A Sticker-Based Model for DNA Computation


Slides by Reem Mokhtar
Sticker(s) Model

\[ N \quad KM \]

\begin{align*}
\text{memory complexes} &= \text{bit string:} \\
\quad &\text{• Memory strand} \\
\quad &\quad (N \text{ bases}) \\
\quad &\text{• Sticker strands} \\
\quad &\quad (M \text{ bases} \times K \text{ regions}) \\
1/\text{on} &= \text{memory strand region with hybridized sticker strand} \\
0/\text{off} &= \text{memory strand region w/out sticker strand}
\end{align*}

Sticker(s) Model

FIG. 1. A memory strand and associated stickers (together called a memory complex) represent a bit string. The top complex on the left has all three bits off; the bottom complex has two annealed stickers and thus two bits on.

Sticker(s) Model

Graphical/Data

- **Purpose:**
  - Ease of interaction and design
  - Aid in validating designs

- **Representations might include**
  - GUI input
  - Rendered results
  - Back-end data structures
Dot Bracket Notation

tram> RNAfold -T 42 -p1
Input string (upper or lower case); @ to quit
............1............2............3............4............
UUGGAGUACACAACCUGUACACUCUUUC
length = 28
UUGGAGUACACAACCUGUACACUCUUUC
.............
minimum free energy = -3.71
.............
free energy of ensemble = -4.39
frequency of mfe structure in ensemble 0.337231

Dot Bracket Notation

Fig. 1. Interactive example run of RNAfold for a random sequence. When the base pairing probability matrix is calculated by symbols ., ||(), are used for bases that are essentially unpaired, weakly paired, strongly paired without preferred direction, weakly upstream (downstream) paired, and strongly upstream (downstream) paired, respectively. Apart from the console output, a, the two postscript files rna.ps, b, and dot.ps, c, are created. The lower left part of dot.ps shows the minimum energy structure, while the upper right shows the pair probabilities. The area of the squares is proportional to the binding probability.

Extended Dot Bracket Notation

A  B  C

Figure 1: A sample secondary structure of an RNA molecule is shown (A) together with the notation of its sequence (B) and its structure in dot-bracket form (C).

matching parenthesis and dots to denote paired and free bases, respectively.

Advantage: string denoting the secondary structure of a nucleic acid is of the same length as the string denoting the nucleotide sequence with a single character for each nucleotide.

Disadvantage:

the secondary structure is of importance, but the detailed sequence that yields the structure is over large stretches arbitrary. In such a scenario the dot-bracket notation is often cumbersome, it leads to large expressions with information that is partially obscured for human readers, because it would require counting identical characters. While the sequence is typically arbitrary in most positions, as long as the structure of the molecule is preserved, nucleic acids with functional properties, such as catalytic activity, often require specific bases in a few positions. If in a few places the nucleotide sequence (i.e. the primary structure) is given, two strings are required: one to specify the structure in dot-bracket-notation and a second string to represent the type of the immutable nucleotides. Furthermore, for communicating structural features among humans a two-dimensional rendering of the one-dimensional dot-bracket notation is often desirable. It would be convenient if specific features in the sequence could be communicated to the rendering software.
Extended Dot Bracket Notation

A \[ \cdots \cdots \cdots \] B CCGAUAGAGGGCGUGCGGUGCACAAGGUGC
C (((((\ldots(((\ldots)))\ldots))\ldots))))

Figure 1: A sample secondary structure of an RNA molecule is shown (A) together with the notation of its sequence (B) and its structure in dot-bracket form (C).

matching parenthesis and dots to denote paired and free bases, respectively.

Advantage: string denoting the secondary structure of a nucleic acid is of the same length as the string denoting the nucleotide sequence with a single character for each nucleotide.

Disadvantage:

the secondary structure is of importance, but the detailed sequence that yields the structure is over large stretches arbitrary. In such a scenario the dot-bracket notation is often cumbersome, it leads to large expressions with information that is partially obscured for human readers, because it would require counting identical characters. While the sequence is typically arbitrary in most positions, as long as the structure of the molecule is preserved, nucleic acids with functional properties, such as catalytic activity, often require specific bases in a few positions. If in a few places the nucleotide sequence (i.e. the primary structure) is given, two strings are required: one to specify the structure in dot-bracket-notation and a second string to represent the type of the immutable nucleotides. Furthermore, for communicating structural features among humans a two-dimensional rendering of the one-dimensional dot-bracket notation is often desirable. It would be convenient if specific features in the sequence could be communicated to the rendering software.
## Extended Dot Bracket Notation

<table>
<thead>
<tr>
<th>Description</th>
<th>Usage</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grouping of base positions</td>
<td>[ .8 ]@{label A}</td>
<td>Eight unbound bases marked as “label A”.</td>
</tr>
<tr>
<td>Parameter delimiter</td>
<td>{ }</td>
<td>see example above</td>
</tr>
<tr>
<td>Set delimiter</td>
<td>.:{A,C}</td>
<td>A single unbound base that can be either A or C.</td>
</tr>
<tr>
<td>Repetition delimiter</td>
<td>A{10}</td>
<td>Always optional.</td>
</tr>
<tr>
<td>Line width</td>
<td>(((([.5]_1)))</td>
<td>Stem-loop structure with bold loop</td>
</tr>
<tr>
<td>Colour</td>
<td>(3[.2]$1(3.4)3)3</td>
<td>A buldge in red.</td>
</tr>
<tr>
<td>Line decoration</td>
<td>.24~1(3.3)3</td>
<td>Binding site marked as crinkled line.</td>
</tr>
<tr>
<td>Annotation marker</td>
<td>.24~1(3.3)3</td>
<td>See first row.</td>
</tr>
<tr>
<td>Multi-molecule binding</td>
<td>(24+1(3.3)3</td>
<td>Sticky end of 24 bases, will bind to site marked 1 on other molecule.</td>
</tr>
<tr>
<td>Base assignment</td>
<td>)):A</td>
<td>Two binding bases, the second one of which is A; See also set delimiter.</td>
</tr>
<tr>
<td>Base exclusion</td>
<td>(((..^U..)))</td>
<td>Stem loop where the central base in the loop is not a uracil.</td>
</tr>
<tr>
<td>Clevage point</td>
<td>(((...%..)))</td>
<td>Between bases, i.e., not a base position.</td>
</tr>
</tbody>
</table>
Then, symbol. Form concatenated string the 5 characters between sequences which are counted as part of the comparison. A complex of two or more oligonucleotides requires to fold into a conformation with two stem-loops. The dot-par

\( [(0,10,15),(35,2,5)] \).

The division feature starts at base 0, has a 10 bp stem, and encompasses of triples (..).

2. ISO

Let \( s \) be a unique representation of secondary structure features. As structure corresponds to an ordered 5

\( p \) by

\( o \)

13

\( i \)

\( T \)

\( G \)

\( U \)

\( C \)

\( A \)

\( U \)

\( A \)

\( G \)

\( C \)

\( U \)

\( C \)

\( G \)

\( C \)

\( U \)

\( C \)

\( C \)

\( C \)

\( U \)

\( C \)

\( C \)

\( C \)

\( G \)

\( G \)

\( C \)

\( G \)

\( C \)

\( U \)

\( C \)

\( C \)

\( C \)

\( U \)

\( C \)

\( C \)

\( C \)

\( U \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)

\( C \)
A diagram illustrating isoform structure.
Languages/Grammar

- **Purpose:**
  - Represent reaction/interaction sequence/network

- **Representations might include:**
  - Regular Grammars
  - Context-free Grammars
  - Graph Grammar
  - etc
Languages

**Alphabet**: Any finite set of symbols \( (\Sigma) \)

Example:

\[ \Sigma_1 = \{1, 2, 3, 4, 5\} \]

26 letters of English alphabet: \( \Sigma_2 = \{a, b, c, d, \ldots, z\} \)

**String**: A finite-length \((n\geq0)\) sequence of symbols (like words) over an alphabet \(\Sigma\)

Example:

a, ab, abc and bba are strings defined over \(\Sigma_2\)


2) Lecture Notes on Regular Languages and Finite Automata
Languages

\( \Sigma^* \) is the set of all strings over an alphabet \( \Sigma \) of ANY finite length.

Null (empty) string \( \rightarrow \varepsilon \)

Example:

If \( \Sigma = \{a, b\} \) then:

\[ \Sigma^* = \{\varepsilon, a, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\} \]

2) Lecture Notes on Regular Languages and Finite Automata
Languages

**Language**: A language is a set of strings all of which choose their symbols from some one alphabet.

--> words in a dictionary, *not* grammatical rules.

**Problem**: Is this string in this language?

2) Lecture Notes on Regular Languages and Finite Automata
Languages

Finite Automata:
Involves:
1) states
2) transitions among states in response to:
3) inputs

Useful for building: compilers, verification systems (ex: circuits, protocols)

2) Lecture Notes on Regular Languages and Finite Automata
Languages

Example of a finite automaton

States: $q_0, q_1, q_2, q_3$.
Input symbols: $a, b$.
Transitions: as indicated above.
Start state: $q_0$.
Accepting state(s): $q_3$.

Retrieved from Prat, A. M., Lecture Notes on Regular Languages and Finite Automata (pp. 11)
Languages

Regular Expressions:

Structural notation for describing a pattern that can be represented by a finite automata

Some RE symbols: ( ) | *

If A, B are languages:

- \((AB)\) → concatenation of enclosed languages A and B (A U B)
- \(A \mid B\) → A or B
- \(A^*\) → 0 or more of A
Languages

A language $L$ over an alphabet $\Sigma$ is just a set of strings $\Sigma^*$. Thus any subset

$$L \subseteq \Sigma^*$$

determines a language over $\Sigma$.

A language determined by a reg. exp. $r$ over $\Sigma$ is

$$L(r) = \{ u \in \Sigma^* | u \text{ matches } r \}$$

Example:

$L((01)^*) \rightarrow$ represents the language $L$ which includes all the strings of alternating 0's and 1's that begin with 0 and end with 1.

2) Lecture Notes on Regular Languages and Finite Automata
Regular language:

A language that can be expressed using a *regular expression*.

Example:

M is a regular language
given by regular expression

\[ r = 1^*(0(1^*)0(1^*))^* \]

Some cannot be represented

Example:

Set of strings \( \{a^n b^n \mid n \geq 0 \} \) cannot be expressed using a finite automaton

DFA

Deterministic Finite Automata (DFA):

Involves:

1) Finite set of states \( (Q) \)
2) Finite set of Input symbols \( (\Sigma) \)
3) Transition function:
   \[ \text{transition
d_function (a state, input symbol) } \rightarrow \text{ another state} \]
   or
   \[ \delta : Q \times \Sigma \rightarrow Q \]
4) Start State (a state from \( Q \))
5) A set of final/accepting states \( F \) such that \( F \subseteq Q \)

2) Lecture Notes on Regular Languages and Finite Automata
DFA

DFA: M with a binary alphabet, which requires that the input contains an even number of 0s.

The state diagram for M

\[ M = (Q, \Sigma, \delta, q_0, F) \]

where

\[ Q = \{S_1, S_2\}, \]
\[ \Sigma = \{0, 1\}, \]
\[ q_0 = S_1, \]
\[ F = \{S_1\}, \] and

Instead of regular expressions...

We will use production rules (which do the same thing as REs):

A production rule is a:

1) 'rewrite rule': rule of the form $A \rightarrow X$, where:
   A 'label'
   X 'sequence of labels or semantic units'
2) which recursively substitutes a **non-terminal**
3) with a string of **terminals** and/or **non-terminals**

**terminal:**

1) literal that actually occurs in the language being expressed.
2) cannot be rewritten/broken down further.
3) cannot be changed using rules of the grammar.

---

# Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Automaton</th>
<th>Production rules (constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
<td>$\alpha \rightarrow $\beta (no restrictions)</td>
</tr>
<tr>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Linear-bounded non-deterministic Turing machine</td>
<td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td>
</tr>
<tr>
<td>Type-2</td>
<td>Context-free</td>
<td>Non-deterministic pushdown automaton</td>
<td>$A \rightarrow \gamma$</td>
</tr>
<tr>
<td>Type-3</td>
<td>Regular</td>
<td>Finite state automaton</td>
<td>$A \rightarrow \alpha$ and $A \rightarrow \alpha B$</td>
</tr>
</tbody>
</table>
Context-Free Languages

Languages that can be represented by context-free grammars

What are context-free grammars?

Formal grammar:

- with a set of production rules
- describes how to generate strings from the language's alphabet
- with a starting non-terminal, or head

Example:

Context-free grammar:

\[
\begin{align*}
A & \rightarrow aB \mid a \mid \epsilon \\
B & \rightarrow bAS \\
S & \rightarrow c
\end{align*}
\]

The following strings can be generated by these production rules:

abc, aabc, abac, ababc, …
### Context-Free Grammars

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENTENCE</td>
<td>SUBJECT VERB OBJECT</td>
</tr>
<tr>
<td>SUBJECT</td>
<td>ARTICLE NOUNPHRASE</td>
</tr>
<tr>
<td>OBJECT</td>
<td>ARTICLE NOUNPHRASE</td>
</tr>
<tr>
<td>ARTICLE</td>
<td>a</td>
</tr>
<tr>
<td>ARTICLE</td>
<td>the</td>
</tr>
<tr>
<td>NOUNPHRASE</td>
<td>NOUN</td>
</tr>
<tr>
<td>NOUNPHRASE</td>
<td>ADJECTIVE NOUN</td>
</tr>
<tr>
<td>ADJECTIVE</td>
<td>big</td>
</tr>
<tr>
<td>ADJECTIVE</td>
<td>small</td>
</tr>
<tr>
<td>NOUN</td>
<td>cat</td>
</tr>
<tr>
<td>NOUN</td>
<td>dog</td>
</tr>
<tr>
<td>VERB</td>
<td>eats</td>
</tr>
</tbody>
</table>

Image from: Pratt, A. M., Lecture Notes on Regular Languages and Finite Automata, (pp 47)
Some other stuff

Chomsky grammars?
Pumping Lemma?
What does this have to do with DNA Self-assembly?

Generative grammars closely correspond to self-assembly and ligation.

Example: DNA-based computing:

DNA strands → set of strings over an alphabet

String over the set \{A, C, G, T\} if you read 5' → 3'

...can use specific notations to deal with:

- Complementarity (ACGTCG' = CGCAGCT)
- Circular strands (prefix symbol ◦)

Use a code book s.t. each symbol in \(\Sigma\) is represented by an \(N\)-base sequence.

\[ C : \Sigma \rightarrow D^N \]

What are Graph Grammars?

A graph grammar, or transformation, is:

Rule based modification of graphs

Core of a rule (or production), $p = (L, R)$

Find a match of $L$ in the source graph and replace $L$ with $R$ to get the target graph of the graph transformation.

Graph Grammars

A graph grammar, or transformation is:

A set of rules or production.

A production $p = (L, R)$ is a pair of graphs $(L, R)$ called LHS $L$ and RHS $R$.

Applying the rule $p = (L, R)$ means finding a match of $L$ in the source graph $(G)$ and replacing $L$ by $R$ → Leading to a target graph $(H)$
Graph Grammars

Before we go into more detail, we discuss the following basic question.

1.1.1 What Is Graph Transformation?

Graph transformation has at least three different roots:
• from Chomsky grammars on strings to graph grammars;
• from term rewriting to graph rewriting;
• from textual description to visual modeling.

We use the notion of graph transformation to comprise the concepts of graph grammars and graph rewriting. In any case, the main idea of graph transformation is the rule-based modification of graphs, as shown in Fig. 1.1.

The core of a rule or production, \( p = (L, R) \), is a pair of graphs \( (L, R) \), called the left-hand side \( L \) and the right-hand side \( R \). Applying the rule means finding a match of \( L \) in the source graph and replacing \( L \) by \( R \), leading to the target graph of the graph transformation. The main technical problems are how to delete \( L \) and how to connect \( R \) with the context in the target graph. In fact, there are several different solutions to how to handle these problems, leading to several different graph transformation approaches, which are summarized below.

1.1.2 Aims and Paradigms of Graph Transformation

Computing was originally done on the level of the von Neumann Machine which is based on machine instructions and registers. This kind of low-level computing was considerably improved by assembler and high-level imperative
Graph Grammars

A graph transformation from G ⇒H, usually contains the following steps:

1. Choose a production rule
2. Check application conditions
3. Remove from G the parts in L that are not in R to get D
4. Glue R to D at the part of L that still has an image in D. Add R/L to D to get E (if an additional embedding exists, embed further) and end with H.
Graph Grammars

Direct derivation of a production rule: consider production rule $p$ as a finite, schematic representation of a potentially infinite set of direct derivations.

If a match $m$ fixes an occurrence of $L$ in a given graph $G$, then $G \Rightarrow H$ denotes the direct derivation where $p$ is applied to $G$ leading to the derived graph $H$.

$$\begin{align*}
p : L & \rightsquigarrow R \\
G & \xrightarrow{p,m} H
\end{align*}$$
Graph Grammars

A match \( m \) for a production rule \( p \) is a graph homomorphism, mapping nodes and edges of \( L \) in to \( G \) s.t. graphical structures and labels are preserved.

\[ H_1 \text{ is constructed as } G_1 - (L_1 - R_1) \cup (R_1 - L_1). \]
Graph Grammars

Fig. 1.2. Graph transformation from an operational point of view
1. What is a “graph”?  
2. How can we match $L$ with a subgraph of $G$?  
3. How can we define the replacement of $L$ by $R$ in $G$?
So what?

- General Function
- Injective, Not surjective
- Surjective, Not injective
- Bijective (injective and surjective)
Problems?

Sometimes the rules can apply in more than one way on the same graph (2).

Figure 3.1: Direct derivations.

Figure 3.2: Example and schematic representation of direct derivation \( G \overset{p,m}{\Rightarrow} H \).
Problems?

It could also result in the deletion vertex that has an edge which is not part of the match (3).

Figure 3.1: Direct derivations.
Approaches

1. Node label replacement approach
2. Hyperedge replacement approach
3. Algebraic approach is based on pushout constructions:
   1. SPO
   2. DPO
4. Logical approach
5. Theory of 2-structures
6. Programmed graph replacement approach
Algebraic Approach

Double or Single pushout approaches.


Handbook of Graph Transformations

Fundamentals of Algebraic Graph Transformations
Algebraic Approach

DPO

![DPO Diagram](image)

**Fig. 1.3.** DPO graph transformation

![Example of DPO graph transformation](image)

**Fig. 1.4.** Example of DPO graph transformation
Algebraic Approach

If, in Fig. 1.4, vertex (2) were to be deleted from $K$, the line condition would not be satisfied in the DPO approach. In the SPO approach, this would mean that vertex (2) is not in the domain of $p$, leading to an edge $e$ in $G$ after deletion of $L \backslash \text{dom}(p)$ in Fig. 1.5. A result, edge $e$ would be deleted in $H$.

A detailed presentation and comparison of the two approaches is given in Volume 1 of the Handbook of Graph Grammars and Computing by Graph Transformation [Roz97]. In this book, however, we consider only the DPO approach as the algebraic graph transformation approach.

Fig. 1.5. Example of SPO graph transformation
Example: Directed Self-Assembly

Alphabet = \{a, b, c, \ldots\}

\[ h : G_1 \to G_2 \]

Function: label preserving embedding if:
1. \( h \) is injective,
2. \( \{x, y\} \in E_{G_1} \iff \{h(x), h(y)\} \in E_{G_2} \),
3. \( l_{G_1} = l_{G_2} \circ h \).

**Definition 1.** A rule is a pair of graphs \( r = (L, R) \) where \( V_L = V_R \). The graphs \( L \) and \( R \) are called the left hand side and right hand side of \( r \) respectively. The size of \( r \) is \( |V_L| = |V_R| \). Rules whose vertex sets have one, two and three vertices are called unary, binary and ternary, respectively.

We may refer to rules as being constructive \( (E_L \subseteq E_R) \), destructive \( (E_L \supset E_R) \) or mixed (neither constructive or destructive). A rule is acyclic if its right hand side contain no cycles (the left hand side may contain cycles).
**Definition 2.** A rule $r$ is applicable to a graph $G$ if there exists an embedding $h : L \rightarrow G$. In this case the function $h$ is called a witness. An action on a graph $G$ is a pair $(r, h)$ such that $r$ is applicable to $G$ with witness $h$.

**Definition 3.** Given a graph $G = (V, E, l)$ and an action $(r, h)$ on $G$ with $r = (L, R)$, the application of $(r, h)$ to $G$ yields a new graph $G' = (V', E', l')$ defined by

\[
V' = V
\]
\[
E' = (E - \{\{h(x), h(y)\} | \{x, y\} \in L\}) \\
\quad \cup \{\{h(x), h(y)\} | \{x, y\} \in R\}
\]
\[
l'(x) = \begin{cases} 
    l(x) & \text{if } x \notin h(V_L) \\
    l_R \circ h^{-1}(x) & \text{otherwise.}
\end{cases}
\]

We write $G \xrightarrow{r,h} G'$ to denote that $G'$ was obtained from $G$ by the application of $(r, h)$. 
**Definition 4.** A graph assembly system is a pair \((G_0, \Phi)\) where \(G_0\) is the initial graph and \(\Phi\) is a set of rules (called the rule set).

We often refer to a system simply by its rule set \(\Phi\) and assume that the initial graph is the infinite graph defined by

\[
G_0 \triangleq (\mathbb{N}, \emptyset, \lambda x.a) \tag{1}
\]

where \(a \in \Sigma\) is the *initial symbol* (here \(\lambda x.a\) is the function assigning the label \(a\) to all vertices).

**Definition 5.** An assembly sequence of a system \((G_0, \Phi)\) is a finite sequence \(\{G_i\}_{i=0}^k\) such that there exists a sequence of actions \(\{(r_i, h_i)\}_{i=1}^k\) where \(r_i \in \Phi\) and

\[
G_i \xrightarrow{r_i, h_i} G_{i+1}
\]

for \(i \in \{0, ..., k - 1\}\).
Definition 6. A connected graph $G$ is reachable in a system $(G_0, \Phi)$ if there exists an assembly sequence $\{G_i\}_{i=0}^{k}$ of $(G_0, \Phi)$ such that $G$ is isomorphic to some component of $G_k$. The set of all such reachable graphs is denoted $\mathcal{R}(G_0, \Phi)$, or just $\mathcal{R}(\Phi)$ if $G_0$ defined by (1).

Definition 7. A graph $G \in \mathcal{R}(G_0, \Phi)$ is stable if for all $G'$ there does not exist an action $(r, h)$ on the disjoint union $G \sqcup G'$ such that $r = (L, R) \in \Phi$ and $h(L) \cap V_G$ is nonempty. The set of all such stable graphs is denoted $\mathcal{S}(G_0, \Phi)$, or just $\mathcal{S}(\Phi)$ if $G_0$ defined by (1).
\[
\Phi_2 = \left\{
\begin{align*}
  a \quad a & \Rightarrow b - c, \\
  b & \Rightarrow d , \\
  a \quad c & \Rightarrow e - d, \\
  b_1 & \Rightarrow f_1 , \\
  a \quad e & \Rightarrow g - f, \\
  b_2 - f_2 & \Rightarrow b_3 \quad f_3
\end{align*}\right\
\]

**Fig. 6.** A particle based embedding of self assembly using graph grammars.
Single Pushout Approach
Double Pushout Approach
Brijder et al
Kawamata et al
McCaskill et al
Klavins et al