Fraglets: Chemical Programming with a Packet Prefix Language

Thomas Meyer and Christian Tschudin
Computer Science Department, University of Basel, Switzerland

15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS’09), Maria Taferl, Oct 13th, 2009
Overview

1. **Functionality** of Fraglets
   - Intro game: Random draws, yet predictable outcome
   - Prefix programs
   - Introductory examples:
     rewriting, FSM, “shuttle service”, active networking
   - A duplicating Quine

2. **Dynamics** of Fraglets
   - See follow up talk by Thomas Meyer
A “Mate-And-Spread” Game

Given: vector of booleans, not uniform

0010011001100

Do rounds, repeat as long as you wish:

Pick two random positions; If content differs then copy randomly

Question: How will the array look, on average, after some rounds?
A “Mate-And-Spread” Game

Given: vector of booleans, not uniform

Do rounds, repeat as long as you wish:

Pick two random positions; If content differs then copy randomly

```c
#define pick(a,b) a=rand()%(len-1);b=rand()%len;if(a==b)a++
main() {char v[]="00100000"; int a, b, len=strlen(v), n=100;
    for (srand(times(0)); n>0; n--) {
        pick(a,b); if (v[a]!=v[b]) {pick(a,b); v[a]=’0’; v[b]=’1’;}
        printf("%s\n", v);
    }
}
```

Question: How will the array look, on average, after some rounds?
A few “Mate-And-Spread” Games, Results of 100 Rounds

<table>
<thead>
<tr>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
<th>00100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100000</td>
<td>10100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00100000</td>
<td>10100000</td>
<td>00100000</td>
<td>00000100</td>
<td>00100000</td>
<td>00000100</td>
<td>00100000</td>
<td>00000100</td>
</tr>
<tr>
<td>00100000</td>
<td>10100000</td>
<td>00100000</td>
<td>00001000</td>
<td>00100000</td>
<td>00001000</td>
<td>00100000</td>
<td>00001000</td>
</tr>
<tr>
<td>00100000</td>
<td>10100000</td>
<td>00100000</td>
<td>00000100</td>
<td>00100000</td>
<td>00000100</td>
<td>00100000</td>
<td>00000100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10010101</td>
<td>11001010</td>
<td>10101001</td>
<td>00101011</td>
<td>10101001</td>
<td>00101011</td>
<td>10101001</td>
<td>00101011</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>00100000</td>
<td>...</td>
<td>11011110</td>
</tr>
<tr>
<td>00100000</td>
<td>00100010</td>
<td>01000000</td>
<td>00100000</td>
<td>01000000</td>
<td>00000100</td>
<td>11011110</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>00100010</td>
<td>01000000</td>
<td>00100000</td>
<td>01000000</td>
<td>00000100</td>
<td>11011110</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>00100010</td>
<td>10000000</td>
<td>00100001</td>
<td>10000000</td>
<td>00000100</td>
<td>11011100</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>01100010</td>
<td>10000000</td>
<td>00100001</td>
<td>10000000</td>
<td>00000100</td>
<td>11011100</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10111111</td>
<td>01010001</td>
<td>01101101</td>
<td>11010001</td>
<td>01101101</td>
<td>01101111</td>
<td>10011001</td>
<td></td>
</tr>
</tbody>
</table>

Asymptotically, all results will have an equal number of zeros and ones!
"Mate-and-Spread": Degrees of Concurrency

- Single thread: sequential program with a master loop
- Two threads (e.g., "mate" and "spread"), two loops
- What about $N$ threads, and no loops?

![Diagram showing degrees of concurrency](image-url)
The term *Fraglets* refers to Mobile Computation Fragments. This is illustrated using a chemical metaphor: Molecules reacting with each other.

### Fraglet Exchange

- Exchange of *fraglets* (=molecules, =packets).
- **Fraglet** = *word of symbols*  
  `[ tag1 tag2 tag3 ... ]`
- **Fraglet Multiset** (=reactor, =node)

### Processing Rules

- Reaction (two fraglets merge)
- Transformation (single fraglet rewriting)
- *Processing order (among molecules) is non-deterministic*
Prefix Instructions: a Reaction and Transformations

<table>
<thead>
<tr>
<th>Op</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>[ match a TAIL1 ], [ a TAIL2 ]</td>
<td>[ TAIL1 TAIL2 ]</td>
</tr>
<tr>
<td>nul</td>
<td>[ nul TAIL ]</td>
<td>– (fraglet is removed)</td>
</tr>
<tr>
<td>nop</td>
<td>[ nop a TAIL ]</td>
<td>[ a TAIL ]</td>
</tr>
<tr>
<td>fork</td>
<td>[ fork a b TAIL ]</td>
<td>[ a TAIL ], [ b TAIL ]</td>
</tr>
<tr>
<td>split</td>
<td>[ split PART1 * TAIL ]</td>
<td>[ PART1 ], [ TAIL ]</td>
</tr>
<tr>
<td>send</td>
<td>$A[\text{send b TAIL}]$</td>
<td>$B[ TAIL ]$ (unreliably)</td>
</tr>
</tbody>
</table>

Language has similarities with (is equivalent to) Post Rewriting Systems
Prefix Programs (or Packets)

- Processing rules have a deliberate constraint: **the first tag fully determines action, no loops**

- Origin in networking: packet processing = (distributed) header rewriting

- A fraglet’s header symbols serve as:
  - instruction
  - related parameters
  - storage
  (the other storage is: other molecules in reactor)
Fraglets Programs are *Distributed Reaction Networks*

Graphical representation of a flow-control protocol
Example 1: Fraglet (header) rewriting

\[
\begin{align*}
\text{In:} & \quad [i \ W] \\
\text{Out:} & \quad [o \ W] \\
\text{Program:} & \quad [\text{match } i \ o]
\end{align*}
\]

Execution Trace:

\[
[\text{match } i \ o] \left\{ [i \ W] \right\} \Rightarrow [o \ W]
\]
Example 1b: The Temporary \texttt{matchp} Hack

Problem: a \texttt{match} molecule (the program) is consumed.
Quick hack: Introduce a “persistent” match, as a catalyst.

\begin{align*}
\text{In:} & \quad [i \ W] \\
\text{Out:} & \quad [o \ W] \\
\text{Program:} & \quad [\text{matchp} \ i \ o]
\end{align*}

Execution Trace:

\[
[\text{matchp} \ i \ o] \\
[\ i \ W]
\Rightarrow [o \ W], [\text{matchp} \ i \ o]
\]
Example 2: (non-deterministic) Finite State Machine

Transitions:  
[ matchp a b ]
[ matchp b c ]
[ matchp c d ]
[ matchp c e ]
[ matchp d a ]
[ matchp e b ]

State token:  [ a ] (initial state)

Possible execution trace (visited states):
[a] → [b] → [c] → [e] → [b] → [c] → [d] → [a] ...
Example 3: A “Shuttle Service”

Service req: \[ \text{shuttle deliver Payload} \]
Shuttle srv: \[ \text{matchp shuttle send B} \]

Execution Trace:
\[ A[ \text{shuttle deliver Payload} ] \Rightarrow A[ \text{send B deliver Payload} ] \Rightarrow B[ \text{deliver Payload} ] \]
Example 4: Active Networking (and Source Routing)

Networking style called “Active Networking”:
– packets contain processing logic
– advantage: minimal preinstalled protocol software requirements

Source routing – the source node prescribes the path
– via H1, H2, ... to some Dest
– Simple active networking (inband) version:

   [ send H1 send H2 send H3 ... send DEST Payload ]
Example 5: Quines

- Classics of informatics:
  - Write a program that outputs its own source code.
  - Named after Willard van Orman Quine (1908–2000)
  - “Quines” exist for every Turing complete language

- How to write a Quine for a prefix language?

Basic Quine structure:
- needs a “blueprint”
- active part (is using the blueprint, and is its activated form)

Plus here: Our Quine duplicates (needed in second part of the talk)
A Duplicating Quine for Fraglets (1)

\[
\text{[match } x \text{ fork fork fork match nop } x] \quad \text{[x fork fork fork match nop } x]\]

A Duplicating Quine for Fraglets (2)

\[
\text{[match } x \text{ fork fork fork match nop } x\text{]} \quad \text{[x fork fork fork match nop } x\text{]}
\]

\[
\text{[fork fork fork match nop } x \text{ fork fork fork match nop } x\text{]}
\]
A Duplicating Quine for Fraglets (3a)
A Duplicating Quine for Fraglets (3b)

[match x fork fork fork match nop x]  [x fork fork fork match nop x]

[fork fork fork match nop x fork fork fork match nop x]  2

[fork match nop x fork fork fork match nop x]
A Duplicating Quine for Fraglets (4)
A Duplicating Quine for Fraglets (5)
This is an autocatalytic reaction network, no need for matchp
Conclusions

- Metaphor of “chemical computations”
  - not wet-lab: string rewriting
  - inherently parallel and non-deterministic execution
  
  We wish to exploit emergent properties of chemical systems like self-organization (see Turing’s work), or robustness

- Active networking, prefix programs: code = data = molecule

- Quines also work that way: what is data and what is code is mere attribution, and not an intrinsic property.

Next: Dynamics of Fraglets – Remember the mate-and-spread game!