Recombinant Programming

Renaud Pawlak, Carlos Cuesta, Houman Younessi

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Recombinant Programming

Renaud Pawlak
Carlos Cuesta
Houman Younessi

N° 5380

Novembre 2004

Thème COM

ISSN 0249-6399

Recombinant Programming

Renaud Pawlak[1], Carlos Cuesta[2], Houman Younessi[3]

Thème COM – Systèmes communicants
Projet JACQARD

Rapport de recherche n° 5380 – Novembre 2004 - 44 pages

Abstract: This research report presents a promising new approach to computation called Recombinant Programming. The novelty of our approach is that it separates the program into two layers of computation: the recombination and the interpretation layer. The recombination layer takes sequences as inputs and allows the programmer to recombine these sequences through the definition of cohesive code units called extensions. The output of such recombination is a mesh that can be used by the interpretation layer in many different ways, depending on the context. To further illustrate our model, we present a language called Grapple that supports Recombinant Programming and show possible applications of this language. In particular, we present RJava, the prototype of a recombinant language that uses Grapple in the context of Java-like programs.

Keywords: language, computing, Recombinant Programming, verification and validation

[1] INRIA Futurs, Projet Jacquard, LIFL – renaud.pawlak@inria.fr
[2] Univ. Valladolid, Spain – cecuesta@infor.uva.es
[3] RPI, Hartford, USA – hyounessi@cox.net

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des vignes
4 rue Jacques Monod - Bât G -91893 Orsay Cedex (France)
Téléphone : +33 1 72 92 59 00— Télécopie : +33 1 60 19 66 08

La programmation Recombinante

Résumé: Ce rapport de recherche présente un nouveau modèle programmation qualifié de recombinant. La nouveauté de cette approche consiste à séparer le programme en deux niveaux de calcul: un niveau de recombinaison et un niveau d'interprétation. Le niveau de recombinaison prend en entrée des séquences et permet au programmeur de recombiner ces séquences via la définition d'unités de code modulaires et cohérentes appelées extensions. La sortie de cette recombinaison peut être utilisé par le niveau d'interprétation de multiples façons, suivant le contexte. Afin d'illustrer notre modèle, nous présentons un langage appelé Grapple qui supporte la programmation recombinante et nous montrons un certain nombre d'applications de ce langage. En particulier, nous présentons RJava, le prototype d'un langage de programmation recombinant qui utilise Grapple dans le contexte de programmes à la Java.

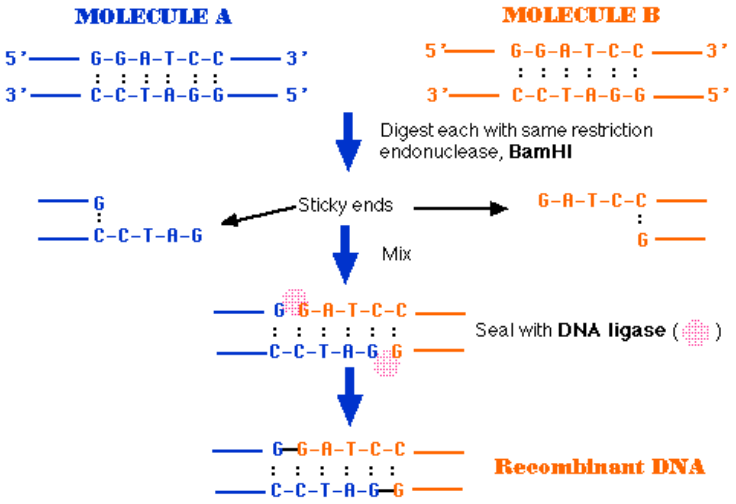
Mots clés: langage, modèle de programmation, programmation recombinante, vérification et validation

1 Introduction to Recombinant Programming: An Analogy

The Recombinant Programming (RP) model is inspired from biology and the Recombinant DNA technology. As such, it introduces a new programming style that differs from the classical functional, procedural, or OO styles. One of the main interests of the model is to provide a framework to Validation and Verification of the programs.

1.1 What Is Recombinant DNA?

As everybody knows, the DNA is the complex molecule formed out of two complementary helicoidal strands that encode the genetic information of living organisms. The genetic information defines ***sequences*** of codons that are used by the cells to synthetize proteins. The codons are encoded by smaller units of information called ***bases***, which are held by the nucleotites in DNA's double helix. One codon corresponds to three consecutive bases. These bases code information other than protein data, such as signalization information that can for example enable or disable the construction of a given protein. As such, the four bases (ACGT) can be seen as the four elementary words of a language called the DNA language.
The chemical process that produces proteins out of the DNA is complex and involves several actors. However, as an approximation, we can see cells as interpreters of specific programs coded in DNA language. The results of their interpretations are proteins. As a consequence, a strong analogy can be made between protein synthesis and computation, as practiced by computer science in general. To simplify, the abstract computational model consists of following the DNA sequence from a starting point to an end point and of constructing an image of this sequence using the protein language.
A ***recombinant*** DNA (rDNA) is a DNA strand that has been created through the recombination of several original DNA strands. The rDNA technology is an artificial technique invented by genetic biologists to create new genetic materials out of existing ones. In a nutshell, the chemical process of recombination involves enzymes (restriction endonuclei) that react with specific sequences of bases in the DNA strands. When one of these sequences, called ***recognition sequence***, reacts with an enzyme, it breaks the DNA double helix, and leaves a pending strand at the end. Two strands coming from different materials may have complementary information on their pending strands. These strands tend to stick together (they are called sticky ends) and the use of another enzyme called a ligase can ***recombine*** the two DNA strands into one rDNA.



From a purely computational perspective, the interesting point of the technique is that it allows the creation of new programs, out of existing ones. Due to the complexity of the process, the actual genetics manipulations are quite limited in practice. They most of the time consist of adding new genes (protein sequences) to an existing DNA. However, if we abstract this model and remove the inherent chemical limitations, it is basically possible to manipulate DNA structures in order to create totally new kinds of programs. The cells can then be used as generic interpreters of these new programs.

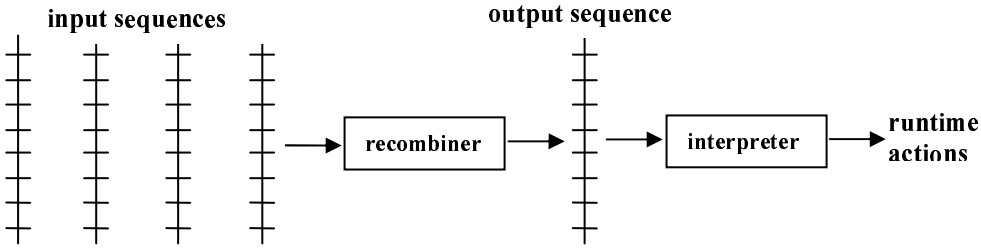
1.2 From Recombinant DNA to Recombinant Programming

The concrete rDNA model presented above can be generalized and abstracted to an actual computation model that we called the Recombinant Programming (RP) model. The basic idea of this computational model is to provide, like rDNA, two levels (or layers) of computation:

1) a sequence interpretation level
2) a sequence recombination level

At the ***sequence interpretation level***, the computation simply consists of interpreting sequences of bases with a simple ***interpreter***. The natural behavior of such an interpreter would be similar to what a living cell does when synthetising proteins. The interpreter would start at a given point of the sequence (most likely at the beginning) and would move forward to the end, interpreting the bases one at a time. Each base can be a word of a language that is linked to a specific action. The semantics of the language can be expressed through an operational semantics. However, this behavior is not mandatory and other kinds of sequence usage can be defined at interpretation level, depending on the context.

At the ***sequence recombination level***, the computation consists of creating new sequences of bases in order to obtain new effects that can be interpreted later on by the interpretation level. This recombination should be ensured by a specific unit, which we call a ***recombiner***.



It is important to note that the manipulated structures are sequences but that sequences are also one particular path in an oriented mesh, which makes it possible to generalize the approach on more complex structures, if needed. Besides, sequences are very commonly encountered in real programs both from a structural and a behavioral perspective. For the structural side, a list, for instance, is a sequence of atoms. In that case a recombiner may be used to sort a list, and a possible interpreter could be a printing program. This way of seeing recombinant computation we call ***Structural Recombinant Computation (SRP)***.

The behavioral side is however more interesting. In that perspective, we come from the point that any execution flow of instructions in a program can be seen as a sequence at runtime. In that context, the recombiner can be used to create computations out of ***primary sequences*** of instructions. The output sequence being seen as the execution flow of the final program. This way of seeing recombinant computation we call ***Behavioral Recombinant Computation (BRP)***.

At this point, we want to emphasize that not all forms of RP are necessarily interesting. An actual model should be defined in order to make it useful for a given context. We have chosen to present an RP model that we defined with the intention to solve challenging problems that we currently encounter in more classical computational models, such as property verification problems. However, we are quite convinced that other recombinant models may be defined to solve other types of problems, such as optimization or constraints solving.

A eneric Recombinant om utation odel

In this section, we present a generic RP model that defines more precisely the basic entities and concepts that are needed for recombinant computation. A real computation model for software construction goes beyond the rDNA analogy and even though the previously mentioned analogies remain, the reader should not try to map all the concepts presented here to rDNA.

2.1 A Two-Layer Architecture

Within a generic computation model, we need a supplementary entity in order to define the recombination parameterization; in other words, how the input sequences are going to be recombined by the recombiner. Indeed, good models for software engineering should ensure separation of concerns and it would not be right to mix this information neither within the input sequences, nor with the recombiner.

This choice of distinguishing the sequence information from the recombination information definitely has important consequences on the way the programs will be built later on. It implies an asymmetric two-layer structure. Note that we focus only on the recombination layer. The interpretation layer can also be architected in similar ways but we do not consider this problem in this report.

Two-layer architectures are a very classic way of architecting programs and systems that we can find in very popular approaches such as reflective approaches and, more recently, Aspect-Oriented Programming. In these kinds of architectures, programs are not symmetric since they contain a base level, which usually describes the basic functionalities of the programs, and one or several meta levels (or aspects), which describe some additional functions that can be defined in a generic way. The interesting point is that the meta-levels have a global impact on the base level. This property helps the programmer to separate concerns that are inherently cross-cutting the base level (hard to modularize). It is important to note that this way of doing things implies that the meta-levels should not completely break the base level but only ***extend*** it. Otherwise, it would be difficult to control the evolution of the program and to compose several meta-levels with each others.

In our approach, the meta-level entities will define how the sequences of bases are recombined. We have chosen to call these entities ***extensions***, in order to reflect the fact that, in a typical program, they should only affect the sequences by extending them. In order to understand how the extensions work, we need to define the recombination process.

2.2 Recombination Basics

The goal if this section is to explain the recombination process and the involved concepts in an intuitive manner and with a gradually growing complexity. We progressively move away from the rDNA analogy and of its limitations in order to define a more generic and flexible model that will fit our needs in terms of computation.

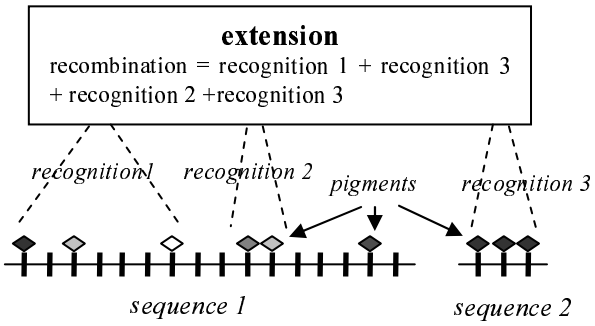
In the rDNA technology, the breaking and recombination of DNA strands occur on particular points because the used enzymes react with specific recognition sequences. These recognition

sequences are well-defined sequences of bases. Thus, within a recognition sequence, bases are not used to encode proteins. They are used for signalizing purposes. Mother Nature mixes the ***encoding data*** (for building proteins) and the ***signalizing data*** (for controlling other parameter in the protein construction or in the DNA recombination) because she has to work with limited capabilities: a four-word language and no ways for attaching any additional information to these bases. Since we are not limited here, it is quite natural to separate more cleanly the encoding data from the signalizing data. Consequently, we will have the following definitions.

Definitions:

A ***sequence*** is a consecutive set of bases.
A ***base*** is a sequence item that encodes a functional element of the sequence. A base can hold several pigments.
A ***pigment*** is a named data item that is attached to a base and is used as signalization information by the recombination process. A set of pigments attached to a set of bases in order to form a consistent recognition pattern is called a ***color***.
A ***recognition sequence*** is any sequence of bases that holds a *color*, which is recognized, extracted, and used during the recombination process. A recognition sequence defines a ***recognition context***, i.e. a set of symbol/value bindings, which are the pigments names/values that are defined for this recognition sequence.
A ***colored sequence*** is a sequence that contains several pigments defining one or several recognition sequences.
A ***recognition expression*** is an expression using some pigment names, which matches a set of recognition sequences within the recognition context. A recognition expression is not a first-order logic predicate since the recognition context defines symbols that values of which may change depending on the sequence's location (a pigment p may have a value v_1 on a base b_1 but a value v_2 on a base b_2). We will describe in further details the recognition expressions later in this report.

Using this model, an extension can be depicted as an entity that defines a set of recognition expressions in order to extract a set of recognition sequences from some original sequence. The extension then defines how these recognition sequences should be recombined to form the output sequence. Finally, the recombiner takes a set of colored (pigmented) sequences and extensions as an input and produces a set of output sequences by recombining the input sequences as defined in the extensions.



2.3 An Incremental Recombination Model

The model presented before is generic enough to deal with any kind of recombination. However, it presents the drawback of having all the recombined sequences at the same degree of importance. Moreover, within the output sequence, all the recombined information is mixed together

in a linear way so that some information on the original sequences is lost somehow. In an ideal two-layer architecture, we want the extensions to be able to *incrementally extend* a given primary sequence. If possible, the original sequence should remain intact so that the extensions usually define alternate sequences (that can possibly completely replace the original sequence, but this is not always the case). A model based on these principles would allow easier programming of complex system because of the leverage of the different concerns of the system, each concern being implemented by an extension, but also because the main functional concern would guide the overall behavior of the system, providing a reliable frame on which the extensions would be plugged in. This vision is similar to the vision adopted by Aspect Oriented Programming (AOP). The context of RP is however different. While AOP deals with weaving of crosscutting concerns, RP deals with the recombination of sequences. Furthermore, AOP is an architectural model that helps the structuration of the programs, while RP is a computation model.

2.3.1 Definitions

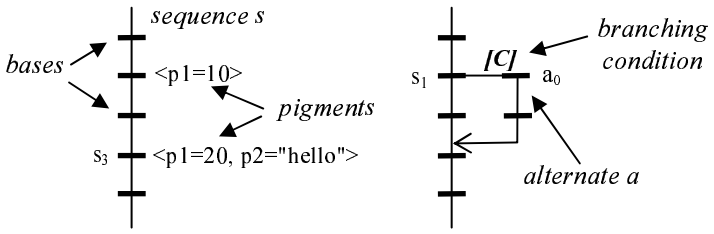
In order to allow incremental extension through recombination, our idea is to define ***branches*** that would define sets of ***alternate sequences*** (***alternates*** for short) relatively to ***primary sequences***. The recombiner then produces as an output result, not a single linear sequence, but a branching mesh. At interpretation time and for one context, this mesh gives only one interpreted recombined sequence, which corresponds to one path in the mesh that is deduced by evaluating ***branching conditions***. This implies that the interpreter defines some ***interpretation context***. Here follow some important definitions.

An ***alternate*** is a sequence that defines an alternate subset of consecutive bases on a primary sequence or on another alternate. Let us call *modified sequence* a primary sequence or an already installed alternate.
A ***branching point*** is the base in the modified sequence from which the alternate starts.
A ***joining point*** is the base in the modified sequence to which the alternate *joins*.
An ***interpretation context*** is a set of symbol/value bindings that defines the values of the expressions that are used at interpretation time.
A ***branching condition*** is a first-order logic predicate that is evaluated by the interpreter in order to know if the interpretation path should branch into a given alternate or not. At a branching point, the interpretation will test all the branching conditions until one is true in the interpretation context (that defines the variable values used in the branching condition). If one branching condition is true, the interpretation path will branch into the corresponding branch. If no condition is evaluated to true, then the interpreter will follow the modified sequence without branching. The evaluation of all the branching conditions within a given context finally produces the final recombined sequence.
A ***branch*** is a recognition expression and template for alternates that will be instantiated wherever the recognition expression matches a modified sequence.

In our model, we always start from a primary sequence, which is enhanced by alternates when the recognition expression of the extension's branches match. An application of a given branch gives a temporary output on which the recombiner reapplies all the extensions, so that the recognition is also performed on the installed alternates. This process will reapply recursively until no more recognition expression is valid for any sequence. This recursive application process entails interesting problems and opportunities that will be discussed later on in the paper.

2.3.2 Notations

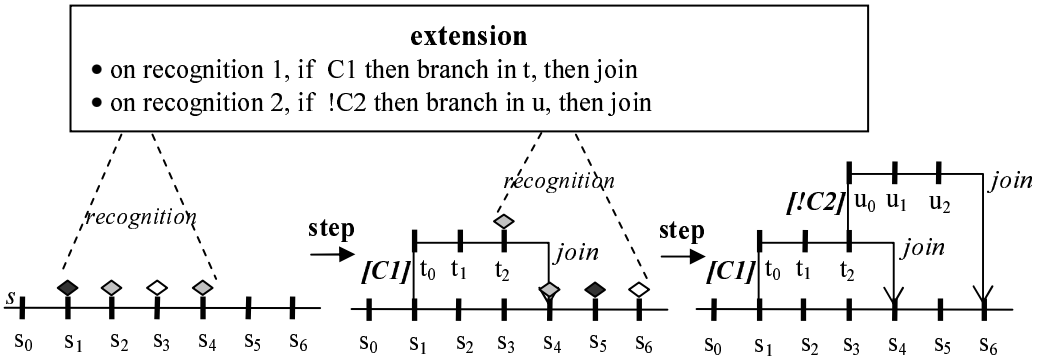
When a colored sequence will be needed for an example, we will use the generic graphical notation below on the left. When an alternate representation is needed, the graphical notation on the right can be used.



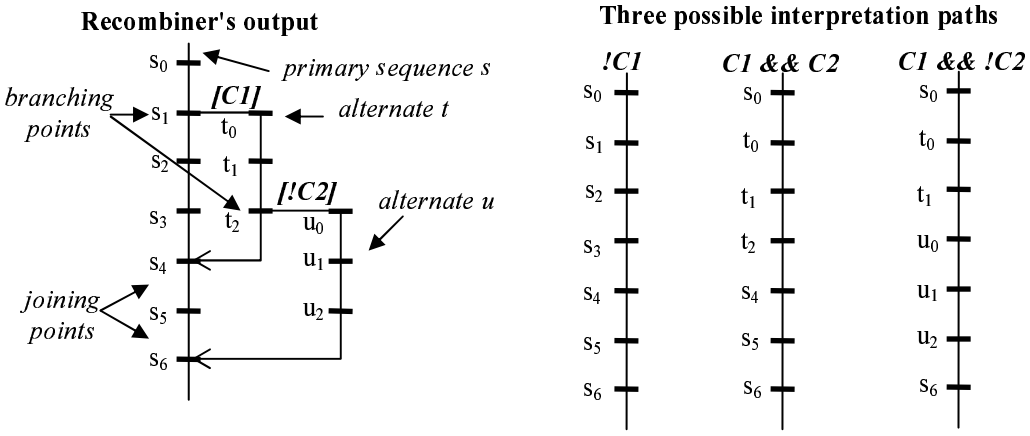
The bases of the sequences are implicitly named with the sequence's name indexed with the base's number, starting with 0. The first base is the top-most base. Names can be explicated for all or some bases (on the left side, the fourth base (s_3) is explicitly named). Pigments and their values are defined using a tag-like notation. Here, the sequence *s* is pigmented with two pigments *p1* and *p2*. The *p1* pigment is only defined for bases s_1 and s_3 that respectively hold the values 10 and 20. The *p2* pigment is defined for base s_3 and its value is the *"hello"* string. Note that this can also be drawn horizontally and read from left to right. Finally, the sequence *s* corresponds to the textual representation $s = (s_0 ; s_1 <p1=10> ; s_2 ; s_3 <p1=20, p2="hello"> ; s_4)$.

When drawing an alternate, as shown on the right part of the figure above, an arrow starting from the branching point and finishing at the joining point should be added. In this case, the alternate is a sequence of two bases, which are implicitly called a_0 and a_1 because the alternate's name is *a*. A condition should be added at the beginning of the alternate (here *C*), but it can be omitted when the context is clear, especially if it is a true condition. Note that when *C* is true, the branched base s_1 will not be part of the interpretation path. More precisely, the right figure corresponds to two possible interpretation paths: $(s_0 ; s_1 ; s_2 ; s_3 ; s_4)$ and $(s_0 ; [C]\ a_0 ; a_1\ [join] ; s_3 ; s_4)$. If *C* is true, then it only corresponds to the latter path.

2.3.3 Example



The figure above shows the creation of two alternates following our incremental recombination process. The extension defines two branches with two recognition expressions. In the first step, *recognition 1* first matches and installs the first alternate, which branches upon the *C1* branching condition and that contains the sequence *t*, formed out of three bases t_0, t_1, and t_2. Then in the second step, *recognition 2* matches and installs the second alternate on the first alternate. This second alternate is composed out of the three bases u_0, u_1, and u_2 (sequence *u*) and branches if *C2* is false.



As shown in the figure above, in our model, the recombiner's output is not a sequence but a branching mesh. In that case, the mesh is centered on one primary sequence *s*, which contains a branching point s_1 leading to alternate *t* upon the realization of the condition *C1*. Alternate *t* itself contains another branching point t_3, which activates alternate *u* if *C2* is not true. This branching mesh finally gives three possible recombined sequences depending on the interpretation context. In a first context, if *C1* is not true, we simply follow the initial primary sequence *s*. In a second one where *C1* and *C2* are both true the path goes through alternate *t*, but not trough *u*. Finally, if *C1* and not *C2*, we branch into both alternates.

This model allows us to build complex recombinations in an incremental way because the branches of the extensions do not break the modified sequences but only install alternates. Besides, the use of branching conditions allows the output of the recombiner to be interpreted differently, depending on an interpretation context. In the next section, we will further explain this model by defining a generic programming language for recombinant programming.

3 A eneric anguage or Recombinant Programming

Before we can further study RP, we feel that it would be of a great help to define a generic language that implements the concepts of the recombination model we have depicted in the previous section. This language, that we call GRPL or ***Grapple*** (Generic Recombinant Programming Language), will allow us to present simple examples in a condensed fashion and will help to clarify the different semantics that are involved in RP.

Grapple is said to be generic because it makes no assumptions on the sequences or the bases and their associated semantics. As a consequence, Grapple mainly focuses on the recombination part of the two-layer architecture that we depicted before and particularly on the definition of the extensions, which are the first class entities for recombinant computation in our incremental model.

3.1 Preliminary Definitions

In this section, we define some key points of RP that are centered around the recognition mechanism of colored sequences. This is a preliminary but necessary step to the Grapple definition.

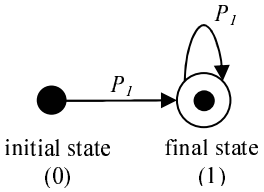
3.1.1 Recognition Expressions

As explained before, a recognition expression is an expression that is applied on a colored sequence in order to recognize strategic points and colors of this sequence for the recombination process, for instance branching and joining points. Since pigment values can change along the

colored sequence, the recognition expression must allow the writer of the expression to specify the location where the recognition predicates need to be applied. These locations should be expressed as generically as possible. For instance, we may want to be able to write that a sequence has to be recognized if a subsequence of consecutive bases hold a pigment p with a value v_1 and that the value changes to v_2 for another set of consecutive bases, following the former subsequence.

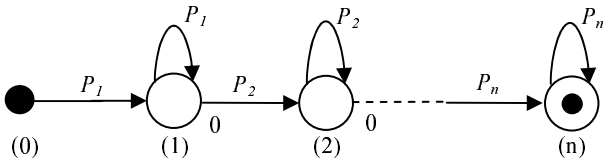
This kind of requirements can be expressed in temporal logics. Indeed, such logics consider time as a sequence of actions, which can be branching or not. They allow the validation of properties that will occur in the future or that happened in the past, or simply for a delimited sequence. Even if we are not talking about time-aware sequences for the moment (we will eventually…), the fact that RP works on sequences revels temporal logic as a particularly well-suited tool to write recognition expressions. However, temporal logic formalisms are not always very intuitive and necessitate some practice in order to be efficiently used. As we want RP to be intuitive and usable as a programming paradigm, we have defined a subset of these logics that have been appearing to be largely sufficient for writing recognition expressions efficiently. We next define the recognition expression language and its semantics.

A basic recognition expression is a first order logic predicate on pigments. For instance P_1 = p_1*>10 && *p_2*=="a string value"*, where P_1 is a predicated and p_1 and p_2 are pigment names. If the recognition expression is limited to this P_1 predicate, the matching recognition sequences will then be all the subsequences within the input sequence where P_1 is true for all their bases. We can define the exact semantics of this expression by using the following state machine.



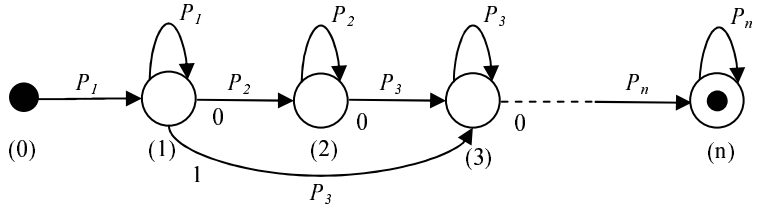
In general, any state machine that corresponds to a recognition expression can be understood as follows. When going through a sequence, a base will potentially be part of the recognition sequence if it triggers a transition of the state machine. If it does not trigger any transition and that we have not reached the final state, then all the potential bases history is forgotten and the current state is reset to the initial state. If the machine has reached a final state and that the next base does not trigger any transition (or that we have reached the end of the sequence), then all the history of the potential bases are part of the recognition sequence. The returned sequence is also called a ***recognition result***, which corresponds to an input sequence and to a given recognition expression.

In generic recognition expressions, we want to be able to match several consecutive predicates. In order to express this, a recognition expression is composed out of several ***recognition segments***. Each segment is a predicate and is separated from the next one by a semi-colon. The general form is then: P_1 ; P_2 ; ..... ; P_n and the corresponding state machine is the following.

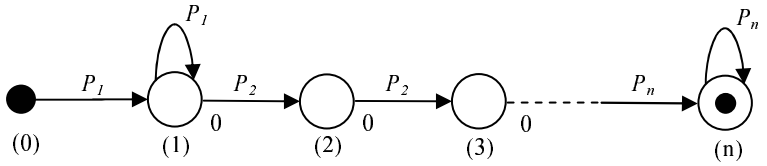


When building this state machine we came across the following question: what happens if P_i and P_{i+1} are true at state *i*? In our semantics, it seems logical to move forward towards the end of the state machine. This is the meaning of the "0" labeling the straight transitions. The transitions with a lower index are triggered before the ones with greater or no indexes.

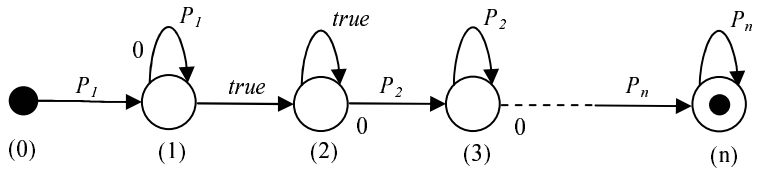
One can also note that this semantics implies that a given predicate has to be verified at least once, that is to say for one or several consecutive bases (*one to many*). In order to make a predicate optional so that it can be verified for 0 or several consecutive bases (*zero to many*), one can use the square bracket around the corresponding segment. For instance, P_1 ; $[P_2]$; P_3 ; ..... ; P_n corresponds to the following state machine.



The recognition of a predicate for an exact number of bases can be achieved by using the curly brackets, followed by an optional number. In P_1 ; $\{P_2\}2$; ... ; P_n, the P_2 predicate has to be valid for exactly two bases and the corresponding state machine is shown below. Note that the number can be omitted. In that case, it takes the default value "1".



Finally, we also want to be able to match no particular predicate for a given recognition segment. The double dots can then be used as in P_1 ; .. ; P_2 ; ..... ; P_n. They are equivalent to a predicate that is always true.



One can note that in that case, the straight transition is not taken over the one that stays in state (1). By defining the state machine this way, we allow the recognition of the longest sequence that verifies P_1.

Note that *{..}* can be noted *.

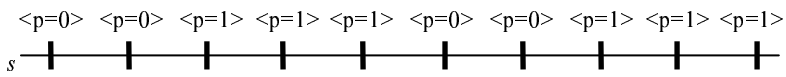
There are several other subtle issues in the definition of the state machines, especially when several optional predicates follow each other or when an optional predicate is placed at the beginning or at the end of a recognition expression. However, these are marginal and resolved cases and we do not wish to enter into to such details in this report.

3.1.2 Recognition Segment Indexes

As we just said, a recognition expression is composed out of a list of recognition segment expressions. When a recognition expression matches a part of a colored sequence in a recognition result, this matching part can then be divided into segments (subsequences) that correspond to

the recognition segment expressions local results. Using the textual form, we can note a segmented result as *((b1 ; b2), (b3 ; b4))*, meaning that this result contains two segments: the *(b1 ; b2)* subsequence, and the *(b3 ; b4)* subsequence.

As an example let us take the following colored sequence *s*.



The recognition expression *RE*: *p==0 ; p>0 ; p==0 ; p>0* gives the result *((s0 ; s1), (s2 ; s3 ; s4), (s5 ; s6), (s7 ; s8 ; s9))*. As a consequence, within the context of *RE*, the segment index 2 denotes the sequence *(s5 ; s6)*. As we will see in the next section, recognition segment indexes are extremely useful in order to refer to a particular subsequence in a generic manner.

3.2 Grapple Definition

Now that the recognition mechanism is fully defined, it becomes possible to define Grapple.

3.2.1 Generic Extensions

In Grapple, an extension follows the syntax defined in the figure below. Note that all the language keywords or symbols are in **`bold`**, `[expression]` denotes an optional expression, `{expression,}` denotes a list of expressions separated by ',' (the separator can be omitted or different). User defined (or undefined) expressions are in *`italic`*. Finally, `( expression1 | expression2 | … | expressionN )` represents a choice between N expressions.

```
extension_definition ::=
    extension extension_name
           [precedes ({extension_name,}|*)]
           [follows ({extension_name,}|*)] {
        [{
        branch[index]([{typed_parameter,}])
            :: recognition_expression [[progression_rule]]
            [{
            case [branching_condition] {
                [{
                ( sequence[(index|name)]([{pigment_name=expr,}]);
                  |
                  nop([{pigment_name=expr,}]); )
                }]
                ( end; | join[index]; )
            }
            }]
        }]
    }
```

Let us ignore the *precedes* and *follows* keywords for the moment and focus on the branch definition. Note that one can define several branches per extension. Remember also that a branch acts as a template for the installation of alternates on a sequence.

A generic branch definition can be split into three main parts.

Firstly, the prototype that defines the parameters of the branches and a branching segment index. The segment index corresponds to the segment of the recognition expression from which we want to branch. We will later see the use of these parameters.

Secondly, the recognition expression, which is placed after "*::*", following the prototype.

Finally, the last element of a branch is a list of cases that all share the recognition expression and parameters defined by the branch.

Each case takes an optional branching condition and defines a body. If the branching condition is omitted, then its value is always *true* and the branch will install alternates into which we always branch (replacing alternates). The body of the case defines the bases of the installed alternates. In Grapple, for genericity purpose, we need to introduce a generic construct in order to refer to generic sequences. This generic construct is given by the *sequence* keyword. The *sequence* keyword takes an index or an identifier. The index identifies the subsequence corresponding to a segment of the recognition result, while the identifier has to be a string that identifies any sequence after its name. The *sequence* keyword also takes a pigment value assignment list that aims to set or modify the values of the pigments for the currently referred sequence. Using this construct, when a branch installs a new alternate, it creates copies of the bases that are matched by the recognition expression. These copies can be pigmented differently than the original ones. If a base of an alternate is the copy of an original base s_n, then we usually note the copy s'_n.

At the end of the case body, the alternate should be terminated, either by joining back the branched sequence through the *join* construct, which also takes a segment index in order to denote the joining point, or by simply ending the sequence by using the *end* keyword.

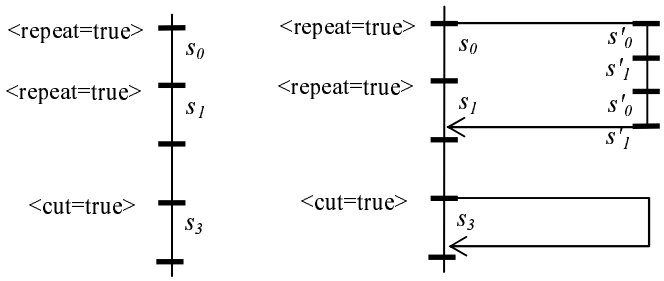
In order to illustrate the definition of generic extensions, we define the two following extensions. The *Repeat* extension repeats a sub-sequence pigmented with a *repeat* pigment.

```
extension Repeat {
  branch[1]() :: repeat
    case {
       sequence[0](repeat=null);
       join[1];
    }
}
```

The *Cut* extension below cuts a sub-sequence pigmented with a *cut* pigment.

```
extension Cut {
  branch[0]() :: cut
    case {
       join[0];
    }
}
```

Here is the recombination effect on a sample sequence.



One have been noticing the use of the *repeat=null* assignment expression in the *Repeat* extension. This allows the removal of the *repeat* pigment in the alternate. Without this removal, the recombiner would reapply the repeating process on the alternate recursively, which would lead to an infinite recursion. In the next paragraph, we further explain how the recombiner progresses when creating the alternates and how this progression can be parameterized.

3.2.2 Recombiner's Progression Model

One can note that the recognition expression is followed by an optional progression rule. This progression rule parameterizes the way the recombiner searches the matching sequences and creates the alternates. This searching and application algorithm is called a ***progression model***. By default, the recombiner will move from the start to the end of the sequence and, at each base, it will try to apply all the branches of the available extensions (the ones that branch at this point). When alternates are created, the recombiner will enter them and recursively apply to them the same progression model. We will justify and explain further this default progression mode later in the paper.

In some cases, the default progression model of the recombiner will not be acceptable and could for example lead to an infinite recursion. Even though infinite recursions can be avoided by redefining the pigments' values in the alternate (like in the *Repeat* extension), these *had hoc* solutions can lead to complex definitions of the extensions. Most of the time, the use of a progression rule is largely preferred. The next simple examples explain the use of these progression rules.

```
extension InfiniteRecursion {
  branch[0]() :: {p==0}
    case {
       sequence[0]();
       join[1];
    }
}
```

The above extension leads to an infinite recursion when a base holds a pigment *p=0*. Indeed, the branch installs an alternate that is the exact copy of the recognition sequence. Nothing then prevents this extension to re-apply to the constructed alternate. The use of the *[>]* progression rule tells the recombiner not to apply a given branch at a given point if one alternate coming from this branch has already been applied to this point. Thus, the extension below applies the extension only once and forces the recombiner to move forward.

```
extension NoInfiniteRecursion {
  branch[0]() :: {p==0} [>]
    case {
       sequence[0]();
       join[1];
    }
}
```

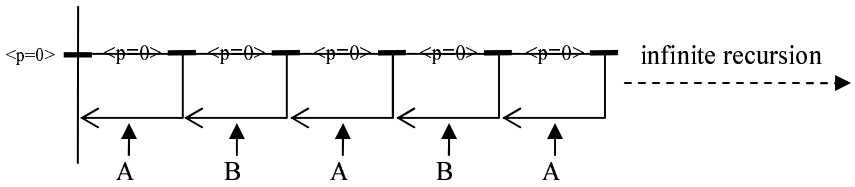
Let us now imagine the following case, defining two extensions.

```
extension A {
  branch[0]() :: {p==0} case { sequence[0](); join[1]; }
}

extension B {
  branch[0]() :: {p==0} case { sequence[0](); join[1]; }
```

```
}
```

In this case, the extension B will apply on extension A's result, and conversely, as shown in the figure. This will lead to a coupled infinite recursion because the *[>]* progression rule is now inefficient.



To deal with such cases, we defined the progression type *[>>]*, which allow the programmer to force the recombiner to move forward if the branch has already been applied to the current point, even if this branch's alternate is hidden by another one. Then, the two extensions below will only be applied once each.

```
extension A {
  branch[0]() :: {p==0} [>>] case { sequence[0](); join[1]; }
}

extension B {
  branch[0]() :: {p==0} [>>] case { sequence[0](); join[1]; }
}
```

It exists a third useful progression rule *[*]* that forces the recombiner to restart the search from the beginning of the sequence each time an alternate is installed. The use of this rule will be illustrated with some examples later on in this report.

The following table recapitulates the progression types and explains their meanings.

Progression	Meaning
[>]	a branch is not applied twice at the same point, except if the alternate coming from this branch is hidden by another branch's alternate
[>>]	a branch is never applied twice at the same point
[*]	when an alternate is installed and branches at a given point, the recognition restarts from the beginning of the sequence (on contrary to the default behavior that moves forward to the next point)

By default, the order the recombiner uses to apply the extensions is not predictable. However, this order may matter. In these cases, the programmer can explicitly parameterize the recombiner by using some ordering clause. Here we force *B* to be applied before *A*.

```
extension B precedes(A) {
  branch[0]() :: {p==0} [>>] case { sequence[0](); join[1]; }
}
```

The *precedes* ordering clause takes a list of extensions, or the * wildcard. The symmetric construct is *follows*.

Within the same extension, the branches and the cases are applied following their declaration's order, so that no ordering clause is necessary for these items. Note that we suppose here that if

two branches/cases of different extensions need to be ordered, it is possible to do it by ordering the extensions or by grouping the branches/cases differently (in other words, by refactoring the extensions).

3.2.3 Branch Parameters and Pigment Bindings

As shown in the syntax definition of an extension, a branch can take parameters. These parameters shall be used in several places.

- In the *sequence* constructs of the branch body in order to colorize a segment. For instance, *sequence(pigment_name=parameter_name)*. Note that a pigment name can be used in place of the parameter name. In that case, the value corresponds to the pigment's value on the currently pigmented base at alternate's installation and evaluation's time.
- In the condition of a case. In general, case conditions use pigment names and their values correspond to the values of the pigments at the branching point. The use of a parameter allows the parameterization of the case condition with a global value.
- In the recognition expression. In the same way and on the same purpose as for the case conditions.

In order to be valid, parameters need to be bounded using the *exists* predicate in the recognition expression. The *exists* predicate takes a pigment name as a parameter and is true on a given base if the pigment is actually defined. Within *exists*, the pigment can optionally be bounded to a name that must correspond to a branch parameter name. For example, the bounded version of the branch: `branch[0]() :: exists(p) && p==0` would be `branch[0](int i) :: exists(p:i) && p==0`. In this case, this branch is equivalent to: `branch[0](int i) :: exists(p:i) && i==0.` However, in the case of more complex branches, and typically branches that have multi-segment recognition sequences, there could be a difference in using the binding or the pigment. For example: `branch[0](int i) :: exists(p1:i) ; .. ; p2 && i==0` is different from `branch[0](int i) :: exists(p1:i) ; .. ; p2 && p1==0`. In the former case, *i==0* refers to the value of the *p1* pigment as defined in the first segment of the recognition's result. In the latter case, *p1==0* refers to the value of *p1* as it is defined in the last segment. These two values may be different, leading to different recognition results. Note that in the case of recognition segments that contains several bases as a result, the bindings take the value of the first base's pigments. In order to avoid possible ambiguities, the bindings should preferably be done within a curly bracketed segment expression.

Beyond referencing pigments, bindings are also used to ensure that pigments correspond to a given type (the parameter type). On type safety purpose, one can declare bindings and parameters that may never be used in the branch.

3.2.4 Special Bases

Like for the DNA, some bases are used for signalizing. Indeed, in some cases, it is preferable to use special bases over pigments, which shall be used for recognition only.

Name	Use/meaning	Keyword
NORMALEND	Seamlessly added to the end of any primary sequence to indicate that we have reached the end of the sequence in a successful manner.	-
JOIN	Indicates that we have reached the end of an alternate and that we join back to the sequence we come from.	join
END	Indicates that we have reached the end of an alternate and that we stop the interpretation right here, whether the sequence we come from has finished its interpretation or not (abnormal end).	end

NOP	A *nop* is like a regular base which as no meanings for the interpretation layer (it is simply ignored). It can then be used by the recombination layer when extra bases are needed.	- / nop

Any of these bases can be pigmented and thus be part of a recognition sequence. Bases that correspond to no keyword (last column) are automatically added by the compiler.

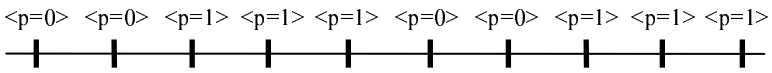
Recombination am le ing ra le

This section illustrates Recombinant Computation with Grapple. The presented examples are selected to cover most of the language's defined features.

4.1 Example 1: changing the color of a sequence

We start with a very simple example that performs no recombination but that installs some replacing alternates in order to change the color of the output sequence.

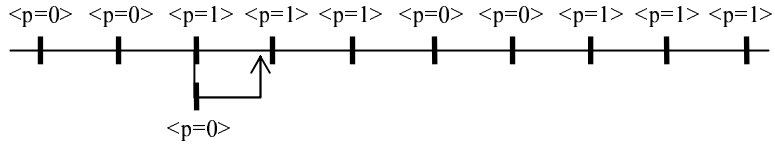
Let us take again the following simple colored sequence.



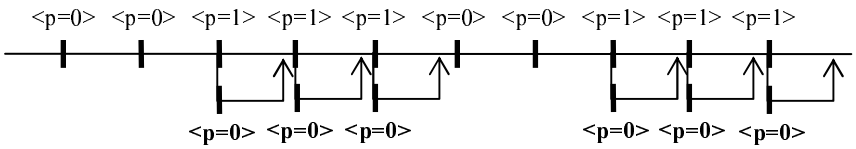
Consider that we want to change the sequence's color so that p is always 0.There are several ways of doing this.

```
extension Reset1 {
  branch[0]() :: {p!=0}
    case {
       sequence[0](p=0);
       join[1];
    }
}
```

The extension *Reset1* given above recognizes all the bases one at a time where p is not zero. Indeed, the recognition expression *{p!=0}* only matches one base at a time because of the curly brackets, meaning that the recognition result is a unique sequence of one base (for instance, base s_2). When applied, the branch creates an alternate that branches from this unique base (**branch**[0]), and has a copy of the original base s_2 but with a p equaling zero (**sequence**[0](p=0)). The created alternate then joins back the original sequence just after s_2 (**join**[1]). If we apply this to base s_2 only, we get the following output.



However, the recombiner will apply the extension's branches wherever the recognition expression matches a given subsequence. Finally, it creates six alternates and gives the following result.

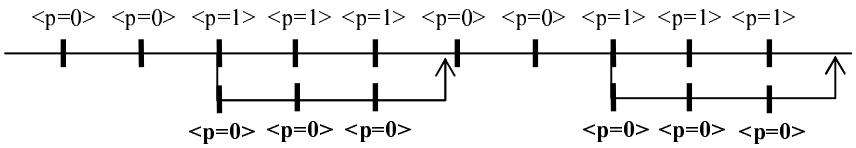


Since all the conditions of all the alternates are true, with a textual representation, it gives the following unique interpretation sequence: *(s_0 <p=0> ; s_1 <p=0> ; [true] s_2 <p=0> [join] ; [true] s_3 <p=0> [join] ; [true] s_4 <p=0> [join] ; s_5 <p=0> ; s_6 <p=0> ; [true] s_7 <p=0> [join] ; [true] s_8 <p=0> [join] ; [true] s_9 <p=0> [join])*, which is a sequence that contains the original sequence's bases (sometimes copies) but with the pigment *p* equaling zero all along the sequence.

A second way of writing the extension would be the following.

```
extension Reset2 {
  branch[0]() :: p!=0
    case {
       sequence[0](p=0);
       join[1];
    }
}
```

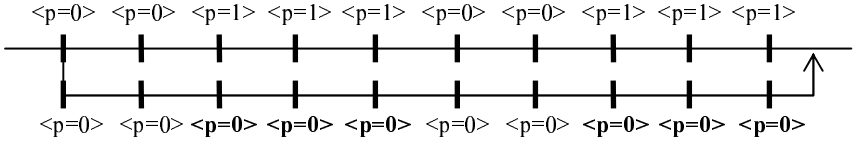
The extension *Reset2* is very similar to *Reset1* but it avoids the construction of several alternates (one per base) by dropping the curly brackets in the recognition expression. As shown next, it leads to the construction of two alternates (one per group of consecutive bases where *p* is not zero).



Finally, another way of writing the extension would be the following.

```
extension Reset3 {
  branch[0]() :: p==0 ; p!=0 ; p==0 ; p!=0
    case {
       sequence[0]();
       sequence[1](p=0);
       sequence[2]();
       sequence[3](p=0);
       join[4];
    }
}
```

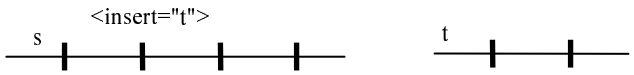
Extension *Reset3* matches and replaces the whole sequence by resetting the *p* value in two segments where it was not zero.



This last solution is the least generic in the sense that the recognition expression is very closely tied to the sequence. It is however sometimes useful to write precise recognition expressions in order to limit the application's scope of an extension.

4.2 Example 2: inserting a sequence within another one

In this simple example, we have the following sequences *s* and *t*.



We can then write the following extension that will insert the *t* sequence in the original *s* sequence, exactly like a new gene would be inserted within an rDNA sequence.

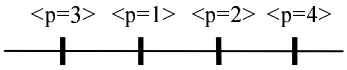
```
extension Insert {
  branch[1](String toInsertName) :: {exists(insert:toInsertName)}
    case {
       sequence[toInsertName]();
       sequence[0](insert=null);
       join[1];
    }
}
```

The result using the textual representation is $(s_0 ; [true]\ t_0 ; t_1 ; s_1\ [join] ;\ s_2 ;\ s_3)$.

The interesting point of this example is the use of the *toInsertName* parameter, which is bound to the pigment *insert*. This pigment acts like a reference to the sequence that needs to be inserted before the s_1 base. Note that if *"t"* does not correspond to any sequence name, then a "*sequence not found*" error can be reported by the recombiner when installing the alternate.

4.3 Example 3: sorting a colored sequence

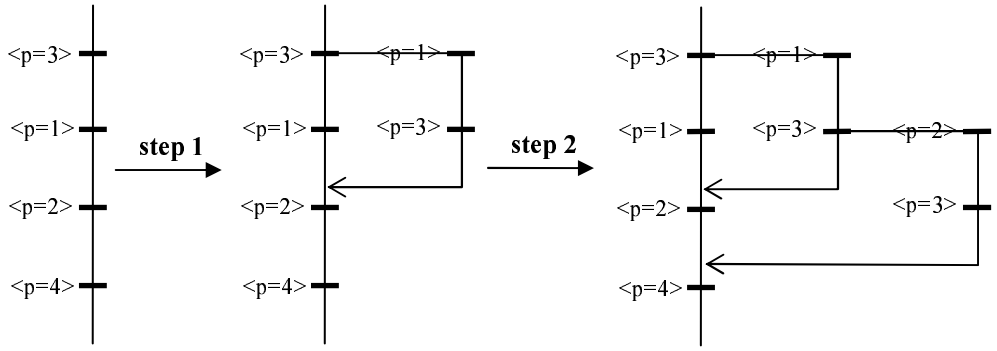
The following example performs some sequence recombination in order to sort the bases on a *p* pigment integer value. Let us have the sequence.



If we want to sort the sequence so that the *p* values increase, we first need to write a recognition expression that detects non-increasing values. This can be done by using the bindings (in the *exists* construct): *RE: {exists(p:v1)} ; {exists(p:v2) && v1>v2}*, which reads that we match a two base sequence where *v1* is the value of *p* for the first base and *v2* is the value of *p* for the second base, under the condition that *v1* is lower than *v2* (otherwise it does not match). Finally, we should write an extension that recombines the primary sequence so that it swaps the two bases of the *RE*'s recognition results.

```
extension Sort {
  branch[0]() :: {exists(p:v1)} ; {exists(p:v2) && v1>v2} [*]
    case {
       sequence[1]();
       sequence[0]();
       join[2];
    }
}
```

The recombiner recursively iterates on the input sequence until the recognition expression does not match any sequence anymore. In our simple sequence, the needed two recombination steps are shown below.



Or, using the textual representation, the recombination trace is:
(s_0 <p=3> ; s_1 <p=1> ; s_2 <p=2> ; s_3 <p=4>)
step 1: *([true] s'_1 <p=1> ; s'_0 <p=3> [join] ; s_2 <p=2> ; s_3 <p=4>)*
step 2: *([true] s'_1 <p=1> ; [true] s'_2 <p=2> ; s'_0 <p=3> [join] ; s_3 <p=4>)*

Finally, the use of the progression rule *[*]* ensures that the recombiner starts over the recognition at the beginning of the sequence each time an alternate is installed. Indeed, an unsorted base can be detected towards the end of the sequence: assume for instance that *<p=1>* is at the last base of the sequence. In that case, this progression rule allows *<p=1>* to be swapped all the way back to the start of the sequence, while it would only be swapped once with the regular progression model.

4.4 Example 4: factorial calculation

In this example, we intend to show that the extension mechanism is generic enough to perform calculations. Recombination and re-coloration of sequences indeed allows a special form of computation that is based on recombinant recursion. We start with a simple example that calculates the factorial of a pigment.

```
extension Factorial {

  // init: initialization branch
  branch[0]():: fac>=0 && (!exists(index))
    case {
      sequence[0](index=1,result=1);
      join[1];
    }

  // calc: calculation branch
  branch[0]():: fac>=0 && exists(index)
    // we iterate until index equals fac
    case index<=fac {
       sequence[0](index=index+1,result=result*index);
       join[1];
    }
}
```

One can note that the condition of the case for the calculation branch is not true. However, since this condition holds on pigments *index* and *fac*, the recombiner will be able to evaluate them totally at recombination time, resulting in true conditions for the alternates, as we show in the following recombination trace (the *init* branch applies first because within a same extension, the recombiner applies the branches following their declaration's order).

(s_0 <fac=4>)

init: *([true] s'$_0$ <fac=4, index=1, result=1> [join])*
calc: *([true] [true] s'$_0$ <fac=4, index=2, result=1> [join] [join])*
calc: *([true] [true] [true] s'$_0$ <fac=4, index=3, result=2> [join] [join] [join])*
calc: *([true] [true] [true] [true] s'$_0$ <fac=4, index=4, result=6> [join] [join] [join] [join])*
calc: *([true] [true] [true] [true] [true] s'$_0$ <fac=4, index=5, result=24> [join] [join] [join] [join] [join])*

When the recombination stops, we finally get the factorial of the *fac* pigment in the *result* pigment. The *index* pigment was used as a counter to calculate the factorial and terminate the recursion.

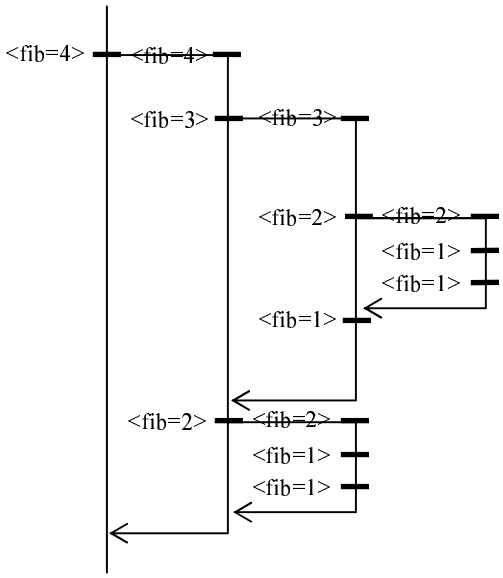
4.5 Example 5: Fibonacci Series

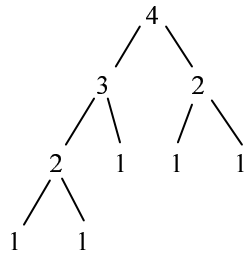
In this more complete example, we propose fairly more complicated calculations on the Fibonacci series. The following *Fibonacci* extension creates a Fibonacci tree out of a base that contains the root number of the tree as a pigment.

```
/**
 * This extension creates a Fibonacci tree structure assuming that
 * a step containing a <code>fib</code> value exists as a root node.
 * @author Carlos E. Cuesta, Renaud Pawlak
 */
extension Fibonacci {

  /**
   * Do the creation work on a given node.
   */
  branch[0]() :: {fib>=1} [>>]
  /**
   * The normal case: the current <code>fib</code> node is divided
   * into two nodes <code>fib-1</code> and <code>fib-2</code>.
   */
  case fib>2 {
    sequence[0]();
    sequence[0](fib=fib-1);
    sequence[0](fib=fib-2);
    join[1];
  }
  /**
   * A final case: when <code>fib==2</code>, the current node is
   * divided into two nodes valuing 1.
   */
  case fib==2 {
    sequence[0]();
    sequence[0](fib=1);
    sequence[0](fib=1);
    join[1];
  }
}
```

The following figure shows the created tree. The final sequence, as shown by the textual representation is not really the Fibonacci tree in itself, but a deep search on this tree. Note that it would have been possible to create the tree itself. The tree could then be used as an input structure by the interpretation level. However, in our case, a deep search is enough to perform calculations on the tree.





([true] <fib=4> ; [true] <fib=3> ; [true] <fib=2> ; <fib=1> ; <fib=1> [join] ; <fib=1> [join] ; [true] <fib=2> ; <fib=1> ; <fib=1> [join] [join])

While the *Fibonacci* extension is applied, it is possible to perform some calculations on the tree. For instance, the *SumAll* extension below sums all the nodes values of the tree. Note that it precedes the Fibonacci extension so that the summation is done before the tree creation. Also note the way we propagate a global result to the whole tree by using a recognition expression of the form: "*{C} ; ..*". The second recognition segment always matches the end of the tree and we can then propagate the *result* value to end of the tree by writing "*sequence[1](result=...);*" in the branch case body. Finally, note the use of the *[>>]* progression rules in all the extension in order to avoid infinite coupled recursions.

```
/**
 * This extension sums all the number "fib" found in the Fibonacci
 * tree in a global "result" variable.
 * @author Carlos E. Cuesta, Renaud Pawlak
 */
extension SumAll precedes(Fibonacci) {
  /** Initializes the result on the tree */
  branch[0]() :: exists(fib) && (!exists(result))
  case {
    sequence[0](result=0);
    join[1];
  }
  /**
   * Adds the current node's fibonacci value to the result (on all
   * the tree)
   * @param i the current node's value
   */
  branch[0](int i) :: {exists(fib:i) && exists(result)};
                      .. [>>]
  case {
    sequence[0](result=result+i);
    sequence[1](result=result+i);
    join[2];
```

```
  }
}
```

As an example of another calculation, the *CountLeaves* extension below counts the number of leaves within a Fibonacci tree. The principles are very similar to the *SumAll* extension.

```
/**
 * This extension counts the number of leaves in a Fibonacci
 * tree and assigns it in a global "result" variable.
 * @author Carlos E. Cuesta, Renaud Pawlak
 */
extension CountLeaves precedes(Fibonacci) {
  /** Initializes the result on the tree */
  branch[0]() :: exists(fib) && (!exists(result))
  case {
    sequence[0](result=0);
    join[1];
  }
  /** Increments 'result' for the nodes having a 1 value (leaves) */
  branch[0]() :: {exists(fib) && exists(result) && fib==1};
                 .. [>>]
  case {
    sequence[0](result=result+1);
    sequence[1](result=result+1);
    join[2];
  }
}
```

It is interesting to note that we can decouple the calculations from each another and from the actual creation of the tree. Indeed, if we name the two results of the *SumAll* and *CountLeaves* extensions differently (for instance *SumAll.result* and *CountLeaves.result*), then we can calculate both results in parallel and easily remove or add one of the calculations. It would be possible to handle this issue more cleanly by defining pigment scopes. However, we do not want to deal with this kind of details in this report.

Finally, this example indicates that Recombinant Computation has a potential for better separating the concerns when defining algorithms. We think that this result may have a positive impact on general computation in the future.

5 o ard e a ioral Recombinant Programming

Until now, we have been focusing on defining the recombination process in a generic way, without assuming any context or any particular form for the bases composing the sequences. Even though we have only been using one of the two layers of the RP model (the recombination layer), we have seen that our extension mechanism allows powerful recombination of sequences and can be regarded as an actual computation model. The applications of this new model still need to be further explored and its repercussions carefully evaluated by subsequent researches.

In this section, we wish to use the characteristics of the two layers (the recombination and the interpretation layers) by providing some insights of an important subset of RP: Behavioral Recombinant Computation (BRP). As we indicated before, the behavioral vision of RP consists of seeing the manipulated sequences as sequences of instructions forming a program's execution. This vision reifies the program's behavior and makes it possible to manipulate it as sequences of elementary instructions: ***behavioral bases***.

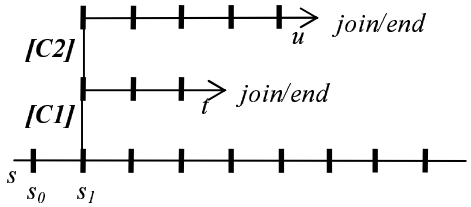
5.1 Branching Time and BRP

The fundamental difference between an ordinary sequence and a behavioral sequence is that the latter is inherently linked to the notion of time. In this section, we would like to show how BRP takes time into account and discuss the impacts on the recombination process.

5.1.1 Relating Sequences to Time

In BRP, a sequence is a list of events that defines a particular time line. Indeed, at the interpreter level, when a given base is interpreted, the base interpretation corresponds to the present event. This present event is also called the ***time cursor***. All the bases that are before the time cursor belong to its past (they have already been interpreted), while all the subsequent bases belong to its future (they will be interpreted). In case the time cursor has one or several alternate sequences, then a branching in time may occur at this particular point. In other words, there are several possible futures starting from this point, and the subsequent bases then belong to a *possible* future.

Next figure shows a typical recombinant graph. From a time-aware standpoint, it can be seen as a main timeline *s*, having two alternate timelines starting from s_1. More precisely, when the time cursor leaves s_0, there are three possible futures: one follows the *s* timeline; one follows the *t* timeline; one follows the *u* timeline. The branching in one future or another is triggered by the evaluation of the *C1* and *C2* conditions in the s_1 entering context. Once the time cursor belongs to a given future, it is not possible to go back and enter another future, even though in our model a given alternate timeline can join back the timeline it comes from (by using a *join* instruction).



As a direct consequence, in BRP, the recombination process can be seen as the creation of a possibility tree, that is to say a mesh structure that defines all the possible futures of a program execution.

In the next sections, we will see that this vision of BRP influences the way the recombiner works, and especially the default progression model.

5.1.2 Studying a Wrong Progression Model

The order in which we apply the extensions is sometimes important. We have previously presented the *precedes* and *follows* constructs and shown examples of use. However, *when* and *where* the extensions are applied is even more important in BRP and we need a clearly defined progression model so that the programmer's task in ordering the extensions is not too tedious. In order to better visualize the potential problems, this section presents a case study using a wrong progression model.

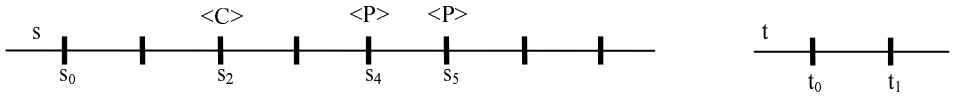
Let us take the two following extensions.

```
extension A {
  branch[0]() :: {C} ; .. ; P    // branch BA
  case C1 { sequence["t"]; join[1]; }
}

extension B {
```

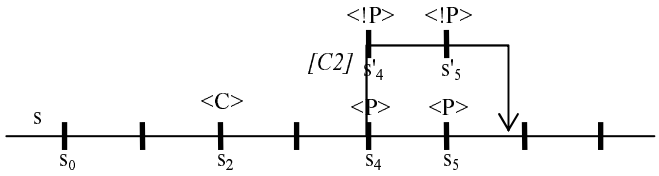
```
  branch[1]() :: .. ; P          // branch B_B
  case C2 { sequence[1](P=!P); join[2]; }
}
```

And let them be applied to the *s* sequence.



Now, let us suppose that the recombiner's progression works as follows: from each base, it applies all the recognition expressions of all the declared branches (here B_A and B_B). If one of them matches, it installs the alternates that correspond to the branch cases, skipping the cases that have a condition interpreted to false (in the context of the branching point). Then, it recursively applies the same progression process in the installed alternates if any, and it moves forward to the next base. The current base is initially set at the first base of the sequence (here s_0).

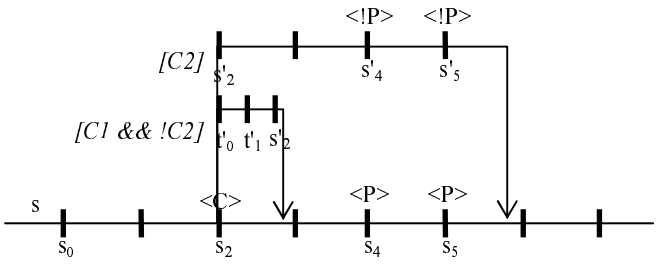
With this progression model, when the current base s_0, the branch B_B matches since the result of the "*.. ; P*" recognition result is *((s0 ; s1 ; s2 <C> ; s3), (s4 <P>; s5<P>))*. This leads to the installation of an alternate timeline that inverts the *P* property, as shown on the following graphics.



Then, when the current base reaches the s_2 base, the recognition expression of the B_A branch ("*{C};..;P*") matches *((s2 <C>), (s3), (s4 <P>; s5 <P>))*, while it does not match the timeline containing s'_4 and s'_5.

The question then arise: in this case, should we install the alternate coming from B_A?

The only case where a clear answer can be formulated is when *C2* is true. Then, the alternate timeline containing s'_4 and s'_5 is a replacing alternate and the only possible timeline is *(s0 ; s1 ; s2 <C> ; s3 ; [true] s'4 <!P> ; s'5 <!P> [join] ; s6)*: the alternate coming from B_A must not be installed because *P* is never verified. In the other cases (when *C2*'s value is unknown at recombination time – but at interpretation time), the job of the recombiner becomes complicated. In fact, the only possible answer to the problem would be to modify the alternate coming from B_A and make it start from s_2, so that two independent timelines are constructed consistently with the two extensions. It would then give the following recombination's result.



This solution may seem acceptable, but in fact it is not. Firstly, it is too complicated. It requires the modification of the already installed alternates in order to make them start from the point where conflicting alternates are installed. It also requires the rewriting of the branching conditions to make sure that the right timeline is taken (here, we rewrite *C1* into *C1 && ! C2*). Secondly, when dealing with several alternates, we have to face a combinatorial explosion since each alternate potentially conflicts with all the other alternates. Finally and most importantly, there is absolutely no guarantee that the *C2* condition can be evaluated in s_2 as it is in s_4 (it may be possible sometimes, but not in the general case, since the interpretation of the bases between s_2 and s_4 may impact the *C2* condition). This last statement makes us definitively reject the current progression model and propose another one where time consistency is better taken into account.

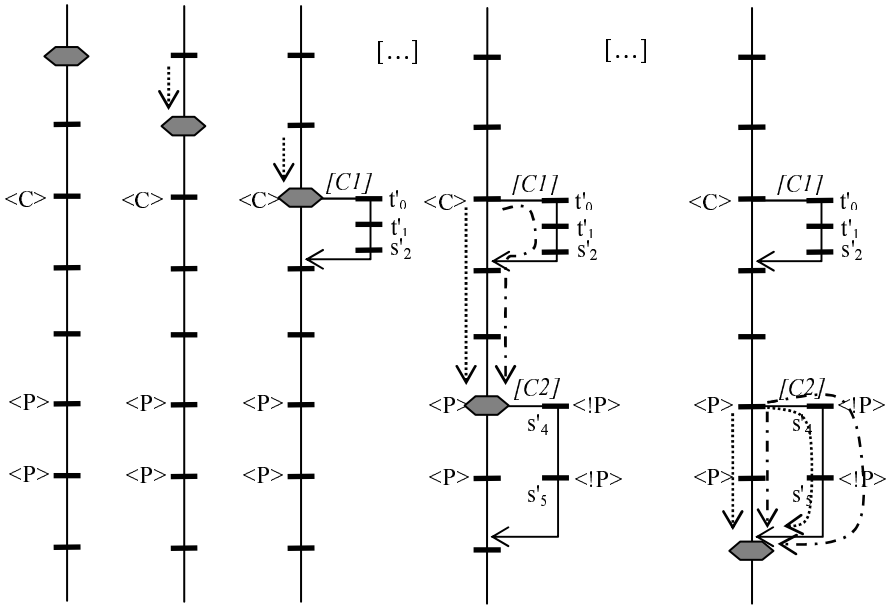
5.1.3 A Time-Aware Progression Model for BRP

If we analyze the previous case study by integrating time properties, we can almost immediately see that most of the issue comes from the fact that the B_B branch creates two possible futures for the s_2 base. In other words, this branch changes its own future. However, one of the inherent property of time is that we cannot change events before they have actually happened. More precisely, the only way we have to change the future is to act right now, in the present (or in the past if we allow "time traveling"). In order to solve the potential contradictions that may occur, logicians have introduced several models of time such as linear time and branching time models. The branching time model is the most interesting in our case. In branching time logic, an introduced event in the present (or in the past) must create a new timeline. When the event takes place, the timeline branches into a new timeline where this event has actually taken place, while the original timeline, in which the event has not taken place, remains. This is consistent with the theory of parallel universes that has been introduced to solve the time travel paradoxes (Whether this theory corresponds to a physical reality or not is far beyond our concerns. It is however a valid theory from a logical perspective and some experiments tend to indicate that it could be physically possible at a quantum scale).

Our default progression model for BRP comes directly from constraints due to branching time logics. Indeed, we consider that the recombiner works following a time cursor (the current base) that corresponds to the present at recombination time. In this context, the main constraint is that an extension cannot modify the future, that is to say none of the bases following the time cursor. In fact, by default, it can only modify the present and branch from the time cursor. As a consequence, the default progression model is defined by the following algorithm (pseudo code).

```
applyBranches(Sequence sequence, Base timeCursor) {
    // 'branches' are all the declared branches of all the extensions
    foreach b in branches
        // 'recognize' returns possible alternates because b's
        // recognition expression matches part of the sequence
        foreach a in recognize(sequence,b)
            // branch only in the present, just forget other alternates
            if(a.branchingPoint == timeCursor) {
                // install the alternate 'a'
                install(sequence,a);
                // recursively apply the same progression model to
                // a sequence that branches into 'a'
                applyBranches(
                     createBranchingSequence(sequence, a),
                     a.getFirstBase()
                );
            }
    // go to the next base
    applyBranches(sequence, timeCursor.getNextBase());
```

```
}
```



By using this progression model, the recombiner installs the alternates as shown on the previous figure, where (⬬) materializes the time cursor. An alternate is installed only if its branching point corresponds to the current time cursor so that only the present is modified. This model ensures that the B_A branch is applied before the B_B branch, which is logical from a time perspective because the modification of the original timeline by B_A occurs before the B_B one.

Note that when the first alternate is installed, the recombination progression forks into two search paths, one following the primary sequence (⋯⋯> path) and one following the alternate (-·-> path). When the second alternate is installed the search is then performed following four different paths, because each search path forks again as shown on the right end of the figure. The programmer should be aware of this behavior because it may lead to inconsistencies in some rare cases. These inconsistencies are due to the joining capabilities of the extensions and are easy to avoid (for instance by always joining at the end of the primary time line). We will not enter a detailed explanation within this report.

In addition to the default over-depicted behavior, the progression model must also include progression rules, which add some conditions on the installation of the alternates. These conditions test that none alternate from the same branch is re-installed twice at the same point (*[>]* and *[>>]* rules). The last progression rule *[*]* is quite different because it restarts the search from the beginning at each alternate installation. More precisely, with our branching time model, the *[*]* progression rule allows branching/modification in the past, or "time traveling", which is logically consistent with our model. Branching in the past can be useful to ensure global properties in the program's execution flow. Of course, we can modify the past during recombination only, where the time progression is simulated by the time cursor. At interpretation time, the time cursor follows the actual physical time and no "time traveling" is possible.

5.2 A Two-Layer Model for BRP

Now that we have clarified the relation between branching time and BRP, we can define further the BRP model. In this section, we define the two-layer model of BRP. We focus on the interpretation layer of the BRP and on showing how it relates to the recombination layer.

5.2.1 Behavioral Bases

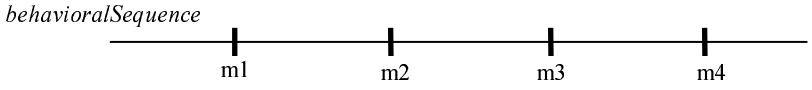
For BRP, we stand in the context of behavior-based programs. It is important to define what we understand by behavior-based definitions of programs. A behavior-based definition basically defines sequences of primitive instructions and the conditions upon which these sequences must be executed. Primitive instructions do not allow control flow manipulations of the program such as branching or iteration. In fact, they can be seen as elementary transitions from one initial state to another within the program's state space. In this vision, these primitive instructions can be directly mapped to the behavioral bases of the recombined sequences. Furthermore, the recombiner's output, which is a branching mesh of behavioral bases, can be directly mapped to the program's state space.

A behavioral program can be partially defined and interpreted using a subset of an imperative language that would contain no control flow possibilities such as loops or if statements (more generally, conditional jumps). Whether the language is object oriented or not is not relevant as a first approximation. For instance, one can code a set of behavioral sequences in the Java programming language. Theses sequences can then be recombined by a recombiner in order to produce the appropriate program. Each sequence is triggered by a branching condition installed by the recombiner.

A typical behavioral sequence can be defined in Java as the following.

```
public behavioralSequence() {
  m1(); m2(); m3(); m4();
}
```

Where *m1*, *m2*, *m3*, and *m4* are methods defined by the current class and that correspond to 4 behavioral bases. Using our graphical notation, it gives the following graphics.



5.2.2 Interpretation Variables

As we have been stating in our two layer RP model, the recombination layer produces a mesh where each branch is entered at interpretation time upon a branching condition. As seen in pure recombinant examples, some branching conditions can be fully evaluated to true. These are conditions that only depend on pigment values. However, in the general case of RP, and especially in BRP, branching conditions may also depend on interpretation variables, that is to say variables of which the values are known only during the interpretation of the recombined program. When a branching condition depends both of pigments and interpretation variables, the recombiner partially evaluates the condition and uses the result as a triggering condition for the corresponding installed alternates.

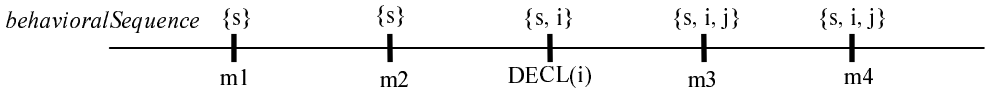
The kind of interpretation variables that are available during the program execution may vary depending on the interpretation layer's semantics. In a typical imperative language, these are local variables and parameters. When the language is object oriented, the interpretation variables may be extended to include object's fields. Depending on the variable type (and sometime on the language semantics), the scope of the variables may differ.

For instance, let us modify the previous Java method.

```
public behavioralSequence(String s) {
  m1(); m2(s); int i=0; int j=m3(i); m4();
```

```
}
```

In this new method, there are 3 interpretation variables: *s*, *i* and *j*. The s variable is defined for the entire sequence, while *i* is defined only for the last 3 bases (a variable declaration is also a base) and *j* is defined for the last 2 bases. In our graphical notation, we note the interpretation variables within curly brackets.



On contrary to pigments, interpretation variables have unknown values before the actual interpretation of the sequence. Some of the variable values could be determined before that through an additional semantics analysis (for instance, *i*'s value is 0 in this case). In that case, they may be handled as pigments by the recombiner.

In many cases, an extension should not refer directly to an interpretation variable. Indeed, the interpretation variable names are chosen by the programmer and referring to them would damage the extensions reusability. In order to overcome this, a pigment can be defined as a ***reference*** to an interpretation variable by using the following notation: *pigment_name=@interpretation_variable_name*.

5.2.3 BRP and Control Flow

In BRP, the only control flow operation that can be achieved by the interpreter is by branching into one alternate when the branching condition is true. On contrary to regular imperative languages where any control flow operation can be performed at runtime, all of the possible execution paths are calculated by the recombiner. This way of programming implies several important properties for BRP.

- BRP makes a clear distinction between control flow parameterization variable, which are basically encoded by the pigments, and the interpretation variables. Interpretation variables are all the external variables, that is to say variables that come from a system that interacts with the currently programmed system (this interacting system can be a human actor). In some formal systems such as π calculus, these kinds of runtime variables are called ***choices***.
- BRP knows all the possible execution paths of the programs at recombination time. This has two important effects.
    - Infinite state spaces, caused for instance by infinite loops, can happen at recombination time (endless installation of alternates), but never at interpretation time. This ensures the termination property at runtime, which is usually difficult to detect in regular computational models.
    - Program properties can be ensured for all possible execution paths (because all of them are reified at recombination time). For instance, it is possible to ensure that any execution path triggers a given event (for example the invocation of a given method in a Java program).

As a simple example, we can simulate an *if* statement using BRP. In that case, we can use control pigments on a sequence. One pigment for the beginning of the *if*, and one pigment for the end. Note that we use an XML-like tag notation to define the pigment within the Java code. All the instructions enclosed within a tag opening-closing pair then hold the defined pigment.

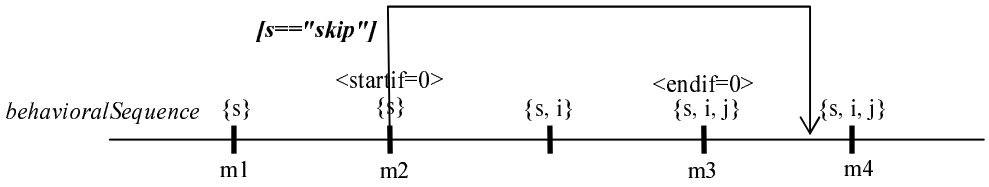
```
public behavioralSequence(String s) {
              m1();
  <startif=0> m2(s);       </startif>
              int i=0;
```

```
  <endif=0>   int j=m3(i); </endif>
              m4();
}
```

In order to simulate an *if*, the following extension installs an alternate path that jumps over the instructions surrounded by the pigments. Note the use of the parameters *sid* and *eid*, which are used to check that the starting identifier of the *if* corresponds to the ending identifier of the *if* (checked by *sid==eid* in the recognition expression).

```
extension If {
  branch[0](int sid, int eid)
    :: {exists(startif:sid)} ; [..] ; {exists(endif:eid) && sid==eid}
  case s=="skip" { join[3]; }
}
```

After recombination, the application of the *If* extension gives the result below. The evaluation of the *s=="skip"* condition can only be done at interpretation time when the value of the *s* parameter is known.



As this simple example shows, BRP can be used to simulate control flow instructions. In the next section, we present and discuss more complex examples based on a programming language for BRP called RJava.

R a a: A e a ioral Recombinant Programming anguage

In this section, we present a language for BRP called RJava, which is an extension of the Java™ programming language. RJava is a proof of concept that intends to demonstrate the possibility of using BRP in regular programming languages. It is under construction and what is presented here is a beta version. Next versions will eventually support more features, even though the underlying syntax and principles should remain the same. The RJava language is freely downloadable from our website [8].

6.1 RJava Basics

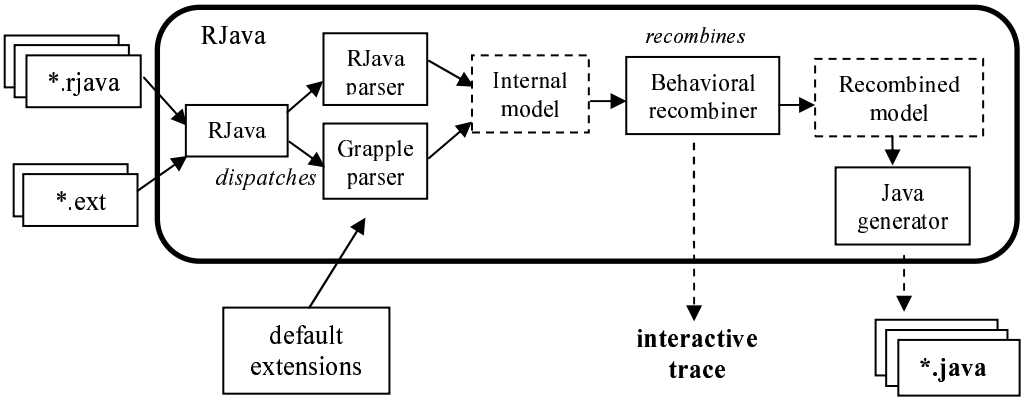
In this section we briefly introduce RJava basic implementation and language features.

6.1.1 Architecture and Implementation

RJava (for Recombinant Java) is an extension of Java 1.4. It is a preprocessor that takes as an input two kinds of files:

- **.rjava* files for Java classes that, in addition to regular methods, possibly contain recombinant methods (i.e. Java-like sequences of behavioral bases)
- **.ext* files that contain extensions, as defined by Grapple, and that are applied to the recombinant methods of the RJava files

As an output, RJava can trace how the recombination occurs on the recombinant methods. The RJava traces can be interactive in order to allow the programmer to debug the recombination process. In the future, RJava will also be able to generate the Java classes that correspond to the RJava classes by replacing the recombinant methods with a pure Java code.



RJava is composed out of several well-defined modules. Most of the modules, except the ones dealing with the **.rjava* files are not RJava dependant. In fact, they implement the generic recombinant API (*grapple* package) that can be used in other contexts and the behavioral recombiner module has been designed to be language-independent. As a consequence, it is easy to develop new recombinant languages by replacing the Java-related modules.

The RJava and the Grapple parsers are implemented with the CUP Java parser generator and JFlex for the lexical analysis part.

6.1.2 Basic Language features

Defining a recombinant method in RJava is done by the use of a new method modifier: *recombinant*. All the methods that are not declared recombinant are regular Java methods and are not subject to recombination.

Within a recombinant method's body, only simple Java instructions that correspond to behavioral bases are allowed. The RJava behavioral bases are:

- Method invocations with constant (integer, string, etc) or variable (variable name) parameters. The method invocation's result can be affected to a variable (that can be declared within the same instruction).
- Simple variable affectations with constants (`i1=0;`) or variable (`i1=i2;`).
- Simple variable declarations (`int i1;`), possibly with a constant (`int i1=0;`) or variable affectation (`int i1=i2;`).

Any other kind of instruction will be simply ignored by the parser that will only report an error in case of a Java actual syntax error. In fact, we intend to support more of the Java language for recombination input in RJava next releases. This is why the whole Java syntax is currently supported, until the experience shows us what should be subject to recombination and what should not.

Still within a recombinant method, an RJava behavioral base can be pigmented by using the XML-tag like notation that we used in the previous section.

If we take again the *If* example, the only change to make in order to get an RJava program is to declare the method recombinant.

```
recombinant public behavioralSequence(String s) {
              m1();
  <startif=0> m2(s);       </startif>
```

```
              int i=0;
  <endif=0>   int j=m3(i); </endif>
              m4();
}
```

Finally, a special interpretation variable is implicitly and locally defined by any method that returns a result (non-void methods). This variable is called *RESULT* and can be used in the case condition expressions. It is only defined when the method result is not affected to a variable (in that case, the programmer can access the result using that variable). For instance, on the *m3* base, if the result was not affected to *j*, a variable *RESULT* would be defined and its value would correspond to *j*'s. We will see some examples using this variable in the next sections.

6.1.3 Using the RJava compiler

The RJava compiler is a regular program written in pure Java. It comes with an *rjava.jar* file that contains all the necessary classes. As such, it can be integrated in any environment and can be used as a command line program. Once the class path contains the jar file, run RJava with:

```
java rjava.Run [options] <FileNames>
  options:
      --quiet: tells the compiler not to print what it is doing
      --generate-java <targetDir> <packageName>: tells the
        RJava compiler to generate a recombined Java program
      --show-alternates: at the end of the parsing, prints out all the
        execution paths that have been constructed on all recombinant
        methods
      --show-construction: for each new installed alternate, prints out
        the weaving state (current time cursor and recombination) and
        stops the weaving until the user press a key
```

For instance, let us have two files: *Example.rjava* containing the class Example that declares the *behavioralSequence* recombinant method presented before, and *If.ext* that contains the *If* extension. If we want to trace the recombination process for these two files, then we can run RJava as:

```
java rjava.Run --show-construction --show-alternates Example.java
If.ext
```

The output is then interactive and waits for the user to press the 'enter' key after the installation of each new alternate.

```
RJava: compiling rjava/examples/simple/builtin_if/Simple.rjava
RJava: pass 1 succeded with 0 warning(s) in 0.26 second(s)
RJava: compiling rjava/examples/simple/builtin_if/If.ext
RJava: pass 1 succeded with 0 warning(s) in 0.13 second(s)
RJava: checking use cases and extensions...
RJava: 0 error(s) and 0 warning(s) while pass 2
RJava: recombine...
Recombiner: starting recombination of 1 primary sequences with 1 exten-
sions...
Recombiner:    builded    alternate    path    from    extension    If,
If.branches(0)::{exists(startif:sid)};[..];{(exists(endif:eid)&&(sid==e
id))}/If.branches(0).cases(0)(int sid,int eid) into behavioralSequence
Recombiner: current alternate path:
 - m1():void - <String s>
   [(s=="skip")] -- join
 - m4():void - <long i, long j, String s>
 - NORMALEND - <>
Recombiner: press 'enter' to continue...
```

Here, the user presses enter only once because there is only one alternate to install.

```
=== Printing all paths for void RESULT behavioralSequence(String s) ===

 - m1():void - <String s>
   [(s=="skip")] -- join
 - m4():void - <long i, long j, String s>
 - NORMALEND - <>

 - m1():void - <String s>
 - m2(String s):void - <startif=0.0, String s>
 - DECL(i) - <String s>
 - m3(long i):long j - <endif=0.0, long i, long j, String s>
 - m4():void - <long i, long j, String s>
 - NORMALEND - <>

=== End of paths for void RESULT behavioralSequence(String s) ===

RJava: compilation terminated in 3.455 second(s)
RJava: job terminated successfully in 3.455 second(s) (including user
interactions)
```

6.2 Control Flow Manipulation in BRP with RJava

In this section, we present the default extensions library that we provide with RJava in order to deal with control flow issues. Since these extensions are not RJava specifics, we also use them as case studies for control flow manipulation in BRP.

6.2.1 RJava Generic Control Flow Extensions

Since no primitive instructions of RJava are control flow instructions, an important part of programming recombinant method consists of controlling the flow of behavioral sequences through the installation of alternates. In order to help the programmer with this task, we intend to provide a library of default generic control flow extensions. In the current version of RJava, these extensions are under construction and correspond to three control flow constructs:

- *If.ext* defines a generic *if* where the condition is the result of a method call,
- *Loop.ext* defines a generic *loop* that repeats a given sequence a number of times that is parameterized by a counter and a step,
- *Retry.ext* defines a generic *loop* that retries a given sequence until the result of a method invocation is true or a counter reaches a maximum repetition bound.

It is interesting to analyze the flow control capability of BRP through these generic examples.

6.2.2 A Generic If Extension

The generic *If* extension is defined as follows.

```
extension If {
  branch[0](int sid, int eid, boolean c)
    :: {exists(startif:sid) && exists(condition:c)};
       [..] ;
       {exists(endif:eid) && (sid == eid)} [>]
  case !c {
    join[3];
  }
}
```

There is not much difference with the previously studied *If* extension, except that, in that case, the branching occurs on a generic *condition* pigment. In order to link this condition to the result of a method invocation, we can for instance define a reference to the *RESULT* variable. Let us

recall that *RESULT* is implicitly defined by RJava when the result of a non-void method is not affected to a variable. A simple use of the *If* extension is shown below.

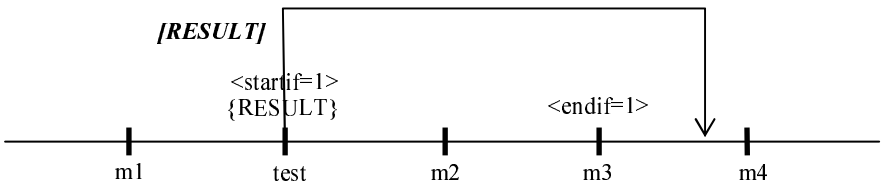
```
recombinant void colored_if_example() {
  m1();
  <startif=1><condition=@RESULT>
  test();
  </startif></condition>
  m2();
  </endif=1>
  m3();
  </endif>
  m4();
}
```

```
recombinant void if_example() {
  m1();
  if( test() ) {
    m2();
    m3();
  }
  m4();
}
```

Both of the previous syntaxes are valid in RJava. On the left, the colored version is of course the one using standard pigment notations to parameterize the *If* extension. On the right, the user-friendly version uses a Java-like syntax which is internally translated by the RJava compiler into the colored version (or a least a similar one). By supporting a Java-like syntax, we want to show that it is possible to use RP within a regular programming language, or at least a language that feels like a regular language.

An important feature to note is that the *test* method must return a *boolean* value. This is checked by the recombiner since the parameter *c* of the *If* extension is declared as *boolean* and is bounded to the *condition* pigment, which is itself a reference to the interpretation variable *RESULT* of the *test* base.

Below, the recombination output of *pigmented_if_example* is very simple.



6.2.3 A Generic Loop Extension

The generic *Loop* extension is defined as follows.

```
extension Loop {
  branch[0](int sid,int eid)
    :: exists(startloop:sid) && exists(counter)
         && exists(endvalue) && exists(step) ;
       [..] ;
       exists(endloop:eid) && (sid==eid)
  case counter!=endvalue {
    sequence[0](startloop=null,endvalue=null,step=null,counter=null);
    sequence[1]();
    sequence[2](endloop=null);
    sequence[0](counter=counter+step);
    sequence[1]();
    sequence[2]();
    join[3];
  }
}
```

This *Loop* extension installs replacing alternates that duplicate the bases of the recognition sequences (segments 0, 1, and 2). The first duplication removes the looping coloration so that the extension does not reapply, and the second duplication increments the *counter* pigment with the *step* pigment. The recursion stops when *counter* equals to *endvalue*. Note that depending on the values, the loop may never terminate. This leads to an infinite loop during the recombination, but not at runtime, which is how RP ensures the runtime termination property.

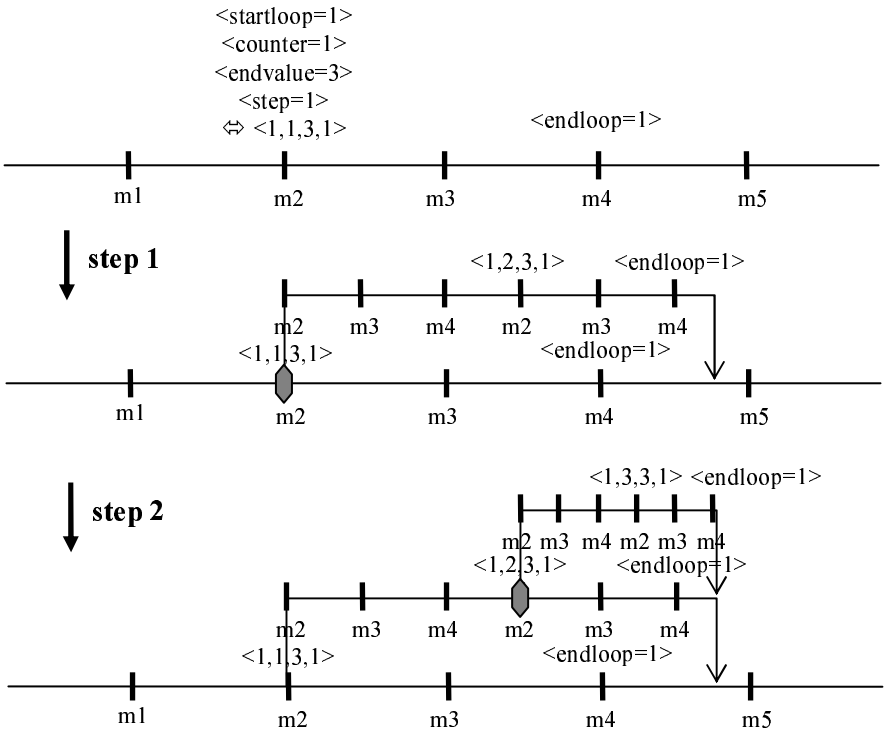
```
recombinant void colored_loop_example() {
  m1();
  <startloop=1><counter=1><endvalue=3><step=1>
  m2();
  </startloop></counter></endvalue></step>
  m3();
  <endloop=1>
  m4();
  </endloop>
  m5();
}
```

⇔

```
recombinant
    void loop_example() {
  m1();
  loop( 1; 3; 1 ) {
    m2();
    m3();
    m4();
  }
  m5();
}
```

Above we show a simple example of use. On the right the Java-like syntax introduces a new *loop* keyword in order to make it intuitive (none of the Java looping constructs can be simply mapped to this one). At this stage, we would like to point out that this loop only allows a number of repetitions that are known in advance. Indeed, at recombination time, we need to know this number in order to be able to install the alternates. This is an important limitation to RP that will be overcome in the future.

The figure below shows the installation of the two alternates needed to repeat the segment *(m2 ; m3 ; m4)* three times. .



It finally corresponds to the recombination output *(m1 ; [true] m2 ; m3 ; m4 ; [true] m2 ; m3 ; m4 ; m2 <1,3,3,1> ; m3 ; m4 <endloop=1> [join] [join] ; m5)*.

6.2.4 A Generic Retry Extension

The generic *Retry* extension is defined as follows.

```
extension Retry precedes(*) {

  branch[3](int sid, int eid, int n, boolean test)
    :: {exists(startretry:sid) && exists(nretry:n)} ;
       [..] ;
       {exists(endretry:eid) && exists(toTest:test) && sid==eid} ;
       *   // '*' is equivalent to '{..}'
  // case 1
  case (!test) && n>0 {
    sequence[0](nretry=n-1);
    sequence[1]();
    sequence[2]();
    sequence[3]();
    join[4];
  }
  // case 2 (termination)
  case (!test) && n==0 {
    end;
  }
}
```

Basically, the retry extension allows the repetition of a given subsequence under two conditions: a variable value is still false (similarly to a repeat loop), and the number of retries to do is not zero yet (this number is being decremented at each repetition).

What is interesting about this extension is that it mixes a pigment based condition (*n*, bounded to the *nretry* pigment), and an interpretation variable based condition (*test*, bounded to the *toTest* pigment, which will be defined as a variable reference). As a consequence, the recombiner will partially evaluate the branching conditions and install alternates having a residual *(!test)* as an interpretation-time branching condition. The recursion applying the alternates will finally end because the *n>0* condition will evaluate to false, making the whole branching condition *(!test) && n>0* totally evaluated to false (because *C && false = false*).

Another particularity about this extension is that it uses the * wildcard for the last recognition segment, which matches one base upon no specific criteria. It basically means that the extension expects that at least one base exists after the retried sequence (we cannot use the *[*]* expression to tell that the base is optional because we actually want a base to branch from). However, the extension works even if no bases follow the retried sequence because of the *NORMALEND* special base, which is seamlessly added to the end of any primary sequence by the compiler. Note that we usually do not show special bases in the sequences unless it is absolutely necessary for the comprehension. The compiler always shows the special bases in the produced traces so that no ambiguity is possible.

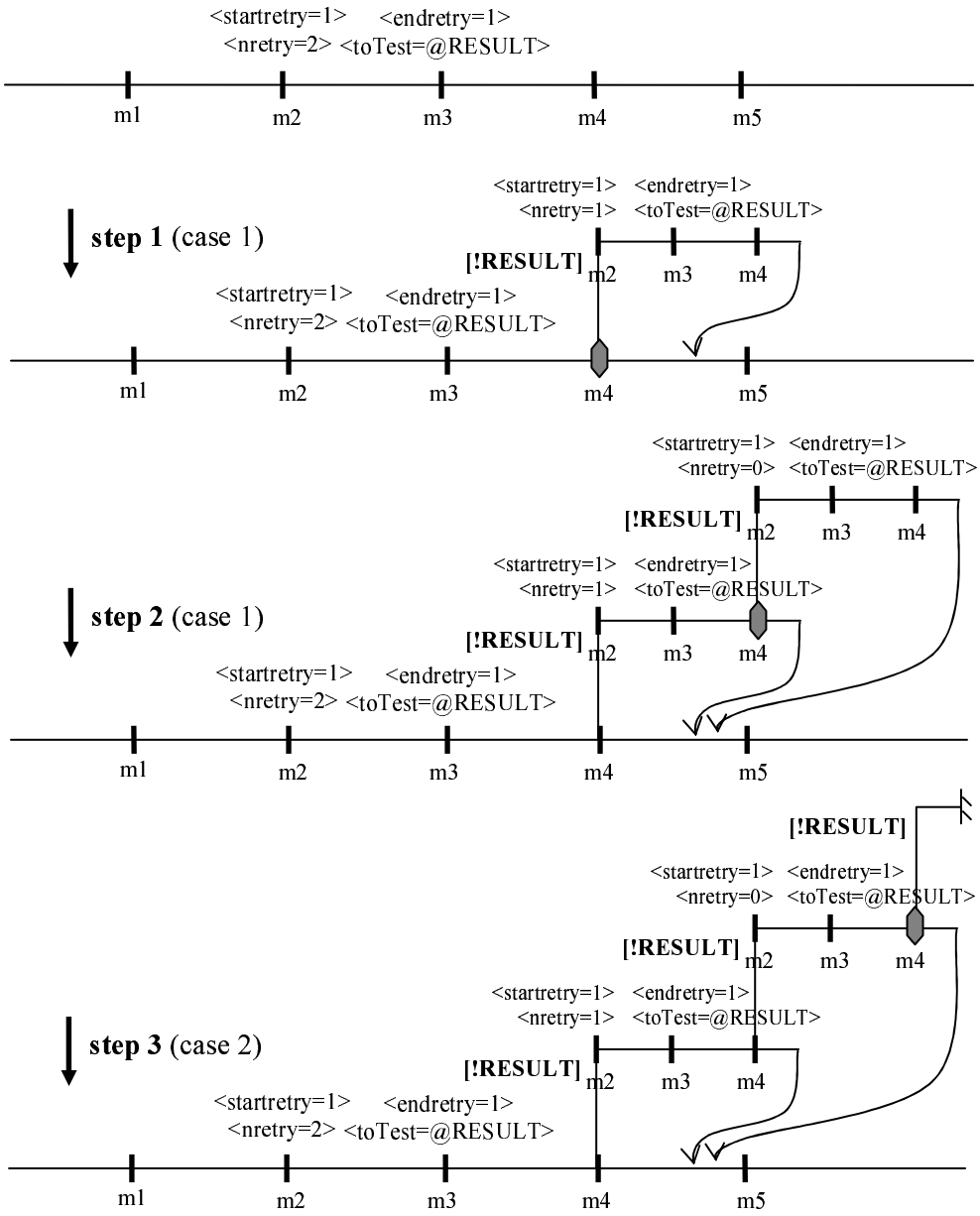
```
recombinant void colored_retry_example() {
  m1();
  <startretry=1><nretry=2>
  m2();
  </startretry></nretry>
  <endretry=1><toTest=@RESULT>
  m3();
  </endretry></toTest>
  m4();
  m5();
}
```

⟺

```
recombinant
    void retry_example() {
  m1();
  retry( 2 ) {
    m2();
  } until( m3() );
  m4();
  m5();
}
```

The next figure draws the installation of the retry extension on the code presented above. Since the *!RESULT* branching condition can only be evaluated in the context of the base it branches from, it gives 4 distinct interpretation paths that we sum up with the textual representation below.

- No retry: *(**m1** ; **m2** <startretry=1><nretry=2> ; **m3** <endretry=1,toTest=@RESULT> ; **m4** ; **m5**)*
- One retry: *(**m1** ; **m2** <startretry=1><nretry=2> ; **m3** <endretry=1,toTest=@RESULT> ;[!RESULT] **m2** <startretry=1><nretry=1> ; **m3** <endretry=1,toTest=@RESULT> ; **m4** [join] ; **m5**)*
- Two retries: *(**m1** ; **m2** <startretry=1><nretry=2> ; **m3** <endretry=1,toTest=@RESULT> ;[!RESULT] **m2** <startretry=1><nretry=1> ; **m3** <endretry=1,toTest=@RESULT> ; [!RESULT] **m2** <startretry=1><nretry=1> ; **m3** <endretry=1,toTest=@RESULT> ; **m4** [join] [join] ; **m5**)*
- Failure: *(**m1** ; **m2** <startretry=1><nretry=2> ; **m3** <endretry=1,toTest=@RESULT> ;[!RESULT] **m2** <startretry=1><nretry=1> ; **m3** <endretry=1,toTest=@RESULT> ; [!RESULT] **m2** <startretry=1><nretry=0> ; **m3** <endretry=1,toTest=@RESULT> ; [!RESULT] END)*


<startretry=1> <endretry=1>
<nretry=2> <toTest=@RESULT>
m1
m2
m3
m4
m5
step 1 (case 1)
<startretry=1> <endretry=1>
<nretry=1> <toTest=@RESULT>
[!RESULT]
step 2 (case 1)
<startretry=1> <endretry=1>
<nretry=0> <toTest=@RESULT>
[!RESULT]
step 3 (case 2)

Related or

This research is closely related to Aspect-Oriented Programming (AOP) [4]. From a structural perspective, RP can indeed be seen as a means to separate concerns within a program. Extensions can be related to aspects while the recombiner is very similar to the weaver as defined in AOP. However, the primary goal of AOP is purely structural since it tries to modularize concerns that crosscut the programs. Even though RP also better modularizes crosscutting concerns, RP is also a true computing model and allows global properties verification of the program, which goes far beyond AOP.

Note that some derivations of AOP try to explore the behavioral side of the programs in order to define better crosscut models. This is the case of use-case level pointcuts [5] and of the Event-Base AOP model [6]. Since they are manipulating sequences of actions, events, or behaviors, these approaches come closer to RP. However, their purposes do not include recombinant calculation.

Since Behavioral RP can be used to calculate the program execution paths, it can also be closely related to partial evaluation [3], and some compile-time program evaluation techniques [7]. However, calculating the program execution paths is only one of the numerous possible applications of RP. Moreover, we believe that RP presents some advantages upon classical techniques because it is focused on separation of concerns. It also contains some inherent validation possibilities, which could make it worth using over more classical approaches.

Finally, Behavioral RP allows the validation of global properties by using recognition expressions that can be seen as simplified temporal logic expressions [1]. Because RP intensively uses recursion and fix points for this kind of calculations/validations, it is also very close to the μ-calculus [2].

8 onclu ion

In this report, we have presented a very new approach for computing called Recombinant Programming (RP). Even though RP is still experimental and at a very preliminary development stage, we think that it is a very promising approach that would be worth being investigated in depth and with more means. Especially, the RJava programming language should be developed in order to obtain a full recombinant Java, which would allow large-scale experimentations of the technique.

As it shows – not enough – in this document, we think that a killer application of RP in Software Engineering is test and validation of programs or specifications through their behavior. Indeed, RP provides a very natural framework to calculate all the possible execution paths of a program (in other words its state space) and to ensure global properties on these calculated paths.

Several important elements still need to be investigated, such as the possibility to deal with special cases of undetermined or infinite state spaces (this could be achieved through JIT recombination technique and partial evaluation). The computation model could also be extended to deal with other data structures than sequences (in order to be able to deal more naturally with more complex contexts). We would like also to investigate the possibility of applying RP to other domains such as electronics, bio-informatics, or general simulation and optimization problems.

Re erence

[1] C. Stirling, "Modal and Temporal Logics", *Handbook of Logic in Computer Science*, Oxford University Press, Oxford, 1991, pp. 477-563.

[2] J.C. Bradfield, and C. Stirling, "Modal Logics and mu-Calculi: an Introduction", *Handbook of Process Algebra*, North-Holland, Amsterdam, 2001, pp. 293-330.

[3] N.D. Jones, "An Introduction to Partial Evaluation", *ACM Computing Surveys* 28(3), Sep. 1996, 480-503.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *Proc. ECOOP'97, LNCS 1241,* Springer, Helsinki, Jun. 1997, pp. 220-242.

[5] J. Sillito, C. Dutchyn, A. Eisenberg and K. de Volder, "Use Case Level Pointcuts", *Proc. ECOOP'2004, LNCS*, Springer-Verlag, Oslo, Jun. 2004.

[6] R. Douence, O. Motelet, and M. Südholt, "A Formal Definition of Crosscuts", *Proc. Reflection'2001, LNCS 2192,* Springer-Verlag, Kyoto, Sep. 2001, pp. 170-186.

[7] K. Czarnecki, and U. Eisenecker, Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[8] R. Pawlak, C. E. Cuesta, H. Younessi, "Recombinant Programming Web Site", http://www.lifl.fr/~pawlak/rp.

10 able o ontent