Towards the automatic programming of H systems: jHsys, a Java H system simulator

Rojas Siles, J.M.[1] and Cruz Echeandía M. de la and Ortega de la Puente, A.[2]

Abstract The main goal of this paper is to describe how we consider that some abstract bio-inspired computing devices (such as splicing systems) can be automatically programmed (designed) in the future. One of the necessary steps is to formally describe the computer being programmed (splicing systems). Some of the authors of this paper have previously solved this problem. Another necessary step is to develop a simulator for H systems. We propose applying Christiansen Grammar Evolution (an evolutionary automatic programming algorithm developed by the authors) to complete the process. This technique includes a fitness function that the simulator requires. This paper is devoted to describe jHSys, a Java simulator for splicing (H) systems.

1 Motivation

The task of automatically writing programs can be seen as a search problem: finding the best in a set of candidate programs automatically generated. Any general search technique can be used to solve this problem.

Conventional personal computers are based on the well known von Neumann architecture, that tries to implement the Turing machine by means of electronic devices: aritmetic and logic unit, registers, data and direction buses, etc. A great effort is being dedicated to one of the current topics of interest of our research group: the design of new abstract computing devices which can be considered as alternative architectures to design new families of computers. Some of them, inspired in the way used by Nature to efficiently solve difficult tasks, are called *natural or unconventional computers*. A few of the natural phenomena inspiring these devices are:

[1]Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Universidad Politécnica de Madrid ·[2]Departamento Ingeniería Informática, Universidad Autónoma de Madrid, e-mail: josemiguel.rojas@upm.es, marina.cruz@uam.es, alfonso.ortega@uam.es

the role of membranes in behaviour of cells, the structure of genetic information, and the way in which species evolve.

Any computer scientist has a clear idea about how to program *conventional* (von Neumann) computers by means of different high level programming languages and their corresponding compilers, which translate programs into machine code. On the other hand, imagining how to program unconventional computers is quite difficult.

One of the main topics of interest of our research group is the formal specification of complex systems that makes it possible to apply formal tools to their design, or study some of their properties. We have successfully applied this approach to different bio-inspired computational devices (L systems, cellular automata [2, 3, 4, 5]) and proposed a new evolutionary automatic programming algorithm (Christiansen Grammar Evolution or CGE [6]) as a powerful tool to design complex systems to solve specific tasks.

CGE wholly describes the candidate solutions, both syntactically and semantically, by means of Christiansen grammars. CGE improves the performance of other approaches, because it reduces the search space by excluding non-promising individuals with syntactic or semantic errors.

Splicing systems are abstract devices with a complex structure, because some of their components depends on others. This dependence makes it difficult to use genetic techniques to search splicing systems because, in this circumstance, genetic operators usually produce a great number of incorrect individuals (both syntactically and semantically).

This paper is focused on one of the steps needed for using CGE to automatically program splicing systems: their simulation.

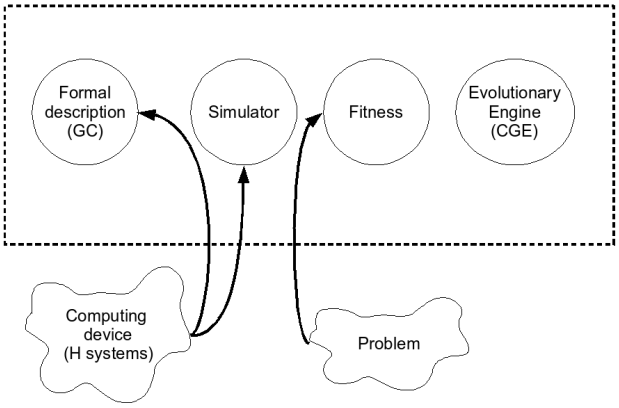

Fig. 1 Blocks of a general way to program natural computers

Figure 1 graphically describes the different blocks which can be considered to propose a general way to program natural computers similar to splicing systems to solve a given problem.

This method takes as inputs the following elements: (1) the *target problem* to be solved and (2) the *computing device* that will be used to solve the problem.

The method consists of the following modules: (1) An *evolutionary engine*, used as an automatic programming algorithm. This engine has to handle candidate solutions with a complex structure. We propose using Christiansen Grammar Evolution. (2) A *formal description of the computing device* being programmed. In [1] the authors design a Christiansen grammar for universal splicing systems. (3) A *simulator for the computing device* that will be used to compute the fitness function. The current paper is focused on this module. (4) The *fitness function*, which must fulfill two roles: simulating the generated solution (in this case, a particular splicing system) and measuring how well the solution solves the target problem.

In the following sections, we will briefly introduce splicing systems, and describe jHsys, a Java simulator for splicing (H) systems. Finally, conclusions and further research lines are discussed.

2 Introduction

2.1 Introduction to splicing systems

Splicing systems were introduced by Head in [7] as a DNA inspired computing device. Splicing systems formalize the DNA recombination operation.

A splicing rule on strings is formally defined by means of four patterns u_1, u_2, u_3 and u_4 and, when applied to two strings $x = x_1u_1u_2x_2$ and $y = y_1u_3u_4y_2$, it can produce two resulting strings $z = x_1u_1u_4y_2$ and $z' = y_1u_3u_2x_2$; although the second one (z') usually is discarded because the simmetric rule has the same result.

Formally they are represented as $u_1\#u_2\$u_3\#u_4$

A splicing system (H) consists of a set of splicing rules (R) that is applied to a set of strings (L), both sets share the same alphabet (V).

Extended splicing systems are one of the best studied variants of the basic model. They use a distinguished subset of the alphabet (T) named *set of terminals* with the same meaning than in Chomsky grammars.

Formally $H = (V,T,L,R)$

The *language generated by* a splicing system ($L(H)$) contains all the possible results of the splicing rules to the initial language (L) The languages generated by extended splicing systems contain only strings of terminal symbols.

For example. It is easy to demonstrate that the following system generates strings with the structure $c_1(ab)^n(ab)^m...c_2$

$$H_0 = \{V_0, L_0, R_0\}$$

- $V_0 = \{a,b,c_1,c_2,c_3,c_4\}$
- $L_0 = \{c_1abc_2, c_3ac_4, c_4bc_3, c_1c_4, c_4c_2\}$
- $R_0 = \{c_3a\#c_4\$c_1\#a^+b^+c_2, c_3a^+b^+\#c_2\$c_4\#bc_3, c_1\#c_4\$c_3\#a^+b^+c_3,$
  $c_1a^+b^+\#c_3\$c_4\#c_2, c_1a^+b^+\#\$c_1\#a^+b^+c_2\}$

The complexity of a splicing system can be studied in terms of the complexity of the initial set and the complexity of the rules in the Chomsky Hierarchy. It is known [8] that the extended splicing system with a finite initial language and with rules whose patterns are specified using regular expressions is equivalent to Turing machines.

2.2 jHSys: a Java splicing systems simulator

2.2.1 jHSys design

jHSys stands for java H System Simulator. It is a multi-threaded Java application to simulate an ample set of families of splicing systems, including those introduced in [8] (regular extended H systems). jHSys allows the specification of a splicing system H in terms of its main components (an alphabet V, an initial language L and a set of splicing rules R).

jHSys fulfills all the features of the original splicing systems model. Moreover, the design of our simulator allows us to easily scale it to support new variants of this kind of systems. An example of this characteristic is the possibility of defining a set of terminals ($T \subseteq V$) to simulate the aforementioned family of extended splicing systems. The simulator can also deal with different stopping conditions, to determine when a particular simulation should stop. Thus, a set of stopping conditions S is added to the initial model. The way of specifying these conditions is shown later on this paper.

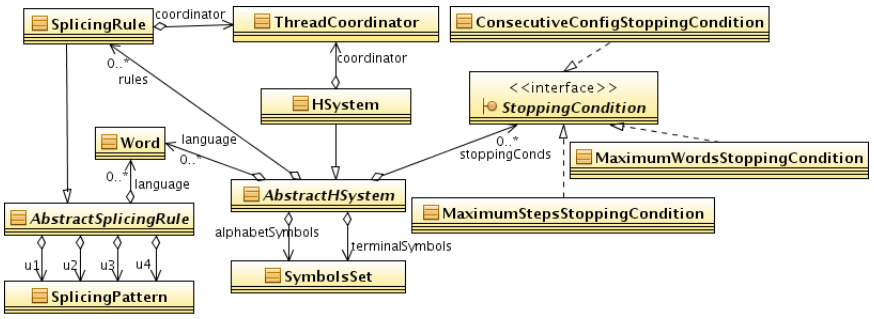

Fig. 2 jHSys Class Diagram

Figure 2 shows a simplified class diagram of jHSys. For the sake of simplicity, the attributes and methods implemented by the different classes are omitted.

We have used a Java interface to represent stopping conditions. This technique will ease the future inclusion of new kinds of conditions. The class `SplincingRules` implements the Java interface `Runnable`. A different Java thread is in this way as-

signed to each rule to improve the efficiency of the simulator on parallel platforms. The `ThreadCoordinator` class properly synchronizes the threads.

jHSys simulations can be summarized as follows:

0. The user has to describe the system under consideration in an XML file.
1. This input file is translated into the corresponding Java data structures.
2. Until computation ends:
   1 An *APPLY_RULE step* is taken to verify and execute all the applicable splicing rules.
   2 An *UPDATE_LANGUAGE step* collects all the new words added in the previous step and includes them in the resulting language of the system.
   3 The stopping conditions are evaluated.

2.2.2 XML specification files

As we have previously mentioned, the input of jHSys is an XML file describing the system to be simulated. Figure 3 shows the XML code corresponding to the splicing system example previously described:

Fig. 3 XML file for the example

```
<?xml version="1.0"?>
<!-- H-System specification file -->
<HSYSTEM>
   <ALPHABET symbols="a b c1 c2 c3 c4"/>
   <TERMINALS symbols="a b c1 c2 c3 c4"/>
   <LANGUAGE words="c1abc2 c3ac4 c4bc3 c1c4 c4c2"/>
   <SPLICING_RULES>
      <SPLICING_RULE u1="c3a" u2="c4" u3="c1" u4="a+b+c2"/>
      <SPLICING_RULE u1="c3a+b+" u2="c2" u3="c4" u4="bc3"/>
      <SPLICING_RULE u1="c1" u2="c4" u3="c3" u4="a+b+c3"/>
      <SPLICING_RULE u1="c1a+b+" u2="c3" u3="c4" u4="c2"/>
      <SPLICING_RULE u1="c1a+b+" u2="" u3="c1" u4="a+b+c2"/>
   </SPLICING_RULES>
   <STOPPING_CONDITION>
      <CONDITION type="MaximumWordsStoppingCondition" maximum="12"/>
   </STOPPING_CONDITION>
</HSYSTEM>
```

- The main tag of the xml structure is `<HSYSTEM>`, that contains the whole description of the splicing system under consideration.
- The tag `<ALPHABET>` declares the available symbols.
- The tag `<TERMINALS>` contains, when simulating an extended splicing system, the set of terminal symbols. As a prerequisite of the simulation, jHSys will check whether it is a subset of the alphabet or not.
- The tag `<LANGUAGE>` contains the list of words of the initial language of the system. All their symbols must belong to the set defined by `<ALPHABET>`.

- The tag <SPLICING_RULES> contains all the splicing rules of the splicing system. Each rule is composed of four patterns ($u1$, $u2$, $u3$ and $u4$), which the user must define as Java regular expressions. For example, the splicing rule formally defined as $(c_3a\#c_4\$c_1\#a^+b^+c_2)$ is translated into the XML component <SPLICING_RULE u1="c3a" u2="c4" u3="c1" u4="a+b+c2"/>.
- Stopping conditions are specified within the tag <STOPPING_CONDITIONS>. jHSys supports three kinds of stopping conditions:
  - **MaximumWordsStoppingCondition**: It stops the simulation when the resulting language contains a certain number of words. The following fragment shows the XML tag for 12 words
    <CONDITION type="MaximumWordsStoppingCondition" maximum="12"/>
  - **MaximumStepsStoppingCondition**: Simulation stops after a given number of steps. The following tag corresponds to 15 steps:
    <CONDITION type="MaximumStepsStoppingCondition" maximum="15"/>
  - **ConsecutiveConfigStoppingCondition**: Simulation finishes when there is no change in the resulting languages between two consecutive APPLY_RULE steps. Its representation in the xml file is
    <CONDITION type="ConsecutiveConfigStoppingCondition"/>

2.2.3 Running jHSys

There are two versions of jHsys. Either can be run by invoking it from the command line or by using an Eclipse plug-in. Eclipse [9] is a free integrated developing environment. It provides the user with a set of tools that includes a rich XML editor. To use jHSys with Eclipse, the user must download, install the plug-in (available online at *http://www.clip.dia.fi.upm.es/∼jmrojas/*), and use the provided toolbar options. In this case, the user can benefit from the features of the built-in XML editor to create and modify the files that describe the splicing systems being simulated. The *console view* of Eclipse will contain the output of the simulations.

The standalone version of the simulator is also available at *http://www.clip.dia.fi. upm.es/∼ jmrojas/*, and it is invoked by the command

```
jhsys [inputFile.xml] [outputFile.log]
```

or

```
java -jar jhsys [inputFile.xml] [outputFile.log]
```

Where

- inputFile.xml describes the splicing system under consideration.
- outputFile.log is a log file that will store the output of the simulation:
  - The first lines briefly describe the splicing system.
  - There is a block for each APPLY_RULE step with a line for each applied rule with the list of words it adds to the resulting language.
  - There is a block for each UPDATE_LANGUAGE step with the current language.

Figure 4 shows the log file corresponding to the splicing system we have previously described.

After two iterations (a total of four steps) the `MaximumWordsStoppingCondition` is reached, i.e., the resulting language contains 12 words. At this moment the simulation stops and the result is the language generated by the second `UPDATE_LANGUAGE` step (the fourth step). Figure 4 shows the complete output for this example.

Fig. 4 Contents of the log file for the studied splicing system

```
XML FILE LOADED AND PARSED SUCCESSFULLY...
H-SYSTEM INFO PARSED SUCCESSFULLY...
STOPPING CONDITIONS INFO PARSED SUCCESSFULLY...
H-SYSTEM XML PARSED SUCCESSFULLY...
RUNNING H-SYSTEM...
*** H-SYSTEM INITIAL CONFIGURATION ***
ALPHABET: [a, b, c1, c2, c3, c4]
TERMINALS: [a, b, c1, c2, c3, c4]
LANGUAGE: [c1abc2, c3ac4, c4bc3, c1c4, c4c2]
SPLICING RULES:
Splicing Rule 0 : [u1="c3a", u2="c4", u3="c1", u4="a+b+c2"]
Splicing Rule 1 : [u1="c3a+b+", u2="c2", u3="c4", u4="bc3"]
Splicing Rule 2 : [u1="c1", u2="c4", u3="c3", u4="a+b+c3"]
Splicing Rule 3 : [u1="c1a+b+", u2="c3", u3="c4", u4="c2"]
Splicing Rule 4 : [u1="c1a+b+", u2="", u3="c1", u4="a+b+c2"]
STOPPING CONDITIONS:
MaximumWords=12
*** APPLY_RULE - STEP 1 ***
Rule 2 adds new words: []
Rule 0 adds new words: [c3aabc2, c1c4]
Rule 1 adds new words: [c3aabbc3, c4c2]
Rule 3 adds new words: []
Rule 4 adds new words: [c1ababc2, c1c2]
*** UPDATE_LANGUAGE - STEP 2 ***
LANGUAGE: [c1abc2, c3ac4, c4bc3, c1c4, c4c2, c3aabc2, c3aabbc3,
c1ababc2, c1c2]
*** APPLY_RULE - STEP 3 ***
Rule 1 adds new words: [c3aabbc3, c4c2]
Rule 2 adds new words: [c1aabbc3, c3c4]
Rule 4 adds new words: [c1ababc2, c1c2, c1ababc2, c1abc2, c1aababc2,
c1bc3, c1aabbabc2, c1c3]
Rule 0 adds new words: [c3aabc2, c1c4]
Rule 3 adds new words: [c1aabbc2, c4c3]
*** UPDATE_LANGUAGE - STEP 4 ***
LANGUAGE: [c1abc2, c3ac4, c4bc3, c1c4, c4c2, c3aabc2, c3aabbc3,
c1ababc2, c1c2, c1aabbc3, c3c4, c1aababc2, c1bc3, c1aabbabc2,
c1c3, c1aabbc2, c4c3]

Stopping condition found: HSystems.stopping.MaximumWordsStoppingCondition
H-System has stopped!!!
```

3 Conclusions and future research

This paper describes a Java simulator for an ample set of splicing systems that includes some well-known universal H systems. The simulator reads the description

of the systems from an XML file and stores the results in a log file. Some of the authors of this paper have previously designed a Christiansen grammar for the same families of splicing systems. In the future, we plan to study the possibility of proposing a methodology to automatically design splicing systems to solve given problems by means of Christiansen Grammar Evolution (an evolutionary automatic programming algorithm proposed by the authors of this paper).

In order to reach this goal, we will need to follow the following steps:

- Selecting a particular problem to be solved with splicing systems.
- Particularizing the Christiansen grammar that we have previously designed according to the particular family that can solve this problem and developing a Java version of the grammar that can be handled by the proposed algorithm.
- Developing a fitness function with the simulator described in this paper.
- Designing a set of experiments to select the proper splicing system.

Acknowledgements This work was partially supported by DGUI CAM/UAM, project CCG08-UAM/TIC-4425. J. M. Rojas Siles is supported by the Spanish Ministry of Science and Innovation under the grant BES-2009-027467, associated to the TIN2008-05624 "DOVES" project. The authors thank Dr. Manuel Alfonseca for his help while preparing this document.

References

1. Cruz M. de la, Ortega A.: A Christiansen grammar for universal splicing systems. In Methods and models in artificial and natural computation, a homage to professor Mira's scientific legacy. LNCS 2009, vol. 5601, p.336–345.
2. Ortega A., Dalhoum A.A., Alfonseca M.: Grammatical evolution to design fractal curves with a given dimension, IBM Jr. of Res. and Dev. Vol. 47:4, p. 483-493 July. 2003.
3. Ortega A., Dalhoum A.L.Abu, Alfonseca M.: Cellular Automata equivalent to DIL Systems, 5th Middle East Symposium on Simulation and Modelling (MESM 2003), Eurosim, 5-7 Ene. 2004, Sharjah, Emiratos Arabes Unidos. Pub: Proceedings, ISBN: 90-77381-06-6, pp.120–124.
4. Dalhoum A.L.Abu, Ortega A., Alfonseca M.: Cellular Automata equivalent to PD0L Systems, International Arab Conference on Information Technology (ACIT 2003), 20-23 Dic. 2003, Alexandria, Egipto. Pub: Proceedings, pp. 819–825.
5. Dalhoum A.L.Abu, Ortega A., Alfonseca M.: Cellular Automata equivalent to D0L Systems, 3rd WSEAS International Conference on Systems Theory and Scientific Computation, Special Session on Cellular Automata and Applications, 15-17 Nov. 2003, Rodas, Grecia. Pub: Proceedings en CDROM.
6. Ortega, A. and de la Cruz, M. and Alfonseca, M.: Christiansen Grammar Evolution: grammatical evolution with semantics, in IEEE Transactions on Evolutionary Computation vol. 11, p. 77-90. 2007
7. T. Head, Splicing schemes and DNA, in: G. Rozenberg and A. Salomaa, eds., Lindenmayer Systems; Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology (Springer, Berlin, 1992) 371-383.
8. Gh. Paun, Regular extended H systems are computationally universal, Journal of Automata, Languages, Combinatorics 1 (1) (1996) 27-36.
9. http://www.eclipse.org/