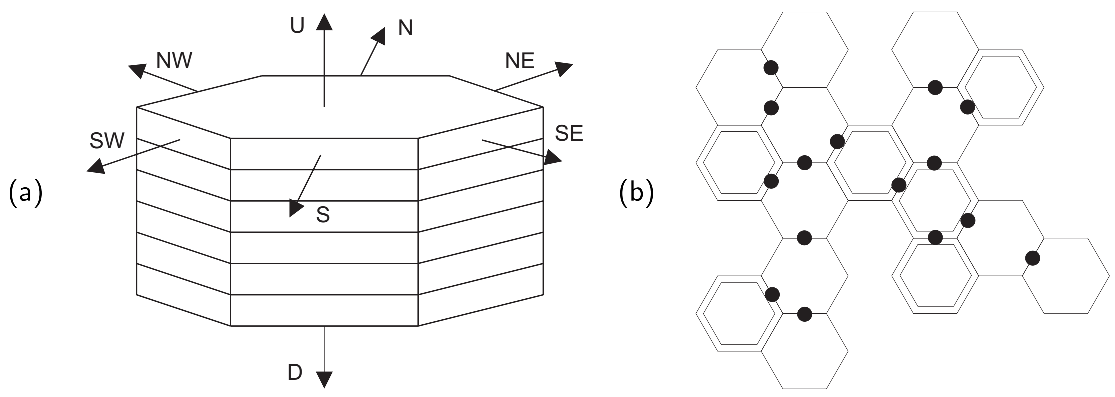
Rozdział 5 zawiera szczegółowy opis środowiska, w dodatku A opisano instrukcję instalacji oraz korzystania ze środowiska.

Opisywane środowisko wykorzystuje niektóre założenia, wcześniejszego środowiska Universum rozwijanego od połowy lat osiemdziesiątych (skrócony opis zawiera rozdział 4.3.3).

5.1 Poziom pierwszy: „termodynamika”

Środowisko zdefiniowane jest w przestrzeni dwuwymiarowej o periodycznych warunkach brzegowych. Podstawowymi obiektami środowiska są *cząsteczki* i *fotony*. Cząsteczki są obiektami trwałymi (nie są tworzone ani niszczone w trakcie symulacji) reprezentowanymi przez sześciokąty o ustalonych wymiarach. Cząsteczki występują w 256 różnych typach. Każdy typ charakteryzowany jest następującymi właściwościami: masa, energia wiązań (energia potrzebna do rozerwania wiązania) oraz energia aktywacji (minimalna energia potrzebna do zainicjowania reakcji). Poza wymienionymi cechami stałymi każda cząsteczka jest charakteryzowana przez aktualne wartości prędkości, położenia i energii wewnętrznej.

Podczas symulacji pomiędzy cząsteczkami mogą powstawać wiązania – dwie lub więcej powiązanych cząsteczek tworzy *kompleks cząsteczek*. Cząsteczki mogą łączyć się na kierunkach: góra (U) i dół (D) tworząc stos. Cząsteczki znajdujące się na spodzie stosu mogą łączyć się na kierunkach poziomych: północ (N), północny zachód (NW),



Rys. 1: Przykładowe kompleksy: (a) stos cząsteczek – widok z boku (b) kompleks utworzony przez wiązania poziome – widok z góry. Sześciokąty narysowane podwójną linią oznaczają stosy, czarne punkty oznaczają wiązania pomiędzy stosami

południowy zachód (SW), południe (S), południowy wschód (SE) i północny wschód (NE). Przykładowe wyglądy kompleksów przedstawia rysunek 1.

Wszystkie procesy w systemie są synchronizowane dyskretnym zegarem. Na początku każdego cyklu czasowego, położenie każdej cząsteczki jest zmieniane odpowiednio do jej prędkości. Jeżeli w trakcie ruchu dwie cząsteczki, nie należące do tego samego kompleksu, znajdą się w odległości mniejszej niż pewna ustalona wartość przyjmowane jest, że doszło do zderzenia. Zderzenia cząsteczek mogą być zarówno sprężyste jak i niesprężyste.

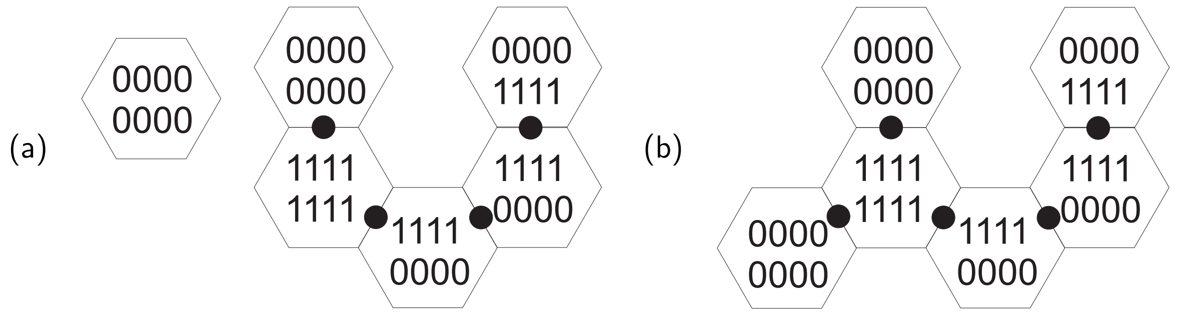
Zderzenie sprężyste modelowane jest zgodnie z zasadami mechaniki klasycznej, przy czym cząsteczki traktowane są tu jako koła o ustalonym promieniu. Podczas zderzenia sprężystego zachowana jest zarówno energia kinetyczna jak i pęd cząsteczek. Podczas zderzenia niesprężystego prędkości zderzających się cząsteczek stają się równe. Wtedy zachowany jest jedynie łączny pęd cząsteczek. Powstający przy tym deficyt energii kinetycznej zrekompensowany zostaje emisją *fotonu*.

Fotony są obiektami nietrwałymi przenoszącymi energię. Powstają w trakcie reakcji rozpraszających energię: zderzeń niesprężystych cząsteczek, tworzenia wiązań pomiędzy cząsteczkami i spontanicznej redukcji energii wewnętrznej cząsteczki. Magazynują one energię traconą przez cząsteczki, dzięki czemu energia całkowita środowiska podczas symulacji pozostaje stała. Fotony są modelowane jako punkty o zerowej masie, poruszające się z ustaloną prędkością.

Fotony mogą zderzać się z cząsteczkami. Podobnie jak w przypadku zderzeń cząsteczek, zderzenia fotonu z cząsteczką mogą być zarówno sprężyste jak i niesprężyste. Zderzenia sprężyste zmieniają jedynie kierunek fotonu, natomiast zderzenia niesprężyste prowadzą do jednej z następujących reakcji: odbicie cząsteczki uderzonej przez foton od cząsteczki sąsiedniej (znajdującej się w odległości nie większej niż pewna ustalona wartość), utworzenie wiązania pomiędzy uderzoną cząsteczką a cząsteczką sąsiednią, zerwanie wiązania pomiędzy cząsteczką trafioną a dowolną cząsteczką związaną, absorpcja fotonu (konwersja energii fotonu w energię wewnętrzną cząsteczki).

Wszystkie wymienione reakcje muszą zachowywać całkowitą energię: jeżeli foton ma zbyt małą energię do zainicjalizowani reakcji, reakcja nie zachodzi. W przypadku gdy energia fotonu jest większa niż zużyta na przeprowadzenie reakcji lub reakcja oddaje energię, po zakończeniu reakcji wygenerowany zostaje foton o energii zapewniającej stałość energii systemu.

Powyżej opisane właściwości środowiska pozwalają, poprzez ustawienie zerowego



Rys. 2: Cząsteczka i kompleks cząsteczek rozpoznawany przez program z rys. 1 (a), struktura po wykonaniu programu (b)

prawdopodobieństwa zderzeń niesprężystych, na przeprowadzenie symulacji procesów zgodnych z mechaniką klasyczną – jest to wtedy równoważne prostej dynamice molekularnej. Z kolei ustawiając niezerowe prawdopodobieństwo zderzeń niesprężystych, w środowisku pojawią się fotony, co w konsekwencji prowadzi do powstawania różnych złożonych kompleksów na skutek losowego tworzenia i zrywania wiązań.

5.2 Poziom drugi: „biochemia"

Poza omówionymi powyżej interakcjami wynikającymi ze zderzeń cząsteczek pomiędzy sobą jak i zderzeń cząsteczek z fotonami, w środowisku zachodzą oddziaływania innego rodzaju. Poszczególne kompleksy są zdolne do rozpoznawania specyficznych struktur cząsteczek znajdujących się w ich otoczeniu (okrąg o zdefiniowanym promieniu) oraz selektywnego tworzenia i rozrywania wiązań pomiędzy cząsteczkami. Funkcja realizowana przez kompleks zakodowana jest przez typy i rozmieszczenie cząsteczek w kompleksie.

Funkcje kompleksów wyrażone są w opisanym poniżej języku, o strukturze zbliżonej do języka Prolog. Występują w nim wyłącznie następujące predykaty wbudowane: `program`, `search`, `action`, `structure`, `exists`, `bind`, `unbind`, `move`, `not`. Predykaty `program`, `search`, `action`, `structure` służą do organizowania struktury programu. Natomiast funkcje rozpoznawania struktur i realizacje zmian w środowisku pełnią predykaty: `exists` (selektywne rozpoznawanie cząsteczek), `bind` (utworzenie wiązania), `unbind` (rozrywanie wiązań) oraz `move` (przesuwanie cząsteczek). Pełną listę predykatów we wszystkich dopuszczalnych wariantach zawiera dodatek C.

Przykładowy program został przedstawiony na List. 1. Jak można zauważyć programu składa się z sekwencji dwóch predykatów `search` i `action`, przy czym pierwszy z nich grupuje predykaty służące do rozpoznawania struktur, natomiast drugi, predykaty umożliwiające manipulowanie rozpoznanymi cząsteczkami.

Każdy predykat `structure` składa się z sekwencji wywołań predykatu `exists` oraz zanegowanego predykatu `structure`. Pozwala to na wykrycie określonej struktury przy warunku nie występowania innych określonych struktur. W przykładowym programie cząsteczka o typie 10101010 jest inhibitorem reakcji. Jej występowanie zablokowałoby zajście przemiany.

Za pomocą predykatu `exists` możliwe jest sprawdzenie istnienia cząsteczki o określonym typie (np. `exists(00001111 ...)`), sprawdzenie czy cząsteczka jest związana z inną cząsteczką na danym kierunku (np. `bound to V2 in N`, oraz czy jest przylegająca do innej cząsteczki (np. `adjacent to V3 in N`). Możliwe jest

```
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: Przykładowy program rozpoznający i modyfikujący strukturę przedstawioną na rys. 2

również oznaczenie cząsteczki spełniajacej określone kryteria jedną z 15 etykiet, od `V1` do `V15` (np. `exists(... mark V1)`).

Przykładowo – polecenie: `exists(11110000 bound to V2 in NW, mark V3)` oznacza: odszukaj cząsteczkę o typie 11110000, związaną na kierunku `NW` z cząsteczką identyfikowaną przez zmienną `V2` i zapamiętaj wynik w zmiennej `V3` (zmienna V3 będzie od tego momentu identyfikowała znalezioną przez polecenie cząsteczkę).

Polecenia zgrupowane w predykacie `action` wykonują sekwencje poleceń umożliwiających zmianę istniejących wiązań oraz położeń cząsteczek. Ze względu na modyfikację właściwości cząsteczek, istotna jest kolejność w jakiej występują poszczególne predykaty, odwrotnie niż w części rozpoznającej cząsteczki. W przykładowym programie polecenie `bind(V2 to V5 in SW)` oznacza: połącz cząsteczkę identyfikowaną przez zmienną `V2` z cząsteczką identyfikowaną przez zmienną `V5` na kierunku `SW`.

W odróżnieniu od Prologu, składnia języka nie determinuje jakie informacje przekazywane są do poszczególnych predykatów. Przyjęta została następująca interpretacja przekazywania parametrów: do predykatów `search` i `action` oraz głównego predykatu `structure` (w programie przykładowym: `structure(0)`) przekazywana jest pełna lista zmiennych, natomiast do pozostałych predykatów `structure` przekazywane są tylko te zmienne które zostały użyte w predykacie nadrzędnym (tj. zostały wyszczególnione w części `mark` predykatu `exists`).

Główną ideą wprowadzonego języka funkcji kompleksów było uzyskanie własności aby niewielkie zmiany w kodzie programu (tzn. w kompleksie cząsteczek kodujących program) na ogół prowadziły do małych zmian w algorytmach części rozpoznającej i wykonawczej programów. Taka własność języka jest podstawowa w przypadku modelowania spontanicznego powstawania złożonych struktur i ich dalszej ewolucji. Brak takich własności był przeszkodą m.in. w przeprowadzaniu takich symulacji w systemie Uniwersum (opis przykładowego eksperymentu w dodatku D).

W opisanym języku małe zmiany w strukturze kompleksów cząsteczek prowadzą do usunięcia lub zmiany pewnych predykatów, co zazwyczaj prowadzi do zmian zdolności rozpoznawczych programu, np. do redukcji jego precyzji.

Interpreter

Programy zakodowane w strukturach cząsteczek są wykonywane przez specjalizowany uproszczony interpreter Prologu. Bezpośrednio przed wykonaniem interpreter tworzy listę faktów o cząsteczkach widzianych przez program.

Jeżeli cześć rozpoznająca strukturę i część wykonawcza programu powiodły się oraz bilans energii przemian jest dodatni interpreter zmienia odpowiednio stan środowiska i wykonanie sie kończy. W przeciwnym wypadku wszystkie kierunki poziome zostają przesunięte o kąt 60^0 i program wykonany zostaje ponownie, aż do zakończenia się sukcesem lub wyczerpania wszystkich kierunków.

Etapy symulacji

Symulacja środowiska jest realizowana w epokach. Każda epoka składa sie z trzech faz. W pierwszej fazie realizowany jest ruch i zderzenia cząsteczek. W fazie drugiej ruch i zderzenia fotonów z cząsteczkami, zaś w fazie trzeciej wykonywane są programy kompleksów. Kompleksy wybierane są w losowej kolejności, a w czasie wykonania programu jednego kompleksu przemiany pozostałej części systemu są zatrzymane – tylko jedna funkcja może być realizowana w danej chwili czasu.

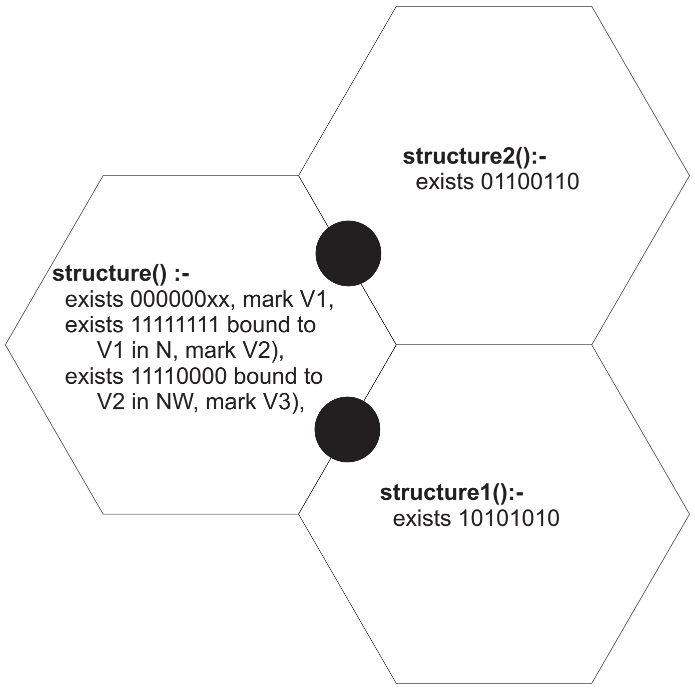
Kodowanie

Kompleks cząsteczek może kodować program. Każdy predykat `structure` jest reprezentowany przez pojedynczy stos cząsteczek w którym zakodowana jest lista predykatów `exists`. Stos kodujący predykat `structure(0)` także koduje predykaty akcji. Stosy związane z tym stosem kodują negację predykatów `structure`. Program przedstawiony na rys. 1 jest reprezentowany przez strukturę cząsteczek pokazaną na rys. 5.7.

Stosy cząsteczek kodujących predykaty są etykietowane specyficznymi typami cząsteczek, więc nie każdy stos zawiera w sobie funkcję.

5.3 Podsumowanie

Środowisko DigiHive zostało zrealizowane poprzez rozszerzenie założeń środowiska Universum (opisane w rozdziale: 4.3.3). Podstawowe różnice to:



Rys. 3: Program z rys. 1 zakodowany w kompleksie cząsteczek. Zauważmy, że predykaty `not(structure)` są kodowane przez stosy cząsteczek przylegajacych do stosu kodującego predykat `structure`(0)

- na poziomie fizyki: zastąpienie dyskretnych położeń cząsteczek położeniami w przestrzeni ciągłej,
- na poziomie oddziaływań pomiędzy cząsteczkami: zastąpienie języka asemblerowego, deklaratywnym językiem wysokiego poziomu

6 Symulacje

W rozdziale 6 opisano wyniki kilku symulacji ilustrujących możliwości środowiska i będących testem dla założeń projektu.

6.1 Proste przykłady zjawisk emergentnych

W rozdziale 6.1 przedstawiono przykłady 3 symulacji ilustrujących powstawanie złożonych struktur na skutek współdziałania zespołu prostych programów.

6.2 Koncepcja budowy systemu samoreprodukującego się

W rozdziale 6.2 opisano wyniki prac nad implementacją zmodyfikowanego modelu von Neumanna (3.2). Oryginalny system von Neumanna zrealizowany był w deterministycznym środowisku automatu komórkowego, zasadniczo różniącym się od środowiska DigiHive. W modelu von Neumanna nie występują cząsteczki stanowiące materiał, z którego budowane są struktury, stan danej komórki może ulegać dowolnej zmianie wynikającej z funkcji przejść automatu.

W opisywanym systemie zrezygnowano z budowy jednej struktury (kompleksu cząsteczek) D na rzecz oddzielnych struktur A, B, $d(A)$ oraz $d(B)$. Startując od początkowej populacji złożonej z co najmniej pojedynczych egzemplarzy poszczególnych struktur, powinniśmy zaobserwować ich powielanie. Na skutek działania struktury B powieleniu powinny ulec opisy tj. $d(A)$ oraz $d(B)$, natomiast struktura A powinna doprowadzić to utworzenia kopii struktur A i B. W tak zaprojektowanym eksperymencie nie ma znaczenia kolejność powielania, nie ma więc potrzeby budowy struktury synchronizującej C. W ramach rozprawy opisano pierwszy etap, tj. implementację uniwersalnego konstruktora A i jego replikację.

Uniwersalny konstruktor

Przez uniwersalny konstruktor będzie rozumiana struktura A (kompleks cząsteczek) zdolna do budowy dowolnej struktury X na podstawie jej opisu $d(X)$. Dopuszcza się wytworzenie przez A na podstawie opisu $d(X')$ struktury $X' \neq X$, która następnie pod wpływem środowiska jest zdolna do przekształcenia się w X.

Opisywany poniżej konstruktor wykonuje następujące zadania:

1. Wyszukanie łańcucha informacyjnego – $d(X)$. Lańcuch informacyjny może być traktowany jako program napisany w języku, zawierającym następujące polecenia: `PUT` (dodaje cząsteczkę do budowanej struktury), `SPLIT` (rozpoczęcie budowy nowego stosu, połączonego na określonym kierunku z wcześniej budowanym stosem), `NEW` (rozpoczęcie budowy nowego stosu, nie połączonego z wcześniej budowanym stosem) i `END` (zakończenie budowy). Lańcuchem informacyjnym jest stos cząsteczek o ustalonym typie. Przykładowo następujący program, tworzący stos dwóch cząsteczek typu 01010101:

   ```
   PUT(01010101)
   PUT(01010101)
   END
   ```

   jest kodowany przez stos cząsteczek:

   ```
   11111111
   01010101
   00000001
   01010101
   00000001
   ```

2. Podłączenie się do łańcucha informacyjnego i rozpoczęcie konstrukcji struktury X.

3. Sekwencyjne przetwarzanie łańcucha informacyjnego:

   (a) Jeżeli napotkano cząsteczkę kodującą polecenie `PUT`, – wyszukanie cząsteczki o określonym typie (argument polecenia) i podłączenie do budowanej struktury X.

   (b) Jeżeli napotkano cząsteczkę kodującą polecenie `SPLIT` – rozłączenie budowanej struktury X na dwa stosy cząsteczek i utworzenie połączenia na zadanym kierunku (argument polecenia) pomiędzy nimi.

   (c) Jeżeli napotkano cząsteczkę kodującą polecenie `NEW` – odłączenie budowanej struktury X od konstruktora i rozpoczęcie budowy nowego stosu na podstawie typu cząsteczki zakodowanej w łańcuchu informacyjnym (argument polecenia).

4. Po przetworzeniu całego łańcucha informacyjnego - odłączenie się od łańcucha i odłączenie budowanej struktury X od siebie (interpretowane jako polecenie `END`).

Spośród wymienionych zadań poważną trudność stanowi zakodowanie poleceń związanych z przetwarzaniem łańcucha informacyjnego. W specyfikacji języka brakuje poleceń umożliwiających wykonywanie instrukcji warunkowych oraz możliwości porównywania typów cząsteczek, które znacząco ułatwiły by zakodowanie wymienionych zadań. Rozwiązaniem problemu jest utworzenie programów modyfikowanych podczas pracy konstruktora (nazywanych dalej *szablonami*): $T1$ „odszukaj cząsteczkę o typie <T> i umieść ją na szczycie budowanego stosu cząsteczek" (gdzie <T> jest dowolnym, podmienianym typem cząsteczki), $T2$ „podziel budowany stos cząsteczek na dwa stosy i utwórz połączenie na kierunku <D> pomiędzy nimi" (gdzie <D> jest dowolnym, podmienianym numerem kierunku).

Szablony zostały zakodowane jako stosy cząsteczek tworzące nieaktywne programy. Każdy szablon zawiera pewien charakterystyczny znacznik (sekwencję cząsteczek o określonym typie). Podmiana cząsteczek pomiędzy znacznikami aktywuje program zdolny do wykonania działania na cząsteczce o określonym typie bądź na określonym kierunku.

Uniwersalny konstruktor został utworzony jako kompleks składający się z 10 działających niezależnie programów, uzupełnionych o cząsteczki specjalne służące m.in. do synchronizacji pracy poszczególnych programów:

P1 Program odnajduje i przyłącza konstruktora do łańcucha informacyjnego. Program przyłącza tzw. wskaźnik czyli stos 3 cząsteczek o określonym typie do łańcucha informacyjnego. Wskaźnik określa miejsce łańcucha, które w danym momencie jest dekodowane. Ponieważ łańcuch informacyjny zawsze zaczyna się od kodowania typu cząsteczki (tj. polecenia `PUT`), program następnie odłącza cząsteczkę kodującą typ od łańcucha informacyjnego i przyłącza w określone miejsce programu P3. Aktywuje program P3.

P2 Program sprawdza czy aktualnie dekodowane jest rozpoznawanie określonego typu cząsteczki (polecenie `PUT`). Jeżeli tak, to odłącza cząsteczkę od łańcucha informacyjnego i przyłącza w określone miejsce programu P4. Aktywuje program P4.

P3 Szablon $T1$ aktywowany na skutek działania programu P1 lub P10. Wyszukuje cząsteczkę o określonym typie (argument pierwszego polecenia `PUT` lub `NEW`) i rozpoczyna tworzenie stosu. Po zakończeniu działania odłącza od siebie fragment przyłączony przez P1 (P10) i przyłącza go z powrotem do łańcucha informacyjnego (deaktywuje się). Poprzez zmianę połączeń pomiędzy cząsteczkami kodującymi stan, aktywuje program P7.

P4 Szablon $T1$ aktywowany na skutek działania programu P2. Wyszukuje cząsteczkę o określonym typie (argument polecenia `PUT`) i dołącza do budowanego stosu. Po zakończeniu działania odłącza od siebie fragment przyłączony przez P2 i przyłącza go z powrotem do łańcucha informacyjnego (deaktywuje się). Poprzez zmianę połączeń pomiędzy cząsteczkami kodującymi stan, aktywuje program P7.

P5 Program sprawdza czy aktualnie dekodowane jest rozpoznawanie określonego kierunku (polecenie `SPLIT`). Jeżeli tak, to odłącza cząsteczkę od łańcucha informacyjnego (argument polecenia `SPLIT`) i przyłącza w określone miejsce programu P6. Aktywuje program P6.

P6 Szablon $T2$ aktywowany na skutek działania programu P5. Odłącza cząsteczkę znajdującą się na wierzchołku budowanego stosu, przesuwa resztę stosu na określonym kierunku (argument polecenia `SPLIT`) i tworzy połączenie pomiędzy cząsteczką a resztą stosu. Cząsteczka ta stanowić będzie nową podstawę dla budowanej konstrukcji. Po zakończeniu działania odłącza od siebie fragment przyłączony przez P4 i przyłącza go z powrotem do łańcucha informacyjnego (deaktywuje się). Poprzez zmianę połączeń pomiędzy cząsteczkami kodującymi stan, aktywuje program P7.

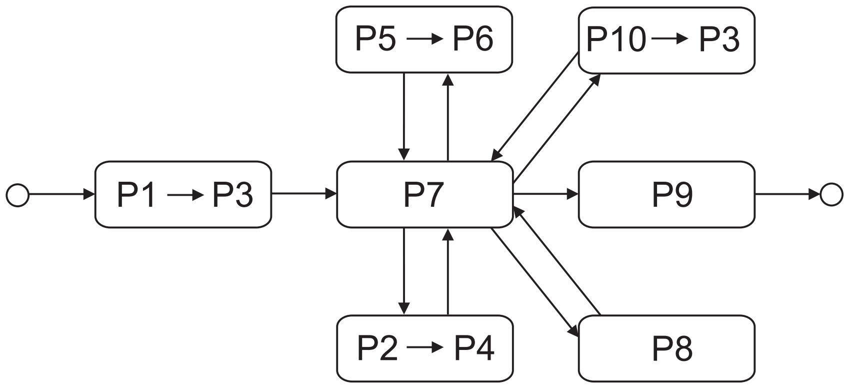
P7 Program przesuwa wskaźnik wzdłuż łańcucha informacyjnego, jeżeli zakończono dekodowanie aktualnej cząsteczki. W zależności od aktualnej cząsteczki aktywuje program P2, P5, P8, P9 lub P10.

P8 Program przesuwa wskaźnik wzdłuż łańcucha informacyjnego, jeżeli w łańcuchu kodującym znajduje się niepoprawna cząsteczka określająca polecenie dla konstruktora (tj. niekodująca jednego z poleceń: `PUT`, `SPLIT`, `NEW`).

P9 Program kończący translację jeżeli osiągnięto koniec łańcucha kodującego (polecenie `END`). Program odłącza budowaną strukturę od konstruktora oraz odłącza konstruktor od łańcucha kodującego.

P10 Program sprawdza czy aktualnie dekodowane jest rozpoczęcie budowy nowej struktury (polecenie `NEW`). Jeżeli tak, to odłącza budowaną strukturę od łańcucha kodującego następnie odłącza cząsteczkę kodującą typ od łańcucha informacyjnego i przyłącza w określone miejsce programu P3. Aktywuje program P3.

Diagram przejść stanów (jako stan konstruktora rozumiany jest aktualnie wykonywany program) przedstawiono na rysunku 4.



Rys. 4: Diagram przejść stanów dla uniwersalnego konstruktora. Zapis „P1→P3” należy rozumieć jako: „wykonywany jest program P1 w rezultacie czego, w kolejnym cyklu wykonywany jest szablon P3” itp.

Przykładowe symulacje

W rozdziale 6.2.2 opisano wyniki przykładowych symulacji z wykorzystaniem uniwersalnego konstruktora: utworzenie kompleksu o kształcie rombu oraz utworzenie kompleksu o kształcie “płatka”. W drugim eksperymencie konstruktor utworzył zespół programów, który następnie zbudował końcową strukturę. Dokładny opis programów składowych zawiera rozdział 6.1.1.

Replikacja konstruktora

W sytuacji gdy konstruktor A działa na opisie $d(A)$, czyli buduje swoją kopię, najważniejszym problemem do rozwiązania jest wpływ aktywności częściowo zbudowanego konstruktora. Struktura ta nie powinna rozpocząć działania zanim nie zostanie całkowicie ukończona, inaczej symulacja może przebiegać w sposób nieprzewidywalny (np. niedokończony konstruktor zacznie budować swoją kopię). Z drugiej strony konstruktor budujący strukturę nie może rozpoznać tej struktury jako części samego siebie (jak opisano poniżej, pewna struktura połączeń pomiędzy pewnymi charakterystycznymi cząsteczkami konstruktora służy do synchronizacji działania poszczególnych jego funkcji). Problem można rozwiązać na dwa sposoby:

1. Poprzez budowę komplementarnego konstruktora A' tj. struktury która ma inne elementy charakterystyczne oraz wymaga innej składni dla opisu budowanej struktury $d'(X)$. System samoreprodukujący składałby się zatem z pary konstruktorów A, A' oraz opisów $d(A')$ oraz $d'(A)$. Konstruktor A tworzyłby zawsze nowy egzemplarz A' natomiast A' tworzyłby A.

2. Poprzez tworzenie pewnej nieaktywnej (nie działającej) struktury E, która po odłączeniu od konstruktora aktywuje samą siebie tj. przekształci się do struktury A. Zauważmy, że w tym podejściu struktura A nie musi być w pełni uniwersalnym konstruktorem – tj. nie musi bezpośrednio odtwarzać pełnej struktury swoich połączeń. Wystarczająca jest możliwość utworzenia pewnej struktury złożonej z połączonych stosów cząsteczek.

W rozdziale 6.2.3 opisano wyniki eksperymentu polegającego na wykonaniu kopii konstruktora zgodnie z założeniami z p. 2. Eksperyment zakończył się powodzeniem, jednakże problemem okazała się niska wydajność procesu (symulacja trwała ok. 3 miesięcy na komputerze klasy PC z procesorem Quad 6600).

6.3 Wnioski i dalsze kierunki badań

W ramach pracy zaprojektowano i zaimplementowano abstrakcyjne środowisko symulacyjne DigiHive spełniające założenia podane przez Langtona. Wyniki przedstawionych symulacji wskazują, że środowisko DigiHive jest dobrym i efektywnych narzędziem służącym do modelowania procesów złożonych.

W rozdziale 6.3 opisano proponowane rozszerzenie środowiska w celu zwiększeniu efektywności symulacji, w szczególności procesu samoreprodukcji. Największym problemem okazał się brak możliwości bezpośredniego porównywania typów cząsteczek, co prowadzi do nadmiernej komplikacji konstruktora, poprzez konieczność tworzenia programów pośrednich, aktywowanych przez inne programy. Planowane jest rozszerzenie listy poleceń poprzez wprowadzenie polecenia `exists like` umożliwiającego odszukanie cząsteczki o typie takim samym lub podobnym (z zadaną dokładnością) jak cząsteczka wcześniej odszukana. Polecenie takie jest nadmiarowe (daje się je zaprogramować za pomocą istniejących predykatów), ale jego obecność pozwoli na znaczne uproszczenia konstruktora a zatem zwiększenia efektywności.

Analiza wyników symulacji, wykazała również konieczność usunięcia ograniczenia polegającego na możliwości tworzenia wiązań poziomych wyłącznie na poziomie cząsteczki znajdującej się na spodzie stosu. Zmiana taka umożliwi równoległe przetwarzanie jednego łańcucha informacyjnego przez kilka konstruktorów lub przez konstruktory i programy tworzące kopię łańcucha.

Planowane są również zmiany w schemacie kodowania poleceń w celu skrócenia zapisu pojedynczych predykatów (rozbicie polecenia `exists` na sekwencję atomowych poleceń) oraz zmniejszenie prawdopodobieństwa zmian w łańcuchu cząsteczek kodujących pojedynczy predykat na skutek losowych zdarzeń. Oczekuje się, że opisane zmiany powinny ułatwić modelowanie procesów samoorganizacji poprzez ograniczenie efektów zdarzeń losowych (zdarzenie losowe powinno prowadzić do niewielkich zmian w kodzie programu).

7 Podsumowanie

W pracy przedstawiono założenia środowiska DigiHive przeznaczonego do modelowania procesów złożonych. Przedstawiono wyniki symulacji ilustrujących możliwości środowiska oraz opracowano założenia eksperymentu symulującego proces samoreprodukcji.

List of abbreviations

General

- ABM: Agent based model
- AC: Artificial chemistry
- ALife: Artificial life
- CA: Cellular automata
- CFD: Computational fluid dynamics
- CSM: Computational solid mechanics
- DPD: Dissipative particle dynamics
- FSM: Finite state automata
- LBM: Lattice Boltzmann method
- LGCA: Lattice gas cellular automata
- MD: Molecular dynamics
- PDE: Partial differential equations
- SPH: Smooothed particle hydrodynamics

DigiHive **physics**

- E_{ac}: Activation energy
- E_i: Internal energy
- E_{pb}: Bond energy
- E_{ph}: Photon energy
- m: Mass of the particle
- M: Mass of the complex of particles
- N: North
- NE: Northeast
- NW: Northwest

- S: South
- SE: Southeast
- SW: Southwest

Chapter 1

Introduction

During recent years, researches in many fields of science and engineering are more and more directed towards explaining or utilizing general or specific phenomenas arising as a result of interactions among many homogeneous or heterogeneous objects of systems. These interactions can be neutral, cooperative or antagonistic and usually lead to complex behaviour of the system as a whole. These fields of research include biology, sociology, computer science and engineering, automatic control and robotics, and generally such fields where the investigated systems consist of many interacting objects.

The question arises how to model the behavior of a system consisting of many interacting objects? The classical mathematical framework, has only limited usefulness because the mathematical models are too complex to be useful or are simply inadequate. The basic tools used in the research are so called individual-based modelling or agent based-modelling or particle methods. The basic idea underlying behind all these approaches is to specify the local rules of behavior of the objects including rules describing the interactions with other objects, and then to simulate, with the help of a computer, the evolution of a system consisting of many such objects. During the simulation, there arises the complex global behavior of a model as the result of a local interactions of its objects.

The aim of the work was to design, implement, and use, in a number of experiments, an abstract software environment (an artificial world) suitable for modelling systems consisting of many moving and interacting objects distributed in space. The environment, named DigiHive is directed towards modeling of complex systems manifested by processes of self-organization, self-reproduction and self modifications. The environment is mainly aimed at modelling and discovering the basic general properties of complex systems consisting of many elements and less at modelling of specific phenomena of real systems.

The environment is a two-dimensional space in which particles and complexes of particles exist, move, and collide. The collisions may result in creating or removing bonds between particles. On a higher level of interactions, complexes of particles may selectively modify the structures of other particles and complexes of particles according to functions encoded in them. The latter property opens unlimited possibilities of modelling the behavior of complex systems – systems the elements of which can mutually modify each other changing their structures and in this way changing their functions. This in turn can lead to the emergence of new functions and interactions which is a crucially novel property of the designed environment. The properties of the environment constitute an artificial world model in which many basic ideas of biological processes, living organisms or software agents' interactions may be investigated by computer simulations.

The system allows to simulate the evolution of various structures, especially self-reproducing "organisms" but without the initially defined fitness criteria (which is the basic need in standard evolutionary algorithms). The criteria are implicitly defined in the rules ("physics") of the environment.

One of the central ideas of contemporary artificial intelligence methods is the existence of agents controlling their own behavior by means of abstract "disembodied" symbol manipulation. This is contrary to the previous cybernetics models developed in the fifties of the twentieth century [90, p. 60]. During last years the idea is emerging that the fully intelligent system should have the intelligence "emobodied" in the hardware like in the living organisms. Attempts are made to build such system [47, 99]. The DigiHive environment can be partly seen as an attempt of realisation of "embodiment" – the structure of complexes of particles is interpreted as a program affecting other complexes.

The functions embedded in the structures of complexes are expressed in a specially designed Prolog-like language. The key feature of the language is the property that small changes in the code of a program should usually lead to relatively small changes in the program's behaviour. Such a property of the language is crucial while using the system to simulate the evolutionary, spontaneous development of complex structures.

Using the DigiHive environment, two main groups of experiments were designed and performed:

1. Various manifestations of emergent phenomena – self-organization of particles starting from initial state.

2. An universal constructor – being the first step to simulate various self-reproduction strategy and its dynamics.

1.1 Thesis

The designed and implemented DigiHive simulation environment is a novel tool for simulating the various processes of self-organization, especially it allows to simulate the various self-replicating structures working in the random environment.

The designed and implemented language encoded in the structures of particles having the property that small changes in the code of a program usually lead to relatively small changes in the program behavior is an important factor for simulation and spontaneous development of complex structures.

1.2 Organization of the dissertation

Chapter 2 contains an abbreviated review of particle methods, with particular emphasis on the abstract ones. Chapter 3 comprises of selected issues related to simulation of complex behaviour, related to an artificial life, self-reproduction and self-organization (in terms of artificial life) researches. Chapter 4 provides a review of selected popular software environments suitable for the simulation of the processes described in Chapter 3. Chapter 5 contains a detailed description of the DigiHive environment – its physics, and the embedded declarative language. Chapter 6 provides a review of the first simulations performed in the DigiHive environment. This chapter describes the simple emergent phenomena simulated in the environment and the results of experiments with the implementation of the universal constructor. At the end of the chapter, results are discussed, and suggestions for further work are included. Chapter 7 contains a summary of the dissertation. Appendix A comprises of a manual for the environment: installation, user interface and input/output files description. The sample input file is presented in Appendix B. In Appendix C, a full review of the DigiHive commands is presented. Appendix D describes the self-organization experiment in the Universum environment, whose results serve as a base to create the DigiHive environment.

Chapter 2

Particle methods

The term *particle methods* describes a group of simulation of physical systems in which global behaviour emerges as the result of a set of finite number of discrete particles to represent the state of a system. Each particle can represent one discrete physical object (from single atoms to astronomical objects)[1] or a group of discrete objects. Every single particle can be described by a set of variables related to specific problems (e.g. mass, momentum, energy, position, charge, vorticity etc.). The formal definition of a particle can be found in: [46]. Applications of particle methods ranges from physical simulations to abstract issues (see chapter 3).

Sect. 2.1 presents a short review of particle methods used in simulations in the field of natural science. Next sections focus on abstract methods: cellular automata (2.2), artificial chemistry (2.3) and agent based modelling (2.4).

2.1 Physical particle methods

In the recent years, tens of particle methods have been developed. Particle methods are currently widely applied to various areas of simulation especially, computational fluid dynamics (CFD) and computational solid mechanics (CSM), etc. The sample particle methods used in physical simulation are:

1. Molecular dynamics (MD). Molecular dynamics is the most popular particle method in scientific and engineering applications. A comprehensive review of the method can be found in plenty of works e.g. [6, 150] etc. It originates in the fifties (e.g. [4]) as a tool for calculating some properties of physical systems, like gases, liquids or solids. The first works, focused on simulation of a set of particles moving freely in space and colliding elastically. The next works [148] introduced the concept of a force field forming modern molecular dynamics.

   MD has been used for various simulation, e.g. simulation of: the viscosity and heat transfer in liquids [28], formation of defects in crystals [27], properties of macromolecules in biological systems [13] etc.

[1]The main concept of particle methods reflects the following Feynmann's sentence [50, chapt. 1.2]: "if, in some cataclysm, all of scientific knowledge were to be destroyed, and only one sentence passed on to the next generations of creatures, what statement would contain the most information in the fewest words? I believe it is the atomic hypothesis (or the atomic fact, or whatever you wish to call it) that all things are made of atoms – little particles that move around in perpetual motion, attracting each other when they are a little distance apart, but repelling upon being squeezed into one another".

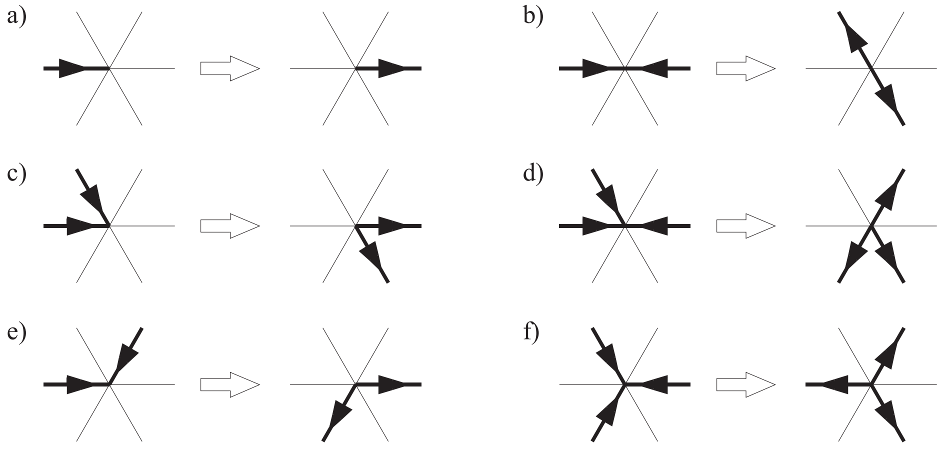


Figure 2.1: Example of resolving particle collisions in LGCA model on nodes of a hexagonal grid.

2. Smoothed particle hydrodynamics (SPH). SPH [109, 127] was developed in 1977 by Lucy [114] and simultaneously by Gingold and Monaghan [62] in order to model astrophysical phenomena. In SPH, a set of discrete astrophysical particles is consolidated into a quasicontinuum [108, p. 25]. The movement of simulated particles is similar to the behaviour of a liquid, thus it is modelled like a fluid governed by the equations of classical hydrodynamics. SPH is now extensively use for astrophysical simulations (e.g. [75]) and CFD/CSM problems (e.g. [126]).

   Some extensions of SPH has been recently proposed, as an example the reproduced kernel particle method has been developed in 1995 [110]. A detailed description of this method and its applications can be found in: [108].

3. Lattice Gas Cellular Automata (LGCA): This method was developed as simplification of molecular dynamics [150, 200]: discretisation of position (molecules are placed on nodes of a hexagonal grid) and time, reduction of impact to the nearest neighbourhood and simplification of rules of collisions (see Fig. 2.1). Using LGCA, Mayer and Rasmussen simulated micellar self-reproduction [120] etc.

   The successor of LGCA is Lattice Boltzmann method (LBM), becoming very popular due to its simplicity and efficiency (easy parallelisation). Recently published applications include: simulation of low Mach number combustion [20], turbulence flow [51] or chemical engineering [41] etc.

Other methods include: dissipative particle dynamics (DPD) [72], particle-in-Cell [66], fluid-in-Cell [61], moving particle semi-implicit [98], discrete element method [34], vortex methods [21] etc. It is common to modify the methods for particular simulations, e.g. the modified DPD was used to simulate of blood dynamics in capillary vessels [14, 45].

Besides the methods closely related to physical problems (CFS, CFD, astrophysics etc.), abstract methods: like cellular automata (2.2), artificial chemistry (2.3) or agent based modelling (2.4) are also considered.

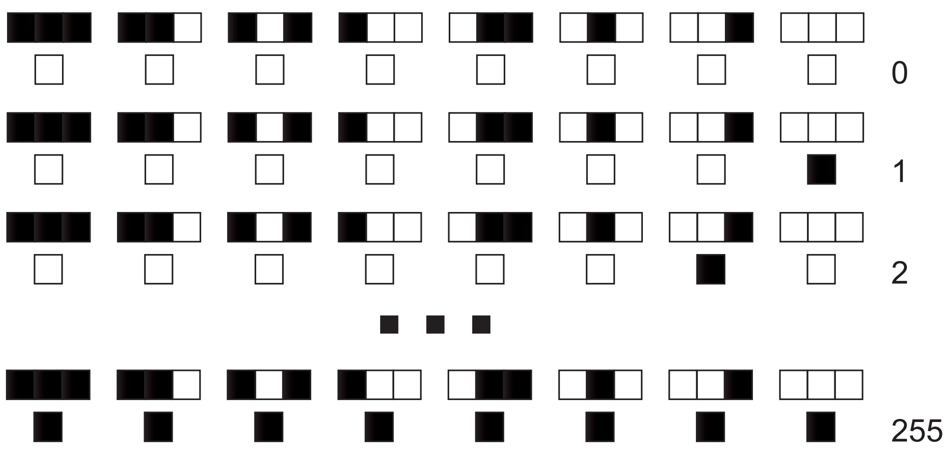


Figure 2.2: The sequence of 256 possible cellular automaton rules for $r = 1$. The number assigned is such that when written in base 2, it gives a sequence of 0's and 1's that correspond to the sequence of new states chosen for each of the eight possible cases covered by the rule [203, p. 53]

2.2 Cellular automata

The history of cellular automata (CA) began in the late forties [203, p. 876]. John von Neumann tried to develop an abstract model of self-reproduction (see 3.2), after his attempt of building a model described by differential equations failed. Following a suggestion from Stanisław Ulam, he simplified his model and introduced the concept of cellular automata [197]. The cellular automata became popular in the seventies after a series of Gardner articles regarding Conway's "Game of life" [57, 58].

The formal definition of CA can be found in [3, p. 28]. A cellular automaton is a lattice of sites. Each site has a finite state automaton (FSM)[2] [73, p. 37–55] inside and can take one of k values. Sites are updated at discrete time steps. At each time step, every FSM changes the state of its site according to a transition function which depends on the state of the site's neighbours.

A simple example of one-dimensional automata is the automata where each site has two neighbours, thus the value of site i at time $t+1$ is determined according to:

$$a_i^{(t+1)} = \phi(a_{i-1}^{(t)}, a_i^{(t)}, a_{i+1}^{(t)}) \tag{2.1}$$

where ϕ is a function of three k-valued variables.

Automata, where only the nearest neighbours are checked, are called $r = 1$ rules, where the two leftmost and two the rightmost checked are called $r = 2$ rules and so on. Each rule can be distinguished by its unique number as explained in Fig. 2.2. Rules which depends on the sum of the values of its neighbourhood are called *totalistic*, the value of site i at time $t+1$ for totalistic automata is described by the equation:

$$a_i^{(t+1)} = \phi(a_{i-1}^{(t)} + a_i^{(t)} + a_{i+1}^{(t)}) \tag{2.2}$$

The cellular automata rule is called *legal* if it satisfies the following conditions:

1. if all neighbourhoods contain cells in a quiescent state (usually 0), the rule should also produce the quiescent state

[2]Abbreviation from other common name: finite state machine

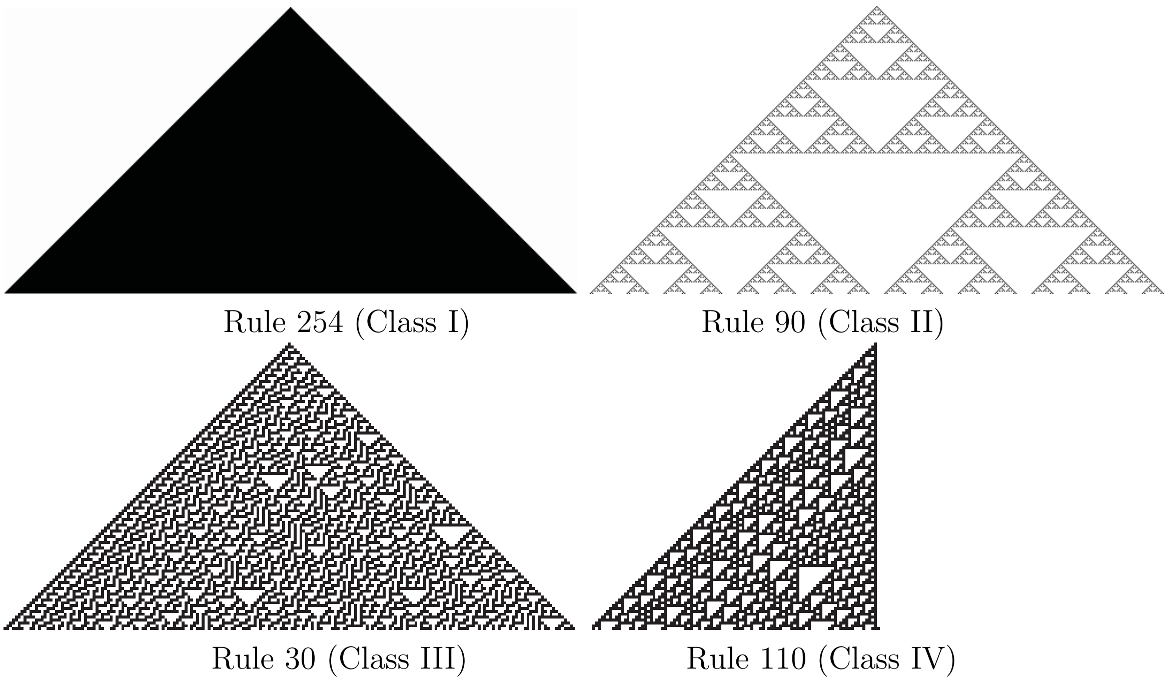


Figure 2.3: Four classes of cellular automata behaviour. 100 steps starting from a single black cell.

2. all rotations of a neighbourhood should map to the same state (spatial isotropy). E.g. the two-dimensional rule must be symmetric.

The automaton state is called *Garden-of-Eden*, if it "cannot occur except at time $T = 0$ [That is] there is no configuration at time $T - 1$ which will give rise to the given configuration at time T by means of the function f which defines the rules for the transition from one state to another" [129].

Wolfram [201, 202] first investigated 1D CAs with $k = 2$ and $r = 1$ (see Tab. 2.1). According to the results of evolution from the initial state, the behaviour of the cellular automata may be classified as belonging to one of the four classes of behaviour:

- Class I: the cellular automata evolve into a homogenous state (rule 254 in Fig. 2.3).
- Class II: the initial state is copied to generate a uniform structure which expands by one site in each direction on each time step (rule 90 in Fig. 2.3).
- Class III: almost all initial states lead to aperiodic, chaotic patterns (rule 30 in Fig. 2.3)
- Class IV: automata that don't fit into any neither of the I-III categories (rule 110 in Fig. 2.3).

The behaviour of class I and II automata is consistent with intuition – simple rules lead to simple behaviour. Contrary to them, the rule 30 in Fig. 2.3 (Class III) produces extremely complex and irregular behaviour. The sequence of states below the initial cell is used as the random number generator for large integers in *Mathematica*[3] [203,

[3]Popular computational software program: `http://www.wolfram.com/products/mathematica/overview/compute.html`.

Rule	Classif.	Rule	Classif.	Rule	Classif.	Rule	Classif.
0	T, I	72	I	128	T, I	200	II
4	II	76	II	132	I	204	II
18	III	90	III	146	III	218	II
22	T, III	94	II	150	T, III	222	II
32	I	104	T, I	160	I	232	T, II
36	II	108	II	164	II	236	II
50	I/II	122	III	178	II	250	I
54	III	126	T, III	182	III	254	T, I

Table 2.1: Table of k=2 and r=1 legal rules and their classification: T–totalistic, Roman number indicate the Wolfram class of automata [3, p. 31].

p. 317]. Interestingly – it is not certain where this complexity came from. The evolution of this automata is as Wolfram said: "the single most surprising scientific discovery I ever made" [203, p. 28].

The most complex and interesting of the CA rules are those that fit into the class IV automata, like rule 110. The behaviour is a strange mixture of regularity and irregularity. It has been argued that such automata are capable of universal computation i.e. patterns may serve as universal Turing machines. This has been proved for rule 110 [32] and Conway's "Game of life" [3, p. 37]. The consequence of this fact is that the behaviour of the cellular automata cannot be predicted in general.

Cellular automata are also generalised to:

1. probabilistic cellular automata [64, 104]: where each cell has a certain probability p ($0 < p \leq 1$) of being updated at each time step,
2. continuous cellular automata [203, p. 155–160]: where cell states are continuous and transition rules are encoded as a continuous function.

Views on cellular automata evolved over time. At the beginning, it was considered as purely abstract "imaginary physics" (as Stanisław Ulam called them). In the 80s due to successful physical simulations (e.g. [194, 201]), CA was regarded as a useful tool that should "attempt not to simulate specific physical phenomena but rather to embody general physics ideas" [194]. In 2002 Wolfram claimed, that CA (or CA-like systems) should form the fundamentals of the "new kind of" science, that will no longer be based on differential equations [203] (in fact the idea appeared much earlier e.g. [191]) – this statement met with a grudging reception amongst physicists (e.g. [162]).

Regardless of this discussion, due to its simplicity and surprising possibilities cellular automata has been used for the simulations of various physical phenomena like crystal growth, turbulent flow, diffusion etc. (see also LGCA/LBM methods in Sect. 2.1) [59, 60], chemistry [94], ecology [9, 179], economy [19], sociology [67] etc. Other classes of cellular automata includes: quantum automata [157], cellular neural networks [161] etc.

2.3 Artificial chemistry

Artificial chemistry emerged as a subfield of artificial life (see 3.1) in 1992 [52]. According to [40]: an artificial chemistry (AC) is a man-made system that is similar the to real chemical system. More formally, it is a triple (S, R, A), where S is a set of possible molecules (basic AC entities), R is a set of rules defining possible reactions between molecules, and A is an algorithm describing how the rules are applied to the molecules. It is also possible to define an artificial chemistry as a tuple (S, I) where I describes interaction between molecules (aggregates R and A). An example of artificial chemistry is Squrim3, described in Sect. 4.3.2 on page 39.

If there is an isomorphism between a molecule or reaction in AC to a molecule or reaction in chemistry, the AC can be called *analogous*, otherwise it is called *abstract*. Analogous ACs are part of *computational chemistry* [33] and are beyond scope of this dissertation (the goal of computational chemistry is to model chemical processes as closely as possible).

2.3.1 Molecules

The set of molecules $S = \{s_1, s_2, ..., s_n\}$ describes all molecules that can appear in an AC. Note that the molecule does not necessarily mean the concept of "real" chemistry, for instance it can be numbers [10], computer programs (see: 4.3.3, 4.2.1, 5) etc. Molecules can be defined:

- *explicit* when the S is given as the enumeration of symbols (e.g.: $S = \{A, B, C\}$)
- *implicit* when S is a description of how to construct a molecule (e.g. $S = \{1, 2, 3, ...\}$).

2.3.2 Rules

The set of rules R describes possible reaction between molecules $s \in S$. It is common to write rules according to the chemical notation:

$$s_1 + s_2 + ... + s_n \rightarrow s'_1 + s'_2 + ... + s'_m \tag{2.3}$$

The rule depicts that n molecules on the left-hand side can react and then be replaced by the m components on the right-hand side. Rules can be defined in two different ways:

- *explicit* when reaction rules are enumerated and explicitly given
- *implicit* when the definition of the interaction refers to molecule structure

2.3.3 Dynamics

The last term of AC's definition describes how rules R are applied to the set of components P. Note that P is not the same as S since molecules may appear in different numbers. Dittrich [40] described two typical methods of the AC dynamics modelling:

- Stochastic molecular collisions: every molecule is explicitly simulated. Typically a sample of molecules is randomly drawn from the population of molecules P then the possibility of applying a rule $r \in R$ is checked. If so, the left-hand-side molecules are replaced by the right-hand-side molecules given by r. This approach provides the lowest level of description thus being the most realistic. The simulation of a large population may be slow due to computational cost.

- Continuous differential or discrete difference equations: the dynamics is described using differential rate equations which reflect the concentration of molecule species. A reaction r can be written as:

$$a_1 s_1 + a_2 s_2 + ... + a_n s_n \rightarrow b_1 s_1 + b_2 s_2 + ... + b_n s_n \tag{2.4}$$

  where a_i, b_i are counterparts of stoichiometric factors of chemical reaction. The change of concentration of s_i is given by the equation:

$$\frac{ds_i}{dt} = \sum_{r \in R} [(b_i^r - a_i^r) \prod_{j=1}^{N} s_j^{a_j^r}], i = 1, \ldots, N \tag{2.5}$$

  Bagley et al. also discussed the possibility that the number of molecular species, thus the number of equations is not fixed and may change over time [8].

2.3.4 Constructive dynamics

If new components can appear the AC is called *constructive*. If they are generated through the action of other components, it is called *strongly constructive* if new components are generated randomly it is called *weakly constructive*. In these terms, natural chemistry is considered to be a strongly constructive system [52]. Constructive ACs are commonly used with implicit defined rules.

An example of a constructive system is number-division chemistry [10]:

- A set of molecules consists of all natural number greater than one: $S = \{2, 3, 4, \ldots\}$
- When two molecules collide, the following reaction occurs:

$$R = \{s_1 + s_2 \rightarrow s_1 + \frac{s_2}{s_1} : s_1, s_2 \in S\} \tag{2.6}$$

  i.e. if s_1 can divide s_2 without remainder s_2 is replaced by value s_2/s_1

Despite being very simple, the behaviour of the system is interesting. The initial, random population of molecules tends to eliminate non-prime numbers until it finally consists of primes number only. Finally, all numbers in the reactor are prime factors of the initial population.

2.3.5 Applications

ACs are used for chemical [119], ecological [15, 43], economical [183] and social [184] modelling. Another common application is the field of information processing e.g. AC was used to get better results with an artificial neural network in robot controller [76]. Ziegler [205] successfully used AC to control the small Khepera robot [128]. Artificial chemistries are also used as an optimisation algorithm, e.g. a Chemical Casting Model has been used to solve the Travelling Salesman Problem [92].

2.4 Agent based modelling

Agent based modelling (ABM) is a modelling and simulation approach applied to a complex system or complex adaptive system in which the model is comprised of a large number of interacting, autonomous elements (*agents*).

The term *agent* has many definitions among researchers, e.g. Elizabeth Sklar [177] used this term to describe entities with the properties of autonomy and self-interested. Mark Bedau used the term to describe "simple, low-level [entities] that simultaneously interacts with each other [...] based on information about, and directly affect, only its own local environment" [11]. Goran Trajkovski analysed different definitions and gave the following list of fundamental properties of an agent (which may be used both alone and in combinations) [192, p. 3]:

1. Autonomy: an ability to operate in an unaided and independent fashion

2. Proactivity: an ability to take initiative to affect actions toward its goals

3. Intentionality: the attribution of purpose, belief, need and desire

Agent-based models are used for various simulations regarding: flocking [156], ant colony optimisation [42], crowd flow[68], economy [189], epidemiology [7] etc. An abbreviated review of agent based-models is presented in Sect. 4.1 on page 29.

Chapter 3

Complex system modelling

This chapter describes selected problems of complex system modelling. Sect. 3.1 consists of introduction to artificial life, i.e. describes its main research directions. Sect. 3.2 contains a review of self-replication issues (mainly based on von Neumann's concept). Sect. 3.3 briefly describes self-organization within the context of artificial life field.

3.1 Artificial life

Artificial life (ALife) is an interdisciplinary study of life and life-like processes. The history of ALife has its intellectual roots in von Neuman's works on cellular automata [197] (see also 3.2), Wiener's works on cybernetics [198] and artificial intelligence (especially genetic algorithms [70]). The name "artificial life" was coined by Langton, who gave the following definition of the subject: "Artificial Life is the study of man-made systems that exhibit behaviours characteristic of natural living systems. It complements the traditional biological sciences concerned with the *analysis* a of living organism by attempting to *synthesise* life-like behaviours within computers and other artificial media. By extending the empirical foundation upon which biology is based beyond the carbon-chain life that has evolved on Earth, Artificial Life can contribute to theoretical biology by locating *life-as-we-know-it* within the larger picture *life-as-it-could-be.*" [103].

Charles Ofria quoted the following John Maynard Smith phrase that points to the importance of ALife research: "So far, we have been able to study only one evolving system and we cannot wait for interstellar flight to provide us with a second. If we want to discover generalisations about evolving systems, we will have to look at artificial ones." [134].

Note, that the above definition of the ALife subject requires an accurate definition of the living system. This is one of the most difficult philosophical and scientific problems in the history of science [56, 167]. According to John Maynard Smith, there are two distinct views of living systems: "One is of a population of entities which, because they possess a hereditary mechanism, will evolve adaptations for survival [evolutionary definition]. The other is of a complex structure which is maintained by the energy flowing through it [ecological definition]" [121, p. 7] (quoted from: [186, p. 9]).

Research directions

At the moment, there are three main branches of artificial life [11]:

- *Soft*: simulations or other purely digital constructions that exhibit life-like

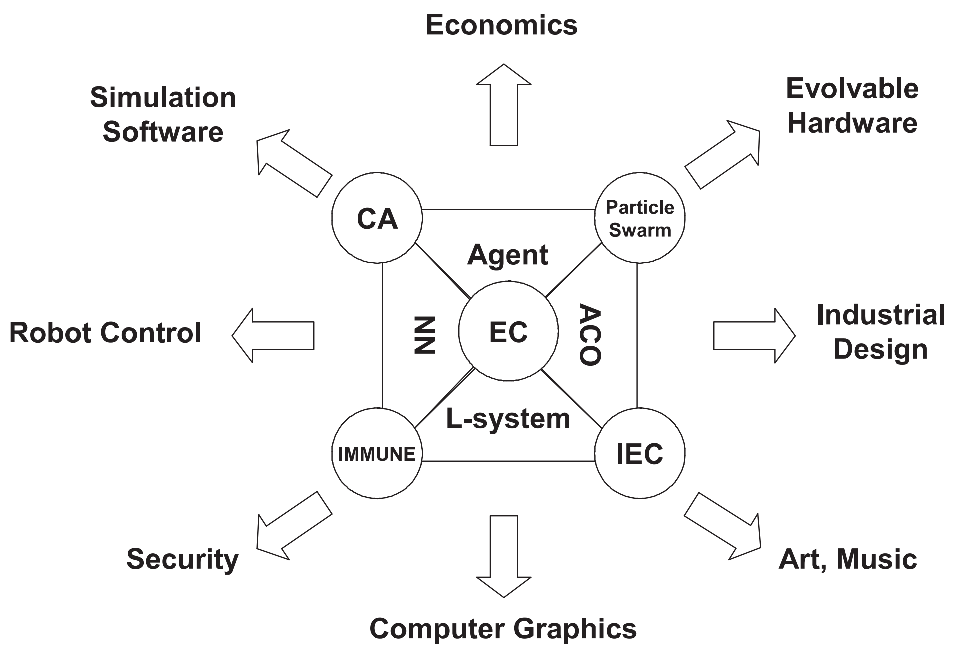


Figure 3.1: Relations of the ALife methodologies (CA = cellular automata, IEC = interactive evolutionary computation, EC = evolutionary computation, NN = neural network, ACO = ant colony optimisation) [95, Fig. 1.1].

behaviour (see chapter 4), software models are generally based on assumptions given by Langton [103]:

1. They consist of population of simple programs or specifications
2. There is no single program that directs all of the other programs
3. Each program details the way in which a simple entity reacts to local situations in its environment, including encounters with other entities
4. There are no rules in the system that dictate global behaviour
5. Any behaviour at levels higher than the individual programs is therefore emergent

- *Hard* artificial life produces hardware implementations of life-like systems, mainly related to work with life-like robots [1], and evolvable hardware [17, 69],
- *Wet* artificial life synthesises living systems out of biochemical substances

Besides the scientific researches, ALife has also been recognized as an interesting tool for artists: [122]

Methodology and applications

Kim and Cho gave the comprehensive review of the applications of artificial life. According to their review, methodology of ALife comprises of: Lindenmayer systems (L-systems), evolutionary computation, interactive evolutionary computation, agent-based modelling (2.4), cellular automata, and ant colony optimisation. Another popular tools, not mentioned by Kim and Cho, are artificial chemistries (2.3). Relations of the methodologies with possible real-life applications are presented in Fig. 3.1.

Interpretation and controversies

An interesting topic is the interpretation of soft ALife simulations [135, 142]. The most popular view called *weak artificial life* [186, p. 38] treats them as models of various processes associated with living organisms, but does not claim that any parts of them are living themselves. On the other hand, proponents of the *strong artificial life*, claim that simulated programs are alive in same manner as their biological counterparts, i.e. life is independent of matter.

Robert Rosen [160] formulated a proof that living systems are fundamentally different from Turing's machines (the conclusion is similar to the one of Roger Penrose in the field of artificial intelligence [143, 144]). His claim, referred as *Rosen's central theorem*, caused a lot of controversy among ALife researches (e.g. [25, 26, 113, 204]). If his proof is correct, not only strong but also weak artificial life are wrong views. An obvious implication of the Rosen's theorem is that most of ALife researches are pointless (as life cannot be computationally modelled, but it is still possible to simulate some of its aspect).

Despite initial doubts [74], artificial life methods have been finally accepted as a part of mainstream science (e.g. [24, 105, 106, 199]).

3.1.1 Open problems

Mark Bedau et al. [12] prepared a list of open problems in artificial life, identifying the main directions of the artificial life research program:

1. How does life arise from the nonliving?

   (a) Generate a molecular proto-organism in vitro. The aim of the task is to create possibly simplest living organism in a laboratory. Molecular life is understood as a self-reproducing entity, constructing itself in a simple environment and capable of evolution. The environment should provide a simple form of energy and nutrients for proto-organisms. At the moment, researches have succeeded in creating artificial self-replicating systems with limited evolution capabilities [151] but combining these in an autonomous, evolvable, self-replicating system still remains an open problem.

   (b) Achieve the transition to life in an artificial chemistry in silico. The task is similar to the previous one as it requires a self-organised collection of separate artificial molecules being able to replicate and evolve. The set of molecules should be created spontaneously in an artificial chemistry started from random initial seed. Artificial chemistry should be stochastic rather than deterministic and constructive rather than descriptive (2.3). Internal organization of generated structures should be emergent phenomena of lower-level primitives.

   Examples of simulation regarding this topic are described in Sect. 3.3.

   (c) Determine whether fundamentally novel living organizations can exist. The task is closely related to molecular biology. It includes the question whether there exists a living organism without some essential features, like: genotype-phenotype distinction, hierarchical organization etc. Answering the question may be helpful in understating what structures may be treated as living organisms (like nanobes [117] or structures found in Martian meteorites [123]).

(d) Simulate a unicellular organism over its entire lifecycle. Contrary to the first two tasks, this one is related to the simulation of biological organism on the computer. Performing the whole simulation is, at the moment, far beyond the abilities of modern computers. Mark Bedau claims that a simplified molecular dynamics version should be achievable this century. Examples of recent works regarding this issue include: simulation of 1 million of atoms for over $50ns$, forming the complete satellite tobacco mosaic virus [53]; simulation of protein folding for over $1\mu s$ [196] etc. Fellermann et al. simulated simple proto-cell using the dissipative particle dynamics approach [49]. It is an open question what sort of simplifications of biological models can be done in order to the keep models accurate.

(e) Explain how the rules and the symbols are generated from physical dynamics in living systems. An example of the symbolic level are genes and rules as for replication or recombination. The task is to study how the underlying physics can form higher level rules that can operate on symbols (with no state transitions defined explicitly).

2. What are the potentials and limits of living systems?

   (a) Determine what is inevitable in the open-ended evolution of life. The task consists of two questions:

   - What are the features common to all evolutionary processes?
   - Do different evolutionary processes contain fundamentally different evolutionary potential?

   The task involves comparing the potential of evolution of living (biological) organisms and the evolution of digital "forms of life".

   (b) Determine minimal conditions for evolutionary transitions from specific to generic response systems. The example of a specific response system is the mechanism present in simple organisms, that recognizes specific foreign molecules and activates the necessary response. On the other hand, multicellular organisms have an almost universal immune system, that can respond to a previously unknown threat. The task is to understand how the generic response system can arise from the specific response system in the course of evolution.

   (c) Create a formal framework for synthesising dynamical hierarchies at all scales. Biological systems are organised in levels: ecosystems, organisms, organs, tissues, cells, organelles, molecules, atoms [180, p. 6]. The first task is to create formal, consistent framework to uniformly describe all levels (e.g. the particle system developed by Rönkkö [159], briefly described on page 33). The second task, is to understand how these hierarchies are generated in biological systems.

   (d) Determine the predictability of evolutionary consequences of manipulating organisms and ecosystems. The task is related to estimating both the cost and benefits of "wet" ALife researches, which may caused an unpredictable impact on the ecosystem. Simulation of consequences of human activity is also related to this task (e.g. modelling the impact of corridors between fragmented habitat on predator-prey oscillations [91]).

(e) Develop a theory of information processing, information flow, and information generation for evolving systems. The task consists of the following sub-tasks:

- There are two complementary kinds of information *transmission* in a living system: hereditary transmission through evolution time, and transmission of information specified by the physical environment. The task is to clarify the range of possibilities for information transmission and determine which of those is exploited by the biosphere
- Components of living system (organisms or groups of organisms) are able to solve problems, in a manner similar to genetic algorithms. The task is to unify evolution with information *processing* in a comprehensive theory.
- Understand how information is *generated* during the evolutionary process.

3. How is life related to the mind, machines, and culture?

(a) Demonstrate the emergence of intelligence and the mind in an artificial living system.

(b) Evaluate the influence of machines on the next major evolutionary transition of life.

(c) Provide a quantitative model of the interplay between cultural and biological evolution

(d) Establish ethical principles for artificial life. This task is closely related with interpretation of an ALife simulation.

3.2 Self-replication

The idea that the complexity of the ecosystem could be explained by the process of biological evolution is commonly accepted among scientists (e.g. [37]). Self-replication is recognized as the crucial phenomenon that is needed to start the evolutionary process, and to sustain growth in the complexity (if greater complexity means better fitness, which is not always true [199]), obtained due to random processes. Tim Hutton briefly explained this process: “in general, where complexity growth occurs and continues to occur, the genomes are storing information against the flow of entropy (like Maxwell’s demon) because of the reproductive advantage of doing so”[78].

A more formal introduction to this issue can be found in various sources e.g. [3]. Eigen and Winkler [48] described various models of population growth, based on game theory, that support claims of displacement of less-adapted organisms by better-adapted ones.

In the context of ALife (3.1), Thomas Ray the author of Tierra (3.2.3) noted: “self-replication is critical to synthetic life because without it, the mechanisms of selection must also be predetermined by the simulator. Such artificial selection can never be as creative as natural selection. The organisms are not free to invent their own fitness functions. Freely evolving creatures will discover means of mutual exploitation and associated implicit fitness functions that we would never think of. Simulations constrained to evolve with predefined genes, alleles, and fitness functions are dead-ended, not alive.” [154, p. 372] (cited from: [186, p. 52]).

The importance of the ALife research of the self-reproduction driven evolution, was summed up by Hutton : “the evolutionary process is often seen as creative in the sense that it has produced many sophisticated organs and behaviours that surprise us with their ingenuity, and yet this self-driven process has never been captured in a computer model” [78].

Beyond the undoubted scientific value, the understanding of the self-replication process plays also a significant role in engineering e.g. Chou and Reggia discussed using cellular automata self-replication in order to obtain a massively parallel machine being able to solve NP-hard problems [23], Eric Drexler sees self-replication as a crucial feature in nanotechnology [44] etc.

3.2.1 Von Neumann’s models

The first attempt to create a formal description of a system that could support self-reproducing was taken by von Neumann. Von Neumann discussed five different models of self-reproduction [197, p. 91–99], but most of them were left unfinished.

The *kinematic* model

The kinematic model is the most general concept which involves a physical machine moving freely in a stockroom of spare parts. The machine should have a memory tape with encoded instructions. According to these instruction, the machine should move in the stockroom and collect the proper parts. The collected parts should be joined together creating in effect an exact copy of itself. When it finishes, it should copy its own memory tape onto the blank tape of its offspring. In spite of referencing to the mechanics or the electronics, the kinematic model was just a mental experiment (in particular it ignores problems of force and energy). The model has been introduced in a series of lectures at The University of Illinois in 1949, and has been popularised in 1955 by Kemeny [93]. The concept was finished in terms of a cellular automata as cellular model (see below).

Freitas and Merkle [55] reviewed tens of both theoretical and experimental works on physical self-replicating systems, which in fact, reflect the von Neumann’s kinematic model. As an example, the idea is present in the concepts of space exploration (e.g. self-replicating, growing lunar factory [54]), a brief discussion on robotics applications was published in Nature [63] etc.

The *excitation-threshold-fatigue* model

This model was designed close to the cellular automata model, but each cell was replaced by an element similar to the neuron. The details of the neuron haven’t been provided, but “through a careful analysis of his work, we can deduce that it would have borne a fairly close relationship to today’s artificial neural networks” [188, p. 35].

The *continuous* model

Details of this model are not known, probably the differential equations would describe “the excitation, threshold and fatigue properties of a neuron” [188, p. 35].

The *probabilistic* model

Von Neumann intended to introduce probabilistic transitions between states instead of deterministic, however no more details are known.

The *cellular* model

The cellular model is the most important model of self-reproduction. In this model, cellular automata (2.2) provide strict theoretical background.

Von Neumann was well aware of the drawbacks of his model: "by axiomatising automata in this manner, one has thrown half of the problem out the window, and it may be the more important half. One has resigned oneself not to explain how these parts are made up of real things, specifically, how these parts are made up of actual elementary particles, or even of higher chemical molecules" [197, p. 77].

The main concept in this model is a *universal constructor*. The universal constructor, denoted by A, is a machine (a particular automaton) capable of constructing any other machine X (i.e. some other automaton) based on its description $\phi(X)$[1] [2] (the result is described in bold):

$$A + \phi(X) \Rightarrow A + \phi(X) + \mathbf{X} \tag{3.1}$$

It is easy to notice, that if the constructor is provided with its own description ($X = A$), then it will construct the copy of itself:

$$A + \phi(A) \Rightarrow A + \phi(A) + \mathbf{A} \tag{3.2}$$

In order to make the full self-replication system, it is necessary to copy also the constructor description $d(A)$. To perform this, a new machine B is introduced, capable of making the exact copy of any description (Fig. 3.2b). The A and B machines can be joined together forming the consistent structure. When this structure is enhanced by an additional machine C, which manages the order of execution, the new machine is formed. The $A + B + C$ machine with its description $\phi(A + B + C)$ forms the full self-replicating system:

$$\begin{aligned} A + B + C + \phi(A + B + C) \ &\Rightarrow \ A + B + C \\ &+ \ \phi(A + B + C) \\ &+ \ \mathbf{A} + \mathbf{B} + \mathbf{C} \end{aligned} \tag{3.3}$$

Due to the capabilities of the universal constructor, it is possible to add the description of an additional automaton D (Fig. 3.2):

$$\begin{aligned} A + B + C + D + \phi(A + B + C + D) \ &\Rightarrow \ A + B + C + D \\ &+ \ \phi(A + B + C + D) \\ &+ \ \mathbf{A} + \mathbf{B} + \mathbf{C} + \mathbf{D} \end{aligned} \tag{3.4}$$

[1]Sometimes denoted as memory tape.

[2]In fact, due to presence of Garden-of-Eden states (described on p. 8), universal constructor is not being able to construct any possible structure, but any structure that can be described upon the description ϕ.

In order to avoid the simple self-reproducing system that resembles growing crystals rather than their biological counterparts, von Neumann postulated that D should posses the ability of performing universal computation.

If the description $\phi(A+B+C+D)$ will be changed in such a way that only the description of D will be affected, the whole system will still be able to reproduce, but the overall result will be slightly different i.e.:

$$\begin{aligned} A+B+C+D+\phi(A+B+C+D') \;&\Rightarrow\; A+B+C+D \\ &+\; \phi(A+B+C+D') \\ &+\; \mathbf{A}+\mathbf{B}+\mathbf{C}+\mathbf{D}' \end{aligned} \tag{3.5}$$

The following set: $A+B+C+D+\phi(A+B+C+D)$ is then able to:

- self-reproduce
- perform any computation
- evolve (as it is robust to at least some mutation)

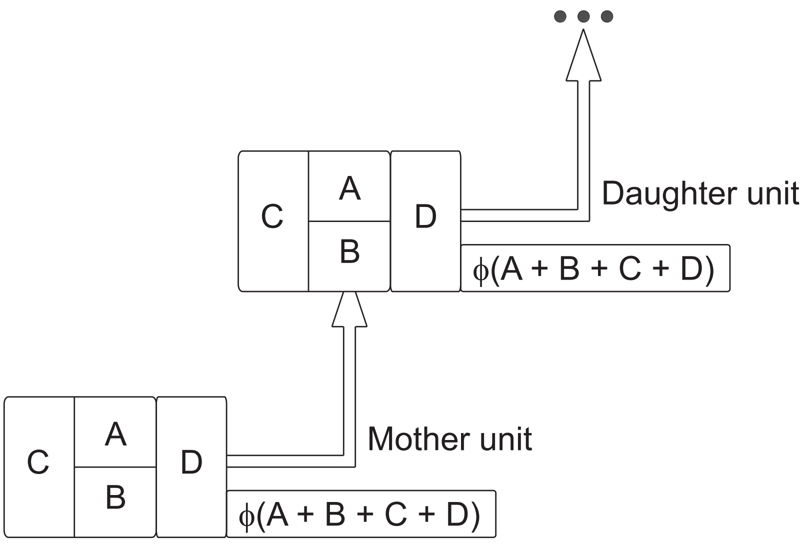


Figure 3.2: Von Neumann's model of self-replication: an universal constructor A, a copying machine B, a control machine C, a universal Turing machine D. The mother automaton, using description $\phi(A+B+C+D)$, constructs the daughter automaton etc.

Von Neumann describes details of implementation in a system with 29 possible states and about 2×10^5 initial cells. Due to technical limitations his work was left unfinished for years. Decades later, in 1995 Pesavento [146] developed a simulation of the universal constructor. The abilities of computers in the nineties were, however, still too small to perform the simulation of full self-replication or to enhance the constructor with the Turing machine. Attempts to simulate the full system are still taken (e.g. in 2005 Buckley published results of an analysis of the constructor [16]).

In 1973 Vitányi discussed the possibility of introducing sexual reproduction into the von Neumann's model [195]. In this model, the constructor uses two descriptions ϕ_1 and ϕ_2, the result is not an exact copy of either parent.

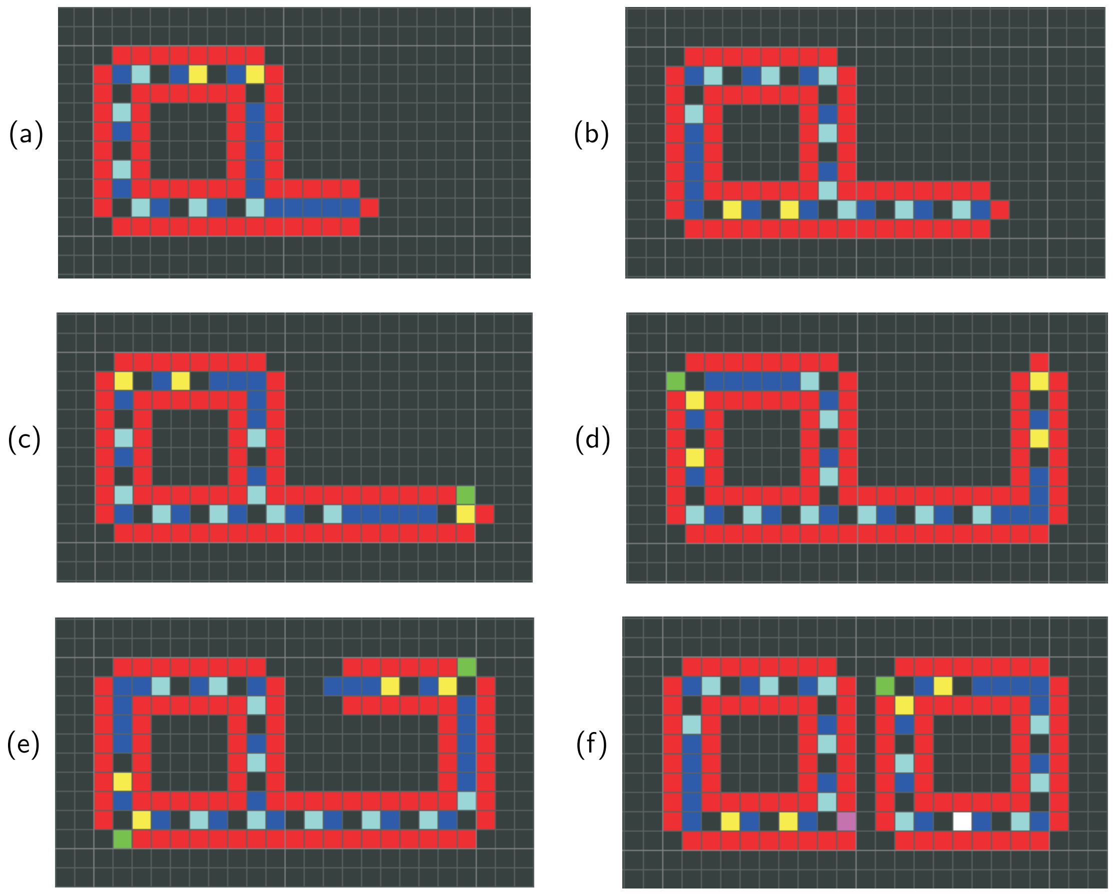


Figure 3.3: Langton's loop after: 0 (a), 16 (b), 32 (c), 64 (d), 96 (e) and 128 (f) time cycles. Pictures generated by Golly 2.1 [158].

In 1968 Codd, by altering the states and transition rules simplified the constructor [30]. Hutton found and corrected four significant errors in Codd's original work and in 2010 published a complete and functioning implementation of Codd's self-replicating automata: "the body of the final machine occupies an area that is 22,254 cells wide and 55,601 cells high, composed of over 45 million nonzero cells in its unsheathed form. The data tape is 208 million cells long, and self-replication is estimated to take at least 1.7×10^{18} time steps" [79].

Beside the omission of energy, there are also other significant distinctions between von Neumann's model and its biological counterpart [186, p. 48], the most important are:

1. biological self-replicators have the ability for self-maintenance in the face of environmental perturbations,

2. von Neumann's self-replicator builds the whole "adult" copy of itself, while biological replication is usually divided into phases (biological organisms grow out from babies, hatch from eggs etc.).

3.2.2 Self-replicating loops

Technical problems with the implementation of the original concept led Langton to rethink its assumption. Because von Neumann's self-replicator is a special case of an extremely complex universal constructor, Langton tried to define the simplest cellular automaton capable of self-replication [102]. Langton distinguished between trivial and non-trivial self reproduction. The trivial self-reproduction is driven by "physics" (i.e. cellular rules) while non-trivial is directed by the configuration itself[3]. An example of trivial self-reproduction is 254 Wolfram automata, when a cell switch its state from quiescent one if any of its neighbour has not-quiescent state (Fig. 2.2).

This approach, known as a *Langton's loop* is loosely based on one of the simplest part of Codd's automaton – *periodic emitter*. The evolution from the initial state is presented in Fig. 3.3.

Langton not also dropped von Neumann's original requirement of universality, which significantly simplified the automaton, but also completely removed its ability to evolve. Taylor noted: "Langton's loop is very fragile in that it cannot in general withstand perturbations and mutations, and is certainly not capable of heritable viable mutation. McMullin [124, p. 181] describes Langton's work as a «cruel (though of course unintentional) parody»of von Neumann's." [186, p. 49]. Despite these objections, Langton's loop became the basis for many followers, who have coped with most of its restrictions.

In 1989 Byl [18] demonstrated an automaton much simpler than Langton's loop, but still capable of non-trivial self-reproduction. In 1993 Reggia et al. designed the smallest known self-replicating loop [155], consisting of just 5 cells embedded in 6-state space.

In 1997 Morita and Imai proposed a simple model of a 12 state automata (the minimum number of cells is only 4) that is capable of self-reproduction, with a large tolerance to initial shapes[130]. This approach was also extended to three-dimensional space [81].

An attempt to extend the Langton's model of the possibility of carrying out universal computation was taken by Perrier and Zahnd [145]. In this approach, a Turing machine with its data was attached to the Langton's loop (theoretically it is possible to attach an universal Turing machine)[4]. The first stage in the Perrier's loop is to create a copy of Langton's loop. The daughter loop then sends a message (special state) to its mother which starts the Turing machine duplication (process is independent of loop duplication). When the copy is finished, another signal activates the reproduction of Turing machine's data. The overall result is a copy of: the loop, the Turing machine and its data.

Similar work was done by Tempesti [187, 188]. Tempesti's loop has "the option of attaching to the automaton an executable program which will be duplicated and executed in each of the copies of the automaton" [187]. Another interesting feature is, that his loop does not "die" after the duplication is complete [5]. Tempesti's loop is thus better suited to finite spaces [6]. Petraglio et al. have implemented arithmetic operations in Tempesti's loop by passing messages from one loop to another after building a network structure through self-replication [147].

Robustness of loops was also discussed by Sayama. In 1998 Sayama introduced the dissolving state into the Langton's loop automaton, forming a structurally dissolvable

[3]Perrier and Zahnd [145] noted that the distinction is unclear, as every cellular automata behaviour results from rules. An interesting discussion of the differentiates between physics-driven and information-driven self-reproduction contains work: [115].

[4]Lerena and Courant attempted to generalise cellular automata into a new class of machines, that are a super-class of them and also of Turing machines [107].

[5]Unlike Langton's loop, which became inactive after replication.

[6]Langton's loop algorithm assumes that there is always enough space for its copy.

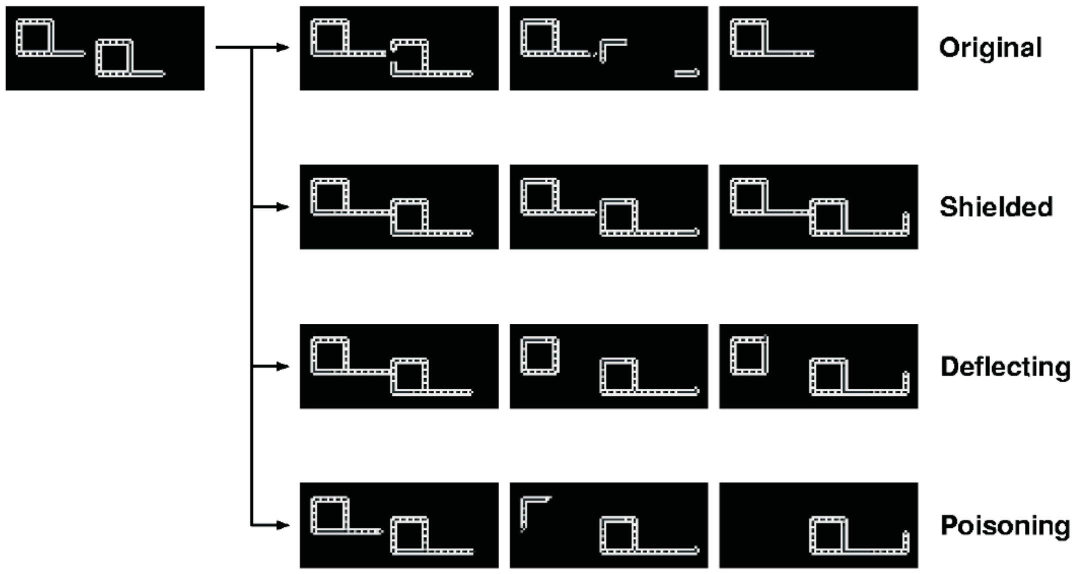


Figure 3.4: Self-protective Sayama loops. “Original: The attacked loop dissolves, and the attacker continues to produce its offspring where the attacked loop was. Shielded: The tip of the attacker’s arm dissolves while the attacked receives no damage, so the same attack continues to occur repeatedly. Deflecting: [...]. The attacker is deceived and begins to produce its offspring in a different direction rotated by 90 degrees counterclockwise. Poisoning: The attacked loop generates a poison [...] at the tip of the attacker’s arm to kill the attacker completely” [166].

self-reproducing (SDSR) loop. As Sayama noted: “the SDSR loop can dissolve its own structure when faced with difficult situations such as a shortage of space for self-reproduction. This mechanism (this disappearance of a subsystem of the whole system) induces [...] potentially evolvable behaviour” [163]. The evolvable behaviour was demonstrated a year later, after introducing several improvements to the SDSR: “though no mechanism was explicitly provided to promote evolution, the loops varied through direct interaction of their phenotypes, smaller individuals were naturally selected thanks to their quicker self-reproductive ability, and the whole population gradually evolved toward the smallest ones.” [164]. The subsequent research led to the observation of the evolutionary growth of complexity: “[...] there were some cases where the structural complexity of organisms characterized by the average length of organisms and the average frequency of branching per organism increased as the population evolved. However, an unlimited increase of structural complexity could not occur in this model.” [165].

The next works concentrated on expanding the capabilities of loop the to protect itself from exogenous attacks. In work [166] the following strategies of *self-protection* were shortly discussed: static barrier between loop and environment, camouflage which causes the loop to be invisible to the attacker, active reaction in order to revert changes, redundancy in the loop’s structure and/or function. Sayama implemented three different mechanisms: shielding, deflecting and poisoning – illustrated in Fig. 3.4, and realized that these improvements significantly increase the diversity of loops.

```
001 nop1    015 sub_ac  029 call    043 nop0     057 nop1  071 nop1
002 nop1    016 mov_ab  030 nop0    044 nop0     058 nop0  072 pop_cx
003 nop1    017 adrf    031 nop0    045 push_ax  059 nop0  073 pop_bx
004 nop1    018 nop0    032 nop1    046 push_bx  060 inc_a 074 pop_ax
005 zero    019 nop0    033 nop1    047 push_cx  061 jnc_b 075 ret
006 shl     020 nop0    034 divide  048 nop1     062 jmp   076 nop1
007 shl     021 nop1    035 jmp     049 nop0     063 nop0  077 nop1
008 shl     022 inc_a   036 nop0    050 nop1     064 nop1  078 nop1
009 mov_cd  023 sub_ab  037 nop0    051 nop0     065 nop0  079 nop0
010 adrb    024 nop1    038 nop1    052 mov_iab  066 nop1  080 if_cz
011 nop0    025 nop1    039 nop0    053 dec_c    067 if_cz
012 nop0    026 nop0    040 if_cz   054 if_cz    068 nop1
013 nop0    027 nop1    041 nop1    055 jmp      069 nop0
014 nop0    028 mal     042 nop1    056 nop0     070 nop1
```

Listing 3.1: Self-replicating program in Tierra.

3.2.3 Core Worlds

Experiments with self-replication in so-called Core Worlds gained great popularity among researchers. This approach was inspired by a popular programming game Core War [39]. In this game, programs, written in a simplified assembly language, compete for the control of the simulated core memory of a computer. The object of the game is to terminate of all hostile programs. The Core War allows programs to self-reproduce itself, and it soon proved to be, that the most successful programs were the ones that used this feature.

In 1990 Rasmussen designed Core World – a system similar to Core War in which the command that copied instructions is defective and sometimes writes a random instruction instead of the one intended [152]. Rasmussen created a simple self-replicator and let it run. Unfortunately his experiment failed. The programs started to copy code into each other, and soon no proper self-replicators survived.

In the next year Thomas Ray [154] designed a Tierra system. Ray developed a set of instructions for performing self-replication which was more robust to mutations. Contrary to Core World, programs in Tierra must allocate memory before they have permission to copy itself.

Ray's first experiment was performed with self-replicating program, presented in List. 3.1. The program shortly filled up the whole available memory, with its multiple copies, some of them were erroneous due to "mutations". When the memory was filled up, Tierra automatically removed the oldest programs to make room for new ones, thus exerting the selection pressure.

Ray observed that behaviour of his programs (or "organisms" as he called them) became intricate: "[...] for example, some organisms were able to shrink further by removing critical portions of their genome and then use those same portions from more complete competitors, in a technique that Ray noted was a form of parasitism. Arms races transpired where hosts evolved methods of eluding the parasites, and they, in turn, evolved to get around these new defenses. Some would-be hosts, known as hyperparasites, even evolved mechanisms for tricking the parasites into aiding them in the copying of their own genomes. Evolution continued in all sorts of interesting manner [...]" [134, p. 6]. It is disputable how these results were related to the knowledge introduced into the system by Ray himself. Timothy Taylor noted: "it may be true to say that most of the interesting behaviour that has been seen to evolve has done so because facilities for these behaviours were engineered into the original language specification. For example, the fact that parasites emerge is a little less surprising when the mechanisms of template-driven branching are considered" [186, p. 54]. Taylor also described some next Tierra improvements which involve adding more high-level operations like insertion, deletion and crossover operators, or new forms of mutation. New features are directly controlled by the Tierra operating system (not by programs themselves), thus they became part of environment's "physics".

In 1992 Adami began experiments with an artificial selection pressure to evolve solutions of specific mathematical problems like adding numbers. Adami tried to define the fitness function (known from genetic algorithms) which would give the successful programs extra CPU cycles as a bonus. Adami was able to evolve some simple tasks but he faced many problems in using Tierra that way. A year later in collaboration with his colleagues, Adami developed Avida which becomes the most advanced tool for studying self-replication and directed evolution among core worlds (see also 4.2.1 for short description) [134, p. 6].

3.2.4 Other approaches

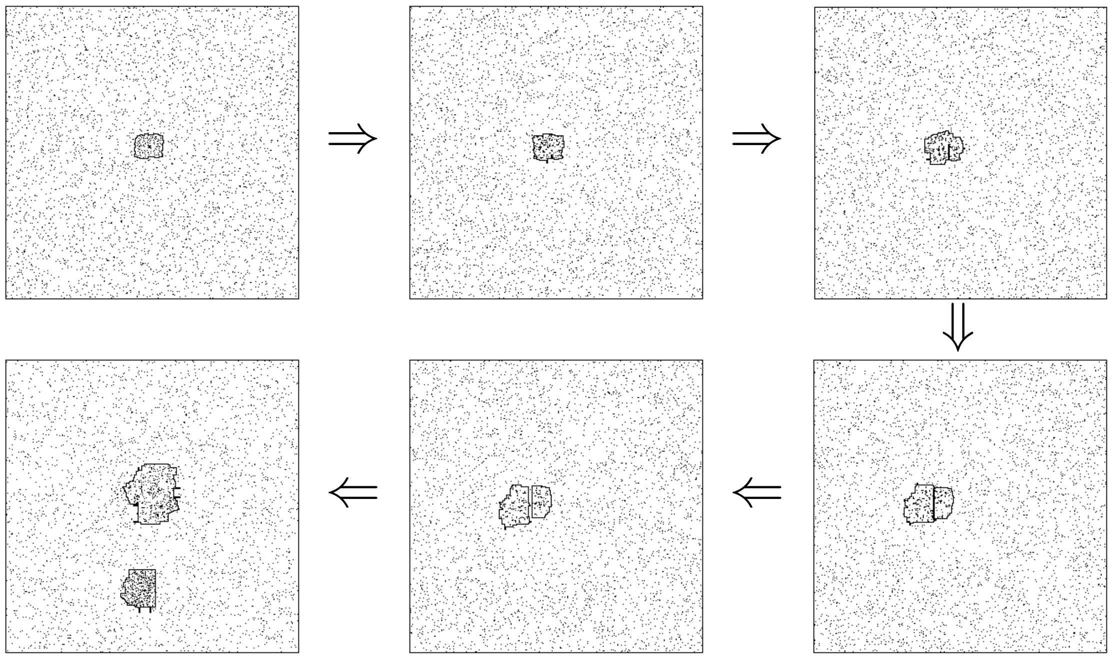


Figure 3.5: Cell division in the Universum environment [85]. Programs build wall inside the cell. When the wall is over, the cell splits into two subcells.

In the seventies John Holland proposed the collection of self-reproduction models referred to as α-Universes. Taylor reported [186, p. 61] that α-Universes were finally developed by McMullin [124], however he found a number of problems which have not been anticipated by Holland. The original theoretical considerations were then continued by Martinez [118] (see also [86, p. 27–32]), which became the inspiration for the Universum environment (4.3.3). In Universum self-replication was designed, but not fully performed. The experiment was planned in a manner reminiscent of cell fission – as presented in Fig. 3.5. The process could not continue due to the lack of program copies.

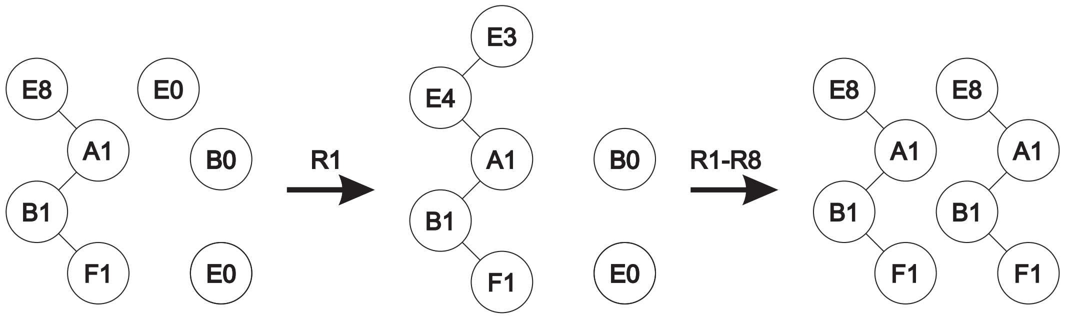


Figure 3.6: Self-replication in Squirm3 [77, Fig. 1].

.

Will Stevens developed the software environment for simulating kinematic self-replicating machines which, as he states, offers a greater physical realism over cellular automata frameworks. In his system "sliding tiles moving in a discrete two-dimensional grid can be put together to build machines. The tiles can perform logical and mechanical functions, and can be connected to each other" [181]. Machines can be constructed from collections of tiles. Stevens also developed 3 dimensional version of his environment in which the von Neumann's constructor has been simulated [182].

Smith et al. [178] simulated self-replication in JohnyVon – an artificial physics with continuous space (see 4.3.1). JohnyVon supports self-replication of strands of "codons" (counterpart of genotype). The authors observed simple evolutionary behaviour, that shorter strands were selected since they can replicate faster than longer ones. The authors plan to introduce phenotypes (bodies) in order to achieve more interesting variants of selection.

In a very similar approach Hutton simulated self-replication in Squirm3 – an artificial chemistry environment (see 4.3.2)[7]. Self-replication occurs when the following reaction set is defined (Fig. 3.6):

$$\begin{array}{llllll} R1: & e8+e0 \to e4e3 & R4: & x3+y6 \to x2y3 & R7: & x2y8 \to x9y1 \\ R2: & x4y1 \to x2y5 & R5: & x7y3 \to x4y3 & R8: & x9y9 \to x8+y8 \\ R3: & x5+x0 \to x7x6 & R6: & f4f3 \to f8+f8 & & \end{array}$$

Results with population of self-replicating strands were similar to those obtained by Smith, as shorter molecules became dominant. In his next work, Hutton introduced rules that allow the emergence of simple membranes, thus self-replication occurs as a division of simple cells. As he noted: "the reproduction of the cells is robust enough to operate many times in a shared environment, leading to competition for resources and space. With a mechanism for mutations to appear, the cells are capable in theory of evolving better-adapted genomes; thus the system provides a framework for exploring the requirements for the evolutionary growth of complexity and evolvability." [78]

Ibáñez et al. discussed the *self-inspection* approach, in which the self-replicating automaton inspects itself and produces an exact copy of the results of its inspection [80].

Lee et al. developed a self-replicating machine in asynchronous cellular automata [104], which more closely resemble natural systems than synchronous models. Asynchronous cellular automata, considered in Lee's article, are probabilistic automata (see 2.2) with assumed spatial isotropy. Authors believes that this approach may be espe-

[7]Previous attempt to simulate self-reproduction within the context of artificial chemistry was performed by Ono and Ikegami [136, 137].

cially beneficial in the field of nanocomputers.

There is also ongoing research work on developing a system of automatic exploration of cellular automata space in order to find new self-replicating automata. In the nineties Lohn and Reggia introduced the use of genetic algorithms to discover automata rules that govern emergent self-replicating processes [111, 112]. The next works focused on using genetic programming rather than genetic algorithm “which produced larger, more rapidly replicating structures than past evolutionary models while requiring only a small fraction of the computational time needed in past similar studies” [138]. The latest results were published in 2010 by Pan and Reggia. As the authors noted: “the replication mechanisms discovered by genetic programming work quite differently than those of many past manually designed replicators: There is no identifiable instruction sequence or construction arm, the replicating structures generally translate and rotate as they reproduce, and they divide via a fissionlike process that involves highly parallel operations. This makes replication very fast, and one cannot identify which descendant is the parent and which is the child” [139].

3.3 Self-organization

Self-organization is a process in which the internal organization of a system increases in complexity. Comprehensive introduction to this topic can be found in: [132]. Various methods of measurements of complexity were presented by Adami [3], this issue was also discussed by Ludwig [115].

Within the context of ALife studies, self-organization is often recognized as the circumstances when the self-replicating structure may be generated spontaneously, as the result of random events. The issue resembles the following John Maynard Smith question from theoretical biology : “what features must be present in a system if it is to lead to indefinitely continuing evolutionary change?” (quoted in [78])

This topic is usually analysed in conjunction with self-replication (3.2), e.g. [71, 22, 77, 140] etc. A detailed description of an unsuccessful self-organization experiment, performed in the Universum environment, is presented in App. D.

Chapter 4

Simulation environments review

This chapter contains a short review of selected environments for simulating complex behaviour: classical agent-based models (4.1), core worlds (4.2) and virtual physics (4.3). Note, that in fact all environments mentioned in Chapt. 4 may be considered as belonging to ABMs, thus the division of this chapter is somewhat arbitrary. The chapter contains only abstract (but mostly real-life inspired) environments, more realistic ones, like e-Cell[1] are not considered.

4.1 Agent based models

Agent-based model (ABM) constitutes the fastest growing group of computer models. At the moment, there are several dozen ABM models in use (it is likely that the overall number of ABM environments exceeds hundreds). A comprehensive comparison of various models can be found in the following papers: [5, 133, 149, 190][2]. This section contains a quick review of the most common approach i.e. general-purpose tools mostly devoted from Swarm (4.1.1) and virtual ecosystems – environments closely related to the aim of this dissertation (4.1.2).

4.1.1 General purpose ABM

The most popular ABM is Swarm [125] which is a software package for multi-agent simulation of complex systems. Swarm was developed at the Santa Fe Institute: its users meet at an annual event SwarmFest. As explained on the Swarm website: "Swarm software comprises a set of code libraries which enable simulations of agent based models to be written in the Objective-C or Java computer languages. These libraries will work on a very wide range of computer platforms. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents: with this architecture, we can implement a large variety of agent based models."[3]. Gulyás et al. noted: "Swarm has a large, and constantly growing user community spanned all over the scientific disciplines, such as chemistry, economics, physics, anthropology, ecology, sociology and political science" [65].

[1]`http://www.e-cell.org/`

[2]Good source of information about several dozens models, though not peer reviewed, is provided by Wikipedia:
`http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software`

[3]`http://www.swarm.org/index.php/Introduction_to_Swarm` (accessed: April 3rd, 2010)

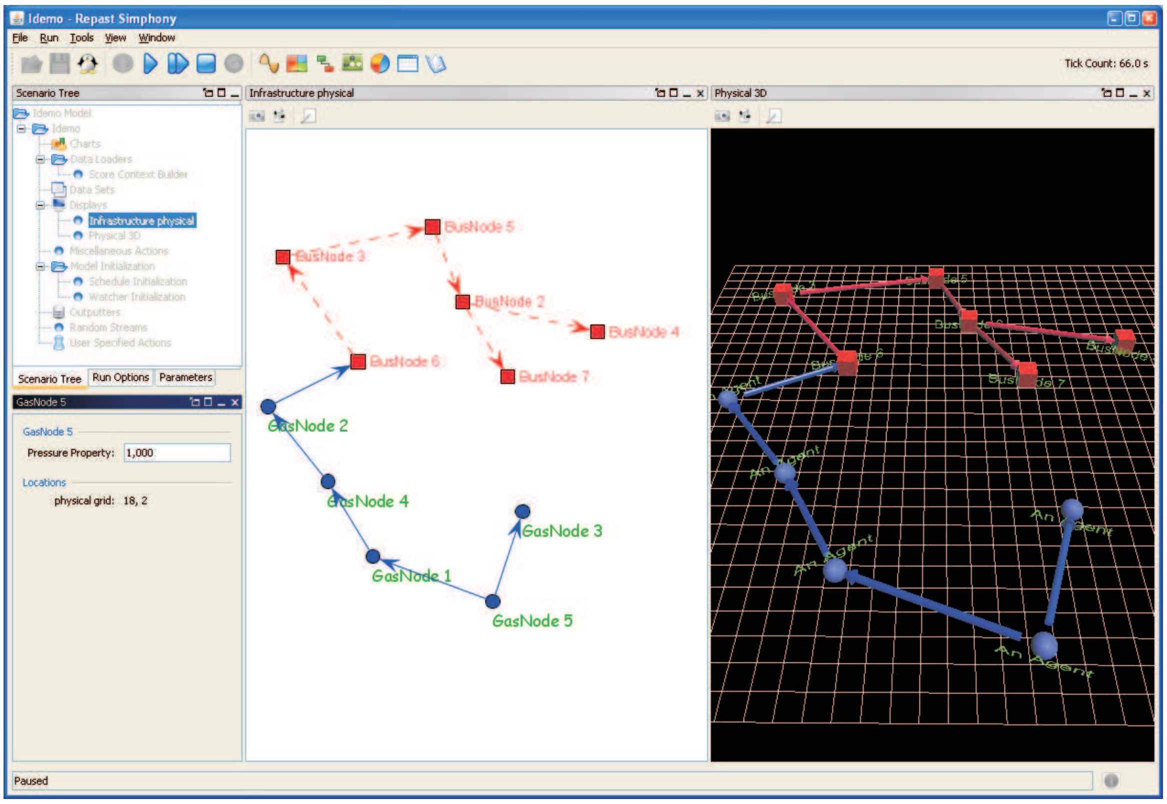


Figure 4.1: Visualisation in Repast Symphony. Picture downloaded from: `http://repast.sourceforge.net/img/runtime_example.PNG` (accessed: April 3rd, 2010)

A similar approach is presented in Repast Simphony [7] (the successor of Repast [31]) developed at the University of Chicago. Repast Simphony contains various libraries e.g.: for genetic algorithms, neural networks, regression, random number generation, specialised mathematics to be used in Java or Groovy computer languages. It also has support for easy modelling and visualisation of two- or three-dimensional environments (Fig. 4.1).

Ascape [82, 141] developed at the Brookings Institute is another tool similar to Swarm, but should be more straightforward to people with minimal technical knowledge. As explained on the Ascape website: "models can be developed in Ascape using far less code than in other tools. Ascape models are easier to explore, and profound changes to the models can be made with minimal code changes. Ascape offers a broad array of modeling and visualization tools. A high-level framework supports complex model design, while end-user tools make it possible for non-programmers to explore many aspects of model dynamics."[4].

There is also some effort focusing on developing high-level languages for ABM purposes, like Multi-Agent Modeling Language (MAML) [65], or Strictly Declarative Modeling Language (SDML) [131] etc.

Recently, solutions based on Logo programming language – like StarLogo or NetLogo [177], gained a large popularity, especially in educational application.

[4]`http://ascape.sourceforge.net/index.htm` (accessed: April 3rd, 2010)

4.1.2 Virtual ecosystems

Virtual ecosystem will be understood by the ABM simulation of environment inspired by the biological system, where a single agent represents the whole organism.

Nerve Garden

Nerve Garden is a representative group of educational rather than scientific programs. Its aim is to "provide a compelling experience of a virtual terrarium that exhibits properties of growth, decay, and energy transfer reminiscent of a simple ecosystem" [35]. The project is available via web browser with VRML support[5]. At the moment Nerve Garden does not support any kind of interactions between its components. The simulation of self-replicating objects is also impossible. The authors plan to develop the project in this direction.

Gene Pool

GenePool[193][6] is a simulation of hundreds of simple organisms called "swimbots" residing in water. The most attractive swimbots, with better swimming skills, have the greater opportunity to mate and to propagate its genetic information. Despite its simplicity, GenePool allows to observe the emergence of local gene pools which constitutes a kind of "species".

LifeDrop

LifeDrop "is a virtual 3D ecosystem simulating a drop of water inhabited by small creatures. Theses creatures are inspired by the Biomorphs created by Richard Dawkins in the Blind Watchmaker. Each Biomorph is acting as an autonomous agent with its own genotype, metabolism, morphology and behaviors. A biomorph is the result of the development process of its DNA in the environment. Then, the success of that phenotype determines whether or not the genes that it bears shall go forward to the next generation. There is no female or male, but two biomorphs can mate to create new biomorphs. Any individual born, therefore, inherits genes that have succeeded in building a long series of successful phenotypes."[7].

SodaPlay

SodaPlay "demonstrates a great variety of form and motion using 2D graphics, in an entertaining format. SodaPlay uses a more molecular style of physics modeling, based on spring forces, to affect the positions and orientations of potentially large-scale spring structures having semicoherent positions and orientations" [193, p. 95]

Evolve 4.0

Evolve 4.0[8] simulates a population of programs residing on a two-dimensional grid. Each cell executes a custom language called KFORTH (portmanteau of "Ken's Forth"), based on the Forth programming language, with the set of high-level commands

[5]At the website: `http://www.biota.org`

[6]`http://www.swimbots.com/`

[7]`http://www.virtual-worlds.net/projects/lifedrop.htm`

[8]`http://www.stauffercom.com/evolve4/`

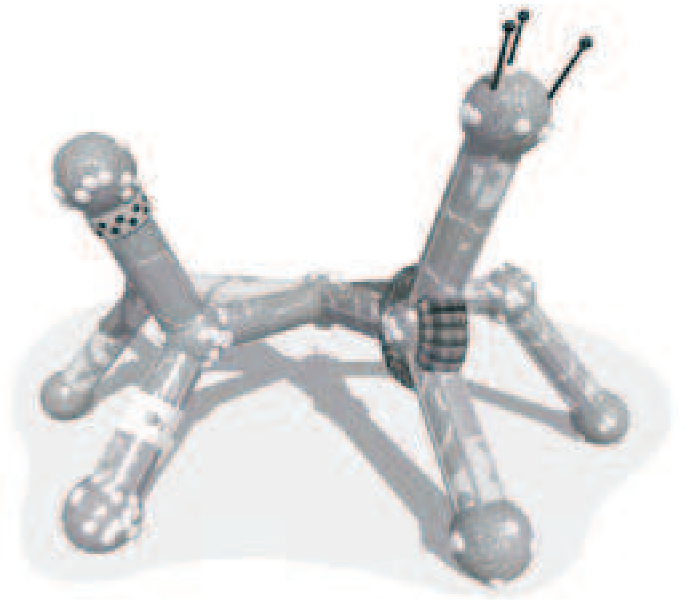


Figure 4.2: Visualisation of Framstick agent [97, Fig. 2.4]

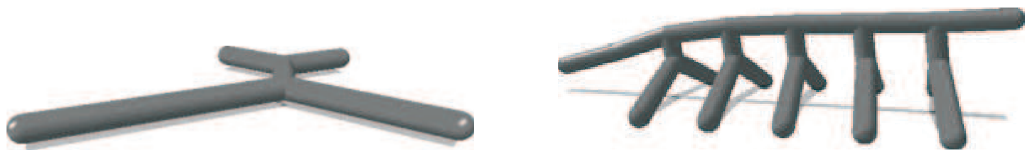


Figure 4.3: Phenotypic interpretation of genotype in Framsticks – left: `XXX(XX,X(X,X))`, right: `rr<X>#55<,<X>RR< <llX>LX>LX> >X` [97, Fig. 2.6].

responsible for the program's activities. The programs can grow, move, eat, reproduce, and eventually evolve over time (algorithms responsible for evolution are embedded into the simulator).

Framstick

Framstick [97] is one of the best known virtual environments for simulating three dimensional agents in virtual environment, inspired by the pioneering work of Karl Sims [175].

The Framstick's world can be both flat or built of blocks. The boundaries can be wrapped or surrounded by walls.

The agents consist of body and brain. The body is made of flexibly joined parts (Fig. 4.2. Each part and joint has the following properties: position, orientation, weight and friction, it is also possible to assign some custom properties: ability to assimilate energy, durability of joints in collisions etc. The body is simulated using the finite element method.

The brain of the agent consists of a simple neural network. The network can have any kind of topology or complexity i.e. neurons may be connected with each other in any way. The environment allows to define types of neurons used in agents, with some predefined types. The brain communicates with the body via three receptors (equilibrium sense, touch sense, smell – detection of energy) and two effectors (for bending and rotating muscles).

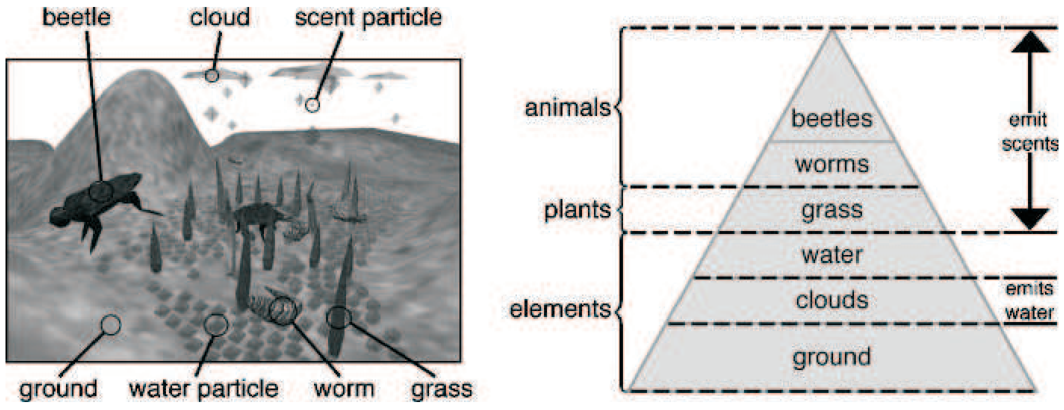


Figure 4.4: The artificial ecosystem: a snapshot on the left, and an ecological pyramid on the right.[159, Fig. 1.1]

Each property of the agent is encoded in the agent's "genotype" [9]. Framstick supports different ways of encoding an agent's genotype – as so-called "genotype languages". Examples are presented in Fig. 4.3.

The environment allows performing various simulations, of both directed and open ended evolution. A number of interesting experiments regarding emergence of complex cognition forms is presented in [116]. Komosiński noted: "in the virtual life lab at the Utrecht University, Framsticks is used for experiments related to evolutionary robotics, animal locomotion, distributed intelligence, artificial ecosystems, biosemiotics, and sexual and natural selection" [97, p. 60].

Rönkkö's artificial ecosystem

Rönkkö's ecosystem was described in [159]. The ecosystem is modelled using a single, uniform, and formalised particle system. The particle system consists of a fixed and finite number of particles. Each particle has the same shape and constant unit mass. The dynamics of the system is governed by four primitive laws: motion (particle's position update), forces (description of reactions, bonds and collisions, energy absorption (transfer of energy from one particle to another) and memory (specific information stored in the particle). Rönkkö's model has much in common with systems reviewed in Sect. 4.3 (virtual physics).

The simulated ecosystem (Fig. 4.1.2) consists of:

1. Ground: 65536 particles forming an uneven landscape (256×256 grid). Ground particles are fixed and immobile.

2. Clouds: A cloud is modelled by a single particle (visualisation, like in Fig. 4.1.2, shows the cloud as a larger object). Each cloud is fixed and immobile, the cloud emits at most one water particle within one iteration round.

3. Water: water is modelled as a collection of particles. Each water belongs to exactly one cloud, the water particles are distributed evenly underneath the cloud. Water may collide with any other particle excluding cloud and scent. Water is absorbed by a root particle of a blade of grass (the initial level of energy is set to 1)

[9]Note, that in the biological systems genotype does not strictly encode every property of body, as exact topology of neural network or size of body is a result of adaptation to changing environmental conditions.

4. Grass: grass consists of blades. A blade consists of 6 particles, two of them have special meaning: root and emitter. The root particle absorbs energy from water, and the emitter emits scent particles (in the same manner as water particles are allocated to clouds). The grass may collide with any other particle excluding the cloud and scent. The worm absorbs energy from the grass (the initial level of energy of each particle is set to 200). When the whole energy is lost, the grass is considered to be eliminated.

5. Worms: A worm is a string of 10 particles, two of them have special meaning: head and emitter. The head particle is used for movement and for absorbing energy. The emitter acts as one in the grass. The worm may collide with any other particle excluding the cloud and scent Worm recognizes scent emitted by the grass (attractive) and beetles (repulsive). The worm loses its energy whenever it moves and if it comes too close to a beetle (the initial level of energy of each particle is set to 200). When the whole energy is lost, the worm is considered to be eliminated.

6. Beetles: A beetle consists of 58 particles with a distinct head particle, an emitter particle, and wing muscle particles. The head particle of a beetle is used for absorbing energy from worms, the emitter particle acts as the one in the worm or the grass, and the wing muscle particles are used for jumping. The beetle loses its energy whenever it jumps (the initial level of energy of each particle is set to 200). When the whole energy is lost, the beetle is considered to be eliminated.

7. Scents: A scent is modelled using a single particle. A scent particle has the initial level of energy set to 1. Its energy is absorbed whenever it is sensed (the particle is eliminated).

Rönkkö described various interesting simulations in the virtual ecosystem: spontaneous formation of rivers and ponds, growth of grass, finding edible grass by worm or hunting dynamics (one beetle and three worms, and many worms and beetles; in both cases all worms have been eliminated).

4.2 Core worlds

The history of core worlds was shortly described in Sect. 3.2.3. The original works of Rasmussen [152], Ray [154] and Adami [2] led to the development of numerous similar environments, like one of Davidge [36], De Dinechin [38], Pargellis [140], Skipper [176] or Taylor [186].

4.2.1 Avida

Avida is the most popular Core World since the nineties and is still under active development at the Michigan State University and at the California Institute of Technology. It may be true to say, that Avida is also the most important and the most influential tool in the whole field of Artificial Life (see 3.1). Papers regarding this environment were published, among many others, in Nature [105, 106, 199] and Science [24]. A description of the system can be found in: [2, 3, 134].

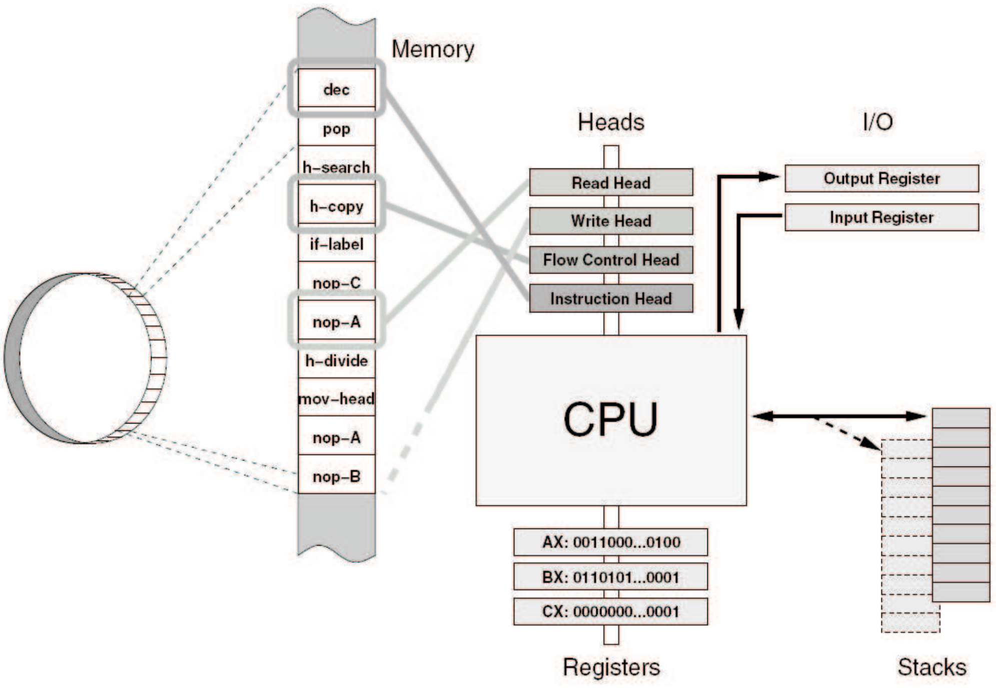


Figure 4.5: Avida: CPU, registers, stacks, heads, memory and I/O functionality. [134, Fig. 1.1]

Simulated world

Avida simulates fixed number of N cells. Each cell can by occupied by exactly one program. Cells may be organised as a 2D grid with Moore neighbourhood (each cell has 8 neighbours) or a fully connected topology (each cell is a neighbour to every other cell) [10]. After performing self-replication, a new program will replace either the parent cell or another cell from the neighbourhood. The most common strategies are: replace random and replace oldest.

The programs existing in each cell are run in within the context of its virtual hardware (see below). Avida simply executes one instruction from each of its embedded programs. In the simplest simulations, all programs are executing at the same speed, which means that during each environment time cycle the same number of commands of each program would be executed (30 instructions by default). It is also possible to differentiate the speed of each program, according to the “merit” value associated with them.

Virtual hardware

Avida consists of a virtual operating system, depicted in Fig. 4.5. The main part of the machine is the virtual central processor unit (CPU) which executes program code (i.e. each instruction in genome in Avida terms). Instructions operate on three registers: AX, BX and CX, which can store a single 32-bit number. Program data can also be stored on one of the two stacks. The memory is initialised with programs code (organism genomes). Instructions are processed sequentially, but because memory is organised in a circular-fashion, the CPU after executing the last instruction will loop back to the beginning and execute again.

[10]It is possible to freely define new kind of topology by explicitly list each cell’s neighbours.

Four components called heads are pointers to locations in the memory. The instruction pointer is the only one that has a counterpart in the classical processor (identifies the instruction being executed at the moment). The read head, and the write head are used in executing the self-replication process. The read head is a pointer to the instructor being read, and the write head is a pointer to the memory where the copied instructions are being written. The flow control head helps to execute jumps and loops (described below).

The last elements of the virtual hardware – input/output registers are used for interaction with the environment. Each program can read data from its input register, perform some computations and then write results to the output.

Software

Programs may consists of hundreds of different instructions, but usually a small subset is used.

An important concept of Avida's language is one of template matching. This is a method of indirect addressing in order to achieve more a robust code[11]. The template matching works as follows: every instruction that needs reference to another position in memory reads its subsequent `nop` instructions, that forms the definition of the template (a template will end with the first instruction other that `nop`). There are three type of `nop`: `nop-a`, `nop-b` and `nop-c`. The instructions are circularly complementary i.e. `nop-a` is complementary to `nop-b`, `nop-b` to `nop-c` and `nop-c` to `nop-a`. The referenced point in memory is defined with the template complementary to the one following the instruction (e.g. for template: `nop-a`, `nop-c`) it would be the first occurrence of the following sequence: `nop-b`, `nop-a`).

Any jump operates in two steps. First, the instruction `h-search` is used to position the flow control head at the desired position in memory (as it searches through complementary templates). In the second step, the instruction `mov-head` perform the jump into the point stored in the flow control head. As an example, in the following program the executed instructions would be: 000 to 010 and 020:

```
000 h-search ; set flow control head
001 nop-a    ; template
002 nop-c    ;
    ...      ; other instructions...
010 mov-head ; jump (to line 201)
    ...      ; instructions to be skipped
018 nop-b    ; complementary template
019 nop-c    ;
020 pop      ; instruction executed after mov-head
```

Self-replication involves three instructions. Initially, the program needs to allocate the memory with `h-alloc` command. The allocation always takes place in the program's local memory, thus it expands the memory between the last and the first instruction of the program (although the memory is considered to be circular, the Avida environment will not execute instruction from just the allocated memory, as it has knowledge of which instruction is the beginning of the program, and which is the end).

[11]Compare to problems with the similar construction in the Universum environment: App D.

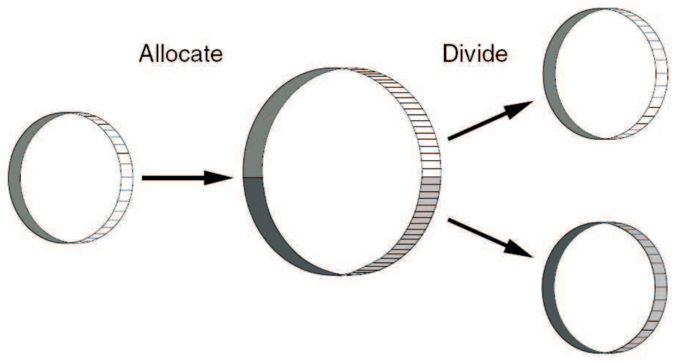


Figure 4.6: Self-replication in Avida. Memory is allocated, program executes sequence copying instruction then program splits into two independent parts [134, Fig. 1.4].

After the allocation, the program can perform the copying process via `h-copy`. This instruction reads one instruction indicated by the read head, and writes it into the place indicated by the write head. In order to achieve its perfect copy, the program needs to initialise its read and write heads, and call `h-copy` for the proper number of times.

The last instruction is `h-divide`, which splits the program between the read head and the write head. If the process has been properly programmed, the parent program should be divided into two exact parts, as presented in Fig. 4.6.

The commands used in self-replication may work erroneously with some predefined probability. E.g. the instruction `h-copy` may write a random instruction to the position of the write head instead of the one indicated by read head. The command `h-divide` may randomly insert instruction into the offspring code or delete randomly chosen of one of its instruction. Avida alsp allows for the occurrence of point (cosmic-ray like) mutations that affect also "living" programs as well. The last source of diversity are "implicit mutations" that are caused by errors in the self-replication algorithm executed directly by the program.

Usually programs compete for limited space and do not interact directly. However, it is possible to enable the `inject` that works similarly to `h-divide` with the exception that the new offspring is inserted into the memory of another program. Such a "parasite", does not harm the attacked program directly, but reduces its efficiency as it consumes its CPU cycles.

The authors are planning to extend Avida with more sophisticated program interactions – like detecting resources, or direct communication between programs [134].

4.3 Virtual physics

The term "virtual physics" was taken from work [178]. In this section three low-level environments will be briefly described: JohnyVon (4.3.1), Squirm3 (4.3.2) and Universum (4.3.3). Their common feature is the apparent similarity to the molecular dynamics method (2.1)[12].

[12]Note, that only purely abstract environment are considered in this section.

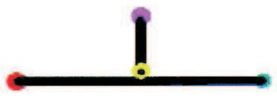
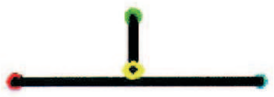
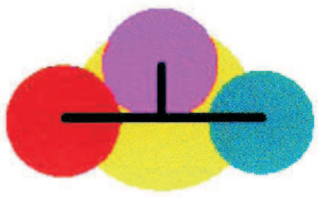
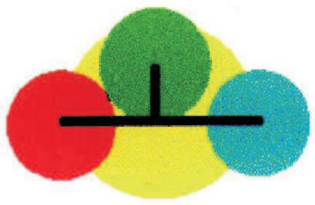
	Type 0 codons (purple codons)	Type 1 codons (green codons)
All fields in their small states		
All fields in their large states		

Figure 4.7: The two types of codons and their field states in JohnyVon. The fields of an unbound codon are always small. The fields become large only when codons bond together. A codon's fields may be in a mixture of states (one field may be small and another large). Note that codons are asymmetric. A black line in a codon that begins in the middle of the codon and ends at the center of a colored field is called *arm* [178, Tab. 1].

4.3.1 JohnyVon

The description of JohnyVon can be found in [178]. The environment simulates the behaviour of T-shaped objects called *codons* (Fig. 4.7) in 2-dimensional continuous space. The authors claim, that due to realistic physics, JohnyVon may have serious implications for research in nanotechnology, theoretical biology, and artificial life. An example of experiment was described on page 26.

Codons and fields

Each codon encodes 1 bit of information, as there are two types of them: 0 and 1. Codons have an attractive or repulsive area associated with it called *fields*. There are five types of fields, which are named according to their colours: purple, green, blue, red, yellow (Fig. 4.7). Fields have two states: small and large. The state is switched as bonds are formed and broken (e.g. small red becomes large red if it is touched by a small blue field and if the arms of their respective codons are aligned linearly). Fields behave somewhat like springs, they can pull codons together, push them apart, or twist them to align their arms (e.g. if two large yellow fields intersect, a repulsive force pushes them apart).

Physics

The environment is synchronised by a discrete clock. Each codon has one associated one unit of mass. Collisions occurs when two codons approaching at a distance less than 1. Codons move in a virtual liquid, a small random value is added to the codon's velocities at each time step to simulate Browniam motion. Liquid viscosity is simulated via multiplying velocities by a fractional factor.

When two codons are bonded together, the *straightening force* twists them to align their bonded arms linearly. *Attractive force* is also exerting which pulls their bonded arms together. The force acts similar to a spring – the strength increases linearly with the distance between the tips (unless the distance is greater than the sum of the radii of the fields).

Between yellow fields there occurs the *repulsive force*, which also act as a spring pushing the codons apart. Contrary to the attractive force, the strength decreases linearly with the distance (unless the distance is greater than the sum of the radii of the fields).

4.3.2 Squirm3

Squirm3 is an abstract artificial chemistry environment (2.3), developed by Tim Hutton [77, 78]. The environment consists of a 2-dimensional discrete lattice. Events are synchronised by a discrete clock.

Components

The basic environment components are called *atoms*. Each atom is of 6 fixed *types* ($type \in \{a, b, c, d, e, f\}$) and has a switchable *state* ($state \in \{0, 1, 2, \ldots\}$). The atoms may be *bonded* due to *reactions*. A *molecule* consists of two or more atoms connected by bonds.

Physics

Each atom occupies exactly one square in the lattice. The atoms are moved around at random – at each time step, each atom may stay remain still or move to one of its neighbour cell (if the cell is empty). Movement is possible if the atom remains within the neighbourhood of every atom it is bonded with. An atom may react with its neighbouring atom.

Chemistry

Squirm3 allows for an arbitrary definition of reaction. As an example, the following reaction: $e8 + e0 \rightarrow e4e3$, means that an atom of type e and state 8 would react with an atom of type e and state 0 forming a bond between them and changing their states to 4 and 3 respectively.

4.3.3 Universum

The full environment description can be found in [83]. Various experiments were presented in a series of papers e.g. [85, 86] etc. An example of an experiment is presented in Fig. 3.5 on page 25.

The Universe is defined over a two-dimensional lattice of identical squares. A square can be empty or contain any number of elements which we called *atoms* and *photons*. The atoms are of two types: 0 and 1, and they are permanent elements of the environment – they are not created nor annihilated during its evolution. The photons are temporary elements, created by a reaction which dissipate energy.

The atoms occupying the same square always forms a one-dimensional string called*particle*. The properties of a particle are fully determined by its constituents

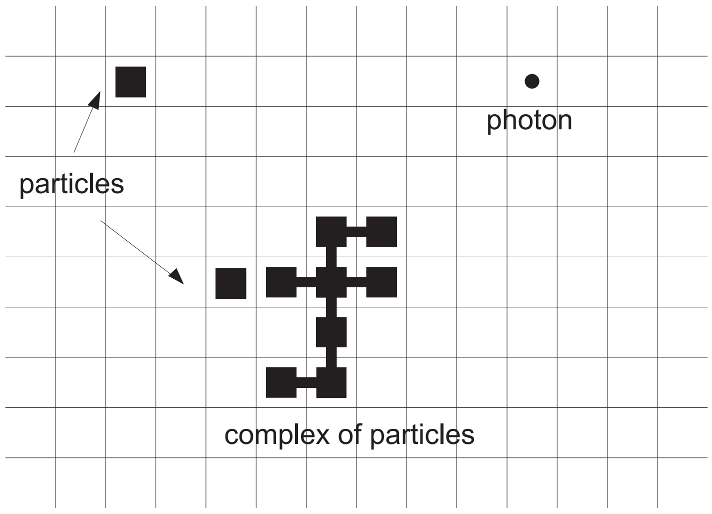


Figure 4.8: An example of the Universum space with particles, complexes and photons

atoms, its velocity, internal energy, orientation, and the possible bonds with adjacent particles. The permanence of a particle depends on its bond energy, which is sum of the bond energies between its neighbour atoms.

The particles occupying adjacent squares can bond together forming a *complex* of particles. Permanence of the complex depends on its bond energy which is sum of the bond energies between its consisting particles.

An example of the Universum space with particles, complexes and photons is presented in Fig. 4.8.

Particles, complexes and photons can move by jumping to adjacent squares. Moves, just like any other transformations, are synchronised by a discrete clock.

Movements and collision

During each time step, particles and complexes can move by jumping randomly to adjoining squares in the x and y directions with probabilities proportional to their velocities in respective directions, v_x and v_y. Such a rule produces random trajectories distributed around the straight lines, which represents the deterministic continuous movement with velocity (v_x, v_y).

Photons move in a similar way, by jumping to adjoining squares in the x and y directions with probabilities: $|\cos\varphi|, |\sin\varphi|$, where φ is a photon direction.

When two particles (unbounded or belonging to different complexes) occupy the adjoining squares or when a photon moves into the square occupied by a particle, a collision occurs and the new state of environment is evaluated.

There are two types of collisions between particles: elastic and inelastic ones. The elastic collision simply changes the participating particles velocity preserving overall kinetic energy and overall momentum. The inelastic collision equalises the velocities of particles preserving only the particles' momentum. The overall environment energy is, however, preserved because of the creation of a new photon having energy equal to particles kinetic energy loss.

Collision between photons and particles can be divided into following types: elastic and inelastic. Elastic collision changes photon direction only. The inelastic collision is classified into the following sub-types:

- rebounding of the hit particle from an adjoining particle.

```
  SHD N ; (Shift pointer)  shift pointer into the adjoining square
        ; in N direction
  PS 1  ; (Put Stone) put the stone number 1 in the square
        ; indicated by the pointer
  SHC   ; (Shift To Corner) shift pointer into the upper left
        ; corner of the activity area Ω
  SCN   ; (Scan) Move the pointer one square in direction E. If the
        ; movement causes crossing of the right boundary of the
        ; program activity area Ω, move the pointer to the leftmost
        ; column in the next row. If before movement the pointer
        ; is in the bottom right square of Ω, terminate the
        ; execution of the program
  CS 0, 1110 ; (Compare Substring) Compare the substring of atoms
        ; beginning at atom 1 (first argument + 1) in the particle
        ; indicated by the pointer with the second argument (1110)
  JF -2 ; (Jump on false) If last comparison failed, go back
        ; 2 commands - to the SCN command
  MOS 1 ; (Move particle to stone) Move the particle indicated
        ; by the pointer, and the pointer itself, into the
        ; square marked by stone 1. If the particle
        ; is a part of a complex, move the whole complex
  SEL S ; (Set link) Set the bond between the particle indicated
        ; by the pointer and the particle occupying the adjoining
        ; square in direction S (if both exist)
```

Listing 4.1: Example of program in the Universum environment.

- setting a bond between the hit particle and the adjoining particle.
- resetting the bond between the hit particle and the adjoining particle.
- changing the order of atoms in the hit particle.
- concatenation of the hit particle and the adjoining particle.
- splitting of the hit particle into two particles.
- increasing internal energy of the hit particle.

Functional interaction

Beside the reactions taking place during collisions of particles and photons, there also exists another class of interaction, in which particles having their individual structure can selectively transform and move other particles in the space around them, according to the function encoded by their atoms. This kind of interaction helps to model various kinds of self-organization or self-modification processes, which were impossible or very difficult to model using molecular dynamics,lattice gas automata or cellular automata methods.

A string of atoms forming a particle is interpreted as a program written in a specially defined assembly-like language. The string is divided into five-bit portions which are mapped into commands (a detailed description of the language and a full list of commands can be found in: [83]).

Functions encoded in particles can recognize the structure of particles and complexes. They can move, split, change bonds, concatenate, change the order of atoms (which can change other particle function) and the orientation of affected particles and complexes.

As an example [83], the following sequence of atoms in a particle: 01100 00010 00001 00000 00001 01111 00000 00111 00000 00000 10100 11100 01010 10000 00010 10001 00000 00001 10010 00000 10101 111 defines the function which moves and attaches to the particles‘s north side a particle which first atoms are 1110. The sequence forms the program depicted in List. 4.1.

Function commands operate on absolute and relative directions. Absolute directions: N, S, W, E describes squares which are north, south west and east of the particle respectively. Relative directions: U (up), D (down), L (left), R (right) are mapped into absolute direction according to the particle‘s orientation, i.e. a particle with orientation N, maps relative direction: U$\leftrightarrow$ N, D$\leftrightarrow$ S, L$\leftrightarrow$ W, R$\leftrightarrow$ E; particle with orientation S maps: U$\leftrightarrow$ S, D$\leftrightarrow$ N, L$\leftrightarrow$ E, R$\leftrightarrow$ W, etc.

Particle orientation can be changed by other particle‘s function. It can also can be changed randomly at the end of each simulation step with preset probability.

The area of the functional activity of a particle is a square region centred around the particle. In a given time step, the functions of particles are activated in a randomly chosen succession. A function activated later may affect the area, which was transformed by functions activated earlier.

The transformation induced by the particle function obeys the momentum and energy conservation laws. There are two sources of energy for the function’s realisation: changing the bonds between participating entities (atoms or particles), and the particle‘s internal energy. If total energy change is positive, a photon is created and changes are saved; otherwise all changes are withdrawn.

Universum has been successfully used for simulations of such physical processes such as diffusion, cluster formation, chemical reaction kinetic, predator-prey system, growth of various complex structures and self-replication processes.

Chapter 5

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 (listed in 3.1.1) 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. 5.1 the first level of interaction (simplified physics) has been described. Sect. 5.2 contains a description of the embedded language. The last section 5.3 contains a discussion of environmental assumptions. The documentation and the results of experiments are included in: [87, 88, 89, 96, 168, 169, 170, 172, 174].

5.1 First level: "Thermodynamics"

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

5.1.1 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]The possible setting are presented in App. A

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 5.2). 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. 5.1.

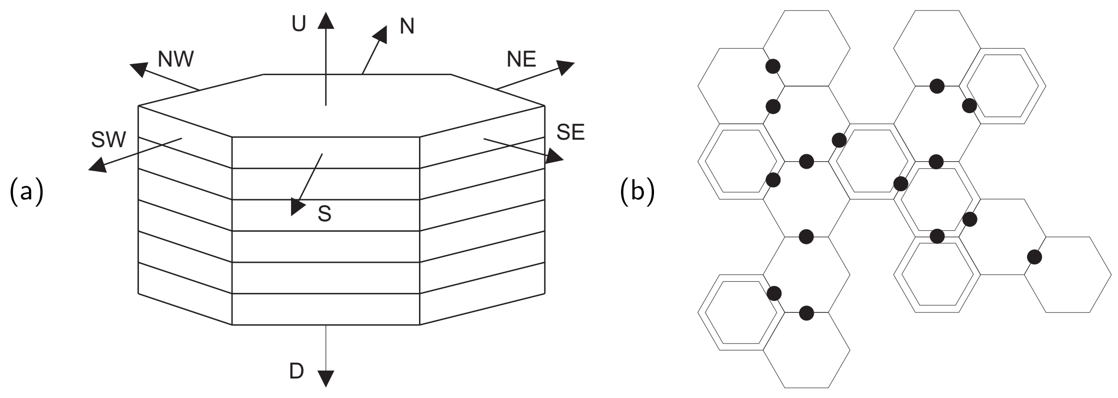


Figure 5.1: Examples of complexes: (a) the horizontal view of single stack of particles with directions shown, and (b) the vertical view of complex formed by horizontal bonds, where hexagons drawn by single lines represent single particles, and by double lines represent stacks of particles and black 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 $R\sqrt{3}$.*

[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:

$$\boldsymbol{s}(i+1) = \boldsymbol{s}(i) + \boldsymbol{v}T \tag{5.1}$$

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:

$$v_1(i+1) = \frac{v_1(i)(M_1 - M_2) + 2M_2 v_2(i)}{M_1 + M_2} \tag{5.2}$$

$$v_2(i+1) = \frac{v_2(i)(M_2 - M_1) + 2M_1 v_1(i)}{M_1 + M_2} \tag{5.3}$$

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:

$$\boldsymbol{v}_1(i+1) = \boldsymbol{v}_2(i+1) = \frac{M_1 \boldsymbol{v}_1(i) + M_2 \boldsymbol{v}_2(i)}{M_1 + M_2} \tag{5.4}$$

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 5.1.2) is generated with the following energy:

$$E_{ph}(i+1) = \frac{M_1 M_2}{2(M_1 + M_2)} |\boldsymbol{v}_1(i) - \boldsymbol{v}_2(i)|^2 \tag{5.5}$$

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

5.1.2 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:

$$\boldsymbol{s}_{ph}(i+1) = \boldsymbol{s}_{ph}(i) + c(\boldsymbol{i}\cos\alpha + \boldsymbol{j}\sin\alpha) \tag{5.6}$$

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:

$$v_{1x}(i+1) = \frac{p_x \pm M_2\sqrt{2\beta M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \quad (5.7)$$

$$v_{1y}(i+1) = \frac{p_y \mp M_2\sqrt{2(1-\beta) M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \quad (5.8)$$

$$v_{2x}(i+1) = \frac{p_x \mp M_1\sqrt{2\beta M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \quad (5.9)$$

$$v_{2y}(i+1) = \frac{p_y \pm M_1\sqrt{2(1-\beta) M_r E_{ph} + (v_{1x}(i) - v_{2x}(i))^2}}{M_1 + M_2} \quad (5.10)$$

   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 (5.4). Besides, new photon energy is:

$$E_{ph}(i+1) = E_{ph}(i) - E_{pb}(t, t') + \frac{M_1 M_2}{2(M_1 + M_2)} |\boldsymbol{v}_1(i) - \boldsymbol{v}_2(i)|^2 \quad (5.11)$$

   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:

$$\boldsymbol{s_1}(i+1) = \boldsymbol{s_1}(i) \tag{5.12}$$

$$\boldsymbol{s_2}(i+1) = \begin{cases} \boldsymbol{i}s_{1x}(i) + \boldsymbol{j}(s_{1y}(i) + R\sqrt{3}) & \text{if } dir = \mathsf{N} \\ \boldsymbol{i}(s_{1x}(i) + 1.5R) + \boldsymbol{j}(s_{1y}(i) + 0.5R\sqrt{3}) & \text{if } dir = \mathsf{NE} \\ \boldsymbol{i}(s_{1x}(i) + 1.5R) + \boldsymbol{j}(s_{1y}(i) - 0.5R\sqrt{3}) & \text{if } dir = \mathsf{SE} \\ \boldsymbol{i}s_{1x}(i) + \boldsymbol{j}(s_{1y}(i) - R\sqrt{3}) & \text{if } dir = \mathsf{S} \\ \boldsymbol{i}(s_{1x}(i) - 1.5R) + \boldsymbol{j}(s_{1y}(i) - 0.5R\sqrt{3}) & \text{if } dir = \mathsf{SW} \\ \boldsymbol{i}(s_{1x}(i) - 1.5R) + \boldsymbol{j}(s_{1y}(i) + 0.5R\sqrt{3}) & \text{if } dir = \mathsf{NW} \end{cases} \tag{5.13}$$

where *dir* is calculated according to:

$$dir = \begin{cases} \mathsf{N} & \text{if } \varphi \geq -\frac{\pi}{6} \text{ and } \varphi < \frac{\pi}{6} \\ \mathsf{NE} & \text{if } \varphi \geq \frac{\pi}{6} \text{ and } \varphi < \frac{\pi}{2} \\ \mathsf{SE} & \text{if } \varphi \geq \frac{\pi}{2} \text{ and } \varphi < \frac{5\pi}{6} \\ \mathsf{S} & \text{if } \varphi \geq \frac{5\pi}{6} \text{ or } \varphi < -\frac{5\pi}{6} \\ \mathsf{SW} & \text{if } \varphi \geq -\frac{5\pi}{6} \text{ and } \varphi < -\frac{\pi}{2} \\ \mathsf{NW} & \text{if } \varphi \geq -\frac{\pi}{2} \text{ and } \varphi < -\frac{\pi}{6} \end{cases} \qquad \varphi = \arctan\frac{s_{2x} - s_{1x}}{s_{2y} - s_{1y}} \tag{5.14}$$

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:

$$\boldsymbol{s_1}(i+1) = \boldsymbol{s_2}(i+1) = \boldsymbol{s_1}(i) \tag{5.15}$$

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:

$$E_{ph}(i+1) = E_{ph}(i) + E_{pb}(t, t') \tag{5.16}$$

   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 (5.13), 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 (5.13) is tried.

6. Photon absorption. The photon is absorbed by the hit particle, and is converted into its internal energy:

$$E_i(i+1) = E_i(i) + E_{ph} \tag{5.17}$$

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).

5.2 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.

5.2.1 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 5.2.2).

Language syntax in BNF notation is presented in Fig. 5.2 (see also: 1.3). An example of a program is presented in List. 5.1 – the program recognizes the structure presented in Fig. 5.3.

Program structure forms a tree with a root element `program()`. The tree representation of the example from List. 5.1 is presented in Fig. 5.4.

5.2.2 Semantics

Program execution is based on the Prolog backtracking algorithm (e.g. [29, 84]).

```
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 5.2: 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 `V`variables) 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 5.1: Example of a program recognising the structure shown in Fig. 5.3.

- *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. 5.5.

   Depending on its final syntax, the predicate is able to (for more details see 5.2.3):

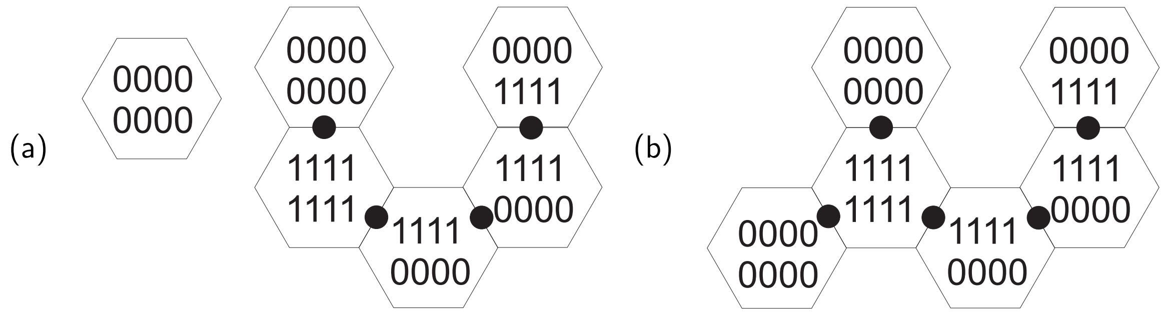


Figure 5.3: Single particle and a complex of particles recognized by the program presented in Fig. 5.1 (a), and the structure after action of the program (b).

(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.:

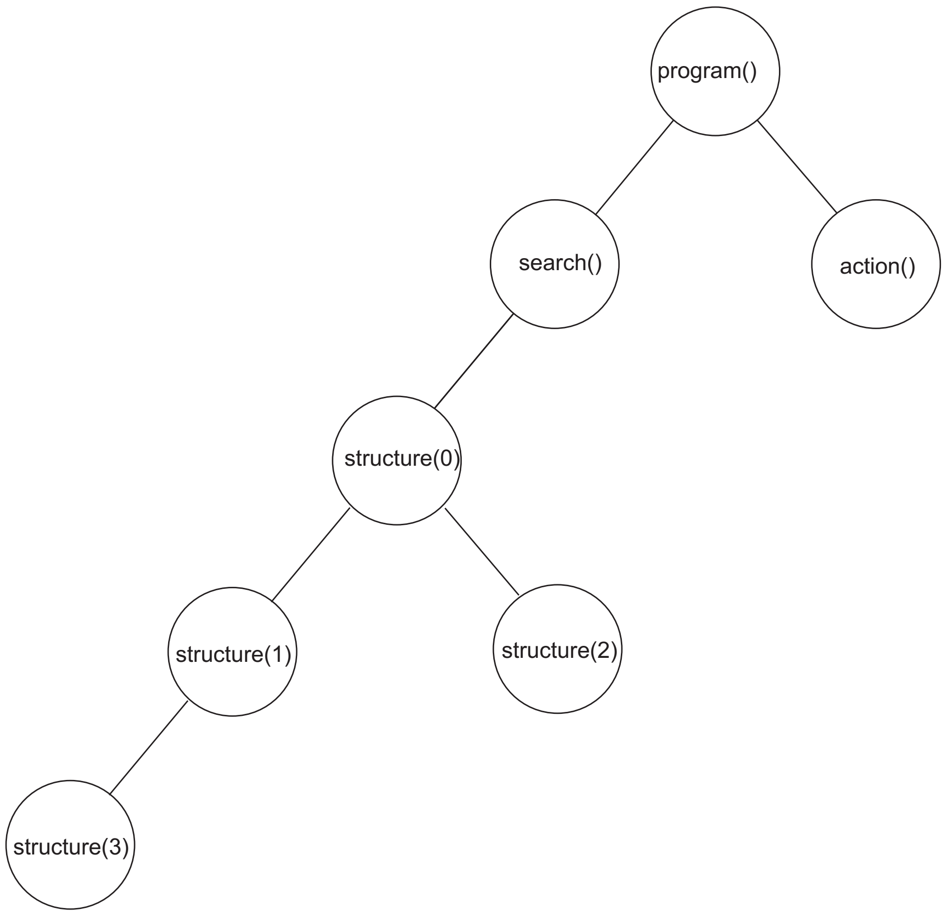


Figure 5.4: Example of a tree structure. The predicates: `exists` and `bind` are omitted.

   i. `exists 11110000 bound in N` – find a particle of type 11110000 which is bound in direction `N` (note: see also 5.2.4 on page 61).
   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 5.2.4 on page 61).

(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 is adjacent` – find a particle of type 11110000
   ii. `exists not 11110000 is 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:

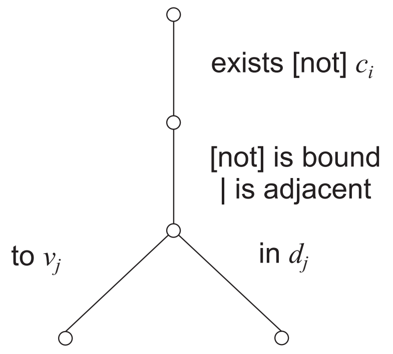


Figure 5.5: Hierarchy

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

i. `exists 11110000 is adjacent in N` – find a particle of type 11110000.
ii. `exists not 11110000 is 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 (5.13) 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 (5.13) 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 5.2.4 on page 61) 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:

$$\boldsymbol{s}_{CM} = \frac{\sum_{i=1}^{\mathsf{N}} \boldsymbol{s}_i m_i}{\sum_{i=1}^{\mathsf{N}} m_i} \tag{5.18}$$

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 5.13 (page 47) 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 44). 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 44). 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 5.18).

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 44). 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 5.2.4).

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**

5.2.3 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 (5.1.1).

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 51). 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 51). 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. C on page 129
`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 5.1: 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 (5.2.2) 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(102,1,1,1,1,1,1,1,1,101,0,103,0,0,0,0,0,28.80,22.09,0.4,0.2,1).
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 5.2: 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. C.

Tab. 5.1 contains a summary of translation rules. An example of a translated program from List. 5.1 is presented in List. 5.2.

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.

5.2.4 Interpreter

After the photons movements (see 5.1.2), every complex in the environment is processed. If the complex encodes one or more programmes (see 5.2.5), 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 5.2.5) 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 (5.1.1). 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 $\rightarrow$ NE, NE $\rightarrow$ SE, SE$\rightarrow$ S, S$\rightarrow$ SW, SW $\rightarrow$ NW, NW $\rightarrow$ 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. 5.6.

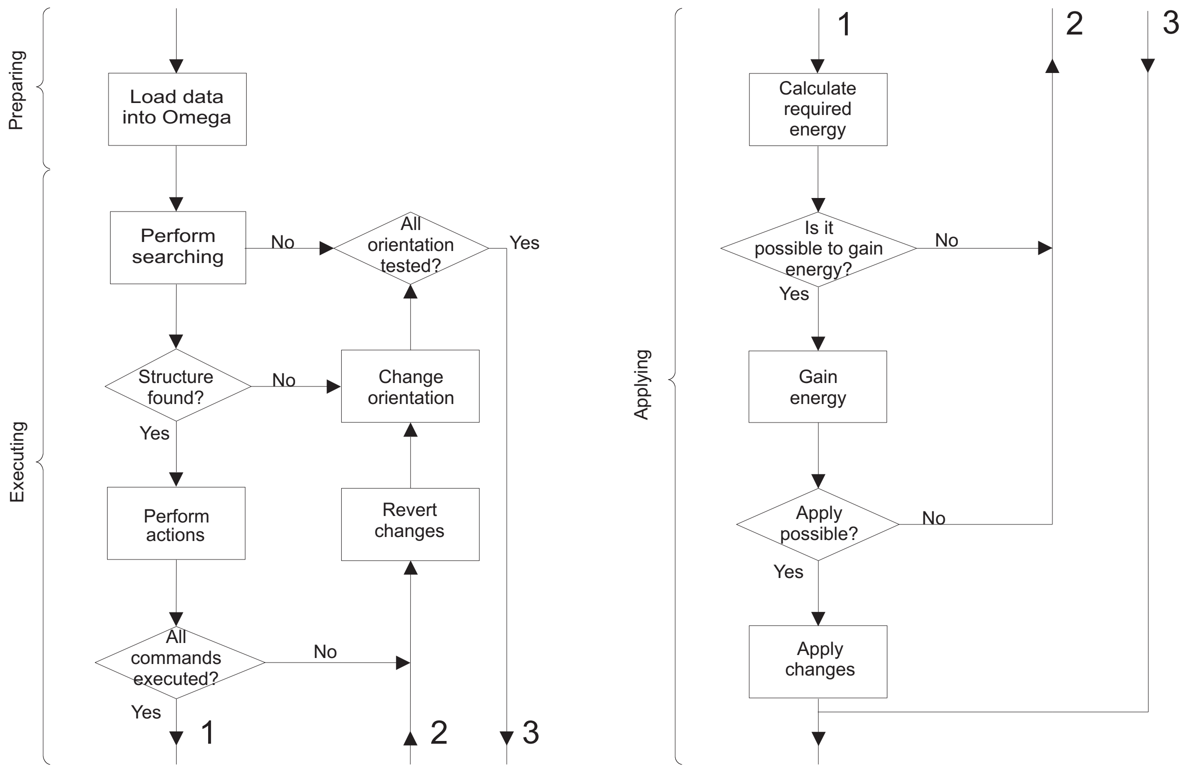


Figure 5.6: Execution algorithm

5.2.5 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. 5.4). 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 5.2.4) is formed as a circle of a given radius from the bottom particle of the main stack. The program introduced in Fig. 5.1 is represented by the structure of the particles as shown in Fig. 5.7.

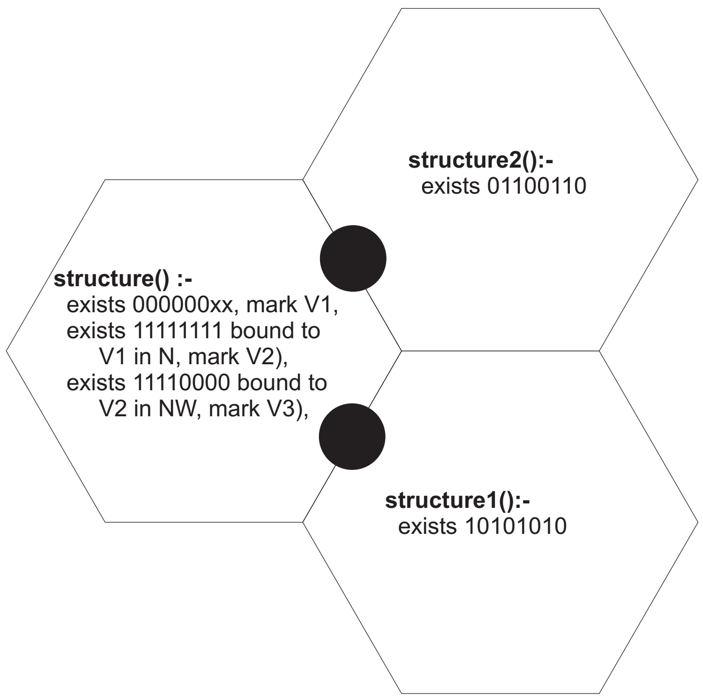


Figure 5.7: An example of a program contained in a complex. Action commands are omitted.

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. 5.8.

Fig. 5.9 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. 5.2.5 for `exists` command and according to Fig. 5.2.5 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

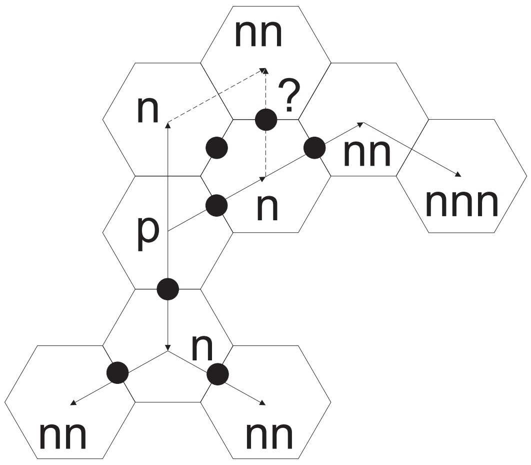


Figure 5.8: Example of encoding of complex program.

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:

$$d = 1 + type \mod 8 \tag{5.19}$$

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

5.3 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 4.3.3. 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. 5.1 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. 5.4).

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 neighborhood.

[5]Similar idea was the reason for the choosing of the LISP representation of trees in genetic programming invented by Koza [100, 101].

top

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

bottom

Figure 5.9: 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 5.10: 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 5.11: 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 [185].

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 6

Simulation experiments

This Chapter contains the description of experiments performed in the DigiHive environment. Sect. 6.1 contains a review of a few experiments that present possibilities of the environment. Sect. 6.2.1 comprises of a detailed description of an universal constructor (3.2.1) implemented in the environment, a few experiments with the constructor are also discussed. Experiment results are discussed in Sect. 6.3, future research directions are also presented in this section (including next experiments and proposed changes in the environment).

6.1 Simple emergent behaviour examples

6.1.1 Snowflake

This simulation was initially presented at the ICANNGA'07 conference [173]. The aim of the experiment was to demonstrate how a set of programs consequently constructs a regular structure starting from the set of randomly distributed particles of two types. There were five types of programs working in a sequence – the state of the structure left by one program was recognized by the second one and so on. The programs were able to:

P1 Joins two unbound particles and deactivates itself by unbinding one particle from itself (removes its header) – as the result, the program acted only once during the simulation – Fig. 6.1.

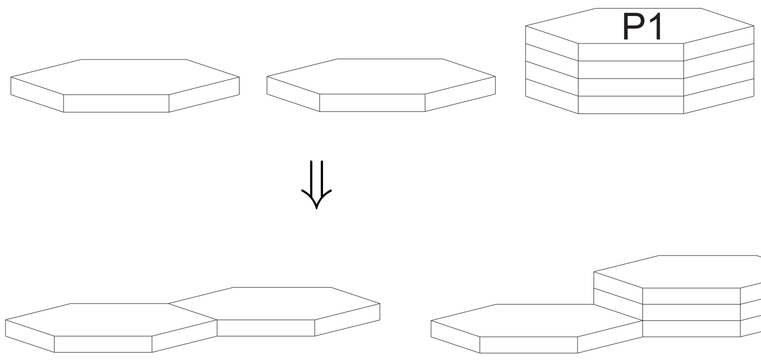


Figure 6.1: Activity of P1. Initial state (uppper): two unbound particles and a stack of particles with P1, state after P1 activity (below) – particles are bound together, P1 deactivated.

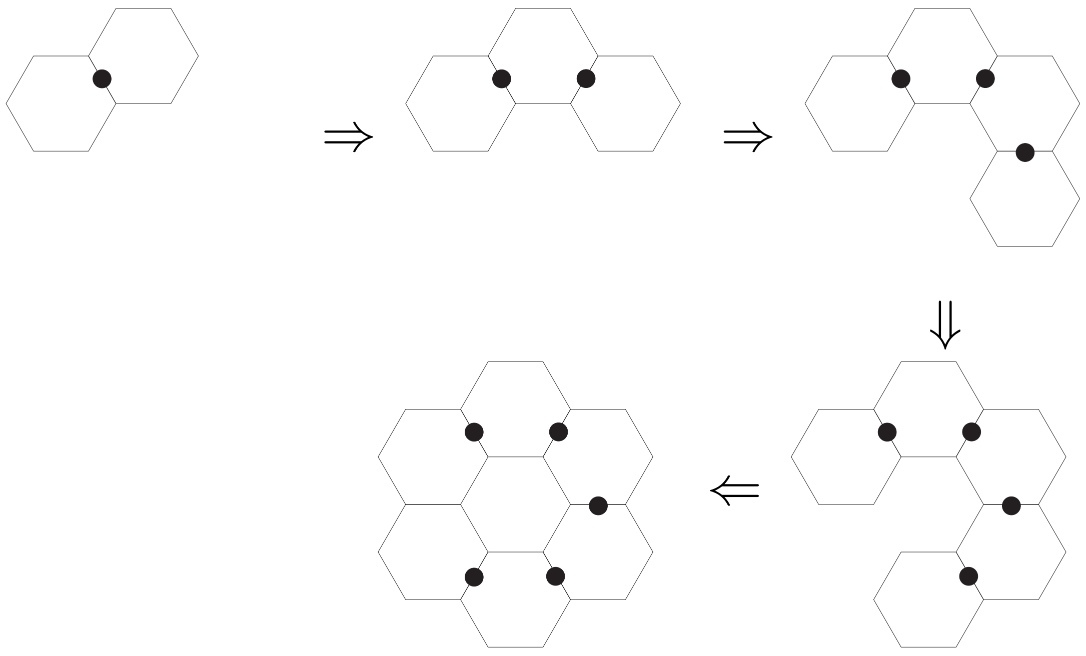


Figure 6.2: Activity of P2. Building a ring using the seed (two bound particles) prepared by P1.

P2 Builds a ring of six particles by adding one particle per run, starting from two bound particles – Fig. 6.2.

P3 recognizes the finished ring of six particles by recognising the two neighbouring particles belonging to the same complex and not bound together. After recognition the program creates a bond between the particles, and puts one unbound particle inside the ring. Then it creates the a between this particle and one particle of the ring. This bond labels finished ring – Fig. 6.3.

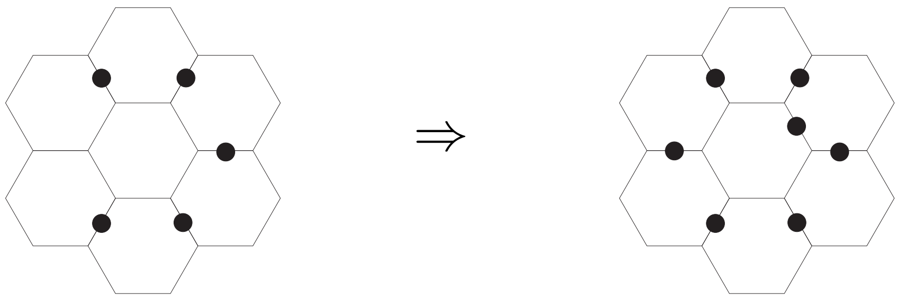


Figure 6.3: Activity of P3. Finishing ring, prepared by P2.

P4 recognizes the finished ring and binds a two-particle stack to the ring, outside of it, which is the beginning of stretching arm – Fig. 6.4.

P5 recognizes the stack put by the program no. 4. If the number of particles in the stack is less than 5, it binds one particle to the arm, binds one particle to the stack and shifts the stack to the end (tip) of the arm.

P6 recognizes the stack of 5 particles and binds to it a two-particle stack, starting a lateral arm. The building of the lateral arm is continued by the program 5.

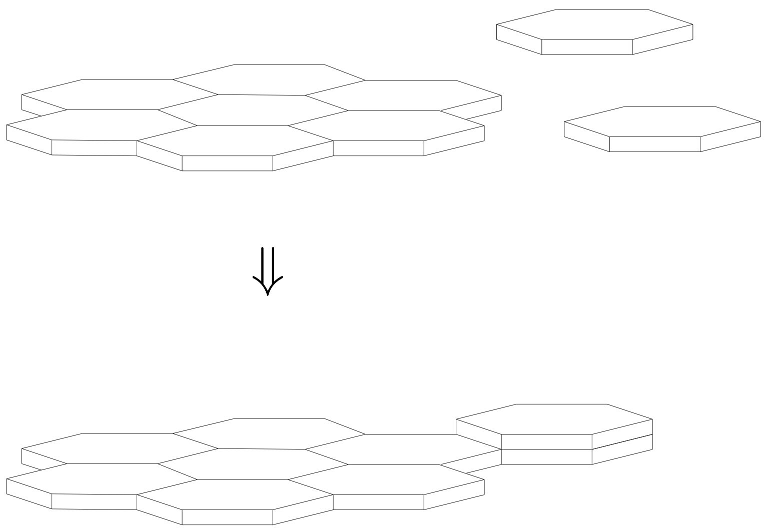


Figure 6.4: Activity of P4. Start building the stretching arm.

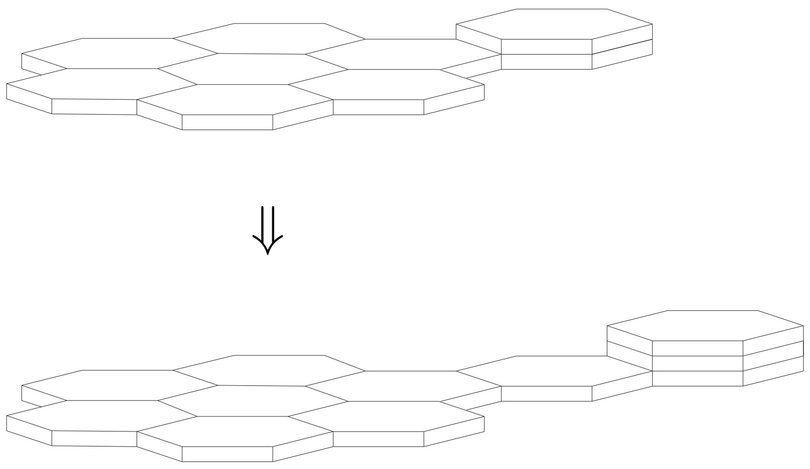


Figure 6.5: Activity of P5. Building the stretching arm.

The initial state of the system and the state of the successfully constructed shape are shown in Fig. 6.8. The activities of individual programs are presented in Fig. 6.7.

To check and demonstrate the "softness" of the language and the coding, a few experiments were run in which some bugs were injected into programs of the presented experiment. This resulted in growth of "flakes" of different shapes, but the erroneous set of programs still worked. The conclusion was, that small changes in programs resulted in relatively small changes in its functioning. Examples of erroneous behaviour are shown in Fig. 6.9.

6.1.2 Snowflake 2

Another example of forming the snowflake-like structure is presented in [96]. The shape emerges as the result of binding the free (unbound) particles to the rising structure. At the beginning of the simulation, the environment consists of:

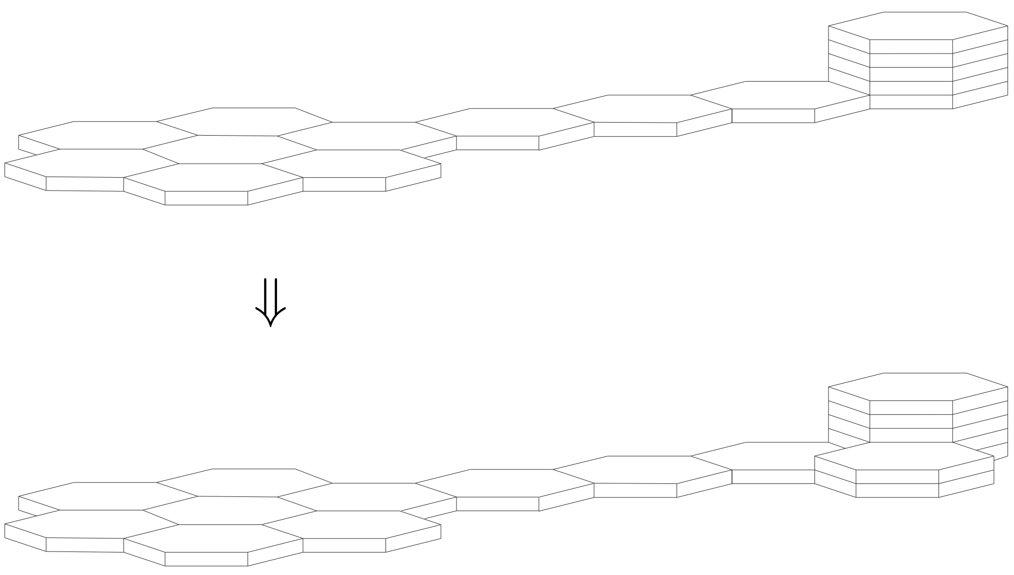


Figure 6.6: Activity of P6. Starts building the lateral arm.

1. The building material: 108 particles of type 1, 18 particles of type 2, 18 particles of type 3.
2. The primitive cells: 2 vertical complexes of exactly 2 particles of type 4
3. 6 different programs (each program occurs twice), which:

   P1 Finds the primitive cell and binds the particle of type 1 to it.

   P2 Finds the particle of type 1 bound to particle of type 2, 3 or 4 on N direction, and binds another particle of type 1 to it.

   P3 Finds the string (part of the complex) of the following particles: 4, 1, 1 and binds the free particle of type 2.

   P4 Finds the particle of type 2 bound to the particle of type 1 and binds the free particle of type 1 to it , changing the direction of the building arm by 60°

   P5 Finds the particle of type 2 bound to the particle of type 1 and binds the free particle of type 1 to it, changing the direction of building arm by −60°.

   P6 Finds the string of the following particles: 2, 1, 1 and bind the free particle of type 3 to it.

As the result of the simulation two structures of snowflake-like shapes emerged (shown in Fig. 6.10).

6.1.3 Wall growth

The simulation presented in [96] shows how set a of programs can be used to expand some existing structures in an organised way. At the beginning of the simulation the environment consists of:

1. The building material: 80 randomly distributed particles of type 0

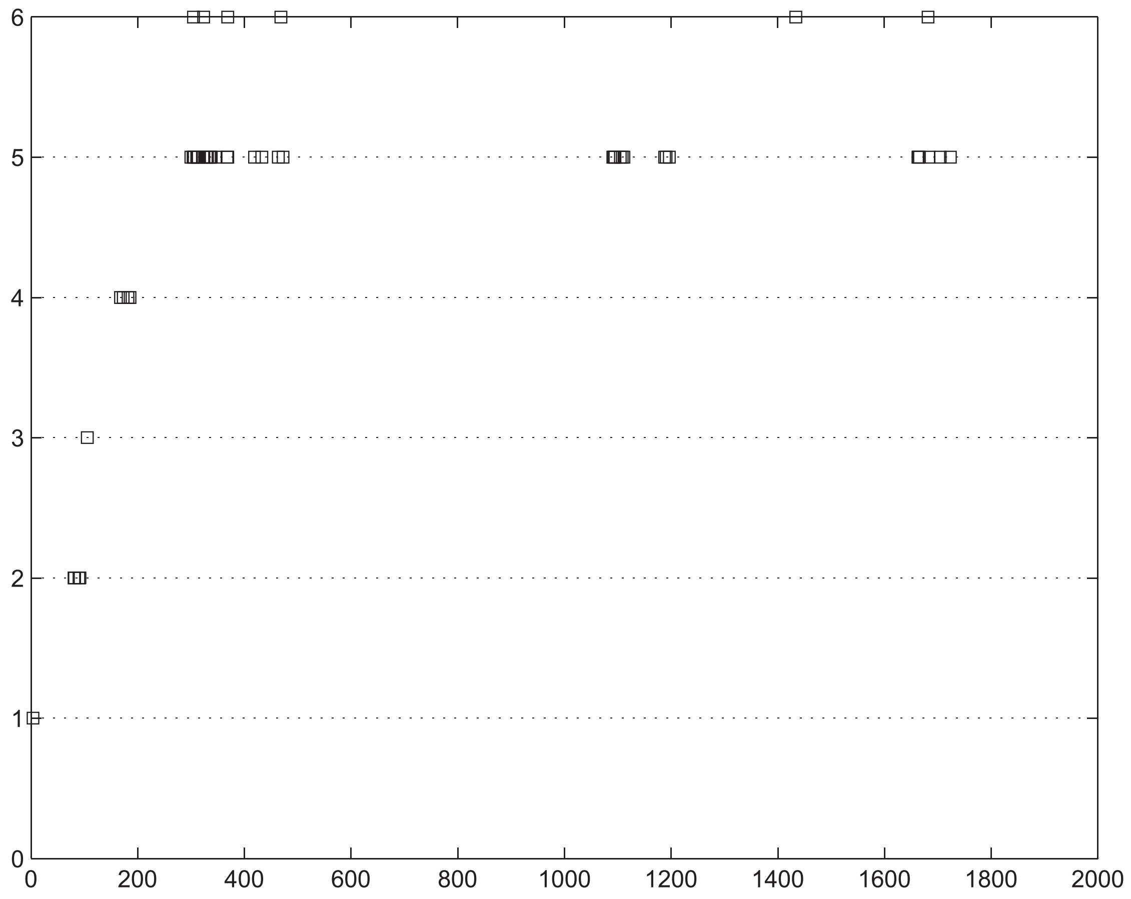


Figure 6.7: Activities of individual programs during the simulation. The number on the vertical axis represents the number of the program. Each square represents a single execution of the program during a time cycle, described on the horizontal axis.

2. Wall: closed, hexagonal ring of particles of type 0 with one particle of type 1 bound inside

3. 4 programs inside the wall, which:

   P1 binds an unbound particle of type 0 to the wall opposite the particle of type 1

   P2 moves a particle of type 1 one position along the wall (counter-clockwise)

   P3 moves a particle of type 1 one position into another wall if movement along current wall is not possible

   P4 moves bonds with particles bound to the particle of type 1 outside the wall

As shown in Fig. 6.11, the area bound with the initial wall expanded. As the result of program cooperation, the particles of type 1 consequently move along the wall (counter-clockwise). In every place it appeared, the wall was pushed outside. As long as the building material is available the wall rises.

6.2 Toward a self-replicating system

This section describes the results of work on the implementation of the modified von Neumann's model (3.2.1). Original von Neumann's model was implemented in a

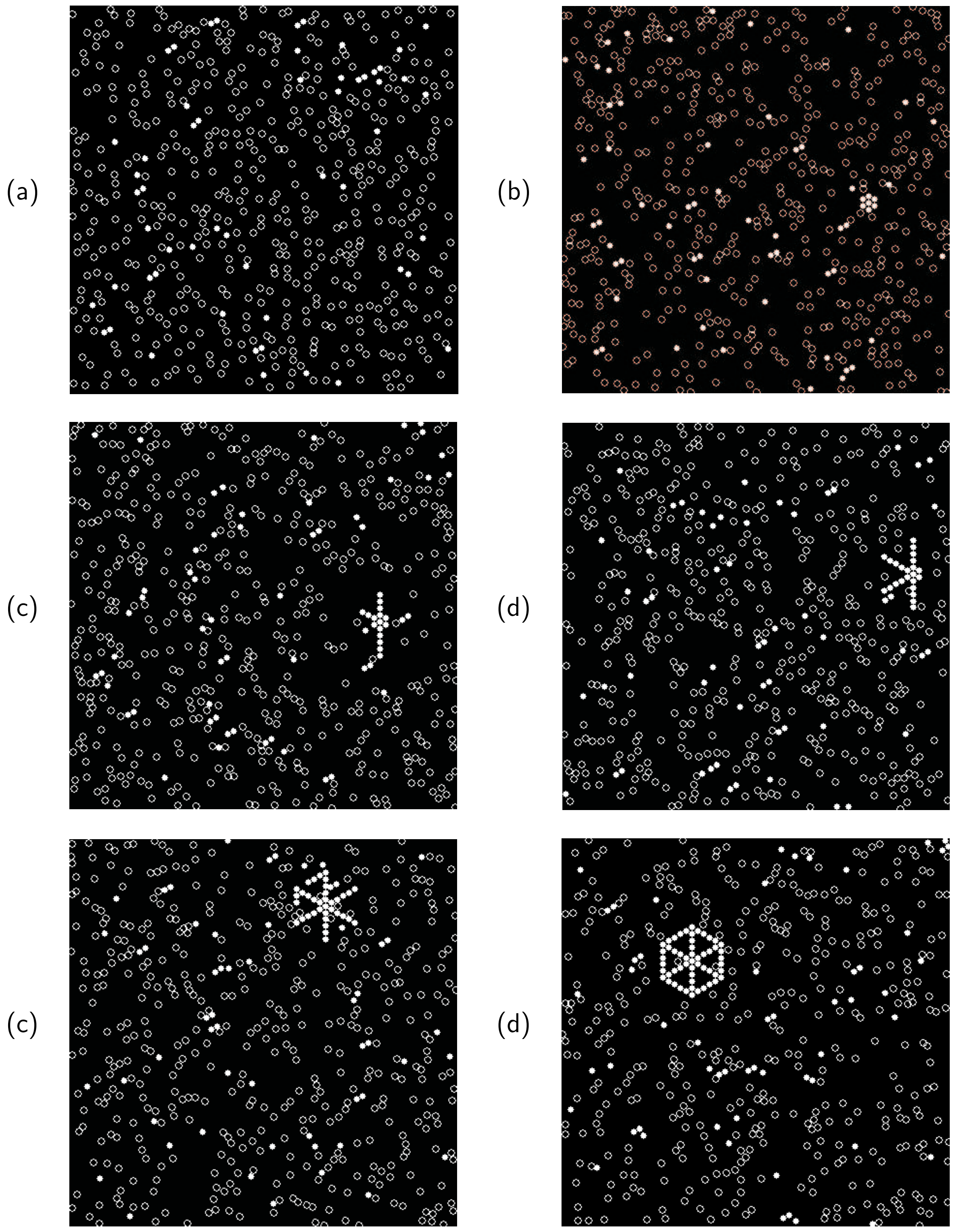


Figure 6.8: Constructing the snowflake. The initial state of the model (a), and the state after: 100 (b), 165 (c), 250 (d), 450 (e) and 2483 (f) time steps. In this figure (and the next pictures in Sect. 6.1), the white dots represents stack of particles – in the first picture (a) every stack of particles contains a program.

deterministic cellular automaton framework, fundamentally different from the DigiHive environment. In the original model, there were no particles forming material from which the structures are built, the state of a cell may undergo any change resulting from the transition function of the other cells.

In the DigiHive implementation, the construction of a single self-replicating structure

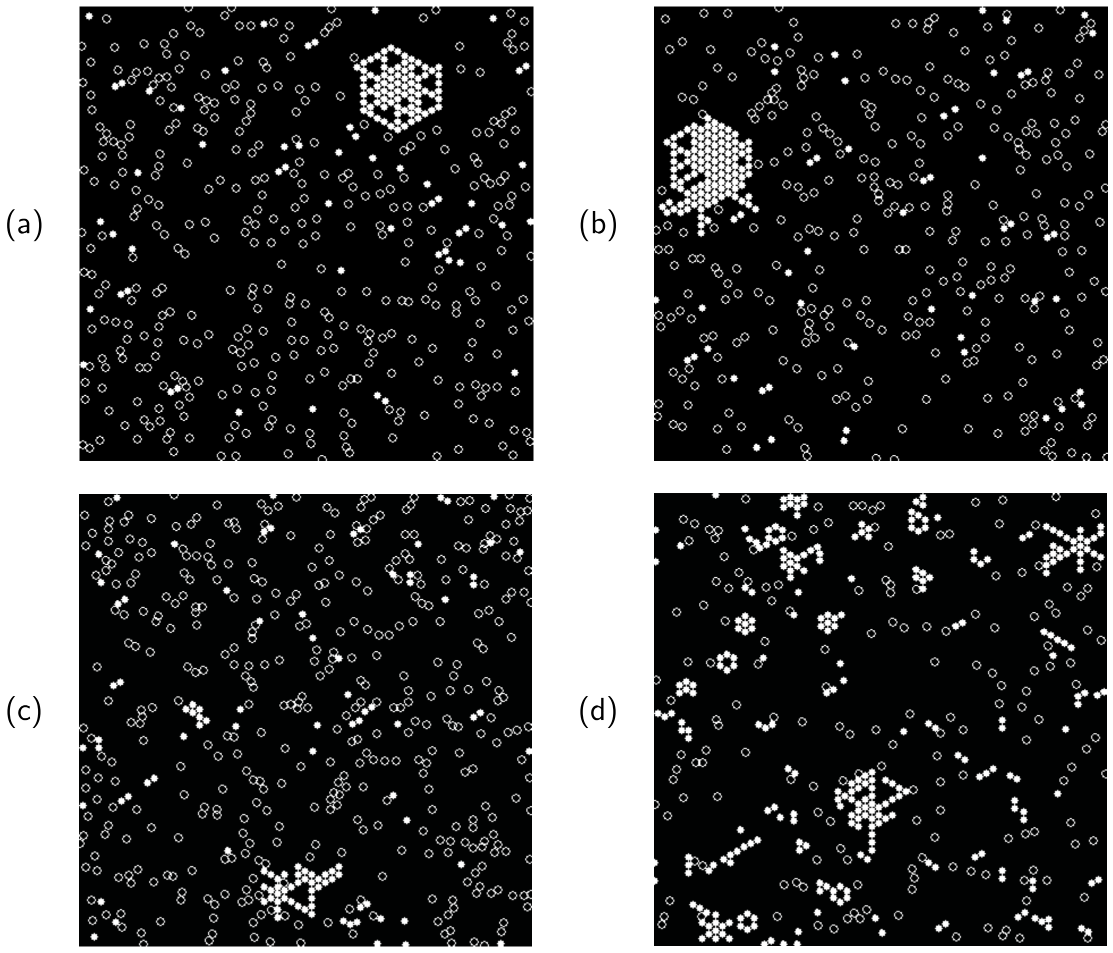


Figure 6.9: Effects of bugs injected into the functions encoded in particles: (a), (b). Effects of running the program in an environment with random events enabled: (c), (d)

(single complex) X was abandoned in favour of separate structures: the universal constructor A, the copying machine B, and their descriptions $\phi(A)$ (or $\phi(A + B)$). Starting from an initial population composed of at least single copies of individual structures, we should observe their reproduction. As the result of the structure B activity, the description $\phi(A+B)$ should be replicated, while the structure of A should lead to the creation of copies of the structures of A and B.

Note, that the experiment is designed so it doesn't matter the order of duplication, there is no need to synchronize the construction with the C structure.

This dissertation concerns with the first stage, i.e. implementation of the universal constructor A and its replication.

6.2.1 Universal constructor

The idea of an universal constructor was developed by von Neumann's during his studies on self-reproduction, the theoretical background was described in Sect. 3.2.1.

The universal constructor in the DigiHive environment is a structure A (a complex of particles) being able to construct any other structure X based on its description $\phi(X)$. It is admissible to construct the X structure via description $\phi(X')$ of an intermediate structure $X' \neq X$, being able to transform itself into the X The transformation should occur through an interaction with the environment in finite time cycles. The universal

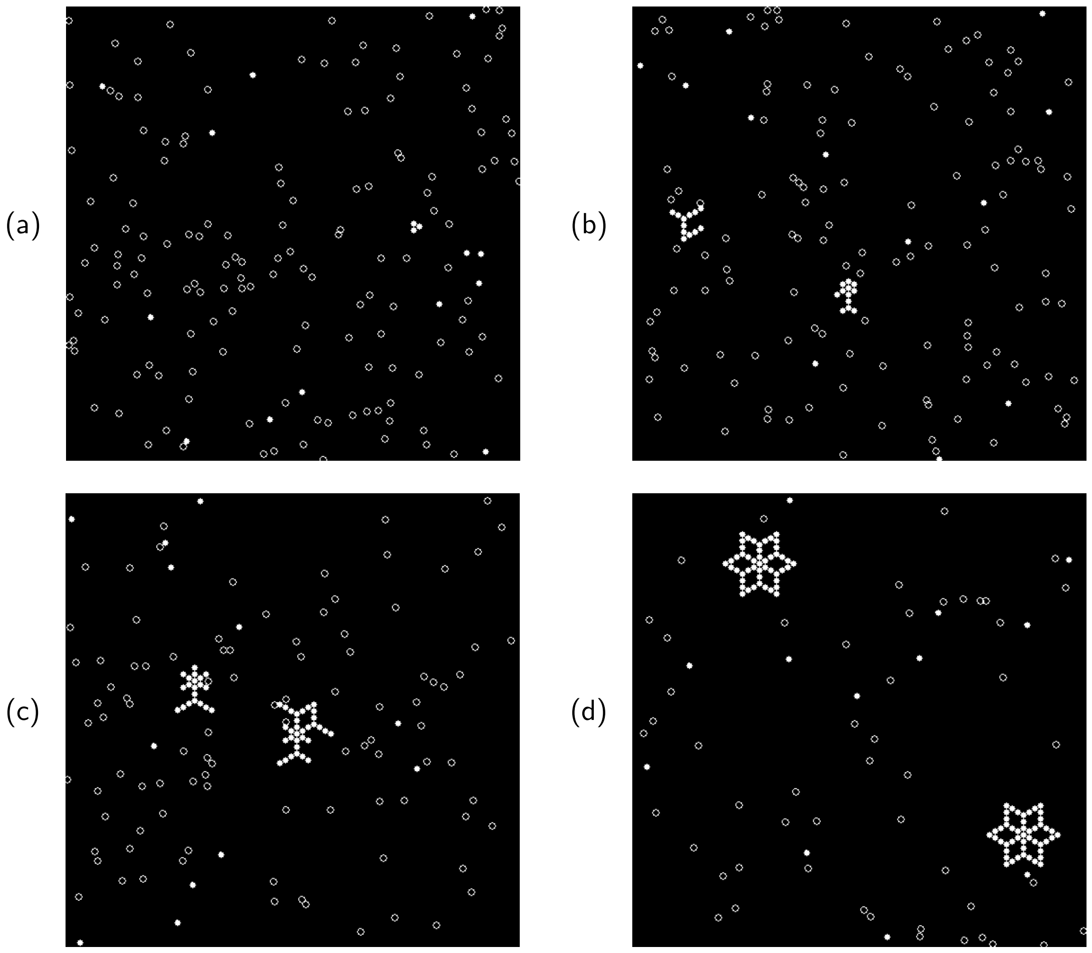


Figure 6.10: The snowflakes after 10 (a), 270 (b), 1670 (c) and 4500 (d) simulation cycle.

constructor can be described by the following equation[1]:

$$A + \phi(X') \Rightarrow A + \phi(X') + X' \Rightarrow A + \phi(X') + \mathbf{X} \tag{6.1}$$

or, when the obvious parts are omitted:

$$A + \phi(X') \Rightarrow X' \Rightarrow \mathbf{X} \tag{6.2}$$

in case when $X = X'$ the equation turns into its simplified form:

$$A + \phi(X) \Rightarrow \mathbf{X} \tag{6.3}$$

The universal constructor is a consistent structure (a set of programs) being able to fulfil the following tasks (see also [174]):

1. search for a valid information structure (information string) – $\phi(X)$. The information string encodes the description of some structure $\phi(X)$, which may be viewed as a program written in a simple universal constructor language. The syntax of the language is presented in Fig. 1. The information string is a vertical

[1]The "⇒" symbol means: "becomes, according to environment laws, in finite number of time steps". The structure obtained as a supposed result of the simulation is marked in bold. This convention is used in other equations presented in this chapter.

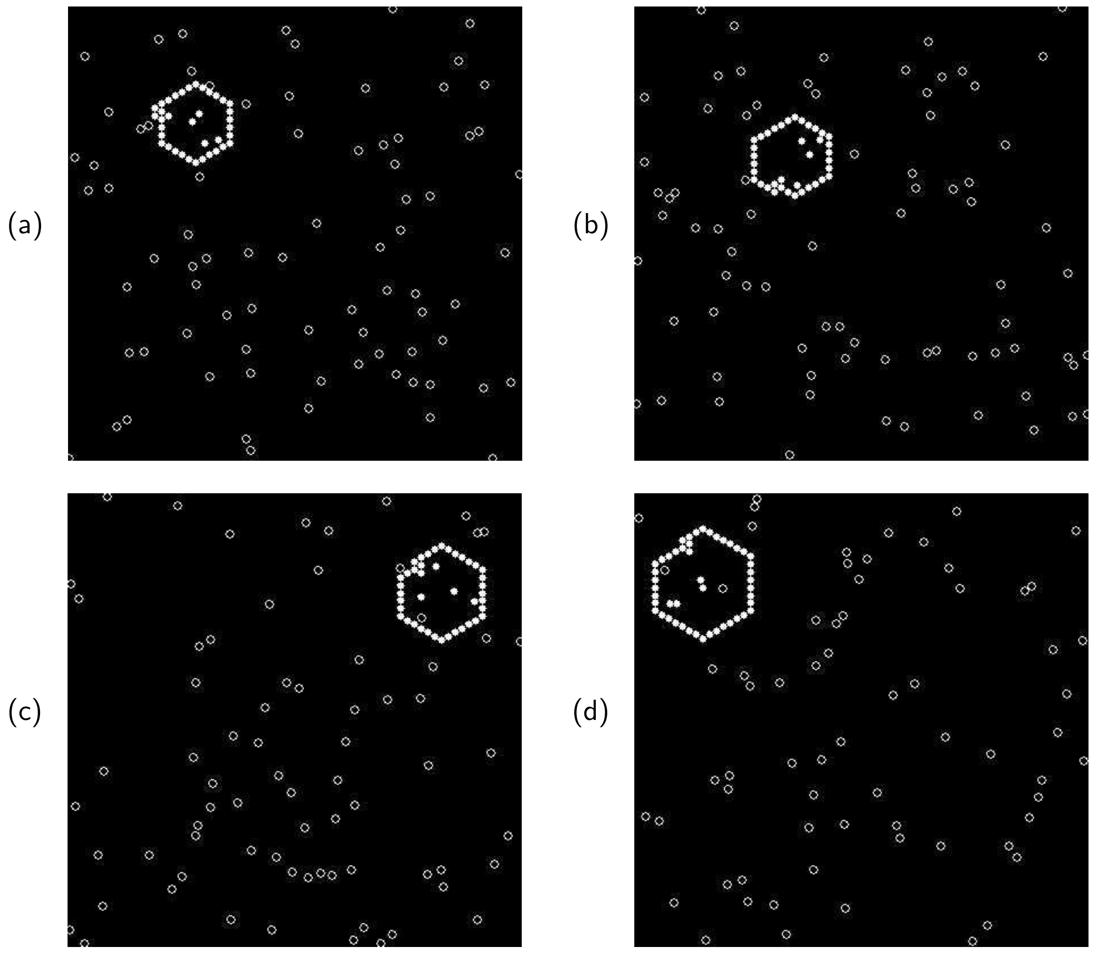


Figure 6.11: Wall growth after 10 (a), 40 (b), 770 (c) and 10950 (d) simulation cycles.

complex of particles of a specified type as described in Fig. 6.13. As an example the following program, which describes a stack of two particles of type 01010101:

```
PUT(01010101)
PUT(01010101)
END
```

can be encoded by the following stack:

```
11111111
01010101
00000001
01010101
00000001
```

2. connect to the found information string and start constructing the structure X. The structure X consists of joined stacks of particles. There is always exactly one stack of particles being build at the moment, called active stack $X^{\star}$.

3. sequentially process the joined string:

   (a) if current particle in the information string encodes particle type (PUT argument) – find the particle of *specified* type in nearest space (Ω – see

```
program ::= PUT( t ) {command} END
command ::= PUT( t ) | SPLIT( d ) | NEW( t )
d ::= N|NE|SE|S|SW|NW
t ::= 0|1, 0|1, 0|1, 0|1, 0|1, 0|1, 0|1, 0|1
```

Figure 6.12: BNF syntax of a simple universal constructor language.

top

`END` (×,×,×,×,×,×,×,×)
⋮
direction (×,×,×,×,×,×,×,×)
`SPLIT` header (×,×,×,×,×,×,1,1)
⋮
particle type (×,×,×,×,×,×,×,×))
`NEW` header (1,1,1,1,1,1,0,1))
⋮
particle type (×,×,×,×,×,×,×,×)
`PUT` header (×,×,×,×,×,×,0,1)

bottom

Figure 6.13: Encoding stack. Bytes taken from type of particle.

page 61) and put it on the top of stack $X^\star$. The found particle is removed from the top of stacks, beginning with particle of type 0000xx10 (Fig. 6.14)[2]. This algorithm is presented in Fig. 6.15(b).

(b) if current particle in the information string encodes particle type on a new stack (`NEW` argument) – disconnect the structure X and immediately start the construction of a new structure with *specified* particle as the beginning of $X^\star$ (a single information string can then encode various structures, e.g. both $\phi(B)$ and $\phi(A)$, please see also additional notes on page 86 regarding this issue).

(c) if current particle in the information string encodes particle direction (`SPLIT` argument) – splits the stack $X^\star$ into two stacks: removes a particle from the top, moves the trimmed stack in *specified* direction, and creates a horizontal bond between $X^\star$ and the removed particle. The particle becomes the bottom of the active stack $X^\star$. This algorithm is presented in Fig. 6.15(b).

4. disconnects from the information string and releases the constructed structure (i.e. a set of stacks being able to create valid programs)

Most of the above tasks can be relatively easy encoded by the language commands

[2]In the first attempts the universal constructor searched for single unbound particles, however in the most complex experiments it turns out that the size of environment grows to a value that causes significant performance problems. Building stacks were introduced as a kind of performance improvement.

top

particle (×,×,×,×,×,×,×,×)
⋮
particle (×,×,×,×,×,×,×,×)
building material header (0,0,0,0,×,×,1,0)

bottom

Figure 6.14: Source of building material.

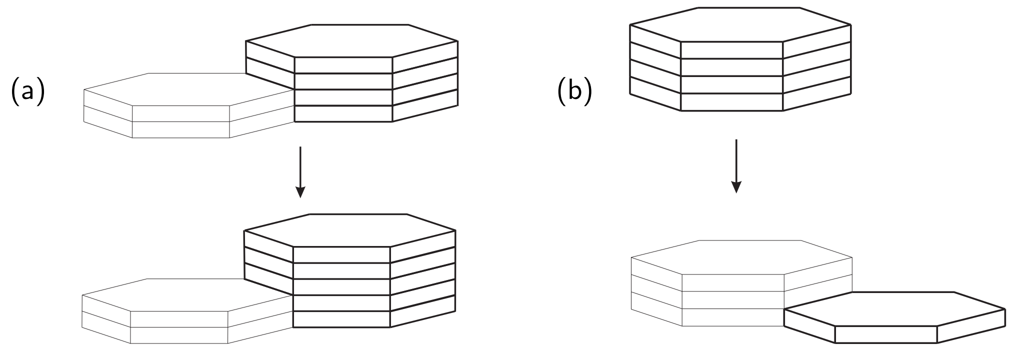


Figure 6.15: Illustration of actions of the universal constructor during processing of the information string: (a) action caused by the command `PUT`. The particle of specified type is put on the top of active stack $X^\star$ (draw using thicker lines), (b) action caused by the command `SPLIT`. The particle is removed from the top of $X^\star$, then the bond is created in the specified direction, the particle becomes a new active stack $X^\star$

(see 5.2). The language, however, does not allow to perform the information string processing (task 3) as there is no conditional nor any comparing commands.

The general idea is to create the following generic programs:

T1 "find the particle <T> and begin construction of the new active stack $X^\star$", where <T> is some particle type (described by the encoding stack)

T2 "find the particle <T> and put it on the top of the constructed active stack $X^\star$", where <T> is some particle type (described by the encoding stack)

T3 "split the active stack $X^\star$ into two stacks and create a bond in <D> direction between them", where <D> is some direction

Obviously, there is no way to directly encode such generic sequences [3] (called *templates*). In fact, templates are encoded as some invalid programs i.e. sequence of inconsistent commands. Each template has also some characteristic mark (a sequence of particles of specified type). Replacing particles between the marks, by particle from encoding string, forms a valid program being able to recognize the encoded particle.

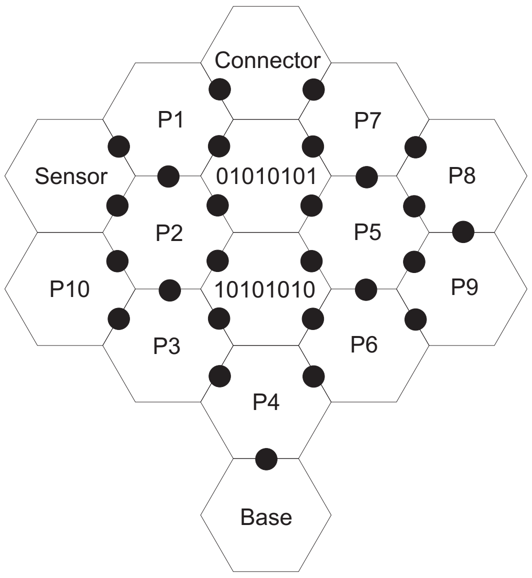


Figure 6.16: Constructor schema – view from top.

(a)

11111111
01010101
00000001
00000000
01010101
00000001

(b)

11111111
01010101
00000001
00000000
11111100
11011100
11001100
00000001

(c)

11111111
01010101
00000001
00000000
11111100
11011100
11001100
01010101
00000001

(d)

11111111
01010101
00000001
11111100
11011100
11001100
00000000
01010101
00000001

(e)

11111111
01010101
11111100
11011100
11001100
00000001
00000000
01010101
00000001

(f)

11111111
11111100
11011100
11001100
00000001
00000000
01010101
00000001

(g)

11111111
11111100
11011100
11001100
01010101
00000001
00000000
01010101
00000001

(h)

11111111
01010101
00000001
00000000
01010101
00000001

Figure 6.17: Example of processing the information string: (a) the initial information string encodes a single stack of two particles of type 01010101 (please compare with the example on page 75), a single particle of type 0000000 plays the role of "junk" information (does not encode any command), (b) the string after P1 activity – the sensor (bold font) is inserted instead of a type particle, a type particle is joined to the P3, (c) after P3 – particle encoding type is back in the string, (d) after P7 – the sensor moved to the next particle in the string, (e) after P8 and P7 – the sensor moved to the next particle in the string ("junk" skipped), (f) after P2 – the type particle joined to the P4, (g) after P4 – particle encoding type is back in the string, (h) after P9 – the sensor is back in the universal constructor, the string is ready for the next processing

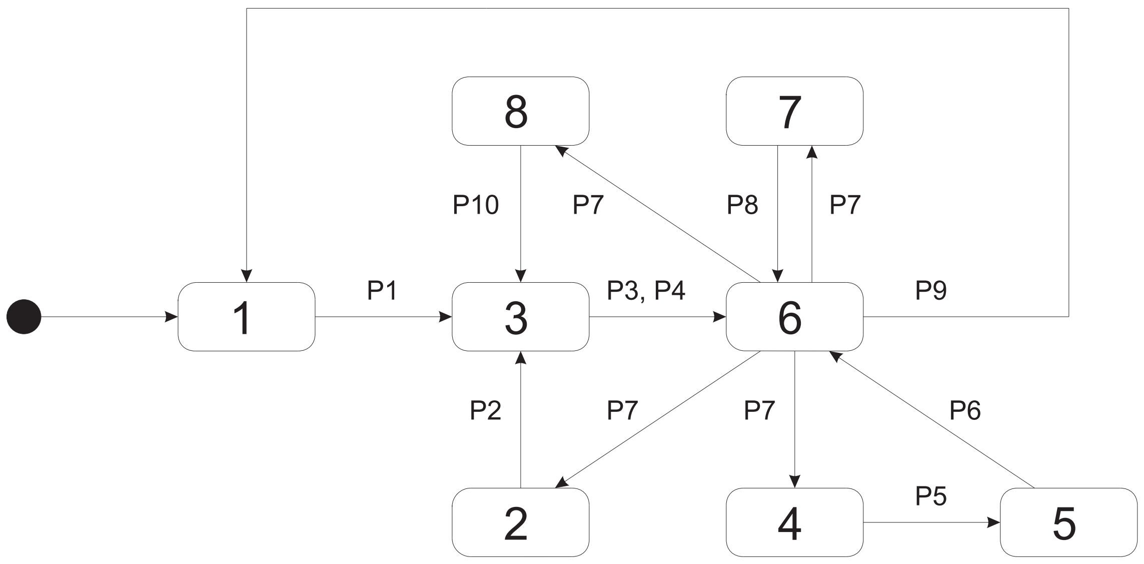


Figure 6.18: Constructor state diagram.

Constructor algorithm

The universal constructor schema is presented in Fig. 6.16. This structure is able to read and interpret the encoding stack (the information string) and translate it into another complex. The encoding stack syntax is described in table 6.13[4]. The constructor consists of two types of stacks: helper stacks and programs (denote as P1, P2, ..., P8). Helper stacks are affected by programs:

1. Connector: a single particle of type 01010001. The universal constructor joins to the information sting by creating a bond between the connector and the bottom of the information string on N direction.
2. Base: a single particle of type 01010000. The structure X is constructed just below the base. The stack directly bound to the base is the structure active stack $X^\star$ (see page 75).
3. Sensor: Stack of the following particles: 11001100, 11011100, 11111100. After the connector (1) is bound to the information string, the program P1 unbinds the sensor particles from the constructor and binds it to the constructor stack
4. State stacks: a stack of exactly two particles of type 01010101 and a stack of exactly two particles of type 10101010. The bond between the bottom stacks describes the current state (described later).

A full list of states with possible transitions between them is presented in Fig. 6.18. Each state is encoded by some specified bonds between the helper stacks, as described

[3]It is possible however to easy encode all of particular instances of the first sequence e.g. "if the examined particle type is 01011001 find the particle 11110000 (...)" *etc.* Every instance requires separate program, so appending the specified particle can be described by 256 different particle stacks i.e. different programs. This approach is then extremely unhandy (besides, the third subtask cannot be resolved that way).

[4]In the first attempt to the experiment the encoding string was designed as a horizontal structure (see [89]). This approach was abandoned due to performance reasons. As turns out later, the vertical structure seems also to be much more "natural" in the DigiHive environment as it can easily emerge during random interaction between particles, or through activities of relatively simple programs.

	Name	Encoding	Programs
1	Disconnected	1. Sensor unbound in D direction	P1
2	Activate type recognition	1. Sensor bound in D direction 2. Sensor bound in U direction to xxxxxx01 particle 3. Sensor bound in U direction to particle $\neq$ 11111101 4. State stacks unbound 5. Connector and upper state stack unbound	P2
3	Process type recognition	1. Sensor bound in D direction 2. Sensor bound in U direction to xxxxxx01 particle 3. State stacks unbound 4. Connector and upper state stack bound	P3, P4
4	Activate splitting	1. Sensor bound in D direction 2. Sensor bound in U direction to xxxxxx11 particle 3. State stacks unbound 4. Connector and upper state stack unbound	P5
5	Process splitting	1. Sensor bound in D direction 2. Sensor bound in U direction to xxxxxx11 particle 3. State stacks unbound 4. Connector and upper state stack bound	P6
6	Process encoding stack	1. Sensor bound in D direction 2. Sensor bound in U direction 3. State stacks bound	P7, P9
7	Find valid specification	1. Sensor bound in D direction to a particle of type $\neq$ xxxxxxx1 2. State stacks unbound	P8
8	New structure construction	1. Sensor bound in D direction 2. Sensor bound in U direction to 11111101 particle 3. State stacks unbound 4. Connector and upper state stack unbound	P10

Table 6.1: List of constructor states

in table 6.1. Constructor states are recognized and changed by the following programs (the universal constructor algorithm is also presented in Fig. 6.19):

P1 Program aimed at finding the encoding stack and the beginning of translation. Detailed algorithm:

   If the constructor state is *Disconnected* then:

   1. Find the beginning of encoding stack (i.e. vertical sequence of at least three particles beginning with 0000xx01) – Fig. 6.17(a)
   2. Unbind the sensor from the constructor
   3. Unbind the second particle from the encoding stack
   4. Insert the sensor to the encoding stack (instead of the substack removed in step 3) – Fig. 6.17(b)
   5. Insert the particle removed in step 3 to the proper place in P3 (P3 becomes the valid program now)
   6. Bind the upper state stack and the connector – change state to *Process type recognition* (execute P3)

P2 Program aimed at preparing for searching for specified particle. Detailed algorithm:

   If the constructor state is *Activate type recognition* then:

   1. Unbind from the encoding stack a particle next to the sensor – Fig. 6.17(f)
   2. Bind the particle removed in step 1 to the proper place in P4 (P4 becomes the valid program now)
   3. Bind the connector to the upper state stack – change the state to *Process type recognition* (execute P4)

P3 Template $T1$, after activation program aimed at searching for the specified particle and beginning the translated stack building. Detailed algorithm:

   If the constructor state is *Process type recognition* **and** P3 is the valid program (possible only via P1 and P10 activity) then:

   1. Find the single particle using the particle joined as a part of P3 by P1. The single particle is removed from the top of stacks described in Fig. 6.14
   2. Bind the found particle to the Base in direction N
   3. Unbind the particle joined by P1 from itself (P3 becomes invalid in the next time cycle)
   4. Unbind the sensor from the encoding stack (joined there by P1)
   5. Insert the particle from step 3 into the encoding stack (instead of the substack removed in step 4)
   6. Insert the sensor again into the encoding stack just above the particle inserted in step 5 – see Fig. 6.17(c)
   7. Unbind the connector and the upper state stack

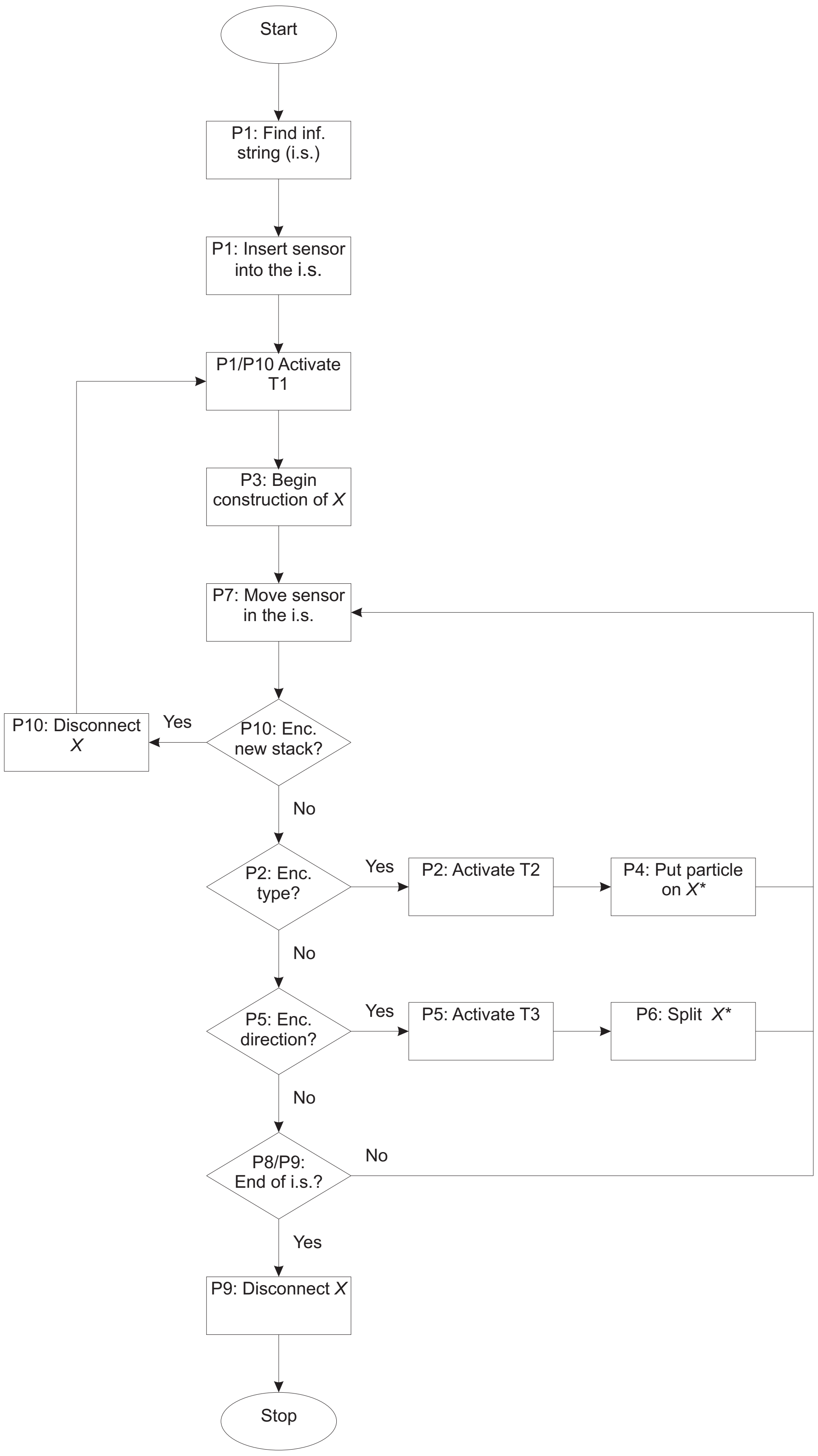


Figure 6.19: Universal constructor flowchart.

8. Bind the state stacks - change the status to *Process encoding stack* (execute P7 or P9)

P4 Template $T2$, after activation, program aimed at searching for specified particle. Detailed algorithm:

If the constructor state is *Process type recognition* **and** P4 is the valid program (possible only via P2 activity) then:

1. Find the single particle using the particle joined as a part of P4 by P2. The single particle can be found on building stacks (similar to P3).
2. Put the found particle on the top of the built stack (bond in N direction to the base) – Fig. 6.17(a)
3. Unbind the particle joined by P2 from itself (P4 becomes invalid in the next time cycle)
4. Unbind the sensor from the encoding stack (joined there by P2)
5. Insert the particle from step 3 into the encoding stack (instead of the substack removed in step 4)
6. Insert the sensor again into the encoding stack just above the particle inserted in step 5 – see Fig. 6.17(g)
7. Unbind the connector and the upper state stack
8. Bind the state stacks - change the status to *Process encoding stack* (execute P7 or P9)

P5 Program aimed at preparing for splitting. Detailed algorithm:

If the constructor state is *Activate splitting* then:

1. Unbind from the encoding stack a particle next to the sensor
2. Bind the particle removed in step 1 to the proper place in P6 (P6 becomes the valid program now)
3. Bind the connector to the upper state stack – change the state to *Process splitting* (execute P6)

P6 Template $T3$, after activation, program aimed at searching for splitting. Detailed algorithm:

If the constructor state is *Process splitting* **and** P6 is the valid program (possible only via P5 activity) then:

1. Unbind the build stack from Base (bond in N direction to the Base)
2. Move the build stack in direction specified by the particle (i.e. its type) joined by P5
3. Remove the top particle from the build stack (previously bond in N direction to the base)
4. Bind the particle removed in 3 to the base
5. Unbind the particle joined by P5 from itself (P6 becomes invalid in the next time cycle)

6. Unbind the sensor from the encoding stack (joined there by P2)
7. Insert the particle removed in step 5 to the encoding stack (instead of the substack removed in step 6)
8. Insert the sensor again into the encoding stack just above the particle inserted in step 7
9. Unbind the connector and the upper state stack
10. Bind the state stacks – change the status to *Process encoding stack* (execute P7 or P9)

P7 Program aimed at moving the sensor along the information string. Detailed algorithm:

If the constructor state is *Process encoding stack* **and** there are at least two particles above the sensor then:

1. Remove the sensor from the encoding stack
2. Insert the sensor again into the encoding stack one particle above the previous place – see Fig. 6.17(d)
3. Unbind the State stack

Note that as a result, the constructor state will be switched into: *Activate type recognition*, *Activate splitting*, *New structure construction* or *Find valid specification*. The final state is determined only by the particle type above the sensor in the encoding stack. In the next step, one of the following program will be executed: P2, P5, P8, P10.

P8 Program aimed at skipping invalid part of information string. Detailed algorithm:

If the constructor state is *Find valid specification* then:

1. Bind the State stack – change the state to *Process encoding stack* (execute P6) – see Fig. 6.17(e)

P9 The program's task is to finish the translation. Detailed algorithm:

If the constructor state is *Process encoding stack* **and** there is only one particle above the sensor in the encoding stack then:

1. Remove the sensor from the encoding stack
2. Bind sensor to the constructor at its initial place (Fig. 6.16 on page 78) – Fig. 6.17(h)
3. Unbind the base and the build structure X – change state to *Disconnected* (execute P1)

P10 Program aimed at preparing to constructing the new structure. Detailed algorithm:

If the constructor state is *New structure construction* then:

1. Unbind the base and the build structure X
2. Unbind from the encoding stack a particle next to the sensor

3. Bind the particle removed in step 2 to the proper place in P3 (P3 becomes the valid program now)
4. Bind the connector to the upper state stack – change the state to *Process type recognition* (execute P3)

Universality of constructor

The universal constructor is not fully universal, i.e. it cannot build any possible spatial structure straightforwardly. Sequential adding the subsequent stacks in various directions may lead to the situation where a built structure will try to occupy a place already occupied by the constructor itself. Important limitation is also lack of possibility of create bond between built stacks in an every possible direction. The universal constructor can only then build a chain of particle stacks. Note, that the stacks of particles itself are not restricted – the constructor can build any possible stack.

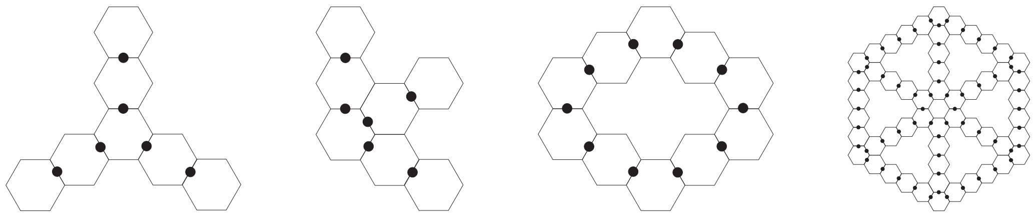


Figure 6.20: Examples of structures, which cannot be directly build by the universal constructor.

Examples of problematic structures are presented in Fig. 6.20. There is no possibility of encode the stack (or particle) with 3 or more horizontal bonds. The universal constructor is also not able to build any closed curve of particles (or of stacks).

In [174] two possible strategies were discussed which can compensate its limitations :

S_1 – construction of set of programs which are able to cooperatively build the desired structure in finite time cycles – Fig. 6.21(a)

S_2 – construction of one intermediate structure which can transform itself into the desired structure – Fig. 6.21(b)

There is also possible to mix the strategies, e.g. by constructing an intermediate structure with set of programs which helps to transforms it into the desired one, etc.

6.2.2 Sample simulations

To illustrate the universal constructor's abilities it was provided with the information string describing a flat, rhombus-shaped complex. As a result, the programmed structure was successfully built. The simulation screenshots are presented in Fig. 6.22.

The universal constructor has also been provided with an information string which encodes a set of programs described in Sect. 6.1.1. After finishing their construction, the programs gradually constructed the shape of a snowflake [170], the result of the experiment is presented in Fig. 6.23.

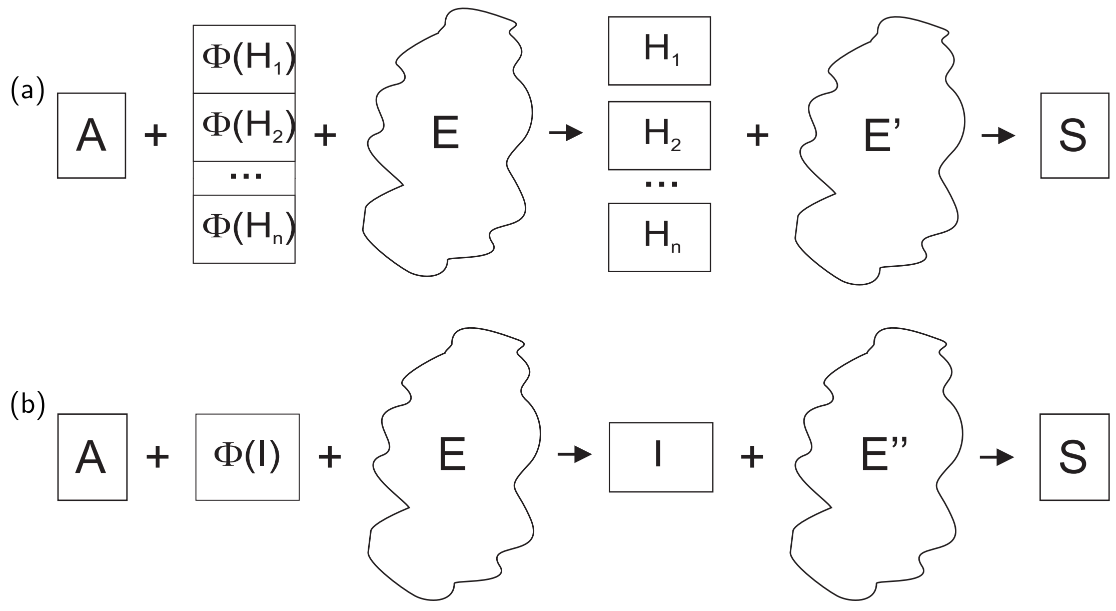


Figure 6.21: Universal constructor strategies: (a) the universal constructor A joins the information string $\phi(H_1)$, $\phi(H_2)$, ..., $\phi(H_n)$ and, using the building material from environment E (random distributed particles and complexes), builds the set of helper $\phi(H_1)$, $\phi(H_2)$, ..., $\phi(H_n)$ programs. Helper programs interacts with environment E' (changed thanks to A activity, note that E' contains also both A and $\phi(H_2)$, ..., $\phi(H_n)$ which in fact becomes a part of the environment at this stage of simulation). As a result they finally build the desired structure S. Note that description of individual programs $\phi(H_1)$, $\phi(H_2)$, ..., $\phi(H_n)$ are separated by the `NEW` command. (b) the universal constructor A joins the information string $d(I)$ and builds the intermediate structure I. The structure I interacts with environment E'' and finally transforms itself into the desired structure S.

Additional notes

As Fig. 6.22(f) shows, after the construction of X is finished, the universal constructor immediately joins the information string again and starts a new translation. As a matter of fact, the constructor A after joining to the structure description $\phi(X)$, forms a consistent structure suitable for performing only one task – the building of the structure X. The desired goal was however slightly different – the A should create different structures regarding the currently processed description.

One of the possible resolutions is to extend the information string with the description of some other structure Z responsible for deactivation of the information string (e.g. by appending some particle of type different than xxxxxx01 at the bottom of the information string). To reuse it, the environment should contain at least one structure Y responsible for activating the information string. The information string can be then enhanced into the $\phi(Y+X+Z)$ (the information string can describe various structures at the same time via the `NEW` command – see page 76). The constructor would build at first the activator for the string Y, then the desired structure X and at the end the structure Z that prevents another construction. While constructing X and Z, the Y structure will probably move away due to random collision with other particles or complexes in the environment. After the whole string is processed it is expected that the constructor will not immediately join the information string (if Y is far away enough). Due to random interaction with the environment, the Y structure

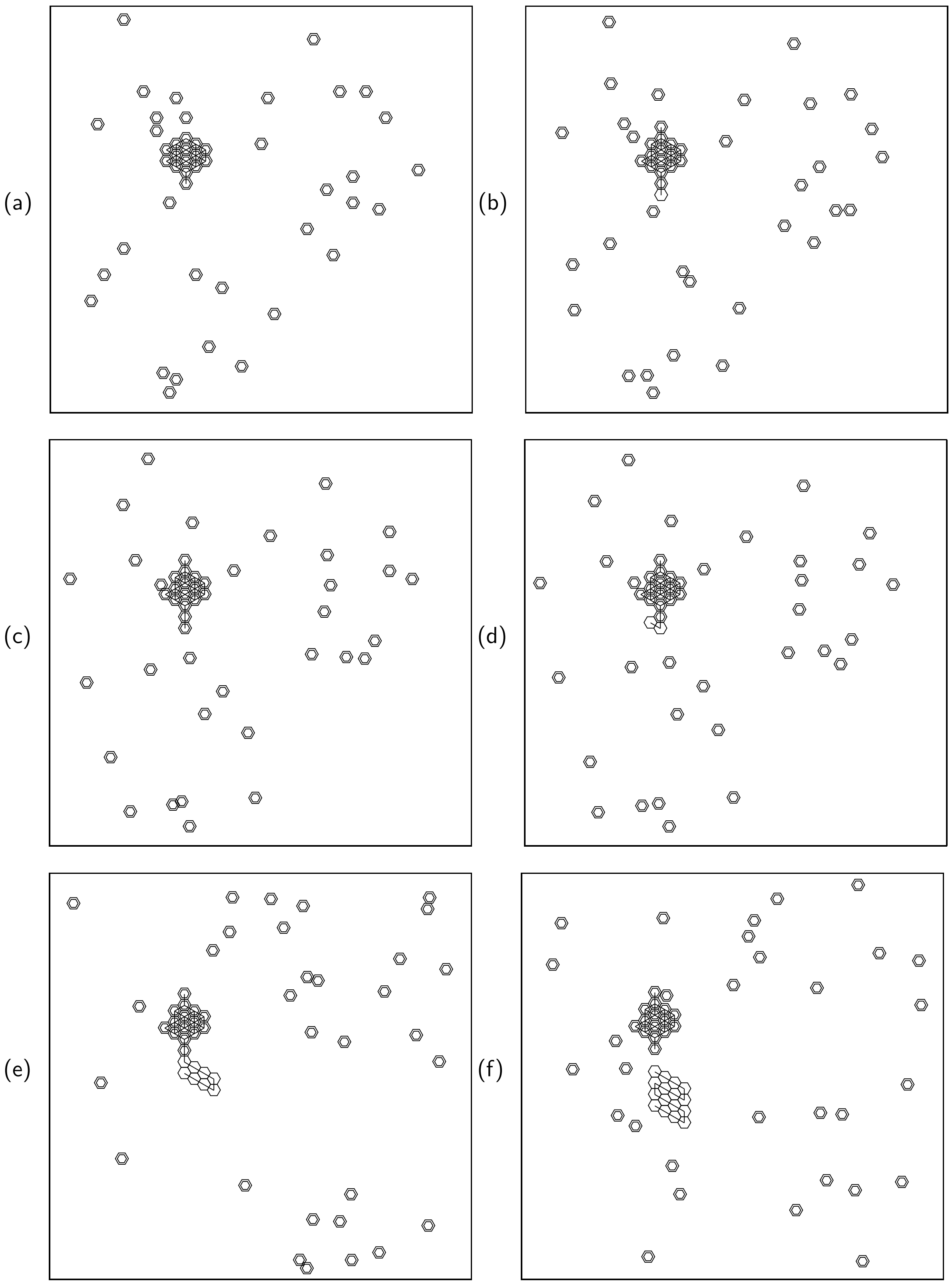


Figure 6.22: The constructor after 0 (a), 1 (b), 4 (c), 5 (d), 33 (e) and 88 (f) simulation cycles. Pictures (a) and (b) show the initial state and the result of P1 and P3 activity i.e. the binding of the constructor to the encoding stack, searching for a free particle and the beginning of building a stack. Pictures (c) and (d) illustrates P5 and P6 activity i.e. splitting . Picture (e) shows the environment after half of the simulation and the picture (f) shows the environment after the simulation has been finished. Note that, the constructor immediately joined the encoding string again and started a new translation.

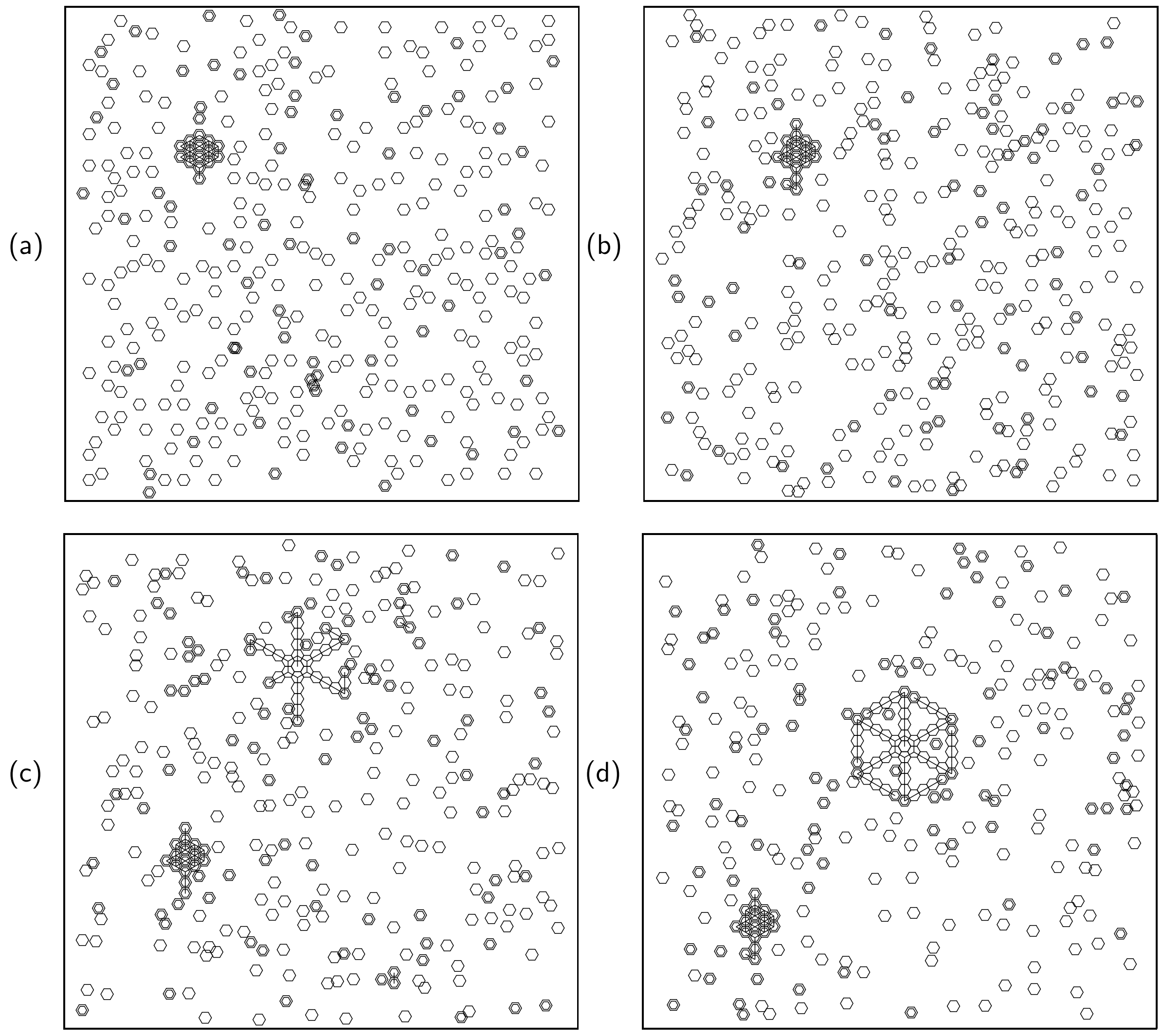


Figure 6.23: Constructing snowflake by the universal constructor: (a) step 0 – initial state; (b) step 110 – building the programs (P1–P6); (c) step 2383 – constructing the ring; (d) step 2910 – finished structure

should at last come close enough to the information string, to successfully activate it again. Because the time between deactivation and activation may be quite long, the constructor A has a chance to join another information string and to start another construction.

A similar strategy would be to perform some modification of the universal constructor – after completing the translation (i.e. after P9 activity) the constructor may turn into an *Inactive* state (that would prevent P1 from running). The state *Disconnected* may be achieved by some other structure Q activity. The description of Q may also be added to the information string that will turn into $\phi(Q+X)$. A drawback of this issue is that the universal constructor is no longer the consistent structure but some distributed set of independent structures.

6.2.3 Universal constructor replication

The original von Neumann's model of self-reproduction (3.2.1) consists of four components: the universal constructor A, the copying machine B, the control machine C and the universal Turing machine. Note, that in order to replicate A it is not necessary to implement the whole model[5]. This experiment was briefly described in: [170].

Starting from the random population of at least one constructor A with its description $\phi(A)$ we should observe the multiplication of instances of A. The most significant problems here are:

1. The universal constructor should not recognize the partially developed structure as a part of itself, e.g. the working P2 (on page 81) should always change the P4 (on page 83) – not recognising the developed P4 inside the new structure *etc.*

2. The constructed programs should not work until the whole structure has been completed.

3. The universal constructor does not allow to create a sophisticated bond structure, as presented in Fig. 6.16, compare to: Fig. 6.20.

The problem can be resolved in one of the following ways (compare to strategies described on p. 85):

1. By constructing another, complementary universal constructor A', i.e. a structure with different characteristic elements interpreting a different description syntax $\phi'(X)$. The self-replicating system should consists of both A, A' and their descriptions $\phi(A')$ and $\phi'(A)$. The process can be described by the equations:

$$\begin{aligned} A + \phi(A') &\Rightarrow A + \phi(A') + \mathbf{A'} \qquad (6.4) \\ A' + \phi'(A) &\Rightarrow A' + \phi'(A) + \mathbf{A} \end{aligned}$$

2. By constructing some intermediate structure I being able to transform itself into the universal constructor in a finite number time cycles:

$$A + \phi(I) \Rightarrow A + \phi(I) + I \Rightarrow A + \phi(I) + A + \mathbf{A} \qquad (6.5)$$

3. By constructing a set of programs $I_1, I_2, \ldots, I_n$ being able to build the constructor in a finite number of time cycles:

$$A + \sum_{i=1}^{n} \phi(I_i) \Rightarrow A + \sum_{i=1}^{n} \phi(I_i) + \sum_{i=1}^{n} I_i \Rightarrow A + \sum_{i=1}^{n} \phi(I_i) + \sum_{i=1}^{n} I_i + \mathbf{A} \qquad (6.6)$$

The resolution described in 2 was chosen as a basis for the self-replication experiment.

Encoding constructor

As described above, the experiment was performed by using the intermediate structure A' being able to transform itself into the universal constructor. The constructed structure was rearranged into the rhombus shape (Fig. 6.24, the shape is the same as in the example simulation in Fig. 6.22) and enhanced by the following activator programs: A1, A2 and A3.

[5]Unsuccessful attempt to construct the self-replication structure in the DigiHive environment is presented in [96]

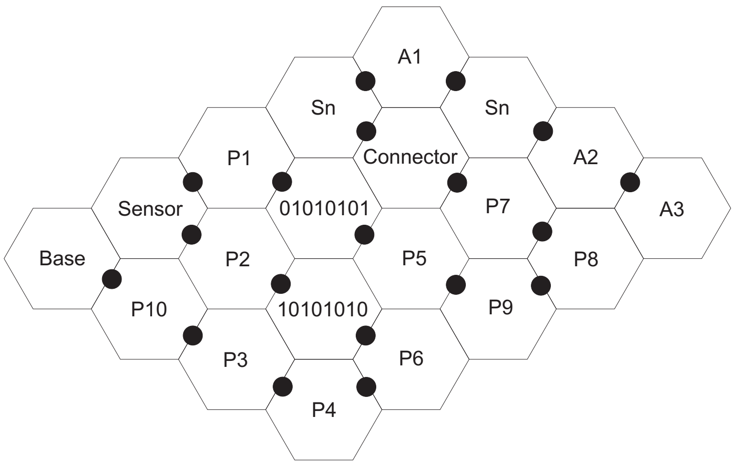


Figure 6.24: Inactive constructor schema – view from top.

The inactive constructor also has to be enhanced by the inhibitor sequences (mentioned earlier). Into all stacks (with the exception of the activator programs), the following sequence of particle types wes introduced: 48, 171, 255, 255, 0. While added to programs, the sequence was translated into the following predicate: `exists 11111111, mark V1` which contradicts other predicates (in every program, the variable `V1` is bound to the particle of type 11001100), i.e. prevents any of the constructor program from being executed. The sequence was also added inside the characteristic marks of the helper stacks and some programs (prevents from recognising the constructed stacks as the part of the universal constructor, until activation).

The existence of the inhibitor sequence should also stops A1 and A2. It is done by constructing the `not(structure)` predicates (see 5.2.2) which recognize the inhibitor sequence. The predicates (stacks described as Sn on structure schema) are bound directly to the A1 and A2 and are constructed before them.

The detailed algorithm of the activator programs:

A1 Program aimed at removing the inhibitor sequences from the surrounding stacks. The program works only if it is not bound to the base of any universal constructor. It should be the last constructed stack (the last active stack in a constructed structure). The program should start immediately after P8 (page 84) activity. Please note that the particles encoding the sequence of predicates `exists` (see 5.2.2) responsible for checking condition, should occur before any action predicate has been encoded. Otherwise the partly created program can work in an unpredictable way.

A2 Program aimed at rearranging half of the structure. Detailed algorithm: if there are no stacks with inhibitor sequences (possible only via A1 activity) then:

1. Find the stack of particles representing the "bottom" part of the constructor
2. Rearrange the bonds between the stacks into the valid part of the constructor (Fig. 6.25)

A3 Program aimed at rearranging the second half of the structure and disconnecting the activator part from the constructor. Detailed algorithm: if there are no stacks with inhibitor sequences (possible only via A1 activity) then:

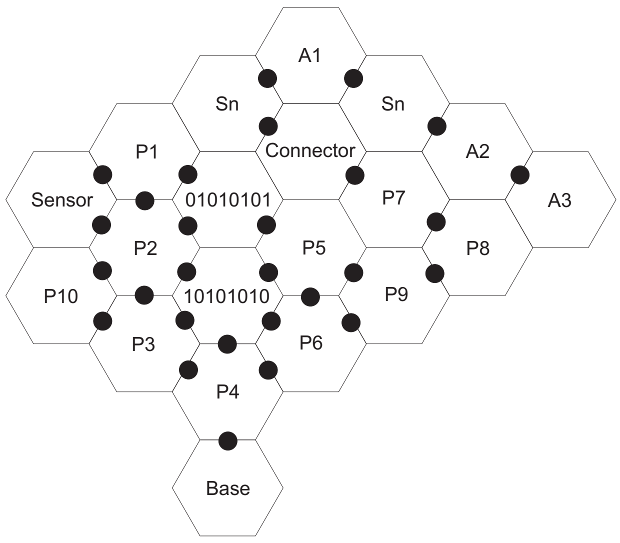


Figure 6.25: Activating constructor – schema after A2 activity.

1. Find the stack of particles representing the "upper" part of the constructor
2. Check how the found "upper" part is connected to the "bottom" part – if the bond connection isn't such as after A2 activity stop the program execution
3. Rearrange the bonds between the stacks into the valid part of the constructor (Fig. 6.26 – left part)
4. Disconnect the activation part from the constructor (Fig. 6.26 – right part)

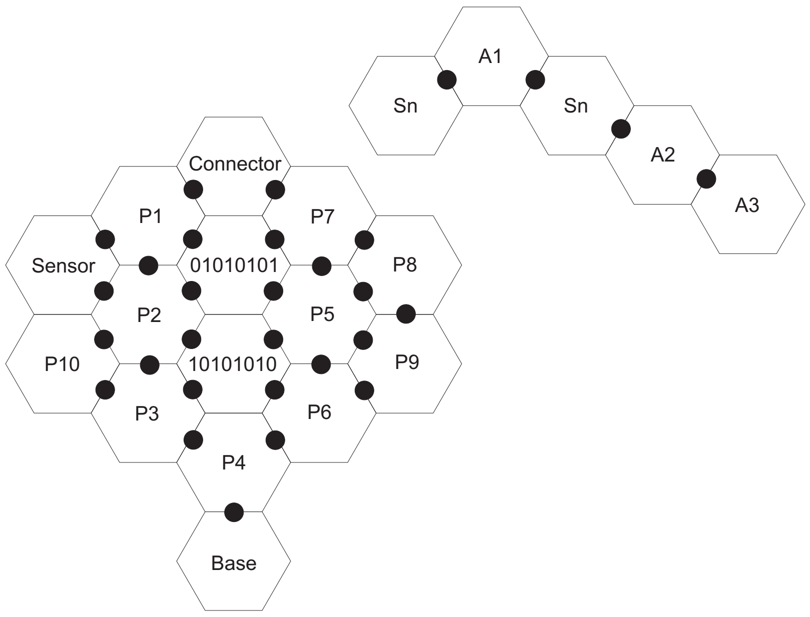


Figure 6.26: Activating constructor – schema after A3 activity. The left part consists of a fully functional universal constructor *A*.

The structure *I* was encoded by a single stack of 3569 particles. Each stack of particles can be interpreted as a program if only there exists particle of type 1111xxxx (5.2.5). The stack indeed contains such a particle, so it forms a valid program of 257 `exists` predicates and 100 action commands. The searching part of the program works properly, but action commands are inconsistent, then there is no impact on environment.

Performing simulation

The simulation has been performed in an environment of size 30×30. The universal constructor A was placed close to the information string $\phi(A')$ in the middle of the environment. The building stacks were placed around, along the circle shape of radius 8 (in a distance of at least 2 particle diameters from the constructor). The prepared building stacks consist always of the same types (apart from the header), i.e. there is only one building stack for each type of particle. The length of stacks has been calculated in accordance with the distribution of types in the universal constructor; the needed number of types were rounded to the minimal number divisible by 50. Because not every type were necessary (e.g. there is no particle of type 00001101 in the universal constructor), the initial state consists of 96 building stacks with length varying from 51 to 351.

In order to provide the source of energy for the programs (5.2.4), each building particle was initially supplied with internal energy (its whole amount was equal to 58387 at the beginning). Because each building stack has also randomly chosen velocity (from range -1 to 1 both vertically and horizontally) another source of energy comes from its kinetic energy – creating a bond between X and the building particle acts as an inelastic collision (see page 45).

The physics setting were set for default values, i.e.:

1. only elastic particle-particle collisions were allowed
2. the probability of a spontaneous emission of photons was set to 0 (see 5.1.2)
3. the only photon-particle collision permitted, was the inelastic one with photon absorption and conversion into particle's internal energy

The aim of the setting was to ensure a fully predictable run of the simulation (no random events caused by the photons) and to guarantee the source of energy for the participating programs.

The Ω size was set to 21, which actually means that the whole environment was in sight of every constructor's program. This helps to reduce the number of wasted time cycles where no program was successfully executed.

The program rotation (5.2.4) was disabled as a very time consuming feature (the single time cycle lasts up to 6 times longer with rotation enabled). Due to asymmetric bonds between its state stacks (Fig. 6.16), the constructor works only with one "default" orientation (examples from Fig. 6.22 and 6.23 were run with orientation enabled).

The simulation has been performed on a computer with Intel Q6600/2.40GHz processor. The 6100 time cycles took about 3 months of continuous calculations. The result is presented in Fig. 6.27. After 6080 cycles the environment consists of both the constructor A and the inactive constructor A'. After 20 more cycles the A' transforms itself, and the environment contains 2 equivalent universal constructors. An universal constructor A that has just finished translation, immediately joins the information string $\phi(A')$ again. Another construction will certainly fail because the second constructor A_c will disturb the synchronisation between the working programs (after the activation of the A_c the constructor A can recognize the pieces of A_c as a part of itself).

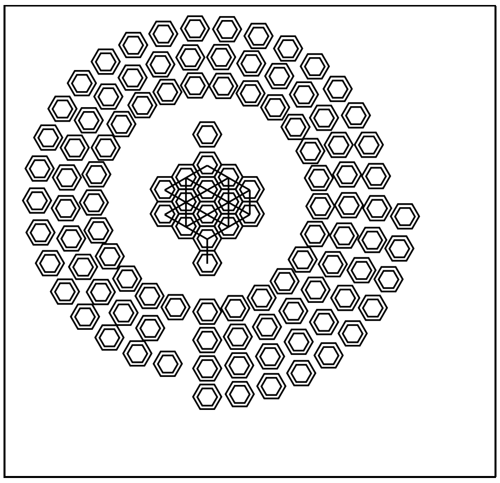
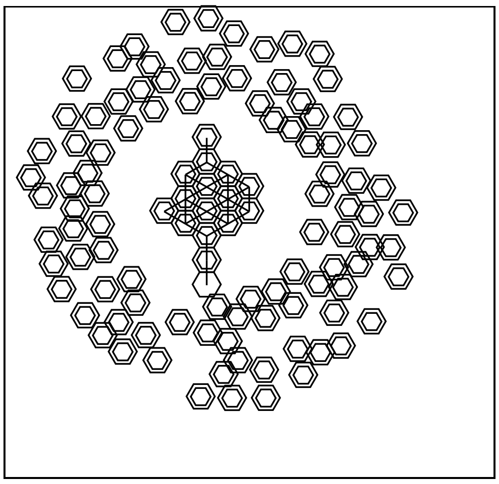
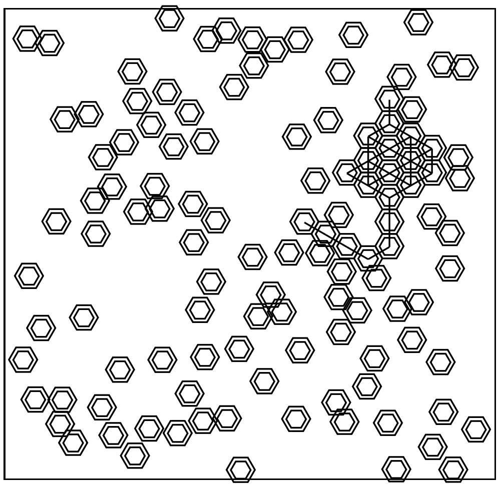
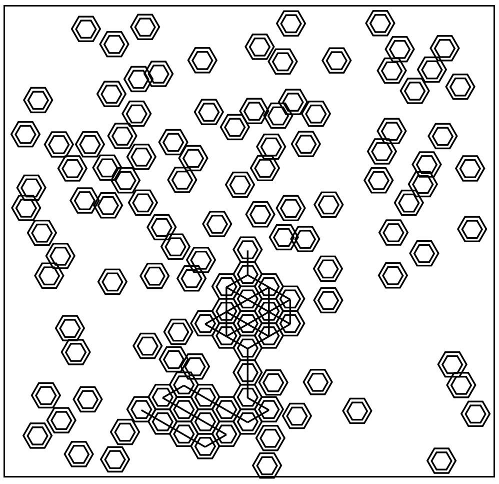
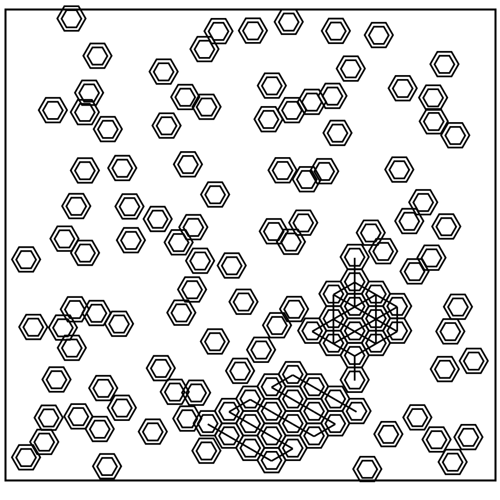
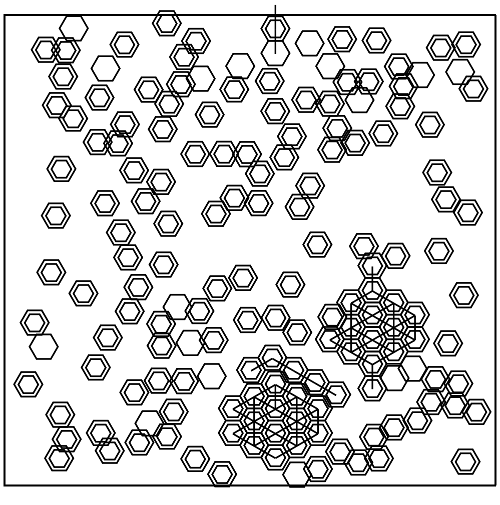
(a)

(b)

(c)

(d)

(e)

(f)


Figure 6.27: Simulation after 0 (a), 1 (b), 1790 (c), 4735 (d), 6080 (e) and 6100 (f) simulation cycles. Note that in the last screenshots there some "flat" structures (particles without any vertical bonds) as the result of A1 activity.

```
exists ::= [not] [[c] | [like f & m]] [[[not] bound [to f] [in d]]
      | [adjacent [to f] [in d] ]], [mark f]
m ::= [0|1|, 0|1, 0|1, 0|1, 0|1, 0|1, 0|1, 0|1]
c ::= [0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×, 0|1|×]
f ::= V short
short ::= 0|1|2|3|4|5|6|7|8|9
d ::= N|NE|SE|S|SW|NW|U|D
```

Figure 6.28: BNF syntax of the `exists` predicate after allowing direct comparing. This figure is an extension of Fig. 5.2 on p. 49.

6.3 Conclusions and further research

The above experiments show that the DigiHive environment is suitable for performing sophisticated simulations of complex systems. Simulations with the universal constructor, which is, at the moment, the most complicated complex system modelled in the environment, highlights some weak points which have to be resolved.

The universal constructor A works, but is too complex to be used in more interesting simulations. Apart from its poor performance, probably the most important drawback is the lack of possibility of simulating the behaviour of a population of more than one constructor. The interference between the constructors ensures, that any such experiment will fail. For the similar reason, extending the constructor with the machine B capable of copying the information string would lead to serious problems. The machine B would be a set of programs very similar to the universal constructor, but slightly less complicated – the main difference is, that the copying machine does not need the programs that encode direction (P5-P6).

The problem of interference between A and B also occurs. For instance, when A is constructing a structure, it injects a sensor into the information string (p. 79). The sensor may be confused with the information string by B, and then may be present in the copy of the string. On the other hand, the sensor part of B may also be confused with the information string by A, and could be interpreted in a wrong way. In fact, such an interference would probably lead to the halting of both A and B. The simple solution is to extend A with a new program capable of recognising the fact of the copying process (by B), and to extend B with a similar program capable of recognising the fact of translation (by A). The A machine would then have some knowledge about B and A would have some knowledge about B. This solution would probably work, but is far from perfect.

6.3.1 Extending list of commands

The poor performance of the universal constructor is the result of technical tricks necessary for the recognising particles encoded in the information string of for recognising direction (see templates on p. 77).

The optimization of recognition of type, which is vital to precisely build programs, seems to be the most important issue. If the environment language had been enhanced with the command that allow comparing types of the particles, the universal constructor would be much simpler.

top

×,×,×,×,×,×,×,×
direction
pointers (2) – 1 byte
like mask
pointer (1)
specification
exists header (0,1,1,0,×,×,×,×)

bottom

Figure 6.29: Encoding of the `exists like` predicate in a stack of particles. Bytes taken from type of particle – extension of Fig. 5.9.

The proposed new form of the `exists` predicate is shown in Fig. 6.28, the encoding schema is presented in Fig. 6.29. Apart from the abilities listed on p. 50, the new predicate would be able to find particles of types like the one stored in a variable. It would also be possible to adjust the accuracy of comparison via an additional mask, e.g.

1. `exists like V1 & 11110000` – find a particle of a type in which 4 first bits are the same as in the `V1` particle,

2. `exists not like V6 & 11110000` – find a particle of a type in which at least one of the 4 first bits are not the same as in the `V6` particle.

The second issue, recognition of direction, can be resolved by the observation that it is not necessary to encode the full version of the `SPLIT` command. The accurate encoding of direction seems to be redundant, as the universal constructor isn't able of directly constructing any possible shape. Program that will interpret all forms of `SPLIT` (`N`, `NW`, `SW`, `S`, `SE`, `NE`) as one particular form, e.g. `SPLIT N`, is sufficient for use in the universal constructor (it is also possible encode other, specific form of `SPLIT` – i.e. 6 different programs for `SPLIT`).

6.3.2 Horizontal bonds

The problem with the population of the constructors is related with the way of processing the information string. The only way of processing a vertical string is to inject some structure inside it (the sensor), which temporary change the string itself. A possible solution is to change the way of constructing the information string into the horizontal structure [89], so that the sensor can join in a way that will not necessarily change the string (e.g. join horizontally to the currently processed part of the string). This solution is unfortunately impractical due to serious performance problems (see also annotation on p. 79).

The best solution is to allow of creating the vertical bonds at different level of stacks, as presented in Fig. 6.30, which is at the moment forbidden by the bond limitation rule (see p. 44. However, without any limitation, the structures as shown in Fig. 6.31, will be allowed to emerge. This is unnecessary complicated, and leads to computational problems during collision resolving (p. 45). In order to keep the model simple, the stacks would be treated as ones of infinite height during collisions.

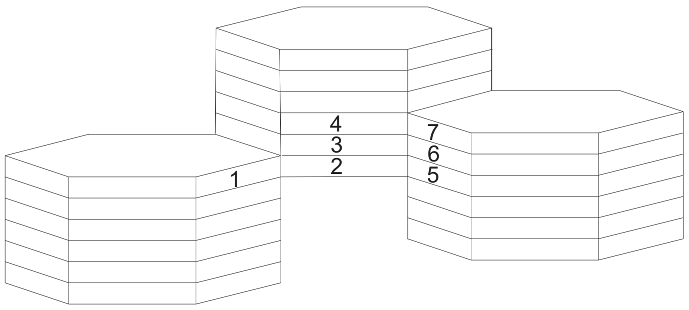


Figure 6.30: New possible horizontal bonds. If particle 1 is bound to 2 and 2 is bound to 5, other possible bonds are 3 to 6 and 4 to 7

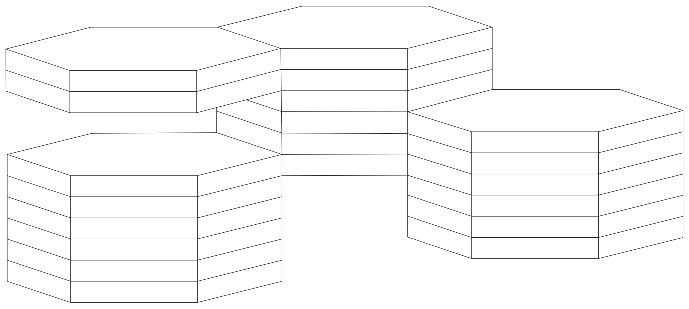


Figure 6.31: Example of a structure that will still be impossible to emerge after introducing of the new rules.

6.3.3 Distributed constructor

The above enhancements make it possible to drastically simplify the universal constructor, which no longer needs to be a consistent spatial structure, but may be formed as a set of independent, distributed in space parts. The proposed schema is presented in Fig. 6.32. Note that the distributed universal constructor does not need any horizontal bonds. Especially, due to elimination of the "activation" states (see Tab. 6.1) the constructor no longer needs to store its internal state. The sensor also plays the role of the connector and the base (Fig. 6.16) – after connecting to the information string via P1, the new structure can be constructed joined directly to the sensor. Note, that the sensor can be encoded as a negated structure containing just one predicate: `exists xxxxxxxx` (means: exists any particle), which prevents any constructed program from running.

The minimal constructor, which is being able to build its copy using a single information string (see 6.2.3) is the set containing the following elements: Sensor, P1, P2, P4 and P5 – note, that `SPLIT` command is not necessary to perform this experiment. In fact, also P5 may be absent at the beginning of the experiment, because it can be encoded in the information string and built via activity of other programs.

A parallel processing of the information string is quite simple, as different sensors can be easily joined to the string.

6.3.4 Distributed self-replicator

In order to build the full self-replicator, the set from Fig. 6.32, should be enhanced with just one program P7 (similar to P1 from Fig. 6.32), which finds a particle of a type pointed by the sensor and puts it on the top of the constructed structure.

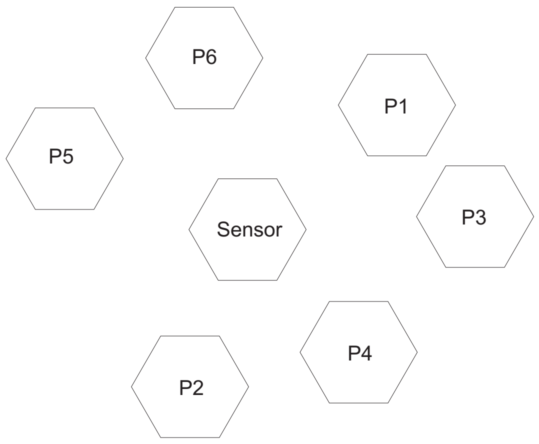


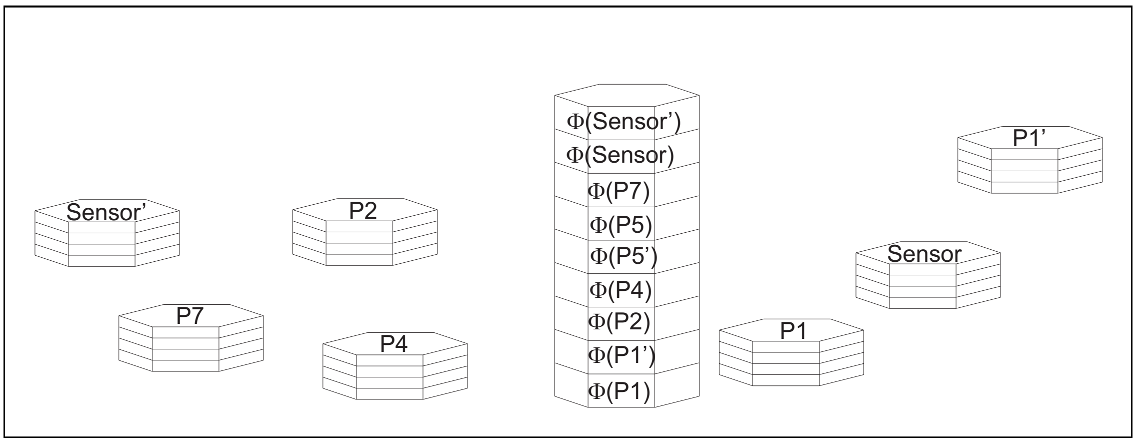
Figure 6.32: Distributed constructor schema – view from top. P1 – program is looking for the information string and connects the sensor to the string, P2 – program encodes the `PUT` command, P3 – program encodes the simplified `SPLIT` command (see discussion on p. 95), P4 – program encodes the `NEW` command, P5 – program encodes the `END` command, P6 – program moves the sensor in case of "junk" information, Sensor – counterpart of Sensor, Connector and Base from Fig. 6.16.

Contrary to other programs, P7 should move the sensor by only one step upwards (i.e. one particle), as there is no difference between the header and the data particle (see Fig. 6.13 and 6.17). For this reason, it is necessary to additionally create slightly different versions of: the sensor (Sensor') – which will not be recognized as a sensor by other programs, P1 (P1') and P5 (P5') – which will operate on Sensor' instead of Sensor.

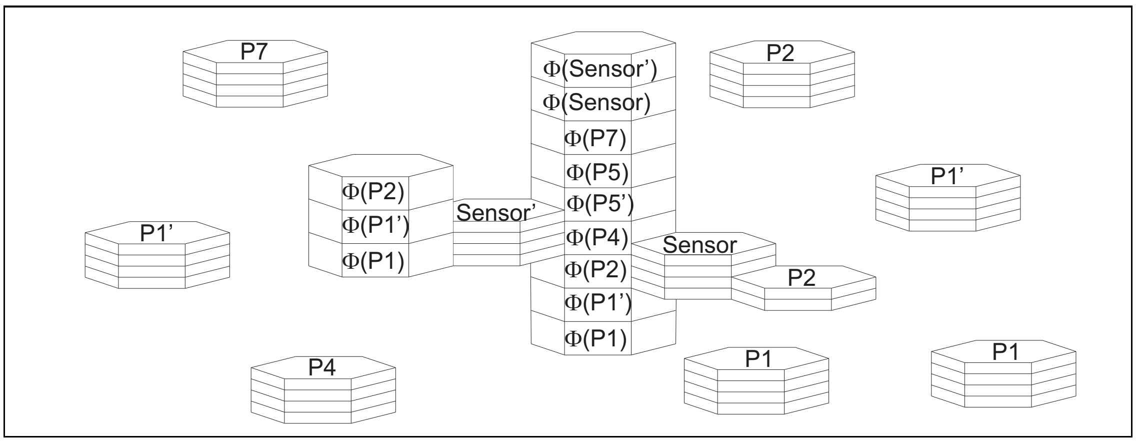
An expected course of the self-reproduction experiment is presented in Fig. 6.33. Note, that the initial state does not contain every program, only the minimal set needed to quickly start the experiment (in fact, the experiment should succeed even in case of an absence of Sensor', P1' and P7' in the initial state).

Due to its distributed form, the self-replication should be understood in terms of increasing the concentration of its component parts. It seems to be an attractive alternative, but may cause other problems in large environments. The concentration of programs may be too low to maintain the efficiency of the process at a reasonable level. The self-replicator may still be the consistent spatial structure (as in the previous version of the universal constructor), but such a "step back" does not seem to be the best direction of DigiHive development. Probably, simplified repulsive and attractive forces will be the introduced in the future, that will lead to the emergence of "cell membranes". The membranes should allow the spatial separation of individual self-replicators in a manner reminiscent of its biological counterparts.

(a)



(b)



(c)

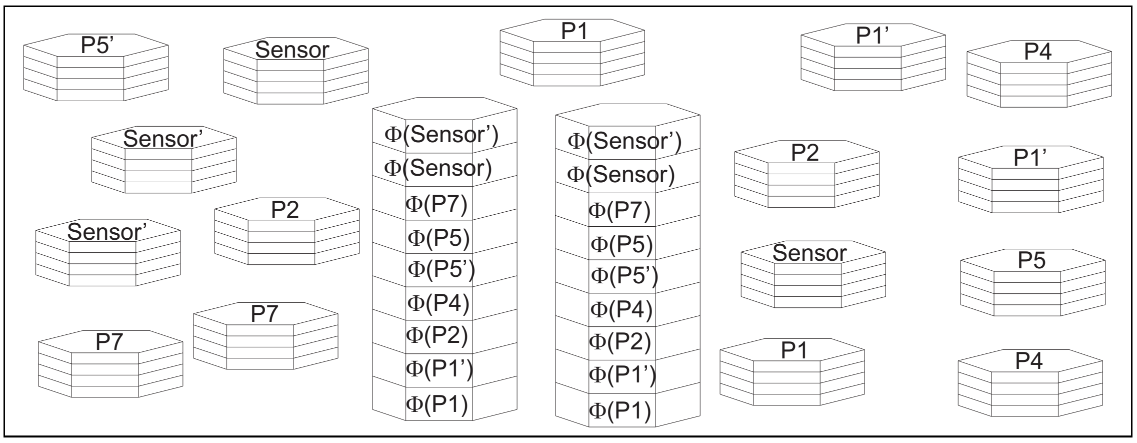


Figure 6.33: Proposed self-reproduction experiment: (a) initial state, (b) state during experiment, (c) expected state after experiment.

Chapter 7

Summary

This dissertation concerns the design, implementation, performing experiments and proposition of the further research in the new simulation environment, the “artificial world”, named DigiHive.

The environment is a complete, low-level, and closed system, obeying the momentum and the energy conservation laws, consisting of the space, moving objects and rules that govern their interactions. The central idea of the environment is that the objects interacts according to the simple rules (“thermodynamics”) an simultaneously, the structure of objects describes specific programs, which are read, interpreted and realized by the environment (“biochemistry”). This allows to model the evolution of objects mutually modifying their internal structures and thus their programs leading to the emergence of new structures containing new, unpredictable programs.

The main achievements of this work are:

1. The concept of the DigiHive environment which includes:
    (a) Two levels of interaction between the objects: On the basic level particles and complexes of particles move and collide. The effects of collisions between the objects are breaking or creation of bonds between them with an assumed probability (from deterministic to completely random rules). This allows for spontaneous creation of various structures. On the second level the structures (complexes of particles) make changes (creating or breaking the bonds) in surrounding objects according to their functions specified in a specially defined language and coded in their structures.
    (b) A new, declarative, high level, Prolog-like language encoded in the structures of the particles. The language has a small set of instructions, that allows the program to selectively create and break bonds between particles in its nearest space. To some extend, the language has also “soft” features, i.e. small changes in program code usually lead to relatively small changes in program behaviour
2. The successful implementation of the DigiHive environment, which was related to the number of technical problems that have been resolved, e.g.:
    (a) the translation of a high level Prolog-like program into the Prolog program,
    (b) the Prolog interpreter has been implemented and embedded into the environment,

3. The various simulations performed, which confirmed the utility of the environment in simulating the various aspects of evolution of complex systems. The simulations may be classified into two groups:
   (a) The interaction of complexes of particles showing emergent behaviour,
   (b) The implementation of the universal constructor which constructs complexes of particles on the base of the description contained in other complexes.

The already performed experiments, especially with the universal constructor, give rise to the ideas for the number of the environment's modifications (described in Sect. 6.3), which are currently being introduced. Especially the poor performance of the environment makes it difficult to perform the large scale simulations. Pending further research include the completion of the self-replication system and comparison of dynamics of various self-replication systems working in the random environment.

7.1 List of author's publications

The main contribution of this dissertation is included in the following papers:

1. Rafał Sienkiewicz and Wojciech Jędruch. Self-organization in artificial environment. In M. Kłopotek and J. Tchórzewski, editors,*Proceedings of Artificial Intelligence Studies*, volume 3 (26), Siedlce, Poland, 2004. Institute of Computer Science University of Podlasie, Publishing House of Univesity of Podlasie.

2. 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.

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

4. 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.

5. Rafał Sienkiewicz and Wojciech Jędruch. Artificial environment for simulation of emergent behaviour. In B. Bieliczynski et al, editor, *Adaptive and Natural Computing Algorithms: 8th International Conference, Icannga 2007, Warsaw, Poland, April 11–14, 2007, Proceedings, Part I*, volume 4431/2007 of LNCS, pages 386–393. Springer, 2007.

6. 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, Łódź, Poland, 2007.* (in Polish, available on-line at: http://www.fccs.wshe.lodz.pl/fccs2007/artykuly/sienkiewicz.pdf).

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

8. 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*, Lecture Notes in Computer Science, 2009. (in press).

9. 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.

Appendix A

The DigiHive environment details

This chapter contains an instruction for use of the DigiHive application. Sect. 1.1 describes program installation. Sect. 1.2 contains executable file specifications and an interface guide. Sect. 1.3 contains a detailed description of state files, which are basic tools for preparing simulations. Sect. 1.4 describes experiment files, that automatise user activities during long-term experiments.

1.1 Installation

Downloadable zip files can be found on the project website [168]. 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 (6.2.3) 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.

1.1.1 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 (1.3.3)
  `+- save` – default folder for experiment definition files (1.4)
    `+- bin` – default folder for binary state files (1.3.1)
    `+- xml` – default folder for xml state files (1.3.2)

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

`+- src` – environment sources

1.2 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 – 1.2.1) 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 (1.4),
`-lb file`: runs DigiHive in GTK mode, loads the specified binary file (1.3.1),
`-lx file`: runs DigiHive in GTK mode, loads the specified xml file (1.3.2),
`-le file`: runs DigiHive in GTK mode, loads the specified experiment file (1.4),
`-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.

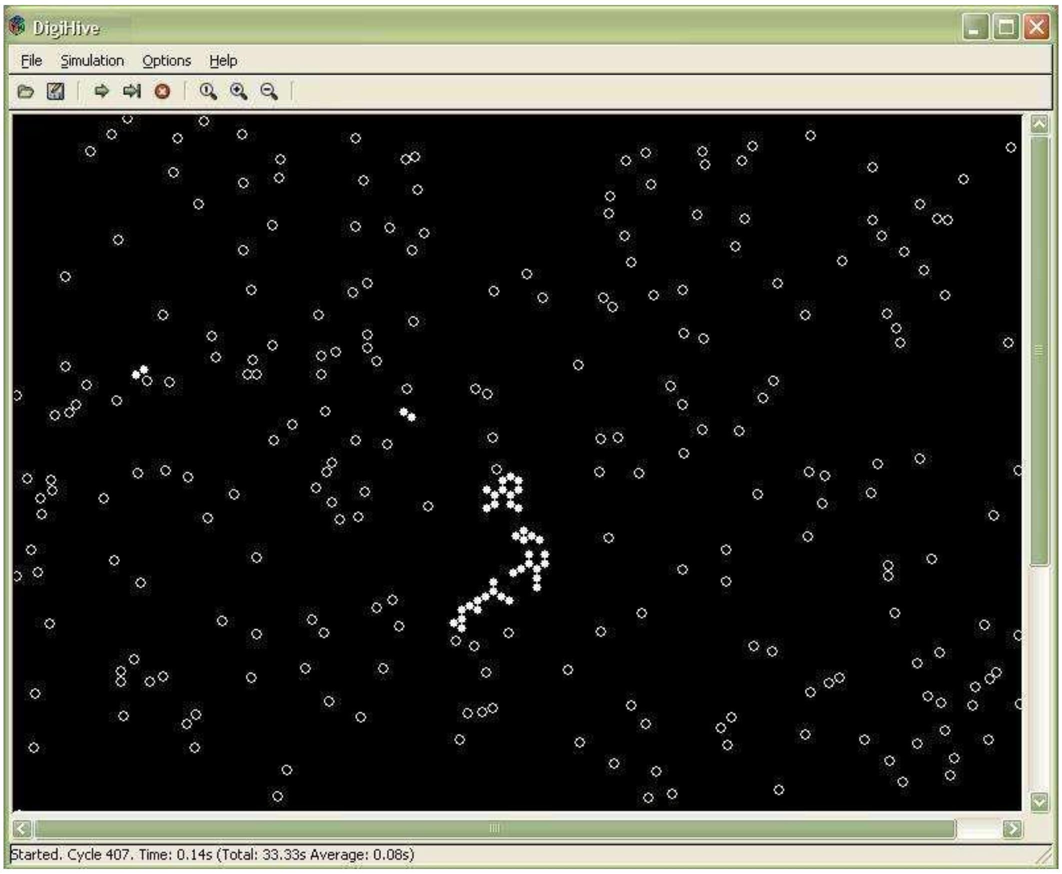
1.2.1 Interface

User interface is displayed only when the environment is running in GTK mode (1.2). The screenshot from the main window is presented in Fig. A.1.

Menu

Program menu contains the following options:

1. File
   (a) Open Binary: loads the environment state from a binary file (see 1.3.1). Option accessible also from the toolbar.
   (b) Open XML: loads the environment state from an XML file (see 1.3.2).
   (c) Save as Binary: saves the current state as a binary file (see 1.3.1). Option accessible also from the toolbar.
   (d) Save as XML: saves the current state as an XML file (see 1.3.2).
   (e) Save as JPEG: saves the current screen as JPEG file.



```
Started. Cycle 407. Time: 0.14s (Total: 33.33s Average: 0.08s)
```

Figure A.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 1.4). Name of the experiment is displayed on the application's title bar.

(h) Exit: closes the environment

2. Simulation

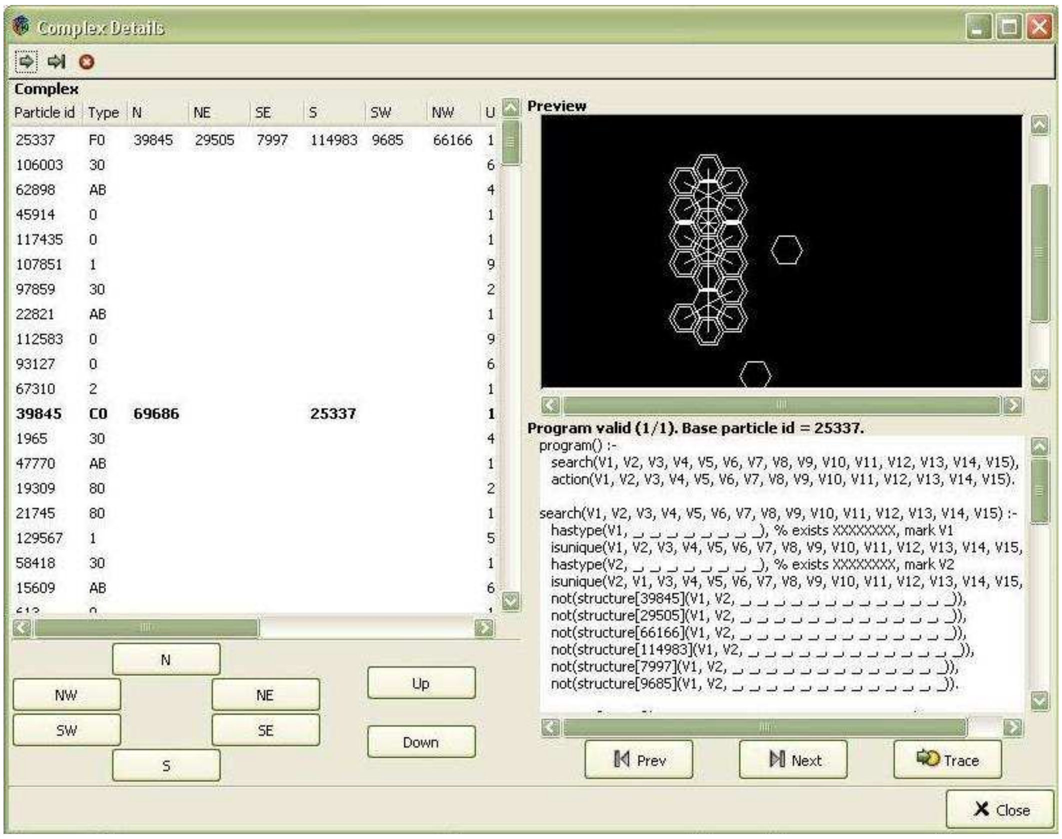
(a) Start: starts the simulation. The environment state should be previously loaded via one of the "Open" options. The option is also accessible from the toolbar. If the state was loaded via the Experiment definition file (1.4) each step may be related with saving additional information into experiment 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 1.4), with the exception of program listings.

3. Options



Particle id	Type	N	NE	SE	S	SW	NW	U
25337	F0	39845	29505	7997	114983	9685	66166	1
106003	30							6
62898	AB							4
45914	0							1
117435	0							1
107851	1							9
97859	30							2
22821	AB							1
112583	0							9
93127	0							6
67310	2							1
39845	**C0**	**69686**			**25337**			**1**
1965	30							4
47770	AB							1
19309	80							2
21745	80							1
129567	1							5
58418	30							1
15609	AB							6

Program valid (1/1). Base particle id = 25337.

```
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) :-
   hastype(V1, _ _ _ _ _ _ _ _), % exists XXXXXXXX, mark V1
   isunique(V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15,
   hastype(V2, _ _ _ _ _ _ _ _), % exists XXXXXXXX, mark V2
   isunique(V2, V1, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15,
   not(structure[39845](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)),
   not(structure[29505](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)),
   not(structure[66166](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)),
   not(structure[114983](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)),
   not(structure[7997](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)),
   not(structure[9685](V1, V2, _ _ _ _ _ _ _ _ _ _ _ _ _)).
```

Figure A.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 1.3.2).

(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

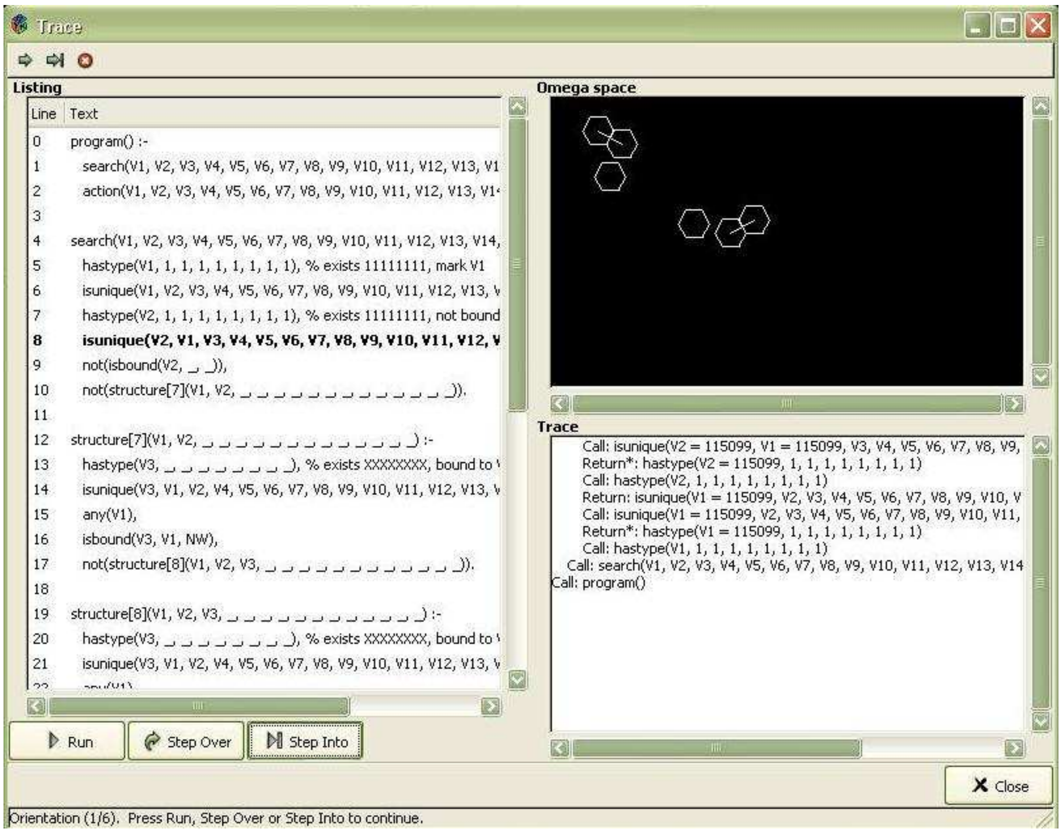
   (a) About: displays the version and the build date.

Complex details

Click the left mouse button on the particle showed in the main window, this displays its details as shown in Fig. A.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



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

search(V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14,
  hastype(V1, 1, 1, 1, 1, 1, 1, 1, 1), % exists 11111111, mark V1
  isunique(V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V
  hastype(V2, 1, 1, 1, 1, 1, 1, 1, 1), % exists 11111111, not bound
  isunique(V2, V1, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V
  not(isbound(V2, _, _)),
  not(structure[7](V1, V2, _, _, _, _, _, _, _, _, _, _, _, _, _, _)).

structure[7](V1, V2, _, _, _, _, _, _, _, _, _, _, _, _, _, _) :-
  hastype(V3, _, _, _, _, _, _, _, _), % exists XXXXXXXX, bound to
  isunique(V3, V1, V2, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V
  any(V1),
  isbound(V3, V1, NW),
  not(structure[8](V1, V2, V3, _, _, _, _, _, _, _, _, _, _, _, _, _)).

structure[8](V1, V2, V3, _, _, _, _, _, _, _, _, _, _, _, _, _) :-
  hastype(V3, _, _, _, _, _, _, _, _), % exists XXXXXXXX, bound to
  isunique(V3, V1, V2, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V
```

Trace

```
    Call: isunique(V2 = 115099, V1 = 115099, V3, V4, V5, V6, V7, V8, V9,
    Return*: hastype(V2 = 115099, 1, 1, 1, 1, 1, 1, 1, 1)
    Call: hastype(V2, 1, 1, 1, 1, 1, 1, 1, 1)
    Return: isunique(V1 = 115099, V2, V3, V4, V5, V6, V7, V8, V9, V10, V
    Call: isunique(V1 = 115099, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11,
    Return*: hastype(V1 = 115099, 1, 1, 1, 1, 1, 1, 1, 1)
    Call: hastype(V1, 1, 1, 1, 1, 1, 1, 1, 1)
  Call: search(V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14
Call: program()
```

Orientation (1/6). Press Run, Step Over or Step Into to continue.

Figure A.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. A.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

1.3 DigiHive state

The environment state can be saved and read in binary files (1.3.1) or *xml* definition files (1.3.2). Sets of environment settings are stored in settings files (1.3.3).

1.3.1 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 (1.3.2), 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 (1.4) are stored as binary files only.

1.3.2 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 [168][3]). A detailed description of creating the XML and DTD files can be found in [153].

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 (1.3.3), 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 (5.1.1): positive real value, e.g. `ei="0.25"`. If absent, the the internal energy is set to 0,

3. `type` – particle type (5.1.1): 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 1.3.2). 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 5.2.5).

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>
```

1.3.3 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 (1.1.1) 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. 45
   (b) `eac` – activation energy – see definition on p. 44
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. 45
   (b) `prob-emit-pho` – the probability of a spontaneous photon emission – see description on p. 45
   (c) `emaxpho` – maximal photon energy after a spontaneous emission – see description on p. 45
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. 46
   (c) `prob-rebound` – the probability of rebounding of the particle hit by the photon from an adjoining particle – see description on p. 46
   (d) `prob-bind` – the probability of creating a horizontal bond between the hit particle and the adjoining particles – see description on p. 46
   (e) `prob-unbind` – the probability of removing the horizontal bond between the hit particle and the bound particles – see description on p. 47
   (f) `prob-concatenate` – the probability of creating a vertical bond between the hit particle and the adjoining particles – see description on p. 47
   (g) `prob-split` – the probability of removing the vertical bond between the hit particle and the bound particles – see description on p. 47
   (h) `prob-add-to-inner` – the probability of photon absorption – see description on p. 48
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. 61.

(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, Drespectively. 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.

1.4 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 (1.3.1). 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 (1.3.2)
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 1.3.2).

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>
```

Appendix B

Sample source file

This chapter contains a sample state file: “snowflake.xml” which contains data for the experiment described in Sect. 6.1.1.

2.1 *Snowflake.xml* listing

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE universum PUBLIC  "-//DIGIHIVE//DTD UNIVERSUM 1.0//EN"
    "http://www.swarm.eti.pg.gda.pl/dtd/universum.dtd">
<universum ccn="0" width="100" height="100" >
    <settings>
        <settings-common time="1.00" eac="0.01"  />
        <settings-particles prob-col-elastic="1.0"
             emaxpho="1.0" prob-emit-pho="0.05"  />
        <settings-photons enabled="true" prob-reflect="0.33"
             prob-rebound="0.0" prob-bind="0.0" prob-unbind="0.0"
             prob-reqroup="0.00" prob-concatenate="0.0"
             prob-split="0.0" prob-add-to-inner="1.0"/>
        <settings-programs enable-self-mutable-programs="true" />
    </settings>
    <particles>
        <particle type="0" id="1" ei="10">
            <velocity x="0.1" y="-0.2"/>
        </particle>
        <particle type="0" id="2" ei="10">
            <velocity x="0.1" y="0.2"/>
        </particle>
        <particle type="0" id="3" ei="10">
            <velocity x="0.4" y="-0.2"/>
        </particle>
        <particle type="255" id="4" ei="10">
            <velocity x="0.4" y="-0.2"/>
        </particle>
    </particles>
    <programs>
     <!--  Program binding two particles and after that
           deactivating itself by unbind from itself
           a particle from the bottom of the stack which
           marks beginning of the program -->
        <program id="3">
            <position x="70" y="10"/>
            <velocity x="0.3" y="0.41"/>
            <!-- Searching for two unbind particles -->
            <search>
                <structure>
                    <exists>
                        <type>00000000</type>
                        <not-isbound/>
                        <mark>V1</mark>
                    </exists>
                    <exists>
                        <type>00000000</type>
                        <not-isbound/>
                        <mark>V2</mark>
                    </exists>
                   <!-- Searching for particle marking
                        beginning of the program -->
                    <exists>
                        <type>11110000</type>
                        <isbound>
                            <in>U</in>
                        </isbound>
                        <mark>V3</mark>
                    </exists>
                    <exists>
                        <type>00110000</type>
                        <isbound>
                            <to>V3</to>
                            <in>D</in>
                        </isbound>
                        <mark>V4</mark>
                    </exists>
                    <exists>
                        <type>10001011</type>
                        <isbound>
                            <to>V4</to>
                            <in>D</in>
                        </isbound>
                        <mark>V5</mark>
                    </exists>
                    <exists>
                        <type>00000000</type>
                        <isbound>
                            <to>V5</to>
                            <in>D</in>
                        </isbound>
                    </exists>
                </structure>
            </search>
            <actions>
```

```
        <action type="bind" par1="V1" par2="V2" dir="S"/>
        <action type="unbind" par1="V3" par2="V4" dir="U"/>
        <action type="bind" par1="V3" par2="V4" dir="NW"/>
    </actions>
</program>
<!-- Program building the ring of particles -->
<program id="4">
    <position x="40" y="80"/>
    <velocity x="-0.3" y="-0.41"/>
    <!-- Recognizing a pair of particles being a part of
         the ring to which another particle can be bind -->
    <search>
        <structure>
            <exists>
                <type>00000000</type>
                <isbound>
                    <in>SW</in>
                </isbound>
                <mark>V1</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V1</to>
                    <in>NE</in>
                </isbound>
                <mark>V3</mark>
            </exists>
            <!-- Checking additional conditions that the
                 pair of particles belongs to the ring -->
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>NW</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>S</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>SW</in>
                </isadjacent>
            </exists>
            <exists>
```

```
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>SE</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>N</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V3</to>
                    <in>SE</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V3</to>
                    <in>NE</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V3</to>
                    <in>S</in>
                </isadjacent>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V3</to>
                    <in>NW</in>
                </isadjacent>
            </exists>
             <!-- Searching for the unbound particle -->
            <exists>
                <type>00000000</type>
                <not-isbound/>
                <mark>V2</mark>
            </exists>
        </structure>
    </search>
    <!-- Binding the particle to the ring -->
    <actions>
        <action type="bind" par1="V1" par2="V2" dir="SE"/>
    </actions>
</program>
<!-- Program closing the ring by checking the existence
     of 3 bound particles and one particle belonging
     to the same complex adjacent but not bound to them.

     Hint: checking of 4 particles (3 and 1) prevent from
     later erroneous recognition -->
<program id="6">
    <position x="20" y="40"/>
    <velocity x="-0.3" y="-0.31"/>
    <search>
        <structure>
            <exists>
                <type>00000000</type>
                <isbound>
                    <in>NW</in>
                </isbound>
                <mark>V1</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V1</to>
                    <in>SE</in>
                </isbound>
                <mark>V2</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V2</to>
                    <in>NE</in>
                </isbound>
                <mark>V5</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isadjacent>
                    <to>V5</to>
                    <in>N</in>
                </isadjacent>
                <mark>V6</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <not-isbound>
                    <in>N</in>
                </not-isbound>
                <mark>V6</mark>
            </exists>
           <!-- Searching for unbound particle 11111111 -->
```

```
                <exists>
                    <type>11111111</type>
                    <not-isbound></not-isbound>
                    <mark>V11</mark>
                </exists>
            </structure>
        </search>
       <!-- Closing the ring and binding the particle 11111111
            into the center of the ring, it marks the completion
            of the ring -->
        <actions>
            <action type="bind" par1="V5" par2="V6" dir="S"/>
            <action type="bind" par1="V5" par2="V11" dir="SE"/>
        </actions>
    </program>
    <!-- Two programs building the lateral branch to the ring.
         The first program binds to the ring a stack of two
         particles. The second program continue building of
         the branch -->
    <!-- First program -->
    <program id="7">
        <position x="60" y="20"/>
        <velocity x="-0.3" y="-0.6"/>
       <!-- Recognition of a particle belonging to the ring -->
        <search>
            <structure>
                <exists>
                    <type>00000000</type>
                    <isbound>
                        <in>NW</in>
                    </isbound>
                    <mark>V12</mark>
                </exists>
                <exists>
                    <type>00000000</type>
                    <isbound>
                        <in>NE</in>
                    </isbound>
                    <mark>V12</mark>
                </exists>
                <exists>
                    <type>11111111</type>
                    <isadjacent>
                        <to>V12</to>
                        <in>S</in>
                    </isadjacent>
                </exists>
                <!-- Searching for two unbound particles -->
                <exists>
                    <type>00000000</type>
                    <not-isbound/>
                    <mark>V13</mark>
                </exists>
                <exists>
                    <type>00000000</type>
                    <not-isbound/>
                    <mark>V14</mark>
                </exists>
            </structure>
        </search>
        <!-- Binding to the ring the vertical stack of two
             particles -->
        <actions>
            <action type="bind" par1="V13" par2="V12" dir="N"/>
            <action type="bind" par1="V13" par2="V14" dir="U"/>
        </actions>
    </program>
    <!-- Second program -->
    <program id="8">
        <position x="75" y="20"/>
        <velocity x="-0.3" y="-0.6"/>
        <!-- Recognition of the stack of two of particles bound
             horizontally to the ring by the action of previous
             program. -->
        <search>
            <structure>
                <exists>
                    <type>00000000</type>
                    <isbound>
                        <in>N</in>
                    </isbound>
                    <mark>V1</mark>
                </exists>
                <exists>
                    <type>xxxxxxxx</type>
                    <isbound>
                        <to>V1</to>
                        <in>D</in>
                    </isbound>
                    <mark>V5</mark>
                </exists>
                <!-- Searching for two unbound particles -->
                <exists>
                    <type>00000000</type>
                    <not-isbound/>
                    <mark>V3</mark>
                </exists>
                <exists>
                    <type>00000000</type>
                    <not-isbound/>
                    <mark>V4</mark>
                </exists>
            </structure>
           <!-- Checking for nonexistence of stack
```

```
            of 4 particles. -->
        <not-structure dir="NE" particleId="9">
            <exists>
                <type>xxxxxxxx</type>
                <isbound>
                    <to>V5</to>
                    <in>D</in>
                </isbound>
                <mark>V6</mark>
            </exists>
            <exists>
                <type>xxxxxxxx</type>
                <isbound>
                    <to>V6</to>
                    <in>D</in>
                </isbound>
                <mark>V7</mark>
            </exists>
            <exists>
                <type>xxxxxxxx</type>
                <isbound>
                    <to>V7</to>
                    <in>D</in>
                </isbound>
            </exists>
        </not-structure>
    </search>
    <!-- Adding one particle to the branch and enlarging
         the stack by one particle. -->
    <actions>
        <action type="bind" par1="V1" par2="V3" dir="S"/>
        <action type="bind" par1="V3" par2="V4" dir="U"/>
        <action type="unbind" par1="V1" par2="V5"/>
        <action type="bind" par1="V4" par2="V5" dir="U"/>
    </actions>
</program>
<!-- Program recognizing the end of the branch (stack of four
     particles) and initializing construction of the lateral
     branch. Initialization is by binding stack of two particles
     to stack of four particles marking the end of branch.  -->
<program id="9">
    <position x="11" y="36"/>
    <velocity x="0.8" y="-0.6"/>
    <!-- Recognition of the stack pf [articles bound
         to a particle in direction N. -->
    <search>
        <structure>
            <exists>
                <type>00000000</type>
                <isbound>
                    <in>N</in>
                </isbound>
                <mark>V1</mark>
            </exists>
            <!-- Additional checking -->
            <exists>
                <type>00000000</type>
                <not-isbound>
                    <in>NW</in>
                </not-isbound>
                <mark>V1</mark>
            </exists>
            <exists>
                <not-type>xxxxxxxx</not-type>
                <isadjacent>
                    <to>V1</to>
                    <in>N</in>
                </isadjacent>
            </exists>
            <!-- Continuation of recognition -->
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V1</to>
                    <in>D</in>
                </isbound>
                <mark>V5</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V5</to>
                    <in>D</in>
                </isbound>
                <mark>V6</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V6</to>
                    <in>D</in>
                </isbound>
                <mark>V7</mark>
            </exists>
            <exists>
                <type>00000000</type>
                <isbound>
                    <to>V7</to>
                    <in>D</in>
                </isbound>
            </exists>
            <!-- Searching for two unbound particles -->
            <exists>
                <type>00000000</type>
```

```
                        <not-isbound/>
                        <mark>V13</mark>
                    </exists>
                    <exists>
                        <type>11111111</type>
                        <not-isbound/>
                        <mark>V14</mark>
                    </exists>
                </structure>
            </search>
            <actions>
                <action type="bind" par1="V1" par2="V13" dir="NE"/>
                 <!-- Binding the particle V14 (11111111) will
                      prevent from later recognition of the
                      stack by the same program.  -->
                <action type="bind" par1="V13" par2="V14" dir="U"/>
            </actions>
        </program>
    </programs>
    <multiply>
        <!-- Making copies of programs  -->
        <multiply-complex id="4" multiply="8">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-complex>
        <multiply-complex id="6" multiply="8">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-complex>
        <multiply-complex id="7" multiply="8">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-complex>
        <multiply-complex id="8" multiply="8">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-complex>
        <multiply-complex id="9" multiply="8">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-complex>
        <multiply-particle multiply="300" id="3">
            <multiply-velocity miny="-1.0" maxy="1.0"
                               minx="-1.0" maxx="1.0"/>
        </multiply-particle>
        <multiply-particle multiply="200" id="4">
            <multiply-velocity miny="-0.5" maxy="0.5"
                               minx="-0.5" maxx="0.5"/>
        </multiply-particle>
    </multiply>
</universum>
```

Appendix C

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 $\in$ {0, 1, _} denote particle type
- v $\in$ {V1, ..., V15 } and vv $\in$ {V0, ..., V16 } denote variables
- d $\in$ {N, NE, SE, S, SW, NW, U, D} denotes direction

No	Name	Prolog	Specification encoding				
			type	relation	to	in	mark

1	`exists t is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is bound in d`	if t $\neq$ 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 is bound, mark vv`	if t $\neq$ 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 is bound`	if t $\neq$ 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 is 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),`
		`not(isbound(VV,V,d)).`
			0	00	11	11	1

34	`exists not t not is bound to v in d`	if t $\neq$ 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 is bound to v, mark vv`	if t $\neq$ 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 is not bound to v`	if t $\neq$ 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 is bound in d, mark vv`	if t $\neq$ 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 is 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 is 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 is 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 is 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 is 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 is 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 is 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 is adjacent in d, mark vv`	if t $\neq$ 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 is adjacent in d`	if t $\neq$ 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 $\neq$ 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 $\neq$ 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

Appendix D

Self-organisation in the Universum environment

This chapter describes the results of a self-organization experiment [171] performed in the *Universum* environment (4.3.3), the ancestor of the DigiHive. The experiment's conclusions were a direct inspiration to create the assumptions of the new environment.

4.1 Simulation experiments

Self-organization experiments started from the initial state of the system consisting of over 60000 five bit particles positioned randomly on a 1000×1000 lattice (32 possible 5 bit strings multiplied by 2048), and were run for 35000 steps.

4.1.1 Environment settings

Environment probabilities were set as shown in table D.1.

N	$P(N)$	Probability of
1	0.20	Particle-particle inelastic collision
2	1.00	Particle-photon inelastic collision
3	0.27	Rebounding particles
4	0.10	Setting bond between particles
5	0.06	Resetting bond
6	0.01	Changing order of atoms
7	0.30	Concatenating particles
8	0.01	Splitting particle
9	0.25	Increase internal energy of particle
10	0.05	Emission of photon
11	0.50	Change of orientation

Table D.1: Environment probabilities, $P(N)$

During each inelastic collision a photon is created so higher values of probability of particle-particle inelastic collision, $P(1)$, leads to more active environment. A similar effect has the probability of particle-photon inelastic collision, $P(2)$. $P(2) = 1$ means that every particle-photon collision causes one of the reactions described in 4.3.3 with probabilities $P(3), P(4), \ldots, P(9)$.

Higher values of probability of setting a bond between the particles, $P(4)$, yields to a creation of bigger complexes. The probability of resetting bond, $P(5)$ has the opposite effect. In the experiment, the bond creation occurs more often than its destruction ($P(4)/P(5) \approx 1.66$).

Most important for self-organization matters, are the probabilities which sets activities, that affect particle‘s functions. The relatively high value of probability of concatenating particles, $P(7)$, causes the average function length to grow and the total number of particles decreases. The probability of splitting particle, $P(8)$, by allowing to split the particle‘s atoms string in random position, causes a destruction of the existing particle‘s function. Particle splitting together with particle concatenation is the source of functional interaction diversity. The probability of changing the order of atoms, $P(6)$, allows the function to be changed in a random way.

The probability of increasing internal energy, $P(9)$, allows photon conversion into the particle‘s internal energy. Internal energy is one of the energy sources for functions realisation (changing of bond energy is another source). The relatively high value of $P(9)$ helps the emerged functions more often affects the environment‘s space. The probability of photon emission, $P(10)$ sets the probability of creating photon from the particle‘s internal energy, at the end of each time step. The low value of $P(10)$ yields to accumulating internal energy for a longer period of time.

The probability of change of orientation, $P(11)$, allows changing the particle‘s orientation at the end of each time step. Orientation defines the direction in which particle function operates. It has been noticed that such variable particle‘s orientation, noticeably helps completing some task, especially with constructing sophisticated spatial structures of particles.

4.1.2 Simulation

Five simulation experiments were realized, each starting with different kinetic energy of particles. They were run for over 30000 time steps, allowing the system to evolve from the initial state to some kind of equilibrium.

Total environment energy is, at the beginning of simulation, a sum of total kinetic energy and total atoms’ bond energy. Later, during the course of simulation, the energy is distributed over kinetic energy, atoms’ bond energy, particles’ bond energy and particles’ internal energy, however total energy remains constant.

All simulations were also repeated in an environment with functional interaction disabled. This enabled to see the influence of functional interactions on the evolution of the system,

4.2 Results

In all of the considered simulations, the total number of particles decreased from 65536 to about 6500–7000 after 34000 time steps and their mass increased (Fig. D.1). After about 10000–15000 time steps, particles functions influence become most important for this aspect of environment evolution and as a result both kind of environment have different attractors.

The number of photons grew, initially reaching its maximum at approximately 400th time step and then gradually decreased to some constant value. Such behaviour is a result of a huge number of particle collisions during the first time steps, and smaller

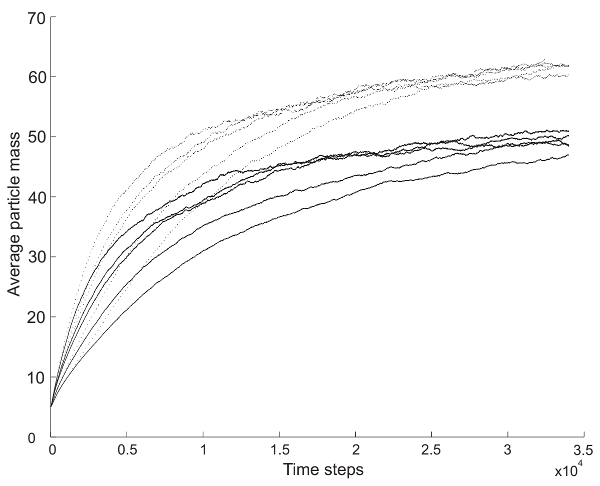


Figure D.1: Average particle mass. In this and in the rest of the figures the continuous lines depict simulation results with the functional activity turned on, and the dotted line with the activity turned off

number of particles and collisions later. The differences between systems with functional activity set on and off were not distinct.

Complexes numbers, as shown in the figure D.2, changed in the way similar to photons numbers. Their number reached its maximum after about 200–600 time steps, then decreased quickly. During the first time steps, the probability of creating a new complex is relatively high, due to many unbounded adjoining particles. When a large portion of particles becomes a part of complexes, the new complexes are hardly ever created (creating a bond reaction during photon-particle collision usually changes the existing complexes' length, as adjoining particles that are still unbounded are very seldom). On the other hand, environment settings yields to particle concatenation rather than splitting them, so there is no source of new unbounded particles.

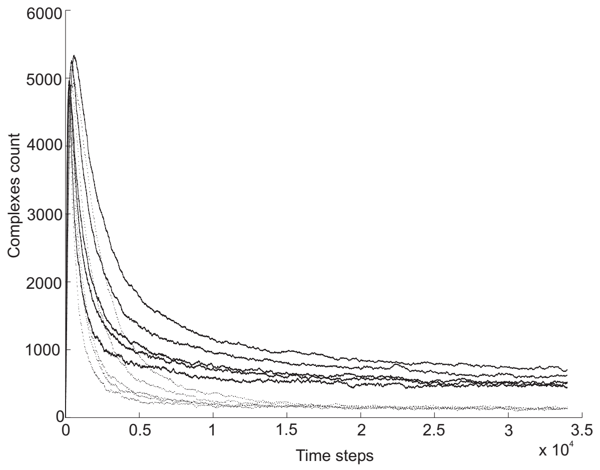


Figure D.2: Complexes number

If functional activities are turned off, complexes number stabilises itself after about 15000 time steps, regardless of initial settings, and remains approximately constant (about 150–250 complexes). Functional activities however, noticeably changes the complexes number evolution – the maximum values are higher than about 500 complexes and, like particles number, yields to different values in the equilibrium (from

500 for higher energy to 900 for the lowest one).

The average complex length, presented in the figure D.3, shows the most visible functional activities' influence on the environment state. If functional activities are turned off the average complex length reaches its maximum after 400 (highest energy) to 1400 (lowest energy) time steps, then decreases to about 2–2.1 particles per complex. Otherwise, the average complex length rises and stabilizes itself after about 2000 steps, at a higher level: from 2.4 to 2.7 particles per complex.

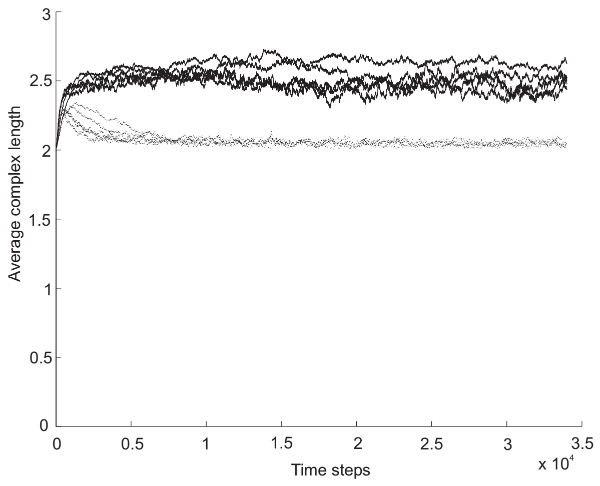


Figure D.3: Average complex length

4.2.1 Emerged functions

During simulations, the length of particles grew which resulted in more complicated functions. This is illustrated in Figure D.4 showing the average length of particles coding completed function. The presented curve depicts the average length of particles which function was executed in a given time step averaged in gate of length equal 100 time steps.

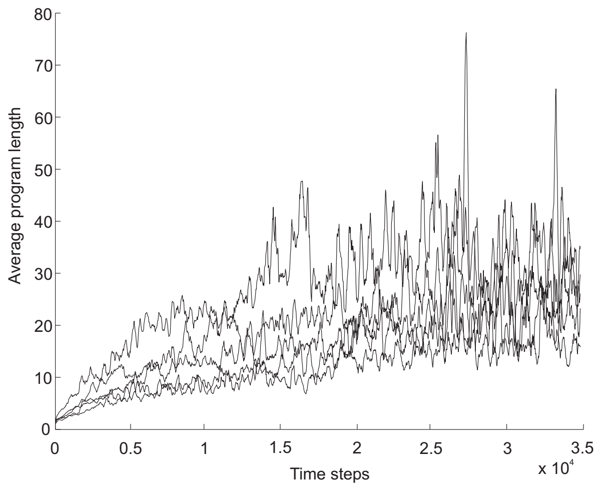


Figure D.4: Average completed programs length (particles string length).

During the first few hundred time steps, the emerged function are only capable of setting new bonds between the particles. This is because this kind of function is coded by short strings [1] so it can emerge after a few time steps. In fact it appeared even after 4-5 steps, when a particle with atom strings 10010 (according to section 4.1 there are 2048 of these particles at the beginning of the simulation) concatenates with any other particle. This simplest function activity noticeably affects the environment evolution.

After about 300-500 time steps, functions become slightly more complex. The new activity, however, does not affect the environment. Mostly, the new atom strings are not interpreted as valid commands (e.g. string 11101 does not encode any command), the new commands search space around the particle but usually ignore the results or are interpreted as a jump to a non existing part of functions etc. Occasionally, new commands present more interesting behaviour, like an example function which emerged at the 364th time step removing the bond to the particle which is left of it, then sets bond with the particle which is above it.

During the next steps, particle strings became longer, but the effect of greater length is reduced due to no correlated activity of consecutive commands. As an example: the function which emerged at the 8123rd time step of the simulation has 28 commands, but only 6 of them are executed. This function removes any bond to the particle left of it, then rotates itself left, then compares its atom string to 515 and (irrespectively of the result) sets the bond again to the previously unbounded particle. Then the function removes any bond between the particle one square below it and the particle one square down and one square west of it (note that depending on the particle‘s orientation “down” could mean any direction, but “west” always describes the same square). The next command is interpreted as a jump 381 commands backwards and – as there are only 5 commands before – terminates execution.

4.3 Conclusion

The presented simulation experiments in an artificial environment has shown the self-organization phenomena in the form of emerging functions of growing complexity. Particle function activities manifested via yielding to different intermediate states of the system and different states in the equilibrium, so functions affected the environment dynamic at another qualitative level.

The functions that emerged during the simulation have not taken full advantage of the language they are encoded in. Especially, no self-replicating system emerged nor any spectacular spatial structure.

Several problems have been noticed with particle’s language features that hampered the development of more complex functions (discussed in work [169]):

1. Conditional commands. The conditional commands are realized via conditional jumps. The serious problem is that the jump destination must be determined in a very precise way. E.g. in the following code:

   ```
   CS 0, 1110
   JT 3
   MOS 1
   END
   MOS 2
   ```

   depending on the result of `CS 0, 1110` the `MOS 1` or `MOS 2` will be executed. Please note that any change in the argument of `JF` command (from -256 to 255)

[1]atom string: 10010 *xxxxx*, where *x* denotes 0 or 1, coding setting bond operation

lead to error, or more likely will terminate the program execution (as the result of a jump to a non existing command). In the experiment, the emerged conditional commands works only as the conditional program terminating.

2. Program loop. Based on environment concepts, any complex behaviour needs some forms of working program loops. In the environment, loops are realized via jump command (both conditional and unconditional) into the precisely chosen part of the program. Spontaneous emergence of valid program loops is then very improbable because of the same reasons as described earlier (during conditional commands).

3. Structure recognition. This basic functionality is realized via a program loop searching through the program activity area Ω. An example of such a loop was presented in the program on p. 41, the working part:

   ```
   SHC
   SCN
   CS 0, 1110
   JF -2
   ```

   Because program loops doesn't exist and the comparative function works improperly, the searching through the activity area wasn't observed. It has been reduced to simply checking the current square and to commands that terminates programs if some condition were (or were not) fulfilled.

4. Marking the places. Thanks to `PS`, it is possible to remember up to 1024 different squares. Any further reference to such square needs to use exactly the same argument as given for `PS`. E.g. the following sequence `PS 535; MOS 536` works well, but the `PS 535; MOS 536` does not. In randomly generated programs, the same argument values for different commands almost never occurred.

5. Command encoding. The encoding algorithm is very vulnerable to any changes in the encoding string (the sequence of atoms). Because every command may be encoded by a different number of data (a different number of atoms), every small change may lead to unpredictable changes in the program's behaviour (change of one command type may change the interpretation of the whole encoding string)

Functional activities, by recognising particular particles and complexes in nearby squares and selectively creating or removing bonds, concatenating, splitting, changing atoms or just moving particles, are able to affect the environment in a very sophisticated manner (e.g. it is possible to simulate a system in which the self-replication ability emerges from these low-level rules). This level of organization tightly depends on the possibility of effectively finding of particular particles or complexes. The above described problems with the particle's language have shown that the probability of spontaneously creating a particle that manifests this kind of behaviour using an existing language is very low.

Bibliography

[1] A. Adamatzky and M. Komosinski. *Artificial Life Models in Hardware*. Springer, 2009.

[2] C. Adami and C.T. Brown. Evolutionary learning in the 2D artificial life system\ avida. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 377–381, 1994.

[3] Christoph Adami. *Introduction to artificial life*. Springer, 1998.

[4] BJ Alder and TE Wainwright. Phase transition for a hard sphere system. *The Journal of Chemical Physics*, 27:1208, 1957.

[5] R. Allan. Survey of Agent Based Modelling and Simulation Tools. Technical report, Computational Science and Engineering Department, STFC Daresbury Laboratory, Daresbury, Warrington WA4 4AD, 2009. `http://epubs.cclrc.ac.uk/bitstream/3637/ABMS.pdf`.

[6] MP Allen and DJ Tildesley. *Computer simulation in chemical physics*. Springer, 1993.

[7] A.H. Auchincloss, D. Roux, and V. Ana. A new tool for epidemiology: the usefulness of dynamic-agent models in understanding place effects on health. *American journal of epidemiology*, 168(1):1, 2008.

[8] R. J. Bagley and J. D. Farmer. Spontaneous emergence of metabolism. In C. G. Langton, C. Taylor, and J. D. Farmer, editors, *Artificial Life II*, pages 93–140. Addison-Wesley, 1992.

[9] H. Balzter, P.W. Braun, and W. Köhler. Cellular automata models for vegetation dynamics. *Ecological Modelling*, 107(2-3):113–125, 1998.

[10] W. Banzhaf, P. Dittrich, and H. Rauhe. Emergent computation by catalyctic reactions. *Nanotechnology*, 7:307–314, 1996.

[11] M.A. Bedau. Artificial life: organization, adaptation and complexity from the bottom up. *Trends in cognitive sciences*, 7(11):505–512, 2003.

[12] 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.

[13] HJC Berendsen. Biophysical applications of molecular dynamics. *Computer Physics Communications*, 44(3):233–242, 1987.

[14] K. Boryczko, W. Dzwinel, and D. A. Yuen. Dynamical clustering of red blood cells in capillary vessels. *Journal of molecular modeling*, 9(1):16–33, 2003.

[15] J. J. Brewster and M. Conrad. Evolve iv: A metabolically-based artificial ecosystem model. *Evolutionary programming*, pages 473–482, 1998.

[16] W.R. Buckley and A. Mukherjee. Constructibility of signal-crossing solutions in von Neumann 29-state cellular automata. *Computational Science–ICCS 2005*, pages 395–403, 2005.

[17] Andrzej Buller. *Sztuczny mózg. To już nie fantazje.* Prószyński i S-ka, 1998.

[18] J. Byl. Self-reproduction in small cellular automata. *Physica D: Nonlinear Phenomena*, 34(1-2):295–299, 1989.

[19] G. Caruso, D. Peeters, J. Cavailhès, and M. Rounsevell. Spatial configurations in a periurban city. A cellular automata-based microeconomic model. *Regional Science and Urban Economics*, 37(5):542–567, 2007.

[20] S. Chen, Z. Liu, C. Zhang, Z. He, Z. Tian, B. Shi, and C. Zheng. A novel coupled lattice Boltzmann model for low Mach number combustion simulation. *Applied Mathematics and Computation*, 193(1):266–284, 2007.

[21] A.J. Chorin and P.S. Bernard. Discretization of a vortex sheet, with an example of roll-up. *Journal of Computational Physics*, 13(3):423–429, 1973.

[22] H.H. Chou and J.A. Reggia. Emergence of self-replicating structures in a cellular automata space. *Physica D: Nonlinear Phenomena*, 110(3-4):252–276, 1997.

[23] H.H. Chou and J.A. Reggia. Problem solving during artificial selection of self-replicating loops* 1. *Physica D: Nonlinear Phenomena*, 115(3-4):293–312, 1998.

[24] S.S. Chow, C.O. Wilke, C. Ofria, R.E. Lenski, and C. Adami. Adaptive radiation from resource competition in digital organisms. *Science*, 305(5680):84, 2004.

[25] D. Chu and W.K. Ho. A category theoretical argument against the possibility of artificial life: Robert Rosen's Central Proof revisited. *Artificial Life*, 12(1):117–134, 2006.

[26] D. Chu and W.K. Ho. Computational realizations of living systems. *Artificial Life*, 13(4):369–381, 2007.

[27] G. Ciccotti, D. Frenkel, and IR McDonald. *Simulation of liquids and solids.* Elsevier Science Pub Co, 1987.

[28] G. Ciccotti, W.G. Hoover, Fisica Società Italiana di, Institute NATO Advenced Study, Fermi" International School of Physics, and Varenna. *Molecular-dynamics simulation of statistical-mechanical systems.* North-Holland Amsterdam, 1986.

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

[30] EF Codd. *Cellular automata.* Academic Press, Inc. Orlando, FL, USA, 1968.

[31] N. Collier. Repast: An extensible framework for agent simulation. In *Swarmfest 2000: proceedings of the 4th annual Swarm User Group Meeting, March 11-13, 2000, Utah State University, Logan, Utah*, page 17. Quinney Natural Resources Research Library, College of Natural Resources, Utah State University, 2001.

[32] M. Cook. Universality in elementary cellular automata. *Complex Systems*, 15, 2004.

[33] Christopher J. Cramer. *Essentials of Computational Chemistry.* John Wiley And Sons, 2002.

[34] PA Cundall. Distinct element models of rock and soil structure. *Analytical and computational methods in engineering rock mechanics*, pages 129–163, 1987.

[35] B. Damer, K. Marcelo, F. Revi, T. Furmanski, and C. Laurel. Nerve garden: germinating biological metaphors in net-based virtual worlds. *Artificial Life Models in Software*, pages 67–80, 2009.

[36] R. Davidge. Processors as organisms. Technical Report CSRP 250, School of Cognitive and Computing Sciences, University of Sussex, 1992.

[37] R. Dawkins. *The blind watchmaker.* Penguin Harmondsworth, 1991.

[38] F. De Dinechin. Self-replication in a 2D von Neumann architecture. In *Fourth European Conference on Artificial Life*, pages 560–565. Citeseer, 1997.

[39] AK Dewdney. In the game called Core War hostile programs engage in a battle of bits. *Scientific American*, 250(5):15–19, 1984.

[40] Peter Dittrich, Jans Ziegler, and Wolfgang Banzhaf. Artificial chemistries – a review. *Artificial Life*, 7:225–275, 2001.

[41] B. Dünweg and A. Ladd. Lattice Boltzmann simulations of soft matter systems. *Advanced Computer Simulation Approaches for Soft Matter Sciences III*, pages 89–166, 2009.

[42] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[43] A. Dorin and K.B. Korb. Building virtual ecosystems from artificial chemistry. *Lecture Notes in Computer Science*, 4648:103, 2007.

[44] K.E. Drexler. *Nanosystems- Molecular machinery, manufacturing, and computation.* New York, NY: John Wiley & Sons, Inc, 1992., 1992.

[45] W. Dzwinel, K. Boryczko, and D.A. Yuen. A discrete-particle model of blood dynamics in capillary vessels. *Journal of colloid and interface science*, 258(1):163–173, 2003.

[46] Witold Dzwinel. *Informatyczna problemy i perspektywy symulacji metodą cząstek.* Wydawnictwa AGH, Kraków, Poland, 1996.

[47] M. Eaton and JJ Collins. Artificial life and embodied robotics: current issues and future challenges. *Artificial Life and Robotics*, 13(2):406–409, 2009.

[48] M. Eigen and R. Winkler. *Laws of the Game.* Princeton University Press Princeton, NJ, 1993.

[49] H. Fellermann, S. Rasmussen, H.J. Ziock, and R.V. Solé. Life Cycle of a Minimal Protocell—A Dissipative Particle Dynamics Study. *Artificial Life*, 13(4):319–345, 2007.

[50] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman Lectures on Physics.* Addison-Wesley, Boston, USA, 2nd edition, 2006.

[51] O. Filippova, S. Succi, F. Mazzocco, C. Arrighetti, G. Bella, and D. Hanel. Multiscale lattice Boltzmann schemes with turbulence modeling. *Journal of Computational Physics*, 170(2):812–829, 2001.

[52] W. Fontana. Algorithimc chemistry. In C. G. Langton, C. Taylor, and J. D. Farmer, editors, *Artificial Life II*, pages 159–210. Addison-Wesley, 1992.

[53] P.L. Freddolino, A.S. Arkhipov, S.B. Larson, A. McPherson, and K. Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14(3):437–449, 2006.

[54] RA Freitas. A self-replicating, growing lunar factory. *Space manufacturing 4*, pages 109–119, 1981.

[55] R.A. Freitas Jr and R.C. Merkle. *Kinematic self-replicating machines.* Landes Bioscience/Eurekah. com, Georgetown, Tex., 2004.

[56] T. Gánti. *The principles of life.* Oxford University Press, USA, 2003.

[57] Martin Gardner. Mathematical games. *Sci. Amer.*, 224:February, 112; March, 106; April, 114, 1971.

[58] Martin Gardner. Mathematical games. *Sci. Amer.*, 224, 1972.

[59] J. R. Gaylord and K. Nishidate. *Modeling Nature: Cellular Automata Simulation with Mathematica.* Springer, 1996.

[60] R.J. Gaylord and P.R. Wellin. *Computer simulations with Mathematica: explorations in complex physical and biological systems.* Springer-Verlag New York, Inc., New York, NY, 1995.

[61] R.A. Gentry, R.E. Martin, and B.J. Daly. An Eulerian differencing method for unsteady compressible flow problems. *Journal of Computational Physics*, 1(1):87–118, 1966.

[62] RA Gingold and JJ Monaghan. Smoothed particle hydrodynamics- Theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, 1977.

[63] S. Griffith, D. Goldwater, and J.M. Jacobson. Self-replication from random parts. *Nature*, 437(7059):636, 2005.

[64] G. Grinstein, C. Jayaprakash, and Y. He. Statistical mechanics of probabilistic cellular automata. *Physical review letters*, 55(23):2527–2530, 1985.

[65] L. Gulyás, T. Kozsik, and S. Fazekas. The multi-agent modeling language. In *Proceedings of Proceedings of the 4th International Conference on Applied Informatics (ICAI)*, pages 43–50, 1999.

[66] F.H. Harlow. The particle-in-cell computing method for fluid dynamics. *Methods Comput. Phys*, 3:319–343, 1964.

[67] C. He, N. Okada, Q. Zhang, P. Shi, and J. Zhang. Modeling urban expansion scenarios by coupling cellular automata model and system dynamic model in Beijing, China. *Applied Geography*, 26(3-4):323–345, 2006.

[68] D. Helbing, F. Schweitzer, J. Keltsch, and P. Molnar. Active walker model for the formation of human and animal trail systems. *Physical Review E*, 56(3):2527–2539, 1997.

[69] T. Higuchi, Y. Liu, and X. Yao. *Evolvable hardware.* Springer, 2006.

[70] J.H. Holland. *Adaptation in natural and artificial systems.* MIT press Cambridge, MA, 1975/1992.

[71] J.H. Holland. Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars. *Automata, languages, development*, pages 385–404, 1976.

[72] PJ Hoogerbrugge and J. Koelman. Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. *EPL (Europhysics Letters)*, 19:155, 1992.

[73] John E. Hopcroft and Jeffrey D. Ulman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[74] J. Horgan. From complexity to perplexity. *Scientific American*, 272(6):104–109, 1995.

[75] J. Hultman and A. Pharasyn. Hierarchical, dissipative formation of elliptical galaxies: is thermal instability the key mechanism? Hydrodynamical simulations including supernova feedback, multi-phase gas and metal enrichment in CDM: structure and dynamics of elliptical galaxies. *Astronomy and Astrophysics-Berlin*, 347:769–798, 1999.

[76] P. Husbands. Evolving robot behaviours with diffusing gas networks. In *Evolutionary Robotics (Proc. EvoRob'98)*, volume 1468 of LNCS. Springer-Berlin, 1998.

[77] Tim J. Hutton. Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8(4):341–356, 2002.

[78] Tim J. Hutton. Evolvable Self-Reproducing Cells in a Two-Dimensional Artificial Chemistry. *Artificial Life*, 13(1):11–30, 2007.

[79] T.J. Hutton. Codd's self-replicating computer. *Artificial Life*, 16(2):99–117, 2010.

[80] J. Ibáñez, D. Anabitarte, I. Azpeitia, O. Barrera, A. Barrutieta, H. Blanco, and F. Echarte. Self-inspection based reproduction in cellular automata. *Advances in artificial life*, pages 564–576, 1995.

[81] K. Imai, T. Hori, and K. Morita. Self-reproduction in three-dimensional reversible cellular space. *Artificial Life*, 8(2):155–174, 2002.

[82] M.E. Inchiosa and M.T. Parker. Overcoming design and development challenges in agent-based modeling using ASCAPE. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3):7304, 2002.

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

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

[85] Wojciech Jędruch and Mariusz Barski. Experiments with a universe for molecular modelling of biological processes. *Biosystems*, 24(2):99–117, 1990.

[86] Wojciech Jędruch and Jeffrey R. Sampson. A universe for molecular modeling of self-replication. *Biosystems*, 20(4):329–340, 1987.

[87] 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.

[88] 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.

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

[90] J. Johnston. *The allure of machinic life: cybernetics, artificial life, and the new AI*. The MIT Press, 2008.

[91] George Kampis and Istvan Karsai. Breaking waves in population flows. In *Advances in Artificial Life, 10th European Conference on Artificial Life, ECAL 2009, Budapest, Hungary, September 13-16, 2009*, Lecture Notes in Computer Science, 2009. (in press).

[92] Y. Kanada. Combinatorial problem solving using randomized dynamic tunneling on a production system. In *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent systems for the 21st Century*, volume 4. New York: IEEE, 1995.

[93] J.G. Kemeny. Man viewed as a machine. *Scientific American*, 192(4):58–67, 1955.

[94] L.B. Kier, C.K. Cheng, and J.D. Nelson. Models of Solute Aggregation Using Cellular Automata. *Chemistry & Biodiversity*, 6(3), 2009.

[95] K.J. Kim and S.B. Cho. A comprehensive overview of the applications of artificial life. *Artificial Life*, 12(1):153–182, 2006.

[96] 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.

[97] M. Komosinski. Framsticks: A Platform for Modeling, Simulating, and Evolving 3D Creatures. In *Artificial life models in software*, pages 37–66. Springer, 2005.

[98] S. Koshizuka, A. Nobe, and Y. Oka. Numerical analysis of breaking waves using the moving particle semi-implicit method. *International Journal for Numerical Methods in Fluids*, 26(7):751–769, 1998.

[99] Z. Kowalczuk and M. Czubenko. Interactive Cognitive-Behavioral Decision Making System. In L. Rutkowski, R. Scherer, R. Tadeusiewicz, L.A. Zadeh, and J.M. Zurada, editors, *Artificial Intelligence and Soft Computing, Part II: 10th International Conference, ICAISC 2010, Zakopane, Poland, June 13-17, 2010, Part II Proceedings*, volume 6114 of *Lecture Notes in Artificial Inteligence*, pages 517–523. Springer, 2010.

[100] 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.

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

[102] C.G. Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):135–144, 1984.

[103] Christopher G. Langton. *Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[104] J. Lee, S. Adachi, and F. Peper. Reliable self-replicating machines in asynchronous cellular automata. *Artificial Life*, 13(4):397–413, 2007.

[105] R.E. Lenski, C. Ofria, T.C. Collier, and C. Adami. Genome complexity, robustness and genetic interactions in digital organisms. *Nature*, 400(6745):661–664, 1999.

[106] R.E. Lenski, C. Ofria, R.T. Pennock, and C. Adami. The evolutionary origin of complex features. *Nature*, 423(6936):139–144, 2003.

[107] P. Lerena and M. Courant. Bio-machines. In *Proceedings of Artificial Life V*, 1996.

[108] S. Li and W.K. Liu. Meshfree and particle methods and their applications. *Applied Mechanics Reviews*, 55:1–34, 2002.

[109] G.R. Liu and MB Liu. *Smoothed particle hydrodynamics: a meshfree particle method.* World Scientific Pub Co Inc, 2003.

[110] W.K. Liu, S. Jun, S. Li, J. Adee, and T. Belytschko. Reproducing kernel particle methods for structural dynamics. *International Journal for Numerical Methods in Engineering*, 38(10):1655–1680, 1995.

[111] J.D. Lohn and J.A. Reggia. Discovery of self-replicating structures using a genetic algorithm. In *1995 IEEE International Conference on Evolutionary Computing*, pages 678–683. Citeseer, 1995.

[112] JD Lohn and JA Reggia. Automatic discovery of self-replicating structures in cellular automata. *IEEE Transactions on Evolutionary Computation*, 1(3):165–178, 1997.

[113] AH Louie. A living system must have noncomputable models. *Artificial life*, 13(3):293–297, 2007.

[114] L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.

[115] M.A. Ludwig. *Computer viruses, artificial life, and evolution.* Amer Eagle Pubns Inc, 1993.

[116] P. Mandik. Synthetic neuroethology. *Cyberphilosophy: The intersection of philosophy and computing*, pages 8–25, 2002.

[117] J. Martel and J.D.E. Young. Purported nanobacteria in human blood as calcium carbonate nanoparticles. *Proceedings of the National Academy of Sciences*, 105(14):5549, 2008.

[118] H.M. Martinez. An automaton analogue of unicellularity. *BioSystems*, 11(2-3):133–162, 1979.

[119] B. Mayer and S. Rasmussen. The lattice molecular automaton (lma): A simulation system for constructive molecular dynamics. *International journal of modern physics C*, 9:157–177, 1998.

[120] B. Mayer and S. Rasmussen. Dynamics and simulation of micellar self-reproduction. *Int. J. Mod. Phys*, 11(4):809–826, 2000.

[121] J. Maynard Smith. *The problems of biology.* Oxford: Oxford Univ. Press, 1989.

[122] J. McCormack. Open problems in evolutionary music and art. *Lecture Notes in Computer Science*, 3449:428–436, 2005.

[123] D.S. McKay, E.K. Gibson Jr, K.L. Thomas-Keprta, H. Vali, C.S. Romanek, S.J. Clemett, X.D.F. Chillier, C.R. Maechling, and R.N. Zare. Search for past life on Mars: Possible relic biogenic activity in Martian meteorite ALH84001. *Science*, 273(5277):924, 1996.

[124] Barry McMullin. *Artificial Knowledge: An Evolutionary Approach.* PhD thesis, Department of Computer Science, University College Dublin, 1992.

[125] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: a toolkit for building multi-agent systems. *Santa Fe NM: Santa Fe Institute Working Paper*, pages 96–06, 1996.

[126] J. Monaghan. Particle methods for hydrodynamics. *Computer Physics Reports*, 3:71–124, 1985.

[127] JJ Monaghan. An introduction to SPH. *Computer Physics Communications*, 48(1):89–96, 1988.

[128] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. *Lecture Notes in Control and Information Sciences*, pages 501–501, 1994.

[129] Edward F. Moore. Machine models of self-reproduction. In *Proceedings of Symposia in Applied Mathematics*, volume 14, pages 17–33, Champaign, IL, USA, 1962. The American Mathematical Society, University of Illinois Press.

[130] K. Morita and K. Imai. A simple self-reproducing cellular automaton with shape-encoding mechanism. In *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, pages 489–496, 1997.

[131] S. Moss, H. Gaylard, S. Wallis, and B. Edmonds. SDML: A multi-agent language for organizational modelling. *Computational & Mathematical Organization Theory*, 4(1):43–69, 1998.

[132] G. Nicolis and I. Prigogine. *Self-organization in nonequilibrium systems: From dissipative structures to order through fluctuations*. Wiley, New York, 1977.

[133] C. Nikolai and G. Madey. Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2, 2009.

[134] C. Ofria and C. Wilke. Avida: Evolution experiments with self-replicating computer programs. *Artificial Life Models in Software*, pages 3–35, 2005.

[135] ET Olson. The ontological basis of strong artificial life. *Artificial Life*, 3(1):29, 1997.

[136] N. Ono and T. Ikegami. Model of self-replicating cell capable of self-maintenance. *Advances in artificial life*, pages 399–406, 1999.

[137] N. Ono and T. Ikegami. Self-maintenance and self-reproduction in an abstract cell model. *Journal of Theoretical Biology*, 206(2):243–253, 2000.

[138] Z. Pan and J. Reggia. Evolutionary Discovery of Arbitrary Self-replicating Structures. *Computational Science–ICCS 2005*, pages 404–411, 2005.

[139] Z. Pan and J.A. Reggia. Computational discovery of instructionless self-replicating structures in cellular automata. *Artificial Life*, 16(1):39–63, 2010.

[140] AN Pargellis. The spontaneous generation of digital Life. *Physica D: Nonlinear Phenomena*, 91(1-2):86–96, 1996.

[141] M.T. Parker. What is Ascape and why should you care. *Journal of Artificial Societies and Social Simulation*, 4(1):5, 2001.

[142] HH Pattee. Artificial life needs a real epistemology. In *Advances in artificial life: Third European Conference on Artificial Life, Granada, Spain, June 4-6, 1995: proceedings*, page 23. Springer Verlag, 1995.

[143] Roger Penrose. *The Emperor's New Mind.* Oxford University Press, 1989.

[144] Roger Penrose. *Shadows of the Mind. A search for the Missing Science of Consciousness.* Oxford University Press, 1994.

[145] J.Y. Perrier, M. Sipper, and J. Zahnd. Toward a viable, self-reproducing universal computer* 1. *Physica D: Nonlinear Phenomena*, 97(4):335–352, 1996.

[146] Umberto Pesavento. An implementation of von neumann's self-reproducing machine. *Artif. Life*, 2(4):337–354, 1995.

[147] E. Petraglio, G. Tempesti, and J.M. Henry. Arithmetic operations with self-replicating loops. In *Collision-based computing*, page 490. Springer-Verlag, 2001.

[148] A. Rahman. Correlations in the motion of atoms in liquid argon. *Phys. Rev*, 136(2A):405–411, 1964.

[149] S.F. Railsback, S.L. Lytinen, and S.K. Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609, 2006.

[150] DC Rapaport. *The art of molecular dynamics simulation.* Cambridge Univ Pr, 2004.

[151] S. Rasmussen, L. Chen, B.M.R. Stadler, and P.F. Stadler. Proto-organism kinetics: Evolutionary dynamics of lipid aggregates with genes and metabolism. *Origins of Life and Evolution of Biospheres*, 34(1):171–180, 2004.

[152] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena*, 42(1-3):111–134, 1990.

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

[154] T. S. Ray. An approach to the synthesis of life. *Artificial Life II*, pages 371–408, 1991.

[155] J.A. Reggia, S.L. Armentrout, H.H. Chou, and Y. Peng. Simple systems that exhibit self-directed replication. *Science*, 259(5099):1282, 1993.

[156] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34. ACM, 1987.

[157] S. Richter and R.F. Werner. Ergodicity of quantum cellular automata. *Journal of Statistical Physics*, 82(3):963–998, 1996.

[158] Tomas Rokicki, Andrew Trevorrow, Dave Greene, Jason Summers, and Tim Hutton. Golly game of life home page. `http://golly.sourceforge.net/`.

[159] M. Rönkkö. An artificial ecosystem: Emergent dynamics and lifelike properties. *Artificial Life*, 13(2):159–187, 2007.

[160] R. Rosen. *Life itself: A comprehensive inquiry into the nature, origin, and fabrication of life*. Columbia Univ Pr, 1991.

[161] T. Roska. *Cellular neural networks*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[162] Rudy Rucker. Review: A new kind of science. *American Mathematical Monthly*, November:851–861, 2003.

[163] H. Sayama. Introduction of structural dissolution into Langton's self-reproducing loop. In *Proceedings of the sixth international conference on Artificial life*, pages 114–122. MIT Press, 1998.

[164] H. Sayama. A new structurally dissolvable self-reproducing loop evolving in a simple cellular automata space. *Artificial Life*, 5(4):343–365, 1999.

[165] H. Sayama. Self-replicating worms that increase structural complexity through gene transmission. In *Artificial life VII: Proceedings of the Seventh International Conference on Artificial Life*, pages 21–30, 2000.

[166] H. Sayama. Self-protection and diversity in self-replicating cellular automata. *Artificial Life*, 10(1):83–98, 2004.

[167] E. Schrödinger and Lewin. *What is life?* Cambridge University Press Cambridge, 1968.

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

[169] 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*, Łódź, Poland, 2007. (in Polish, available on-line at: `http://www.fccs.wshe.lodz.pl/fccs2007/artykuly/sienkiewicz.pdf`).

[170] 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.

[171] Rafał Sienkiewicz and Wojciech Jędruch. Self-organization in artificial environment. In M. Kłopotek and J. Tchórzewski, editors, *Proceedings of Artificial Intelligence Studies*, volume 3 (26), Siedlce, Poland, 2004. Institute of Computer Science University of Podlasie, Publishing House of Univesity of Podlasie.

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

[173] Rafał Sienkiewicz and Wojciech Jędruch. Artificial environment for simulation of emergent behaviour. In B. Bieliczynski et al, editor, *Adaptive and Natural Computing Algorithms: 8th International Conference, Icannga 2007, Warsaw, Poland, April 11-14, 2007, Proceedings, Part I*, volume 4431/2007 of *LNCS*, pages 386–393. Springer, 2007.

[174] 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*, Lecture Notes in Computer Science, 2009. (in press).

[175] K. Sims. Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

[176] J. Skipper. The computer zoo-evolution in a box. *Computational intelligence in software engineering*, page 151, 1998.

[177] E. Sklar. NetLogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311, 2007.

[178] A. Smith, P. Turney, and R. Ewaschuk. Self-replicating machines in continuous space with virtual physics. *Artificial Life*, 9(1):21–40, 2003.

[179] B.S. Soares-Filho, G. Coutinho Cerqueira, and C. Lopes Pennachin. DINAMICA—a stochastic cellular automata model designed to simulate the landscape dynamics in an Amazonian colonization frontier. *Ecological Modelling*, 154(3):217–235, 2002.

[180] Eldra Solomon, Linda Berg, and Diana W. Martin. *Biology*. Brooks Cole, 7 edition, 2004.

[181] W.M. Stevens. Simulating self-replicating machines. *Journal of Intelligent and Robotic Systems*, 49(2):135–150, 2007.

[182] W.M. Stevens. Parts closure in a kinematic self-replicating programmable constructor. *Artificial Life and Robotics*, 13(2):508–511, 2009.

[183] B. Straatman, R. White, and W. Banzhaf. An artificial chemistry-based model of economies. *Artificial Life*, 11:592, 2008.

[184] T. Szuba. Evaluation measures for the collective intelligence of closed social structures. In *International Conference on Intelligent System and Semiotics (ISAS'97)*, pages 401–406. Gaithersburg, MD, 1998.

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

[186] T.J. Taylor. *From artificial evolution to artificial life*. PhD thesis, The University of Edinburgh, 1999.

[187] G. Tempesti. A new self-reproducing cellular automaton capable of construction and computation. *Advances in artificial life*, pages 555–563, 1995.

[188] G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired By Biological Processes.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1998.

[189] L. Tesfatsion. Agent-based computational economics: A constructive approach to economic theory. *Handbook of computational economics*, pages 831–880, 2006.

[190] R. Tobias and C. Hofmann. Evaluation of free Java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1), 2004.

[191] T. Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D: Nonlinear Phenomena*, 10(1-2):117–127, 1984.

[192] G. Trajkovski. *An Imitation-Based Approach to Modeling Homogenous Agent Societes.* Idea Group Publishing, 2006.

[193] J. Ventrella. GenePool: Exploring the Interaction Between Natural Selection and Sexual Selection. *Artificial Life Models in Software*, pages 81–96, 2009.

[194] Gérard Y. Vichniac. Simulating physics with cellular automata. *Physica D: Nonlinear Phenomena*, 10(Issue 1-2):96–116, 1984.

[195] P. Vitányi. Sexually reproducing cellular automata. *Mathematical Biosciences*, 18(1-2):23–54, 1973.

[196] V.A. Voelz, G.R. Bowman, K. Beauchamp, and V.S. Pande. Molecular Simulation of ab Initio Protein Folding for a Millisecond Folder NTL9 (1- 39). *Journal of the American Chemical Society*, pages 342–346, 2010.

[197] John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[198] N. Wiener. *Cybernetics: or, Control and Communication in the Animal and the Machine.* The MIT Press, 1965.

[199] C.O. Wilke, J.L. Wang, C. Ofria, R.E. Lenski, and C. Adami. Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, 412(6844):331–333, 2001.

[200] D.A. Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction.* Springer Verlag, 2000.

[201] Stefen Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55:601–644, 1983.

[202] Stefen Wolfram. Universality and complexity of cellular automata. *Physica D*, 10:1–35, 1984.

[203] Stephen Wolfram. *A New Kind of Science.* Wolfram Media, 2002.

[204] O. Wolkenhauer. Interpreting Rosen. *Artificial life*, 13(3):291–292, 2007.

[205] J. Ziegler and W. Banzhaf. Evolving control metabolisms for a robot. *Artificial Life*, 7(2):171–190, 2001.