Dublin City University
*Research Institute for Networks and Communications Engineering*
*School of Electronic Engineering, Dublin, Ireland*

# The Holland Broadcast language

James Decraene

November 2006

**Artificial Life Laboratory**

Technical report ALL-06-01

James Decraene
School Of Electronic Engineering
Dublin City University
Dublin 9, IRELAND

Telephone: +353-1-700 7696
E-mail: james.decraene@eeng.dcu.ie

**Abstract**

The broadcast language is a programming formalism devised by Holland in 1975, which aims at allowing Genetic Algorithms (GAs) to use an *adaptable representation*. A GA may provide an efficient method for adaption but still depends on the efficiency of the fitness function used. During long-term evolution, this efficiency could be limited by the fixed representation used by the GA to encode the problem. When a fitness function is very complex, it is desirable to adapt the problem representation employed by the fitness function. By adapting the representation, the broadcast language may overcome the deficiencies caused by fixed problem representation in GAs.

This report describes an initial detailed specification and implementation of the broadcast language. Our first motivation is the fact that there is currently no published implementation of broadcast systems (broadcast language-based systems) available. Despite Holland presented the broadcast language in his book "Adaptation in Natural and Artificial systems", he did not support this approach with experimental studies.

Our second motivation is the affirmation made by Holland that broadcast systems could model biochemical networks. Indeed Holland also described how the broadcast language could provide a straightforward representation to a variety of biochemical networks (Genetic Regulatory Networks, Neural Networks, Immune system etc). As these biochemical models share many similarities with Cell Signaling Networks (CSNs), broadcast systems may also be considered to model CSNs. One of our goals, within the ESIGNET project, is to develop an evolutionary system to realize and evolve CSNs in Silico. Examining the broadcast language may provide us with valuable insights to the development of such a system.

In this paper, we initially review the Holland broadcast language, we then propose a specification and implementation of the language which is later illustrated with an experiment: modeling different chemical reactions.

# Contents

# 1 Introduction

Our current investigation is concerned with the development of an evolutionary system platform to evolve a closed control system of CSNs. This evolutionary system is based on our novel class of string-based Artificial Chemistries: the Molecular Classifier system (MCS). The main inspiration of MCS comes from Holland Classifier systems [3] which differs from the MCS by distinguishing a demarcation between *messages* and *classifiers*. However this specific demarcation property is also shared by the precursor of Holland Classifier systems : the broadcast language which was proposed by the same author in 1975 [4].

Holland proposed the broadcast language to solve some undesirable issues met with using Genetic Algorithms (GAs) [4, 1]. Holland argued that GAs provide an efficient method of adaptation, however in the case of long-term adaptation, the efficiency of GAs could be limited by the representation used to encode the problem. This problem representation is fixed and influences the complexity of the fitness function, during long-term evolution, this may limit the performances of the GA. To overcome this limitation, Holland proposed to adapt the problem representation used by the fitness function. Adapting the representation may then generate correlations between the problem representation and the GA performance.

Another interesting feature discussed by Holland is that the broadcast language is a Turing Complete programming language. Nevertheless this Turing Completeness property was only claimed and has never been formally demonstrated. If the broadcast language is indeed Turing Complete then no long-term limits will be given by the language itself. Although this issue is not covered in this paper, it is still interesting to mention this property as it may be of some relevance at a later stage.

Following this, Holland argued that the broadcast language provides a straightforward representation to a variety of models such as the operon-operator model (a Genetic Regulatory Network model). As Genetic Regulatory Networks (GRNs) and CSNs share many properties, it is common that a modeling technique applied to GRNs could also be employed for the modeling of CSNs. Thus, if the broadcast language can model GRNs then it might as well model CSNs.

However, after describing some of the potential merits of the broadcast language, we need to consider the fact that Holland did not support this approach with experimental studies. We may also note that there is currently no published studies on broadcast systems (broadcast language-based systems) in the literature. After presenting the broadcast language in his book, Holland did not pursue this idea but proposed another system which is a simplification of the broadcast language: the Learning Classifier system (LCS) [3]. In 2001, Holland [2] also discussed on the use of an LCS-based agent model to study the evolution of complexity of signaling networks. Nevertheless, we still focus on the broadcast language because on the contrary to LCS, broadcast system do not make the distinct demarcation between message and rules. This property is shared with our MCS and is motivated by biological plausibilities (a protein may act as a message/substrate *and* as a rule/catalyst).

We believe that the examination of broadcast systems will provides significant insight for the development of our evolutionary simulation platform, for the following reasons:

- No demarcation between rules/messages in broadcast system, a key property shared with the MCS
- Modeling CSNs may be a natural application of broadcast system as claimed by Holland
- This will at least provide more understanding on Holland broadcast language
- The broadcast language is supposedly a Turing Complete programming language

This report is organized as follows: we first provide a detailed specification of the broadcast language, we then present the broadcast system implementation. This is followed by an experiment in which we evaluate the modeling of different chemical reactions using the broadcast system.

# 2 The Broadcast Language

In this section we first present an overview of the broadcast system, in which we make an analogy between broadcast systems processes and real chemical reactions. We then discuss in detail the language specification: the syntax and semantic.

## 2.1 An overview

We introduce the broadcast language specification by providing an overview of the broadcast language. The broadcast language basic components are called *broadcast units* which can be viewed as condition/action rules. Whenever a broadcast unit conditional statement is satisfied, the action statement is executed. This means that whenever a broadcast units detect in the environment the presence of (a) specific signal(s), including themselves, then the broadcast unit would broadcast an output signal.

As an example, we may consider a given broadcast unit that upon detecting signals $I_1$ and $I_2$ would broadcast an output signal $I_3$. This is analogous to catalysts which would form a product upon the binding of a specific substrate to its binding region. In this example a catalyst can be thought of as a broadcast unit, a substrate would be a detected signal, the catalyst binding region would refer to the broadcast unit condition, the product is the output signal and finally the environment would the reaction space (e.g. the cell).

Following above analogy, a substrate can be degraded during catalysis, we address this issue in the signal processing ability of broadcast units. Indeed signal processing can also be performed with broadcast units: i.e. a broadcast unit may detect a signal $I$ and broadcast a signal $I'$, so that $I'$ is some modification of signal $I$.

Some broadcast units may broadcast a signal that may constitute a new broadcast unit. Similarly, a broadcast unit can be interpreted as a signal detected by another broadcast unit. As a result, a broadcast unit may create new broadcast units or detect and modify an existing broadcast unit.

A set of broadcast units, combined as a string, designates a *broadcast device*. A broadcast device can be viewed as a protein complex in which interactions between the several proteins result in complex functional behavior of the complex. A collection of broadcast devices could then be regarded as a computer program.

Table 1: Comparison of biological and broadcast language terminology

| Biology | broadcast language |
|---|---|
| sequence of amino acids from $\{A, R, N, D, C, E, \ldots\}$ | string of symbols from $\Lambda = \{0,\ 1,\ *,\ :,\ \Diamond,\ \triangledown,\ \blacktriangledown,\ \triangle,\ p,\ '\}$ |
| substrate | input signal |
| product | output signal |
| protein with no enzymatic function | null unit |
| enzyme | broadcast unit |
| protein complex | broadcast device |
| cellular milieu | list of strings from $\Lambda$ |

As a summary, the above table presents a comparison between the biological and the broadcast system terminology.

## 2.2 The syntax

We first describe the different structures constituting the language: the symbols, the broadcast units and broadcast devices. The interpretation of the symbols, broadcast units and broadcast devices will the follow.

The broadcast language alphabet $\Lambda$ is finite and contains ten *symbols*, $\Lambda^*$ is the set of strings over $\Lambda$. The symbols constitute the atomic elements of the language.

$$\Lambda = \{0,\ 1,\ *,\ :,\ \Diamond,\ \triangledown,\ \blacktriangledown,\ \triangle,\ p,\ '\}$$

Let $I$ be an arbitrary string from $\Lambda^*$, in $I$, a symbol is said to be quoted if it is preceded by a symbol $'$. A broadcast unit $I_n$ is an arbitrary string from $\Lambda^*$ which does not contain neither unquoted $*$ nor unquoted $:$. A set of broadcast units may be concatenated to form a *broadcast device*, a broadcast device $I$ may contain $0 \leq n \leq \infty$ broadcast units $I_1, \ldots, I_n$. If $n = 0$ then $I$ does not contain any broadcast unit and $I$ is then called

a *null* device. A null device is only interpreted as a signal and does not broadcast a signal under any circumstances. A broadcast device which is not null is said to be *active* any may broadcast an output upon the detection of appropriate signals.

$$I = p10 * 11' * \triangle 0 : 1\triangle * : 11\triangledown : 11\triangledown$$

$$I' = 011p' * *\triangledown : \Diamond 1011\triangle$$

$$I'' = p11' * \triangle 0 : 1\triangle' * : 1p1\triangle : 0001\Diamond$$

The above is an example of broadcast devices, note that $I''$ is a null device. A broadcast device $I$ is parsed into broadcast units as follows:

- Any prefix occurring to the left of leftmost $*$ is ignored.
- The first broadcast unit is designated from leftmost unquoted $*$ to (not including) the next unquoted $*$ on the right if any.
- Following broadcast units are obtained by repeating above procedure for each successive unquoted $*$ from the left.

For example the broadcast device $I$:

$$I = p10 * 11' * \triangle 0 : 1\triangle * : 11\triangledown : 11\triangledown$$

designates two distinct broadcast units $I_1$ and $I_2$:

$$I_1 = *11' * \triangle 0 : 1\triangle$$

$$I_2 = * : 11\triangledown : 11\triangledown$$

Four types of broadcast unit can be constructed other than the null unit. To determine the broadcast unit type, we first need to identify the number $s$ of unquoted : occurring in $I$. If $s \geq 3$ then the third : and anything to the right of it are ignored.

1. $*I_1 : I_2$
2. $* : I_1 : I_2$
3. $*I_1 :: I_2$
4. $*I_1 : I_2 : I_3$

For example:

$$I_1 = 11' * \triangle 0 : 1\triangle \text{ is of type 1}$$

$$I_1 = 11' * \triangle 0 : 1\triangle \text{ is of type 2}$$

The interpretation of the different types of broadcast unit will be presented in the following section.

## 2.3 The semantic

In this part, we first describe interpretation of the different types of broadcast units. We then present how the symbols are interpreted in a broadcast unit. We finally show how to resolve semantical conflicts that may appear in some broadcast units.

### Broadcast Units

The finite collection of broadcast devices can be described by its *state* $S$ at each time step $t$. For example $S(0) = \{011 : *\triangle 011 : 11, 101, 100, 0111 : 01 :\}$ describes the set of broadcast devices at time step $t = 0$, which corresponds to the initial state of the collection. Four types of broadcast unit can be distinguished, any other broadcast units that do not follow one of those four schemes are null units. broadcast units may engage in the following interactions based on discrete timesteps:

1. $*I_1 : I_2$

   If a signal of type $I_1$ is detected at time $t$ then the signal $I_2$ is broadcast at time $t + 1$.

2. $* : I_1 : I_2$

   If there is no signal of type $I_1$ present at time $t$ then the signal $I_2$ is broadcast at time $t + 1$.

3. $*I_1 :: I_2$

   If a signal of type $I_1$ is detected at time $t$ then a *persistent* string of type $I_2$ (if any) is removed from the environment at the end of time $t$.

4. $*I_1 : I_2 : I_3$

   If a signal of type $I_1$ and a signal of

type $I_2$ are both present at time $t$ then the signal $S_3$ is broadcast *at same time* $t$ unless the string $I_3$ contains unquoted symbols $\{\triangledown, \blacktriangledown, \triangle\}$ or singly quoted occurrence of $*$, in which case the string $I_3$ is broadcast a time $t+1$.

For broadcast units of type 1 and 2, the string $I_2$ refers to the output signal. Whereas $I_1$ is said to be a broadcast unit argument, and this is the case for any types of broadcast unit. Nevertheless, we also have additional broadcast unit arguments $I_2$ for broadcast unit of type 3 and 4. Finally, in the case of type 4 broadcast unit, $I_3$ correspond to the output signal.

When a broadcast unit of type 2 is fired at time $t$, this implies the deletion of a persistent signal of the. Persistent signals include signal starting with an unquoted occurrence of $p$ but also active broadcast devices.

Also when an output signal is interpreted for broadcast, one quote is removed from each quoted symbol. This allows one to use the quote symbol to "protect" special symbols to be passed into the output signal, see next section for an example. A broadcast unit may broadcast only once at each time step.

### The symbols

The interpretation of each symbol in $\Lambda = \{0,\ 1,\ *,\ :,\ \Diamond,\ \triangledown,\ \blacktriangledown,\ \triangle,\ p,\ '\}$ is now presented. When a symbol is said to be ignored, this means that the symbol is not interpreted by the broadcast unit.

$\{0,1\}$ 0 and 1 are the basic elements to specify a signal. A string such as 010110 can be regarded as the signature of a particular signal. This signature can be employed by a broadcast unit to detect and identify a signal. For example: let $I_1 = *10111:00$ be a broadcast unit and $I_2 = 10111$ a signal, both strings are present at time $t$ in the environment: $S(t) = \{*10111:00, 10111\}$. At time $t$, $I_2$ is detected by $I_1$, this triggers the activation of broadcast unit $I_1$, as a result: $S(t+1) = \{*10111:00, 10111, 00\}$.

$*$ As mentioned earlier, this symbol indicates that the following symbols until the next unquoted $*$ (if any) are to be interpreted as a broadcast unit. If a broadcast device $I$ does not contain any unquoted $*$ then $I$ is a null unit.

: This symbol is used as a punctuation mark to differentiate the arguments of a broadcast unit. The symbol : also determines the type of the broadcast unit as presented earlier. If more than two unquoted : are found in a broadcast unit then the third : and anything to the right of it are ignored.

$\Diamond$ When this symbol is met in the argument of a broadcast unit, it indicates that a signal detected by the broadcast unit may present any symbol at this position. This specific symbol occurring in the detected signal does not affect its acceptation or rejection by the broadcast unit.

For example, with $S(t) = \{*10\Diamond 11 : 00, 10011\}$ we still obtain $S(t) = \{*10\Diamond 11 : 00, 10011, 00\}$, $*10\Diamond 11$ would broadcast 00 if a signal containing $10\ldots11$ is present ($\ldots$ indicates an arbitrary symbol from $\Lambda$).

Also if $\Diamond$ occurs at the rightmost position in the argument of a broadcast unit, then it indicates that a signal detected by the broadcast unit may present any suffix without affecting acceptance or rejection.

For example, with $S(t) = \{*1011\Diamond : 00, 10011\}$ we obtain $S(t+1) = \{*1011\Diamond : 00, 101101101, 00\}$, $*1011\Diamond : 00$ would broadcast 00 if any signal containing the prefix 1011 is detected.

$\triangledown$ When this symbol occurs in the arguments of a broadcast unit, it designates an arbitrary initial (prefix) of terminal (suffix) string of symbols. This allows one to pass string of symbols from input signal to the broadcast signal ($\approx$ unit processing).

For example, with $S(t) = \{*10\triangledown : \triangledown, 10011\}$ we obtain at $t+1$: $S(t+1) = \{*10\triangledown : \triangledown, 10011, 011\}$. In this case $\triangledown$ designates the suffix 011 occurring in the input

signal 10011. whereas if $S(t) = \{*10\triangledown : \triangledown, 100100101\}$ then we obtain at $t+1$: $S(t+1) = \{*10\triangledown : \triangledown, 100100101, 0100101\}$. If several occurrences of $\triangledown$ are found in a given broadcast unit then they all designate the same substring.

▼ This symbol is similar to $\triangledown$ but can concatenate different inputs signals.

For example, with $S(t) = \{*10\triangledown : 11\blacktriangledown : 000\triangledown\blacktriangledown, 10111, 1100\}$ we obtain at $t+1$: $S(t+1) = \{*10\triangledown : 11\blacktriangledown : 000\triangledown\blacktriangledown, 10111, 1100, 00011100\}$. In this case $\triangledown$ designates the suffix 111 occurring in the input signal 10111 and ▼ designates the suffix 00 found in the detected signal 1100. The format of the broadcast signal is $000\triangledown\blacktriangledown$, therefore we replace and concatenate $\triangledown$ and ▼ accordingly and we obtain the output signal 00011100.

$\triangle$ This element is employed in the same manner as $\triangledown$ and ▼ but designates an arbitrary *single* symbols which position can be anywhere in the argument of a given broadcast unit.

For example, with $S(t) = \{*11\triangle 0 : 1\triangle, 1100\}$ the output signal 10 is broadcast and thus $S(t+1) = \{*11\triangle 0 : 1\triangle, 1100, 10\}$. Whereas if $S(t) = \{*11\triangle 0 : 1\triangle, 1110\}$ the output signal 11 is broadcast and $S(t+1) = \{*11\triangle 0 : 1\triangle, 1100, 11\}$.

$p$ When this symbol occurs at the first position of a string, it designates a *persistent* string which persists over time until deleted even if the string is not an active broadcast unit. A null device occurring at time $t$ which is not persistent exists only for one timestep and is removed at the end of time $t$.

$'$ This symbols is used to quote a symbol in the arguments of a broadcast unit.

For example, with $S(t) = \{*11'\triangle 0 : 11, 11\triangle\}$ the input signal $11\triangle$ is detected by the broadcast unit and thus the output string 11 is broadcast at $t+1$: $S(t+1) = \{*11'\triangle 0 : 11, 11\triangle, 11\}$

**Semantical conflict resolution**

In some cases, the interpretation of some broadcast units and symbols may rise ambiguities, we present how those conflicts are resolved:

- If the arguments of a broadcast unit contain at least one unquoted occurrence of a symbol from the set $\{\triangledown, \blacktriangledown\}$ then these symbols must occur at the *first* or *last* position to be interpreted of the broadcast unit arguments, otherwise they are treated as null symbol without any interpretation.

  The symbols $\triangledown$ or ▼ are specifically designed to designate a prefix or suffix string. If they are met elsewhere within the arguments of a broadcast unit then they are simply ignored as it may otherwise leads to ambiguous interpretation. This applies to any types of broadcast unit.

- Still regarding $\triangledown$ and ▼ symbols, if the first broadcast unit argument $I_1$ contains more than one unquoted occurrence of a symbol from the set $\{\triangledown, \blacktriangledown\}$ then only the first one is operative and only if it occurs at the first position, the other ones are ignored.

  This also applies for the second broadcast unit argument $I_2$ in the case of type 4 broadcast units. However there is then an additional stipulation: If an operative symbol from the set $\{\triangledown, \blacktriangledown\}$ is the same in both $I_1$ and $I_2$ then only the one occurring in $I_1$ is interpreted, the other symbol met in $I_2$ is ignored.

- Moreover, if a broadcast unit contains more than one $\triangle$ symbols then only the leftmost occurrence of $\triangle$ is operative and is to be interpreted by the broadcast unit. The other occurrences of $\triangle$ found in the broadcast unit argument are ignored. A $\triangle$ symbol may occur anywhere within a broadcast unit argument. This applies to any types of broadcast unit.

- If a broadcast unit output signal contains an unquoted occurrence of $\triangle$, $\triangledown$ or ▼ and if there is no interpretable occurrences in the broadcast unit argument, then this symbol

occurring in the output signal is treated as a null symbol and is not broadcast.

- Similarly, if a single broadcast unit argument at time $t$ is satisfied simultaneously by two or more detected signals, then each signal is assigned a probability $p = 1/n$ where $n$ is the number of satisfying signals. Following this, a single detected signal is picked at random for interpretation by the broadcast unit using this distribution. This features allows the representation of *stochasticity* in the broadcast system.

### Examples

To illustrate above presentation of broadcast units and their interpretation, we present some examples of broadcast devices and how they are interpreted:

- $S(t) = \{*'\triangledown 10\blacktriangledown 01\triangledown : 11, 10001\}$
  $S(t+1) = \{*'\triangledown 10\blacktriangledown 01\triangledown : 11, 10001\}$

  In this example, the first occurrence of $\triangledown$ is quoted which means that this symbol is not interpreted, this indicates that the broadcast is looking for an input signal which starts with a $\triangledown$. The symbol $\blacktriangledown$ is ignored as it does not occur at the first or last position of the broadcast unit argument. The second occurrence $\triangledown$ occurs a the end of the argument, meaning that it is operative. This implies that an input signal may possess any suffix without affecting its acceptance or rejection. In order to get activated, the broadcast unit $*'\triangledown 10\blacktriangledown 01\triangledown : 11$ needs to detect a signal of the form $\triangledown 1001\ldots$ where the dots means any suffix. As there is no such signals in the environment, no signal is broadcast.

- $S(t) = \{*\triangledown 110 : \triangledown 001 : \triangledown 1111\blacktriangledown 01, 001, p10\triangle 110\}$
  $S(t+1) = \{*\triangledown 110 : \triangledown 001 : \triangledown 1111\blacktriangledown 01,\ 001,\ p10\triangle 110,\ p10\triangle 111101\}$

  This broadcast unit is of type 4, as the symbol $\triangledown$ occurs at the first position of both $I_1$ and $I_2$, only the first occurrence is operative whereas the second one is ignored. This broadcast unit is looking for two distinct signals, $I_1$ can be satisfied by a signal of the form $\ldots 110$ and $I_2$ is satisfied by a signal of the form 001. In this case a signal 001 is present, and the input signal $p10\triangle 110$ satisfies $I_1$ therefore the broadcast unit gets activated. Again this broadcast unit is of type 4 and may broadcast at the same time $t$, however the output signal is not broadcast at $t$ as the output contains an unquoted occurrence of $\triangledown$. As a result, the output signal is broadcast at time $t+1$. We note that an occurrence of $\blacktriangledown$ is present in the output signal, however there is no $\blacktriangledown$ occurring in both $I_1$ and $I_2$. Thus the symbol $\blacktriangledown$ can not be interpreted and is tread as null symbol. Finally, the output signal is built as follows: First, we interpret $\triangledown$ so that $\triangledown = p10\triangle$, then the strings 1111 and 01 are concatenated resulting in the output signal: $p10\triangle 111101$.

- $S(t) = \{*0110\triangle\triangle :: 1111\triangle, 11110, 01100, 0110\Diamond\}$
  $S(t+1) = \{*0110\triangle\triangle :: 1111\triangle, 01100\}$

  This broadcast unit is of type 2 meaning that if $I_2$ is satisfied at time $t$ then a persistent signal of the form $I_2$ will be removed at time $t+1$. In $I_1$ there are two occurrences of $\triangle$ therefore only the first one is operative and the second one is ignored. $I_1$ can be satisfied by a signal of the form 0110. where the dot stands for an arbitrary symbol. In this example, two environmental signals 01100 and $0110\Diamond$ are present and satisfy $I_1$. Each signal is assigned the probability $p = 1/2$, we pick one at random, in this case let us say the string 01100 is picked. $I_2$ is then be interpreted as a string of the form 11110 (using $\triangle = 0$). Finally, at time $t+1$ the signal 11110 is then removed from the environment.

  We note that, if the signal $0110\Diamond$ was picked instead of the signal 11110, then $\triangle$ would

have been interpreted as $\triangle = \Diamond$. Thus the broadcast unit would have been looking for a signal of the form 1111$\Diamond$ which is not present in the environment. As a result, no signal would have been removed at time $t + 1$.

# 3 Implementation

In this section we present our implementation of the Holland broadcast system. We first depict an overview of the system and we then describe each part of the system in detail. The broadcast system was implemented using the C++ language, see Appendix A for source code.

In this object oriented implementation we may distinguish three main classes:

- `Env` represents the environment, this object holds a list of all current existing devices.
- The class `BDevice` designates a broadcast device, an instantiation of `BDevice` may hold from 0 to 3 `BUnit` objects.
- The `BUnit` class refers to a broadcast unit, it may contain one or two argument(s) and an output signal, all represented by strings of characters.

At time $t$, all broadcast devices including null devices are stored in a vector of devices $S$, this vector is held by an instance of `Env` , at time $t = 0$, $S$ is empty. A vector of character strings $A$ is employed to hold signals (strings) to be added to $S$ at the beginning of $t$, at time $t = 0$ $A$ represents the initial set of broadcast devices. $D$ is a vector of strings holding signals to be removed from $S$ at the end of $t$.

Figure 3 presents an overview of the system from its initialization to its termination. We now discuss in detail each step presented in this diagram:

1. Initialization: an `Env` object is instantiated, vectors $S, A$ and $D$ are created and are empty by default.

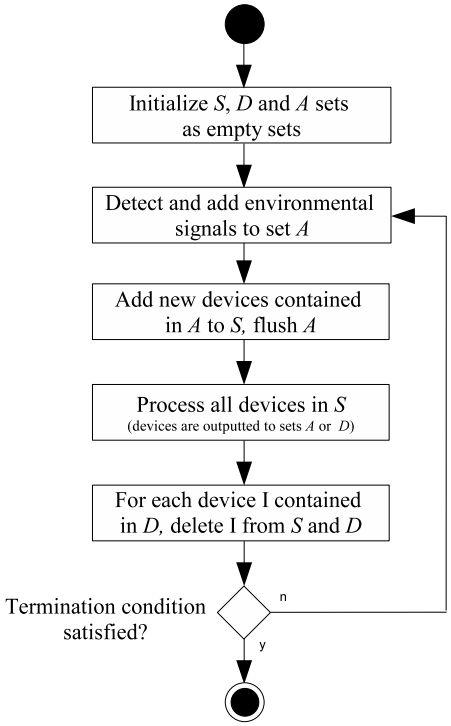


Figure 1: Broacast system flowchart

2. Environmental signals: at this step, input signals (strings of character) given by the environment are added to set $A$. At time $t = 0$, the input signals correspond to the initial set of signals. A *detector* may be built to probe the environment and insert new signals into set $A$.
3. Transferring signals from set $A$ into $S$: signals contained in set $A$ are inserted in set $S$. Set $A$ is then flushed. Each signal inserted in $S$ is processed into broadcast devices (`BDevice` objects), if a signal generates an active broadcast device then this broadcast device is parsed into broadcast units (`BUnit` objects).
4. Process signals in $S$, this step is broken up into two sequential sub-processes:
   (a) we first look for broadcast units of type 4 that are able to broadcast at same time $t$. If those broadcast units can be satisfied by other signals (including themselves) then they broadcast their output signals, the latter are

then inserted in $S$. As those new inserted signals may satisfy other similar type 4 broadcast units, we need to repeat this process until no new signal gets inserted. This step is needed to be performed first because those type 4 broadcast units may output signals that will contribute to other broadcast unit contained in $S$.

(b) Then each broadcast device in $S$ is processed in a sequential order: if a broadcast device $I$ is active then each broadcast unit $I_i$ contained in $I$ may broadcast its output signal upon detecting the adequate signals. A broadcast unit which have already been activated at time $t$ may not broadcast again under any circumstances. Output signals issued by type $1, 2$ and 4 broadcast unit are stored in set $A$. If a type 2 broadcast unit is activated then its output signal is inserted in set $D$. Also if a broadcast device $I$ is a null device and is not a persistent signal, then this device signal is added to set $D$.

5. Delete signals from sets $S$ and $D$: for each signal $I_d$ contained in set $D$, if there is a signal of the form $I_d$ present in $S$ then this signal is removed from $S$. If there are $n$ signals in $S$ that are of the form $I_d$ then only one of those signals is to be deleted. To determine which signal to delete, we assign a probability $p = 1/n$ to each signal and we pick one at random and remove it from $S$. $D$ is then flushed.

6. Termination condition: this condition is set by the user, it may simply be an integer $T$ indicating the maximum number of steps. If the termination condition is not satisfied then go to 1.

# 4 Case studies

## 4.1 Building a NAND gate

In this section we describe the construction of a NAND gate using the broadcast language. This is intended to demonstrate how the broadcast language can be considered as a logical universal computational formalism. Moreover using the Boolean abstraction, it is also possible to build qualitative models of natural networks such as Genetic Regulatory Networks. With the Boolean abstraction, a molecule is considered as a logical expression having two different possible states. One possible state is the ON state meaning that the molecule is present in the environment. When a molecule state is OFF, this indicates that the particular molecule is not present in the environment (cell).

In the remainder of this section we first present a simple example in which a NAND gate is constructed within a static environment (the inputs values do not change over time), then a second example follows in which the same gate is adapted to be used with a dynamic system:

1. We consider a NAND gate having for inputs signals $A$ and $B$ and for output signal $C$. To construct this logical gate with the broadcast language, we first represent each signal $A$, $B$, $C$ as null broadcast devices (substrates): $A = p001$, $B = p010$ and $C = p000$.

   We then declare the following active broadcast devices (enzymes): $I_1 = *p001 : 011$ and $I_2 = *p010 : 100$, these devices emit signaling molecules $S_1 = 011$ and $S_2 = 100$ upon detecting $A$ and $B$ respectively. Similarly, we define $I_3 = * : p001 : 101$ and $I_4 = * : p010 : 110$ which would emit $S_3 = 101$ and $S_4 = 110$ if $A$ or $B$ are not detected.

   Finally the following broadcast devices are employed to output $C$ according to the intermediary states of signaling molecules $S_1$,$S_2$,$S_3$ and $S_4$: $I_5 = *011 : 110 : p000$, $I_6 = *100 : 101 : p000$ and $I_7 = *101 : 110 : p000$. Using these broadcast devices, it is possible to obtain the state of $C$ according

to the states of input signals $A$ and $B$, 2 time steps are necessary to propagate and process the signals $A$ and $B$.

2. Within a dynamic system, some modifications are necessary to maintain our NAND gate. These modifications are intended so as to allow the output signal $C$ to degrade over time. First, the broadcast devices $I_1$,$I_2$ and $I_3$ are modified as follows: As currently defined, those broadcast devices output the signal $p000$ (which designates the persistent signal $C$) when satisfied, these output signals are replaced with an additional signaling molecule $S_5 = 111$. As a result, we obtain the broadcast devices: $I'_5 = *011 : 110 : 111$, $I'_6 = *100 : 101 : 111$ and $I'_7 = *101 : 110 : 111$.

   We then declare a broadcast device $I_8 = *111 : p000$ that upon detecting $S_5$ would emit the output signal $C$. Finally we declare a broadcast device $I_9 = p000 :: p000$ that deletes (degrades) $C$ upon detecting $C$. We note that as soon as a signal $C$ appears at time $t$, it would be removed by $I_9$ at the end of time $t$. To counter balance that effect, we double the concentration of broadcast devices $I_8$ so that the production rate of $C$ is higher than its degradation rate.

In Fig. 2 we present a simulation using such a NAND gate specified with the broadcast language. In this simulation, the inputs $A$ and $B$ are manually switched ON/OFF at different timesteps. We detail the states of the system at timestep 0,1 and 2:

$$\begin{aligned} S(0) = \quad & \{A = p001\ ,\ I_1 = *p001 : 011, \\ & I_2 = *p010 : 100,\ I_3 = * : p001 : 101, \\ & {}^*I_4 = * : p010 : 110,\ I'_5 = *011 : 110 : 111, \\ & I'_6 = *100 : 101 : 111,\ I'_7 = *101 : 110 : 111, \\ & I_8 = *111 : p000,\ I_8 = *111 : p000, \\ & I_9 = p000 :: p000\} \end{aligned}$$

At $t = 0$, the system is initialized with above broadcast devices, $A$ is ON, and both $B$ and $C$ are OFF. We note that both broadcast devices $I_1$ and $I_4$ are satisfied leading to the production of $S_1$ and $S_4$ at $t = 1$:

$$\begin{aligned} S(1) = \quad & \{A = p001\ ,\ I_1 = *p001 : 011, \\ & I_2 = *p010 : 100,\ I_3 = * : p001 : 101, \\ & {}^*I_4 = * : p010 : 110,\ I'_5 = *011 : 110 : 111, \\ & I'_6 = *100 : 101 : 111,\ I'_7 = *101 : 110 : 111, \\ & I_8 = *111 : p000,\ I_8 = *111 : p000, \\ & I_9 = p000 :: p000,\ S_1 = 011, S_4 = 110\} \end{aligned}$$

At $t = 1$, $I'_5$ is activated due to the presence of $S_1$ and $S_4$. $I'_5$ is a type 4 broadcast device that is able to output $S_5$ signal during same timestep. As a result, both $I_8$ broadcast devices are now activated and produce two instances of $C$ molecule at $t = 2$. As $S_1$,$S_4$ and $S_5$ are not persistent, these signals are removed at the end of $t = 1$. However, as $A$ is still ON and $B$ is OFF, $S_1$ and $S_4$ are again produced at $t = 2$.

$$\begin{aligned} S(2) = \quad & \{A = p001\ ,\ I_1 = *p001 : 011, \\ & I_2 = *p010 : 100,\ I_3 = * : p001 : 101, \\ & {}^*I_4 = * : p010 : 110,\ I'_5 = *011 : 110 : 111, \\ & I'_6 = *100 : 101 : 111,\ I'_7 = *101 : 110 : 111, \\ & I_8 = *111 : p000, I_8 = *111 : p000, \\ & I_9 = p000 :: p000,\ C = p0000, \\ & S_1 = 011, S_4 = 110\} \end{aligned}$$

At the beginning of $t = 2$, two instances of $C$ are contained in the system. However the $I_9$ broadcast device is now satisfied by instances of the $C$ molecule resulting in the removal of one instance of $C$.

In this section we present a case study on the use of the broadcast system to represent and study biochemical networks.

## 4.2 Regulatory aspects of cell signaling networks

In this first case study, we present how the broadcast system can be employed to model a signaling pathway where only the regulatory aspects are covered.

One way to represent the regulatory aspects of CSNs is through the use of the Boolean formalism. With the Boolean abstraction, a molecule is considered as a logical expression having two different possible states. One possible state is the *on* state meaning that a protein is present in the environment (the gene coding for this protein is

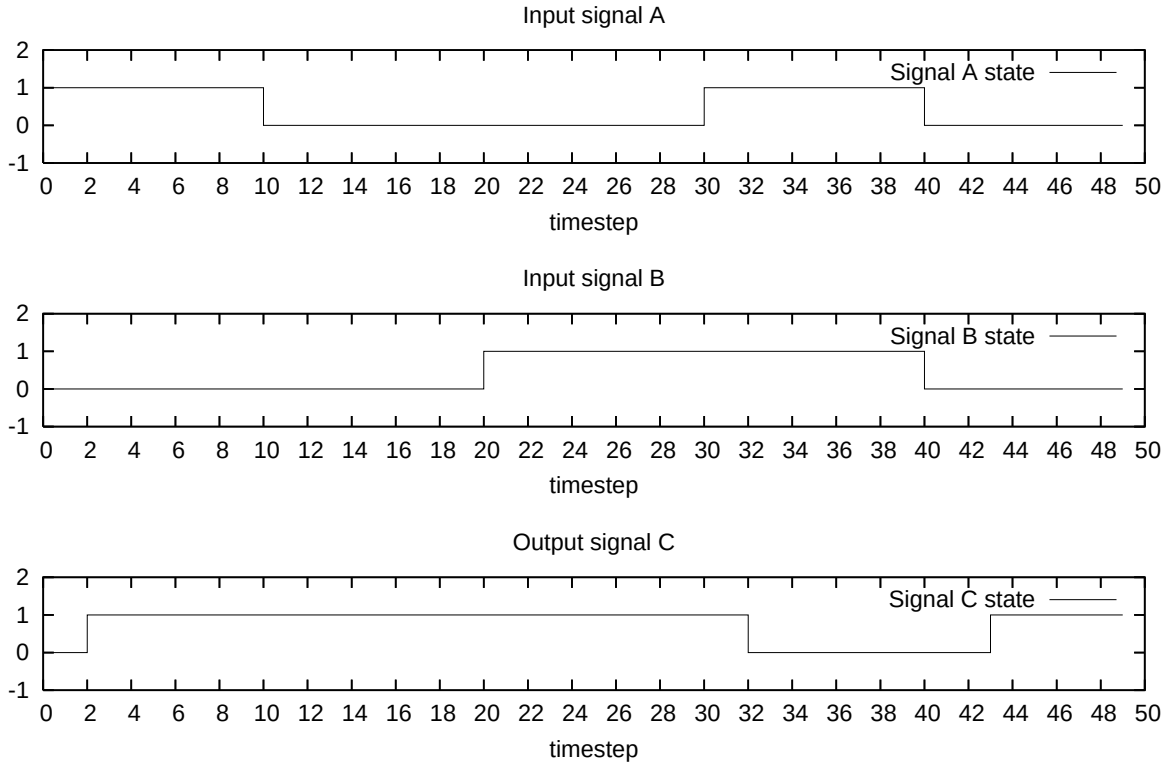


Figure 2: NAND gate specified with our implementation of the broadcast language. The output signal $C$ state is initialized as OFF (0), at timestep 10,20,30 and 40, inputs $A/B$ are manually switched ON/OFF (1/0), We note that the propagation time needed to process the switching of inputs $A$ or $B$ differs according to the nature of the switching involved and present states of $A$,$B$ and $C$.

being expressed). On the contrary, when a protein state is *off*, this indicates that this particular molecule is not present in the milieu.

A simple Boolean network only requires three types of information about the network: the connectivity (node to node), the sign of interaction (inhibitory or excitatory) and the nature of the summation (how do input signals combine to generate output). Figure 3 provides an example of a graphical boolean representation of a signaling pathway.

In this case study, we propose to use the broadcast language to mimic the boolean network of the CSN presented in Figure 3. Thus our broadcast system model will be able to predict the state of output molecules according to the state of input molecules.

To accomplish this, we first need to assign to each molecule (substrate) $PhyA$, $PhyB$, $Eth, \ldots$ a string representation (signal) such as $p0000000$, $p0000001$, $p0000010, \ldots$, see Appendix B for the complete molecule-string mapping. We then provide the broadcast devices (enzymes) responsible for the reactions to occur in this experiment. In this case the broadcast devices stand for the boolean functions shown in Fig.3.

$$(PR1PR5) = (\neg PSI2 \wedge (PhyA \vee PhyB)) \wedge SA$$

The above equation describes the state of $PR1PR$ according to the states of $PSI2$, $PhyA$, $PhyB$ and $SA$. We now present how to represent this Boolean expression using the broadcast language:

In order to represent an `OR` gate that takes for input signals $PhyA$ and $PhyB$ we generate the following broadcast device:

$$I_1 = *p000000\Diamond : 1000000$$

This broadcast device indicates that whenever persistent signals $p0000000$ or $p0000001$ ($PhyA$

or $PhyB$) is detected, the signaling molecule 1000000 is broadcast. This example also demonstrates how to represent *crosstalk* phenomena in the broadcast language. The purpose of using signaling molecules will be shown in the description of the third broadcast device $I_3$.

The `NOT` gate is expressed through the use of type 2 broadcast unit, to represent `NOT` $p0000010$ ($PSI2$), the following broadcast device is employed:

$$I_2 = * : p0000010 : 1000001$$

The above broadcast device stipulates that when no persistent $PSI2$ molecule is present then the signaling molecule 1000001 is broadcast.

Following the given example, we want to express an `AND` gate. The expression (($p0000000$ `OR` $p0000001$) `AND` (`NOT` $p0000010$)) can be translated into the following broadcast device:

$$I_3 = *1000000 : 1000001 : 1000010$$

$I_3$ would broadcast 1000010 only if 1000000 is detected, meaning that either $p0000000$ ($PhyA$) or $p0000001$ ($PhyB$) is on, **and** only if 1000001 is also detected meaning that $p0000010$ ($PSI2$) is not detected.

$$I_4 = *p0000011 : 1000011$$

The broadcast device $I_4$ is used to broadcast a signaling molecule 1000011 if $p0000011$ ($SA$) is detected.

$$I_5 = *1000010 : 1000011 : 1000100$$

$I_5$ is similar to $I_3$ and represent an `AND` gate taking in account the results of $I_3$ and $I_4$. This broadcast device, if satisfied, broadcast a signaling molecule that is employed to activate $PR1PR5$ ($p0000101$), see following broadcast device:

$$I_6 = *1000100 : p0000101$$

See appendix B for the full broadcast system model and for experimental results. We may note that because some broadcast units broadcast at time $t + 1$, a cascade of similar reactions may then take a certain amount of timesteps. This is indeed necessary so that every boolean functions described in the model are actually processed. In the current example, 4 timesteps are necessary to obtain the output states accounting every boolean gates.

# 5 Conclusion and future work

It was demonstrated that the Broadcast Language can model Genetic Regulatory Networks (GRNs). This was due to the ability of the Broadcast Language to mirror Boolean networks which illustrates the wide ranging processing power that Broadcast Systems are capable of. Nevertheless, it was also highlighted that the Broadcast Language is limited regarding the representation and simulation of CSNs. To address this issue, we propose to combine the MCS concept with the Broadcast Language in a new system termed "MCS.b". The MCS.b complements the broadcast language (syntax and semantics) and extends it by including the following refinements:

- Instead of processing all broadcast devices sequentially and deterministically during a time step, the MCS.b processes as follows: at each time step $t$, we pick $n$ pairs of broadcast devices at random. For each pair of devices, one of the broadcast devices is designated (at random) as the *catalyst device* and the second one as the *substrate device*. If the conditional statement of the catalyst device is satisfied by the signal of the substrate device, then the action statement of the catalyst device is executed upon the substrate device.

- $n$ is a constant and designates the number of pairs of broadcast devices that will interact during a timestep. It is also plausible to consider $n$ as the temperature in real chemistry. Temperature has an important role in chemical reactions, indeed molecules at higher temperature have a greater probability to collide with one another. In the

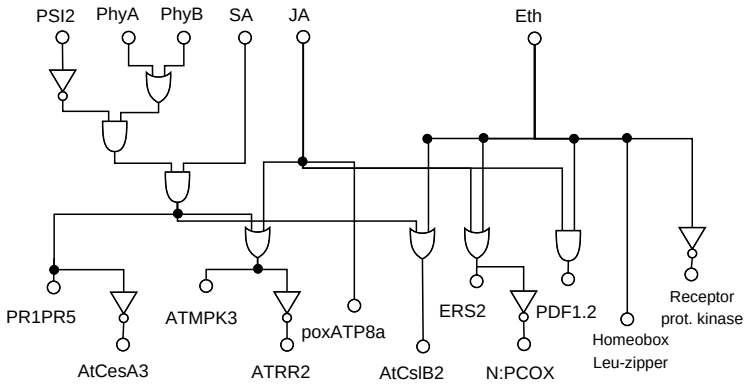


Figure 3: Boolean representation of the signal transduction network controlling the plants defense response against pathogens adapted from [5].

broadcast language "universe", in order to increase the "temperature", one may increment the integer number $n$.

- In the broadcast language specification given by Holland, additional rules were required to resolve some ambiguities raised by the interpretation of broadcast devices. To facilitate this, the MCS.b simplifies the interpretation of broadcast units by preserving broadcast units of type 1 only.
- Similarly the notion of non-persistent devices is removed: by default all devices are considered as persistent molecules.
- As type 3 broadcast units and non-persistent devices no longer exist in this proposal, no molecule can be deleted from the population. However the deletion of molecules is needed to obtain evolutionary pressure. Our suggestion is as follows: each time two molecules react together, we pick a molecule at random and delete it from the population.

By combining the strength of both the MCS and Broadcast Language, we expect the MCS.b to be capable of modeling, simulating and evolving ACSNs in a more fateful manner. At present, we have conducted a number of preliminary experiments examining the spontaneous emergence of collective autocatalytic sets among others. This was expected to be trivial as this phenomenon was already demonstrated with other Artificial Chemistry Systems (such as Tierra, Alchemy, etc.). Initial results suggest that the MCS.b performs as expected, however before these results can be presented to the research community, validation against empirical biological data is required.

# 6 Acknowledgments

This work was funded by ESIGNET (Evolving Cell Signaling Networks in Silico), an European Integrated Project in the EU FP6 NEST Initiative (contract no. 12789).

# References

[1] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[2] J.H. Holland. Exploring the evolution of complexity in signaling networks. *Complexity*, 7(2):34–45, 2001.

[3] J.H. Holland and J.S. Reitman. Cognitive systems based on adaptive algorithms. *ACM SIGART Bulletin*, pages 49–49, 1977.

[4] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[5] Marcela B. Trevino Santa Cruz Thierry Genoud and Jean-Pierre Mtraux. Numeric simulation of plant signaling networks. *Plant Physiology*, 126:1430–1437, August 2001.

# A Source code

FILE *main.cpp*

```
1 /**
2  * \file BUnit.cpp
3  * \author James Decraene
4  */
5
6 /*
7 This file is part of BL.
8
9 BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24 #include "env.h"
25
26 using namespace std;
27
28 int main(int argc, char * argv[]){
29    srand(time(NULL));
30    int runtime=50;
31    string buf;
32    vector<string> myInitialDevices;
33    map<const string, string> myMolecules;
34    ifstream in;
35
36    if (argc==1)
37       in.open("case3.dat");
38    else
39       in.open(argv[1]);
40
41    while (in){
42       getline(in,buf);
43       if (buf[0]!='%' && buf[0]!=' ')
44          myInitialDevices.push_back(buf);
45       if (buf[0]=='%' && buf[1]=='%'){
46          int loc1=buf.find(' ',0)+1,loc2=buf.find(" ",loc1)-loc1;
47          myMolecules.insert(make_pair(buf.substr(loc1,loc2), buf.substr(loc2+loc1+1,buf.size
                ())));
48       }
49    }
50
51    Env myEnv(myInitialDevices, myMolecules);
52    myEnv.run(runtime);
53    //myEnv.printMoleculesConcentration();
54    //myEnv.printPersistentMolecules();
55    myEnv.printMoleculesState();
56    system("pause");
57    return EXIT_SUCCESS;
58 }
```

FILE *env.h*

```
1 /**
2  * \file env.h
3  * \author James Decraene
4  */
5
6 /*
7 This file is part of BL.
8
9 BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24 #ifndef ENV_H
25 #define ENV_H
26
27 #include "BDevice.h"
28
29 /**
30  * An instance of the env class holds the population of broadcast devices
31  * and environmental parameters, in here are also declared function to compute statistics
32  */
33
34 class Env{
35 public:
36    Env(vector<string> initialDevices, map<const string, string> myMolecules);
37    ~Env();
38    void step();
39    int getNumBDevices();
40    int getTimeStep();
41    void run(const int nbSteps);
42    int getConcentration(const string aSignal);
43    void printMoleculesConcentration();
44    void printMoleculesState();
45    void printPersistentMolecules();
46    /** Return the set of environmental signals
47    */
48    vector<string> getEnvSignals();
49
50 private:
51    bool remove(const string aSignal);
52    void doMatches();
53    /** Set of the environment broadcast devices
54    */
55    vector<BDevice> bdevices;
56    /** Set of broadcast devices to be removed at next time step
57    */
58    vector<string> delDevices;
59    /** Set of broadcast devices to be added at next time step
60    */
61    vector<string> addDevices;
62    /** Mapping between molecules (broadcast devices) signals and their name
63    */
64    map<const string, string> molecules;
65    void doPIT4Bunits();
```

```
66      void flushAddDevices();
67      void initDevices();
68      void flushDelDevices();
69      void removeNonPSignals();
70      /** Current Simulation time step
71      */
72      int timeStep;
73      /** Number of time step to be run during simulation
74      */
75      int runtime;
76
77  };
78
79  /** Return the number of broadcast devices held in the environment
80  */
81  inline int Env::getNumBDevices(){
82      return bdevices.size();
83  }
84
85  /** Return current simulation time step.
86  */
87  inline int Env::getTimeStep(){
88      return timeStep;
89  }
90
91  #endif
```

FILE *env.cpp*

```
1  /**
2   * \file env.cpp
3   * \author James Decraene
4   */
5
6  /*
7  This file is part of BL.
8
9  BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24 #include "env.h"
25 /** A constructor
26 * \param initialDevices is a set of broadcast devices to be included at time t=0, these
       initial devices are added in addDevices set
27 * \param myMolecules is a set of pairs [broadcastDeviceSignal / nameOfMolecule] e.g.
       p00000001 PhyA etc
28 */
29 Env::Env(vector<string> initialDevices, map<const string, string> myMolecules){
30    for(int i=0; i<initialDevices.size(); i++)
31       addDevices.push_back(initialDevices[i]);
32    initialDevices.clear();
33    molecules=myMolecules;
34 }
35
36
37 /** A destructor
38 */
39 Env::~Env(){
40 }
41
42 /** Print out concentration of each molecule in myMolecules
43 */
44 void Env::printMoleculesConcentration(){
45    map<string,string>::iterator anIterator;
46    for( anIterator = molecules.begin(); anIterator != molecules.end(); anIterator++ )
47      cout<<"\t"<<getConcentration((*anIterator).first)<<"\t["<<molecules[(*anIterator).first
          ]<<"]"<<endl;
48 }
49
50 /** Print out state (On or Off) of each molecule in myMolecules, a molecule/broadcast device
       is On if its quantity is > 0
51 */
52 void Env::printMoleculesState(){
53    map<string,string>::iterator anIterator;
54    cout<<"Molecules state:"<<endl;
55    if (!molecules.empty())
56       for( anIterator = molecules.begin(); anIterator != molecules.end(); anIterator++ )
57         cout<<"["<<molecules[(*anIterator).first]<<"] "<<(getConcentration((*anIterator).
              first)>0?"on":"off")<<endl;
58    else
59       cout<<"No molecule specified"<<endl;
```

```
60 }
61
62 /** Print out persistent (starting with symbol p) broadcast devices present in the population
63 */
64 void Env::printPersistentMolecules(){
65    cout<<"Persistent molecules:"<<endl;
66    for(int i=0; i<bdevices.size(); i++)
67       if ((bdevices[i].getSignal())[0]=='p')
68          cout<<bdevices[i].getSignal()<<endl;
69 }
70
71 /** Return a set of existing broadcast devices in the simulation
72 */
73 vector<string> Env::getEnvSignals(){
74    vector<string> res;
75    for(int i=0; i<bdevices.size(); i++)
76       res.push_back(bdevices[i].getSignal());
77    return res;
78 }
79
80 /** Process matching for broaccast devices
81 */
82 void Env::doMatches(){
83    vector<string> envSignals=getEnvSignals();
84
85    for(int i=0; i<bdevices.size(); i++)
86       bdevices[i].match(envSignals, addDevices, delDevices);
87 }
88
89 /** Return the concentration of a given broadcast device
90 \param aSignal is the signal of the broadcast device we want the concentration
91 */
92 int Env::getConcentration(const string aSignal){
93    int res=0;
94    for(int i=0; i<bdevices.size(); i++)
95       if (bdevices[i].getSignal()==aSignal)
96         res++;
97    return res;
98 }
99
100 /** Remove broadcast devices that matches a given signal
101 \param aSignal is the signal of the broadcast device(s) we want to remove from the
      environmnet (bdevices set)
102 */
103 bool Env::remove(const string aSignal){
104    bool res=false;
105    for(int i=0; i<bdevices.size(); i++)
106       if (bdevices[i].getSignal()==aSignal){
107         res=true;
108         bdevices.erase(bdevices.begin()+i);
109       }
110    return res;
111 }
112
113 /** Look for type 4 broadcast devices that can react at same time t, if there are any, make
      them react and insert them in current population
114 of broadcast devices, so that they can contribute at current time step.
115 */
116 void Env::doPIT4Bunits(){
117    vector<string> envSignals=getEnvSignals();
118    vector<string> PIT4;
119
120    for(int i=0; i<bdevices.size(); i++){
121       PIT4=bdevices[i].processInstantType4BUnits(envSignals);
122       if (!PIT4.empty()){
123          for(int j=0; j<PIT4.size(); j++)
124             bdevices.push_back(PIT4[j]);
125          i=0;
126       }
```

```
127     }
128 }
129
130 /** Empty the set of broadcast devices contained in addDevices (the set of broadcast devices
        to be added at t+1)
131 */
132 void Env::flushAddDevices(){
133     if(!addDevices.empty()){
134         for(int i=0; i<addDevices.size(); i++)
135             bdevices.push_back(BDevice(addDevices[i]));
136         addDevices.clear();
137     }
138 }
139
140 /** Initialize every broadcast devices in the population
141 */
142 void Env::initDevices(){
143     for(int i=0; i<bdevices.size(); i++)
144         bdevices[i].init();
145 }
146
147 /** Empty the set of broadcast devices contained in delDevices (the set of broadcast devices
        to be removed at t+1)
148 */
149 //to be done: if different persistent signals are matched then remove only one at random
150 void Env::flushDelDevices(){
151     if(!delDevices.empty()){
152         for(int i=0; i<delDevices.size(); i++)
153             remove(delDevices[i]);
154         delDevices.clear();
155     }
156 }
157
158 /** Remove non persistent signals (null broadcast devices that do not begin with symbol p)
        contained in the population (set bdevices
159 These broadcast devices are inserted in set delDevices (to be removed at next time step then)
160 */
161 void Env::removeNonPSignals(){
162     for(int i=0; i<bdevices.size(); i++)
163         if (bdevices[i].getNbUnits()==0 && (bdevices[i].getSignal())[0]!='p')
164             delDevices.push_back(bdevices[i].getSignal());
165 }
166
167 /** Process a simulation time step
168 */
169 void Env::step(){
170     flushAddDevices();
171     initDevices();
172     doPIT4Bunits();
173     doMatches();
174     removeNonPSignals();
175     flushDelDevices();
176     timeStep++;
177 }
178
179 /** Run the simulation
180 \param nbSteps is the number of time step to be run during the simulation
181 */
182 void Env::run(const int nbSteps){
183     runtime=nbSteps;
184     ofstream f;
185     f.open("dynNand.dat");
186
187     cout<<"Running the simulation..."<<endl;
188     for(int i=0; i<runtime; i++){
189         cout<<"step:"<<i<<" "<<endl;
190         //events:
191         if(i==10)
192             delDevices.push_back("p001");
```

```
193
194         if(i==20)
195             bdevices.push_back(BDevice("p010"));
196
197         if(i==30)
198             bdevices.push_back(BDevice("p001"));
199
200         if(i==40){
201             delDevices.push_back("p001");
202             delDevices.push_back("p010");
203         }
204
205
206         step();
207         f<<i<<" "<<getConcentration("p001")<<" "<<getConcentration("p010")<<" "<<
                getConcentration("p000")<<endl;
208     }
209     f.close();
210     system("getPlot.bat");
211 }
```

FILE *BDevice.h*

```
1 /**
2  * \file BDevice.h
3  * \author James Decraene
4  */
5
6 #ifndef BDEVICE_H
7 #define BDEVICE_H
8
9 #include "BUnit.h"
10
11 /** An instance designates a broadcast devices.
12 * A broadcast device may be of any length and may any number of broadcast unit
13 */
14
15 class BDevice{
16 public:
17     BDevice(string newSignal);
18     ~BDevice();
19     string getSignal();
20     vector<string> processInstantType4BUnits(const vector<string>  envSignals);
21     void init();
22     int getNbUnits();
23     void match(const vector<string> envSignals, vector<string> &envAddDevices, vector<string>
           &envDelDevices);
24
25 private:
26     /** Signal of the broadcast device represented as a string
27     */
28     string signal;
29     /** hasReacted is true if the broadcast device has already successfully reacted with
           another broadcast device during a given time step
30     */
31     bool hasReacted;
32     /** nbUnits designates the number of broadcast unit that the broacast device hold, if
           nbUnits = 0 then the broadcast device is null
33     */
34     int nbUnits;
35     /** Set of broadcast units held by the broadcast device
36     */
37     vector<BUnit> bunits;
38     int parseToUnits(const string aSignal);
39 };
40
41 /** Return the number of broadcast units held by the broadcast device
42 */
43     inline int BDevice::getNbUnits(){
44     return nbUnits;
45 }
46
47 /** Return the broadcast device signal
48 */
49 inline string BDevice::getSignal(){
50     return signal;
51 }
52
53 #endif
```

FILE *BDevice.cpp*

```
1  /**
2   * \file BDevice.cpp
3   * \author James Decraene
4   */
5
6  /*
7  This file is part of BL.
8
9  BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24 #include "BDevice.h"
25
26 /** A constructor
27 \param newSignal is the signal (string) from which the broadcast device is built on
28 This signal is parsed into broadcast units if any
29 */
30 BDevice::BDevice(string newSignal){
31     signal=newSignal;
32     nbUnits=parseToUnits(signal);
33 }
34
35 /** A destructor
36 */
37 BDevice::~BDevice(){
38     bunits.clear();
39 }
40
41 /** Initialize every broadcast units held by the broadcast device
42 */
43 void BDevice::init(){
44     for(int i=0; i<bunits.size();i++)
45         bunits[i].init();
46 }
47
48 /** Look for matching signals in envSignals and act according the type of the broadcast
      device
49 \param envSignals is the set of all environmental signals
50 \param envAddDevices is the set of all the broadcast devices to be added at time t+1
51 \param endDelDevices is the set of all the broadcast devices to be removed at time t+1
52 For every broadcast units held by the broadcast device that have not already fired during the
       same time step:
53     according to the type of the broadcast unit, the output of the reaction is added to
         appropriate set (add/del)
54 */
55 void  BDevice::match(const vector<string> envSignals, vector<string> &envAddDevices, vector<
      string> &envDelDevices){
56     bool b=false;
57     string inputSignal1, inputSignal2;
58
59     for(int i=0; i<nbUnits; i++)
60         if (!bunits[i].hasFired()){
```

```
61          switch (bunits[i].getType()){
62             case 1:
63                if ((inputSignal1=bunits[i].findMatchIn(0,envSignals))!="none")
64                   envAddDevices.push_back(bunits[i].reactWith(inputSignal1,""));
65                break;
66             case 2:
67                if ((inputSignal1=bunits[i].findMatchIn(0,envSignals))=="none")
68                   envAddDevices.push_back(bunits[i].reactWith(inputSignal1,""));
69                break;
70             case 3:
71                if ((inputSignal1=bunits[i].findMatchIn(0,envSignals))!="none")
72                   {
73                   envDelDevices.push_back(bunits[i].reactWith(inputSignal1,""));
74                }
75                break;
76             case 4:
77                if ((inputSignal1=bunits[i].findMatchIn(0,envSignals))!="none" && (
                      inputSignal2=bunits[i].findMatchIn(1,envSignals))!="none")
78                   envAddDevices.push_back(bunits[i].reactWith(inputSignal1,inputSignal2));
79                break;
80          }
81       }
82 }
83
84 /** Return a set of broadcast devices, this set contains all broadcast devices resulting from
       broadcast unit of type 4 that are able to react
85  * during the same time step, as these may contribute to other broadcast devices later during
      the same time step
86  */
87 vector<string> BDevice::processInstantType4BUnits(const vector<string> envSignals){
88    vector<string> res;
89    bool b=false;
90    string inputSignal1, inputSignal2;
91
92    for(int i=0; i<nbUnits; i++)
93       if (bunits[i].getType()==4 && !bunits[i].hasFired()){
94          if ((inputSignal1=bunits[i].findMatchIn(0,envSignals))!="none" && (inputSignal1=
                bunits[i].findMatchIn(1,envSignals))!="none"){
95             for(int j=0; j<bunits[i].getI(2).size();j++)
96                if(!isQuoted(bunits[i].getI(2),j) && (bunits[i].getI(2)[j]=='v' || bunits[i].
                      getI(2)[j]=='v' || bunits[i].getI(2)[j]=='^')
97                   || (isQuoted(bunits[i].getI(2),j) && bunits[i].getI(2)[j]=='*'))
98                   b=true;
99             if (!b)
100               res.push_back(bunits[i].reactWith(inputSignal1, inputSignal2));
101         }
102      }
103   return res;
104 }
105
106 /** Return the number of broadcast units created. The latter are parsed from the broadcast
       device signal
107 */
108 int BDevice::parseToUnits(const string aSignal){
109    int res=0,pos1=0,pos2=0;
110    string newBunitSignal;
111
112    while(aSignal.find('*',pos1)!=string::npos){
113       pos1=aSignal.find('*',pos1);
114          if (pos1==0 || pos1!=0 && !isQuoted(aSignal,pos1)){
115             if ((pos2=aSignal.find('*',++pos1))!=string::npos && !isQuoted(aSignal, pos2)){
116                newBunitSignal=aSignal.substr(pos1,pos2-pos1);
117                pos1=pos2;
118             } else {
119                newBunitSignal=aSignal.substr(pos1, aSignal.size());
120                pos1=aSignal.size();
121             }
122          }else pos1++;
123
```

```
124         if (!newBunitSignal.empty()){
125             bunits.push_back(BUnit(newBunitSignal));
126             res++;
127         }
128     }
129     return res;
130 }
```

FILE *BUnit.h*

```
1 /**
2  * \file BUnit.h
3  * \author James Decraene
4  */
5
6 /*
7 This file is part of BL.
8
9 BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24 #ifndef BUNIT_H
25 #define BUNIT_H
26
27 #include <iostream>
28 #include <fstream>
29 #include <sstream>
30 #include <vector>
31 #include <ctime>
32 #include <string>
33 #include <map>
34
35 using namespace std;
36 #include <greta/regexpr2.h>
37 using namespace regex;
38
39 /**
40  * An instance of BUnit designates a broadcast unit which may be hold by a broadcast devices.
41  * A BUnit may hold 2 or 3 arguments according to the type of the broadcast unit (1,2,3 or 4)
        .
42  * The signal  of a broadcast unit is designated as a string.
43  * For matching purpose, regular expressions are used, the regex form of a broadcast unit
       argument is extracted from the compacted form (discarding null symbols of a signals
       argument)
44  */
45 class BUnit{
46 public:
47    BUnit(string newBunitSignal);
48    ~BUnit();
49    int getType();
50    bool match(const int argRegexFormindex, const string aSignal);
51    string getI(const int index);
52    bool hasFired();
53    void init();
54    string findMatchIn(const int argRegexFormindex, const vector<string> envSignals);
55    string reactWith(const string inputSignal1, const string inputSignal2);
56
57 private:
58    /** Type (1,2,3 or 4) of the broadcast unit
59    */
60    int type;
61    /** Set of the broadcast unit's arguments, a type 1,2,3 broadcast unit contains 2
        arguments I[0] and I[1] (I[0]=1 input/ I[1]=1 output)
```

```
62      a type 4 broadcast unit hold 2 inputs: I[0], I[1] and 1 output: I[2]
63      */
64      string I[3];
65      /** Signal of the broadcast unit represented as a string of symbol fron the broadcast
            language alphabet
66      */
67      string signal;
68      /** Compacted form of the signal (for direct translation to regex form)
69      The compacted form discards all null (ignored) symbols
70      */
71      string signalCompactForm;
72      /** Set of regex form of the 1,[2] broadcast units arguments, contains only 1 if broadcast
             unit if of type 1,2 or 3, 2 if type 4
73      */
74      string argRegexForm[2];
75      /** fired is True or Not if the broadcast unit has already fired or not during the
            timestep
76      */
77      bool fired;
78      int setType();
79      string getSymbolValueFromSignals(const char symbol, const string inputSignal1, const
            string  inputSignal2);
80      string getSymbolValueFromSignal(const char symbol, const string inputSignal, const int
            argIndex);
81      string setCompactForm();
82      string setRegexForm(const string arg);
83      void buildArguments();
84  };
85
86  /** Return true if this particular broadcast unit has already fired, otherwise return false
87  */
88  inline bool BUnit::hasFired(){
89      return fired;
90  }
91
92  /** Return the broadcast unit argument (1,2 for type 1,2,3 broadcast devices, 1,2,3 for type
        4 broadcast devices)
93  \param index is the index of the broadcast unit argument
94  */
95  inline string BUnit::getI(const int index){
96      return I[index];
97  }
98
99
100 /** Return the type (integer between 1 and 4) of the broadcast unit,
101 */
102 inline int BUnit::getType(){
103     return type;
104 }
105
106 /** Return True or Not if the given symbol is quoted (preceded by a ') or not
107 \param s is the signal in which the symbol occurs
108 \param index is the index of the symbol we are testing
109 */
110 inline bool isQuoted(const string s, const int index){
111     return (index>0 && (s[index-1]=='\''))?true:false;
112 }
113
114 #endif
```

FILE *BUnit.cpp*

```
1  /**
2   * \file BUnit.cpp
3   * \author James Decraene
4   */
5
6  /*
7  This file is part of BL.
8
9  BL is free software; you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation; either version 2 of the License, or
12 (at your option) any later version.
13
14 Foobar is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with Foobar; if not, write to the Free Software
21 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
22 */
23
24  #include "BUnit.h"
25 /** Construct a broadcast unit
26 \param newBunitSignal is the string aka signal of the broadcast unit
27 */
28 BUnit::BUnit(string newBunitSignal){
29     signal=newBunitSignal;
30     buildArguments();
31     type=setType();
32     if(type!=0){
33         signalCompactForm=setCompactForm();
34         for(int i=0; i<(type==4?2:1); i++)
35             argRegexForm[i]=setRegexForm(I[i]);
36     }
37 }
38
39 /** Initialize the broadcast unit
40 */
41 void BUnit::init(){
42     fired=false;
43 }
44
45 /** Return the interpreted value of a broacast unit symbol
46 \param symbol designates the symbol for which we want the interpretation/value given the
       input signals
47 \param inputSignal is the current processed signal
48 \param argIndex is the index of the current broadcast units argument processed
49 */
50 string BUnit::getSymbolValueFromSignal(const char symbol, const string inputSignal, const int
       argIndex){
51     string res;
52     int wildcardEdge=0,pos;
53
54     if ((pos=I[argIndex].find(symbol))!=string::npos){
55         //is there a v or V symbol located at the beginning (designating a prefix: -1) or at
               the end (designating a suffix: 1) of the signal
56         if (I[argIndex][0]=='v' || I[argIndex][0]=='V')
57             wildcardEdge=-1;
58         else if (I[argIndex][I[0].size()]=='v' || I[argIndex][I[0].size()]=='V')
59             wildcardEdge=1;
60
61         if(symbol=='^')
```

```
62          res=(wildcardEdge==-1)?inputSignal[inputSignal.size()-I[argIndex].size()+pos]:
                inputSignal[pos];
63
64      if(symbol=='v' || symbol=='V' && wildcardEdge!=0)
65          res=(wildcardEdge==-1)?inputSignal.substr(0 ,inputSignal.size()-I[argIndex].size()
                +1):inputSignal.substr(I[argIndex].size()-1, inputSignal.size()-1);
66    }
67    return res;
68 }
69
70 /** Return the value of an interpreted symbol to be concatenated in the output signal, next
       this function calls getSymbolValueFromSignal for each input signal
71 \param symbol designates the symbol for which we want the interpretation/value given the
       input signals
72 \param inputSignal1 is the signal of the first input signal which is necessary for all types
       of broacast units
73 \param inputSignal2 is the second optionnal input signal (for type 4 broadcast units only)
74 */
75 string BUnit::getSymbolValueFromSignals(const char symbol, const string inputSignal1, const
       string  inputSignal2=""){
76    string res;
77    res+=getSymbolValueFromSignal(symbol, inputSignal1, 0);
78
79    if (!inputSignal2.empty())
80       res+=getSymbolValueFromSignal(symbol, inputSignal2,1);
81
82    return res;
83 }
84
85 /** Return the product from the reaction between the broadcast unit and input signal(s), 2
       input signals are taken with type 4 broadcast units
86 \param inputSignal1 is the signal of the first input signal, this regards all type of
       broadcast units
87 \param inputSignal2 is the signal of second input signal used with type 4 broadcast units
88 */
89 string BUnit::reactWith(const string inputSignal1, const string inputSignal2=""){
90    string res;
91    fired=true;
92    map<const char, string> symbolsValue;
93
94    symbolsValue['0'] = "0";
95    symbolsValue['1'] = "1";
96    symbolsValue['p'] = "p";
97    symbolsValue['v'] = getSymbolValueFromSignals('v', inputSignal1, inputSignal2);
98    symbolsValue['V'] = getSymbolValueFromSignals('V', inputSignal1, inputSignal2);
99    symbolsValue['^'] = getSymbolValueFromSignals('^', inputSignal1, inputSignal2);
100
101   int j=type==4?2:1;
102   for(int i=0; i<I[j].size();i++)
103      if(isQuoted(I[j],i))
104         res+=I[j][i];
105      else
106         res+=symbolsValue[I[j][i]];
107
108   return res;
109 }
110
111 /** Return "none" if the broadcast unit has not find any matches in the given set of signals
112 otherwise return the signal of a matched signal
113 \param argRegexFormindex is the index of the Broadcast unit's argument regex form (1 for and
       type 1,2,3 b. devices , and may also be 2 with type 4 broadcast devices)
114 \param envSignals is a set of environmental signals in which this broadcast unit will be
       looking for a match
115 */
116 string BUnit::findMatchIn(const int argRegexFormindex, const vector<string> envSignals){
117    vector<string> matches;
118
119    for(int i=0; i<envSignals.size(); i++)
120       if (match(argRegexFormindex, envSignals[i]))
```

```
121             matches.push_back(envSignals[i]);
122
123     if (matches.empty())
124         return "none";
125     else return matches[rand()%matches.size()];
126 }
127
128
129 /** Return the regex form of the compacted form of the broadcast unit signal
130 */
131 string BUnit::setRegexForm(const string arg){
132     string res;
133     for(int i=0; i<arg.size();i++){
134         if (!isQuoted(arg,i) && arg[i]=='v' || arg[i]=='V')
135             res+=".+";
136         else{
137             if(i==0)res+="^";
138
139             if (arg[i]=='0' ||  arg[i]=='1' || arg[i]=='p')
140                 res+=arg[i];
141
142             if (isQuoted(arg,i)){
143                 if (arg[i]=='^' || arg[i]=='*')
144                     res+="\\";
145                 res+=arg[i];
146             }
147
148             if (!isQuoted(arg,i) && (arg[i]=='^' || arg[i]=='x'))
149                 res+=".";
150
151             if(i==arg.size()-1)res+="$";
152         }
153     }
154     return res;
155 }
156
157 /** return true if aSignal matches the Bunits conditions
158 \param argRegexFormindex is the index of the Broadcast unit's argument regex form (1 for and
        type 1,2,3 b. devices , and may also be 2 with type 4 broadcast devices)
159 \param aSignal is the given signal the broadcast unit is trying to match/bind to.
160 */
161 bool BUnit::match(const int argRegexFormindex,const string aSignal){
162     match_results results;
163     rpattern p(argRegexForm[argRegexFormindex], NOCASE);
164
165     match_results::backref_type br = p.match(aSignal, results);
166     if (br.matched)
167         return true;
168     else
169         return false;
170 }
171
172 /** Parse the broadcast unit signal into 2/3 arguments (2 for type 1,2,3 broadcast devices, 3
        for type 4 broadcast devices)
173 */
174 void BUnit::buildArguments(){
175     for(int i=0,j=0; j<3 && i<signal.size(); i++)
176         if (signal[i]==':' && !isQuoted(signal,i))
177             j++;
178         else
179             I[j].push_back(signal[i]);
180 }
181
182 /** Compute the type (1,2,3 or 4) of the broadcast unit, this is done according to the states
        (empty or not) of the 3 possible broadcast unit arguments
183 */
184 int BUnit::setType(){
185     if (!I[0].empty() && !I[1].empty() && I[2].empty())
186         return 1;
```

```
187     else if(I[0].empty() && !I[1].empty() && !I[2].empty()){
188        I[0]=I[1];
189        I[1]=I[2];
190        I[2].clear();
191        return 2;
192     }else if(!I[0].empty() && I[1].empty() && !I[2].empty()){
193        I[1]=I[2];
194        I[2].clear();
195        return 3;
196     }else if(!I[0].empty() && !I[1].empty() && !I[2].empty())
197        return 4;
198     else
199        return 0;
200 }
201
202 /** Return the compacted form (discarding null symbols) of the broadcast unit signal, this
       string is used later to carry
203 out a direct mapping to regex form
204 */
205 string BUnit::setCompactForm(){
206     string res,arg;
207     int i,j,k,pos;
208     bool f=false;
209
210     for(j=0; j<(type==4?2:1); j++)
211        for(i=0; i<I[j].size(); i++)
212           if (!isQuoted(I[j],i) && (I[j][i]=='v' || I[j][i]=='V') && i!=0 && i!=I[j].size()-1)
213              I[j].erase(i--,1);
214
215     for(j=0; j<(type==4?2:1); j++)
216        if ((I[j][0]=='v' || I[j][0]=='V') && !isQuoted(I[j],I[j].size()-1) && (I[j][I[j].size
             ()-1]=='v' || I[j][I[j].size()-1]=='V'))
217           I[j].erase(I[j].size()-1,1);
218
219     if (type==4)
220        for(i=0; i<I[1].size();i++)
221           if (!isQuoted(I[1],i) && (I[1][i]=='v' || I[1][i]=='V') && (pos=I[0].find(I[1][i]))
                !=string::npos  && !isQuoted(I[0],pos))
222              I[1].erase(i--,1);
223
224     for(j=0; j<(type==4?2:1); j++)
225        for(i=0; i<I[j].size(); i++)
226           if(!isQuoted(I[j],i) && I[j][i]=='^')
227              if(!f) f=true;
228                 else
229              I[j].erase(i--,1);
230
231     k=type==4?2:1;
232     arg=(type==4)?I[0]+I[1]:I[0];
233
234     for(i=0; i<I[k].size(); i++)
235        if (!isQuoted(I[k],i) && (I[k][i]=='v' || I[k][i]=='V' || I[k][i]=='^') && ((pos=arg.
             find(I[k][i]))==string::npos  || (pos!=string::npos && isQuoted(arg,pos))))
236           I[k].erase(i--,1);
237
238     for (i=0; i<3;i++)
239        res+=(i==0 || i==2 && I[i].empty())?I[i]:':'+I[i];
240
241     return res;
242 }
243
244 /** A destructor
245 */
246 BUnit::~BUnit(){
247 }
```

# B Case studies

FILE *case1.dat, "Modeling a simple biochemical network"*

```
1 %-----------------------------------------------------
2 %Mapping table of molecules:
3
4 %signal      molecule
5
6 %% p0000000 PhyA
7 %% p0000001 PhyB
8 %% p0000010 PSI2
9 %% p0000011 SA
10 %% p0000100 JA
11 %% p0000101 PR1PR5
12 %% p0000111 ATMPK3
13 %% p0001000 AtCesa3
14 %% p0001001 ERS2
15 %% p0001010 N:PCOX
16 %% p0001011 ETH
17 %% p0001100 poxATP8a
18 %% p0001101 ATRR2
19 %% p0001111 ACslB2
20 %% p0010000 PDF1.2
21 %% p0010001 HomLeuZip
22 %% p0010010 RePrkinase
23
24 %-----------------------------------------------------
25 %initial molecule(s):
26
27 %PhyA
28 p0000000
29
30 %PhyB
31 p0000001
32
33 %PSI2
34 p0000010
35
36 %SA
37 p0000011
38
39 %JA
40 p0000100
41
42 %ETH
43 p0001011
44
45 %-----------------------------------------------------
46 %rules:
47 %if PhyA or PhyB then 1000000
48 *p000000x:1000000
49
50 %if not PSI3 then 1000001
51 *:p0000010:1000001
52
53 %if (PhyA or PhyB) and not PSI3 then 1000011
54 *1000000:1000001:1000010
55
56 %if SA then 1000100
57 *p0000011:1000011
58
59 %if ((PhyA or PhyB) and not PSI3) and SA then 1000101
60 *1000010:1000011:1000100
61
62 %if (((PhyA or PhyB) and not PSI3) and SA) then PR1PR5
```

```
63 *1000100:p0000101
64
65 %if (((PhyA or PhyB) and not PSI3) and SA ) then ACslB2
66 *1000100:p0001111
67
68 %if (((PhyA or PhyB) and not PSI3) and SA) then ATMPK3
69 *1000100:p0000111
70
71 %if JA then ATMPK3
72 *p0000100:p0000111
73
74 %if JA then ERS2
75 *p0000100:p0001001
76
77 %if JA then poxATP8a
78 *p0000100:p0001100
79
80 %if Eth then ACslB2
81 *p0001011:p0001111
82
83 %if Eth then ERS2
84 *p0001011:p0001001
85
86 %if Eth and JA then PDF1.2
87 *p0001011:p0000100:p0010000
88
89 %if Eth then Homeobox leu zipper
90 *p0001011:p0010001
91
92 %if not Eth then Receptor prot. kinase
93 *:p0001011:p0010010
94
95 %if not JA then 1000111
96 *:p0000100:1000111
97
98 %if not Eth then 1001000
99 *:p0001011:1001000
100
101 %if not Eth and not JA then N:PCOX
102 *1000111:1001000:p0001010
103
104 %ATRR2 is on by default
105 p0001101
106
107 %if not JA then remove ATRR2
108 *p0000100::p0001101
109
110 %if ((PhyA or PhyB) and not PSI3) and SA) then  remove ATRR2
111 *1000100::p0001101
112
113 %AtCesAa3 is on by default
114 p0001000
115
116 %if ((PhyA or PhyB) and not PSI3) and SA) then  remove AtCesAa3
117 *1000100::p0001000
```

 FILE *results.dat, "Series of results from Case study 1"*

```
Results:

--TEST 1--------------------------------
Inputs:

[PhyA] on
[PhyB] off
[PSI2] off
[SA] on
[JA] off
[ETH] on

Outputs:

[PR1PR5] on
[ATMPK3] on
[AtCesa3] off
[ERS2] on
[N:PCOX] off
[poxATP8a] off
[ATRR2] off
[ACslB2] on
[PDF1.2] off
[HomeoboxLeuZipper] on
[ReceptorProt.kinase] off

--TEST 2--------------------------------
Inputs:

[PhyA] off
[PhyB] on
[PSI2] off
[SA] off
[JA] on
[ETH] off

Outputs:

[PR1PR5] off
[ATMPK3] on
[AtCesa3] on
[ERS2] on
[N:PCOX] off
[poxATP8a] on
[ATRR2] off
[ACslB2] off
[PDF1.2] off
[HomeoboxLeuZipper] off
[ReceptorProt.kinase] on
```

FILE *case2.dat, 'Realizing a NAND gate-example I'*

```
1 %% p000 output
2
3 %input1
4 p001
5
6 %input2
7 p010
8
9 *p001:011
10 *p010:100
11
12 *:p001:101
13 *:p010:110
14
15 *011:110:p000
16 *100:101:p000
17 *101:110:p000
```

FILE *case3.dat, 'Realizing a NAND gate-example II'*

```
%% p000 output

%input1
 p001

%input2
%p010

*p001:011
*p010:100

*:p001:101
*:p010:110

*011:110:111
*100:101:111
*101:110:111

*111:p000;
*111:p000;

*p000::p000
```

# C Installing BC

FILE *readMe.txt*

```
1 ####BC was compiled using the following:
2
3 -Dev-C++ 4.9.9.2 IDE: http://www.bloodshed.net/devcpp.html
4 -G++ 3.3.3 (cygwin)
5 -Greta regex:http://research.microsoft.com/projects/greta/
6 -windows xp
7
8 ####To compile BC:
9
10 -It is suggested to use Dev-C++ (open "bc.dev")
11 -The GRETA library is necessary, the devpak (to be used in conjunction with dev-c++ can be
       found here: http://devpaks.org/details.php?devpak=10, parameters for linker: -lgreta -
       lmsvcp60
12
13
14 ####To run BC:
15
16 -BC is a console based application
17 -BC may take for argument the file containing the definition of the broadcast devices, e.g. "
       bc.exe case1.dat"
18 -Examples are presented in Appendix B of the report ALL-06-01 ("case1.dat" and "case2.dat")
19
20 ####Notes on how to define broadcast devices:
21
22 -The following syntax is used to define broadcast devices (in a text file, e.g. "case1.dat"):
23 -"%% p0000000 PhyA": When %% occurs at the beginning, this indicates that we desire a mapping
       between the signal string and the name of the broadcast device/molecule (this can be
       used by the program to output information about molecules
24 -"%PhyA": A single % is used to comment
25 -"p00000000": The system adds the broadcast devices "p000000" in the initial set of broadcast
       devices
```

FILE *Makefile.win*

```
# Project: bc
# Makefile created by Dev-C++ 4.9.9.2

CPP  = g++.exe -D__DEBUG__
CC   = gcc.exe -D__DEBUG__
WINDRES = windres.exe
RES  =
OBJ  = main.o BDevice.o env.o BUnit.o $(RES)
LINKOBJ  = main.o BDevice.o env.o BUnit.o $(RES)
LIBS =  -L"lib" -lgreta -lmsvcp60  -lgmon -pg  -g3
INCS =  -I"include"
CXXINCS =  -I"lib/gcc/mingw32/3.4.2/include"  -I"include/c++/3.4.2/backward"  -I"include/c++/3.4.2/mingw32"  -I"include/c++/3.4.2"  -I"include"  -I"C:/Dev-Cpp/include/greta"
BIN  = bc.exe
CXXFLAGS = $(CXXINCS)   -pg -g3
CFLAGS = $(INCS)   -pg -g3
RM = rm -f

.PHONY: all all-before all-after clean clean-custom

all: all-before bc.exe all-after


clean: clean-custom
	${RM} $(OBJ) $(BIN)

$(BIN): $(OBJ)
	$(CPP) $(LINKOBJ) -o "bc.exe" $(LIBS)

main.o: main.cpp
	$(CPP) -c main.cpp -o main.o $(CXXFLAGS)

BDevice.o: BDevice.cpp
	$(CPP) -c BDevice.cpp -o BDevice.o $(CXXFLAGS)

env.o: env.cpp
	$(CPP) -c env.cpp -o env.o $(CXXFLAGS)

BUnit.o: BUnit.cpp
	$(CPP) -c BUnit.cpp -o BUnit.o $(CXXFLAGS)
```