Exploring the design space of recombinase logic circuits.

Sarah Guiziou†[1*#], Guillaume Pérution-Kihli†[2], Federico Ulliana[2], Michel Leclère ‡[2], Jérôme Bonnet ‡[1*].

[1] Centre de Biochimie Structurale (CBS). INSERM U154, CNRS UMR5048, University of Montpellier, France.

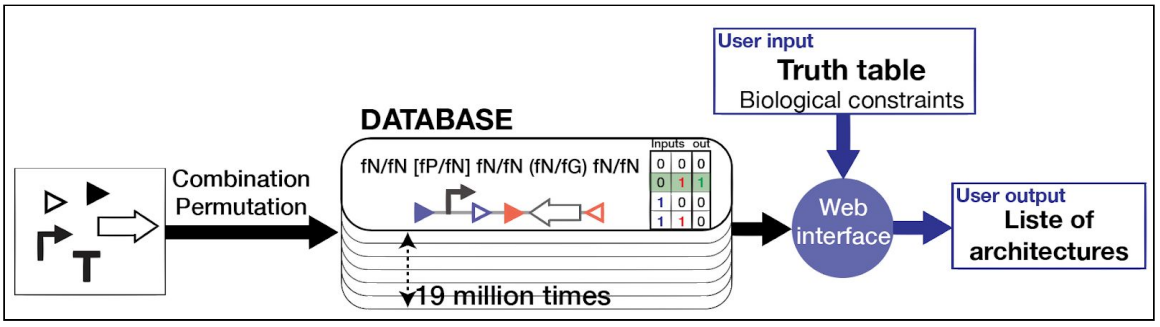
[2] Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM). CNRS UMR 5506, University of Montpellier, France.

† These authors contributed equally to this work

‡ Co-last authors

[#] Current address: Department of Biology, University of Washington, Seattle, Washington 98195, USA

*to whom correspondence should be addressed: guiziou.sarah@gmail.com, jerome.bonnet@inserm.fr



Abstract

Logic circuits operating in living cells are generally built by mimicking electronic layouts, and scale-up is accomplished using additional layers of elementary logic gates like NOT and NOR gates. Recombinase-based logic, in which logic is implemented using DNA inversion or excision, allows for highly efficient, compact and single-layer design architectures. However, recombinase logic architectures depart from electronic design principles, and gate design performed empirically is challenging for an increasing number of inputs. Here we used a combinatorial approach to explore the design space of recombinase logic devices. We generated combinations and permutations of recombination sites, genes, and regulatory elements, for a total of ~19 million designs supporting the implementation of all 2- and 3-input logic functions and up to 92% of 4-input logic functions. We estimated the influence of different design constraints on the number of executable functions, and found that the use of DNA inversion and transcriptional terminators were key factors to implement the vast majority of logic functions. We provide a user-friendly interface, called RECOMBINATOR (http://recombinator.lirmm.fr/index.php), that enable users to navigate the design space of recombinase-based logic, find architectures implementing a specific logic function and sort them according to various biological criteria. Finally, we define a set of 16 architectures from which all 256 3-input logic functions can be derived. This work provides a theoretical foundation for the systematic exploration and design of single-layer recombinase logic devices.

Keywords: synthetic biology, biological computing, recombinase, logic gates, database, circuit design.

[MAIN TEXT]

INTRODUCTION

The field of synthetic biology aims at engineering new biological systems and functions to solve pressing challenges in health, environment, or manufacturing, and to answer basic research questions [1]. Most engineered biological systems are designed using sensor, signal processing, and output modules [2] [3]. In the past years, large efforts have been devoted to reprogram cellular behavior by engineering logic devices operating within living cells using transcriptional regulators [4,5], RNA molecules [6,7], proteins [8,9], or site-specific recombinases [10–12].

Recombinase logic is of particular interest as it supports the implementation of complex logic functions using reduced, single-layer designs [10]. Recombinase logic devices operate via recombinase mediated irreversible inversion or excision of regulatory elements controlling gene expression. Recombinase logic devices are highly modular (i.e. inputs are easily modified by changing the control signal driving integrase activity), are capable of data storage [13–15], and can be adapted to various species with minimal modifications [11,16–19].

Several recombinase logic systems operating in single-cell or multicellular systems have been described in recent years, mostly using serine integrases [10–12]. All 2-input logic functions have been realized through Boolean Integrase Logic (BIL) gates, using a combination of DNA inversion, excision, asymmetric terminators and promoters, and one pair of sites and integrase per input [10] (Table S1). Another approach, termed BLADE, used excision-based recombinase devices and integrase site variants to implement all 2- and 3-input logic gates in single-layer single-cell devices [11]. BLADE offers a single, modular genetic layout albeit with a less compact design since the number of required integrase site variants increases exponentially in the number of inputs. The BIL gate strategy on the other hand is more flexible and can potentially lead to most compact genetic layouts, which could be easier to debug, implement, and scale, while however being highly divergent and *ad-hoc.*

Here we aimed to explore the design space of recombinase logic devices built using the BIL gate strategy. Other types of cellular logic (e.g. transcription-factor based) have explored circuit design spaces using electronic principles [4]. Recombinase logic devices however mark a departure from electronic layouts mimikry, and methods to explore their design space have been lacking. Systematic design rules have been defined to generate a reduced set of recombinase logic devices [20], but these approaches do not support the design of all possible single-layer recombinase logic devices.

In this work, we set up two objectives: (i) to systematize the design of recombinase logic devices based on the BIL gates strategy (Table S1) for an increasing number of inputs (i.e 3 and 4 inputs), and (ii) to determine if all 3-input and 4-input logic functions could be implemented using BIL gate design.

We used a combinatorial approach in which we generated million of combinations and permutations of recombination sites, genes, and regulatory elements. The ~19 million architectures generated in this work can implement all 2- and 3-logic functions and 92% of 4-input logic functions. We provide a web-interface, called RECOMBINATOR, from which users can obtain all possible architectures for any desired logic functions, and sort them according to specific biological constraints. Finally, we defined a reduced set of sixteen logic functions and corresponding architectures which once optimized should support the implementation of all 3-input logic functions. The RECOMBINATOR database supports the exploration of a wide range of single-layer recombinase-logic circuits designs and will facilitate the deployment of recombinase logic for many synthetic biology applications.

RESULTS

1 - Formalizing logic architecture generation.

Recombinase-based logic devices are built by composing recombinase sites with biological parts controlling gene-expression, plus at least one gene to provide the device output. In order to generate our library of logic designs, we first developed a formalization for biological parts functionality (semantics to which are associated biological elements) (Supplementary text 3.1). We then generated recombinase site arrays (structures) that were functionalized by combinatorially inserting semantics at each intersite position. These functionalized structures were then converted to architectures by replacing semantics by their corresponding biological element (Fig S1).

Formalizing functionality of biological parts.

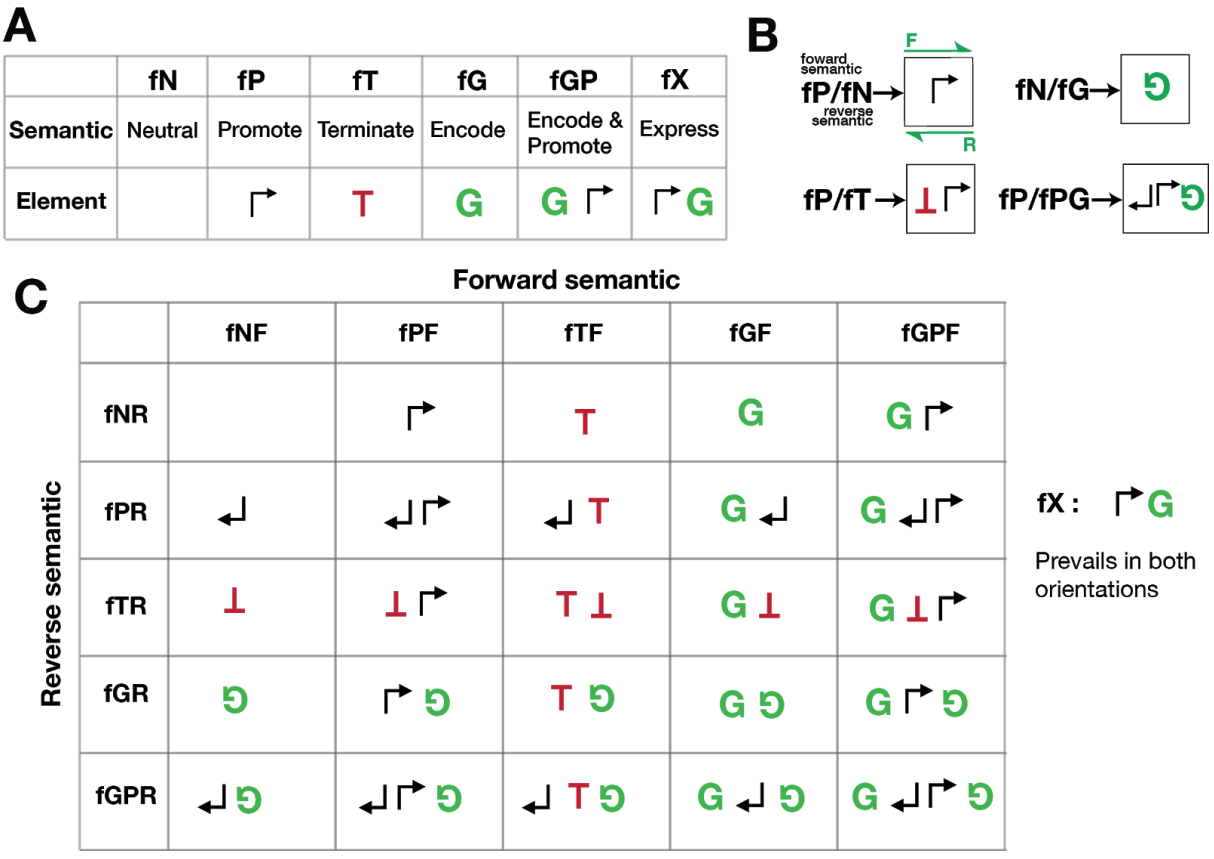


Figure 1: Formalizing functionality of biological parts and part concatenations into a reduced set of semantics (A) Formalism for the 6 semantics for a single DNA orientation and corresponding biological elements. *In terms of gene expression, DNA parts and their composition can have 6 different functionalities in a single DNA orientation, called semantics: neutral (no function), promote (promoter), terminate (terminator), encode (gene coding sequence), encode and promote (concatenation of gene followed by a promoter), and express (concatenation of promoter then a gene).* ***(B) Example of forward and reverse semantics for parts and part concatenations.*** *Each part and part concatenation have a semantic in forward and in reverse orientation represented as f(forward)/f(reverse).* ***(C) Correspondence between semantics and elements.*** *The columns correspond to the forward semantics and the lines to the reverse semantics. In each cell of the table,*

the element corresponding to semantics concatenation is represented. As the "express" semantic (fX) prevails over any other semantic in both orientations, it is represented separately at the right of the table.

In our design, we used three types of biological parts: promoter, terminator, and gene. While these parts for gene expression can be composed in an infinite manner, the parts and the composition of these parts are reducible to a limited number of semantics corresponding to their function in the context of gene expression. We formalized here this finite set of semantics.

Each biological part implement one semantic: a promoter promotes the initiation of transcription (fP), a terminator terminates transcription (fT), and a gene encodes for a function (encoded by a RNA molecule and/or a protein) (fG) (Fig1C). We assume that the semantic of these parts is neutral in reverse orientation (fN). Each part has consequently a forward and reverse semantic (f-/f-), e.g. a promoter in forward orientation has a fP semantic and a fN semantic in reverse orientation, i.e. (fP/fN) (Fig1D).

The combination of parts two-by-two leads to only two novel semantics, as several combinations are otherwise being simplifiable in the four previously defined semantics (Table S2). The two novel semantics are: (1) the expression of a gene (fX) corresponding to the concatenation of a promoter with a gene, and (2) the double semantic encapsulating both promotion and encoding (fGP) corresponding to the concatenation of a gene with a promoter (Fig1C).

To summarize, in one orientation (e.g. 5'->3'), all concatenations of biological parts can be reduced to 6 semantics: neutral (fN), promote (fP), terminate (fT), encode (fG), express (fX), encode and promote (fGP). Considering both orientations, 26 semantics exist, as the semantic "express" prevails over all other semantics in both orientations (Fig1; Table S2; Supplementary text 3.1).

Representing recombination site arrays as structures composed of well-balanced sequences of brackets and parentheses

Site-specific recombinases drive the logic devices by catalyzing transitions between different recombination intermediates having different output states. Each input signal induces the activation of a single recombinase using either transcriptional, translational or post-translational activation. A recombinase recognizes two specific DNA sequences called pair of recombination sites and mediates excision of the DNA sequence placed between sites in parallel orientation, or inversion for sites placed in antiparallel orientation (Fig2A-B).

In order to generate all possible site permutations, we used brackets and parentheses to represent recombination sites respectively in inversion and excision orientations. We called *structure* the sequence of parentheses and brackets representing a recombination site array. As we focused on combinatorial logic, recombination reactions must be independent, therefore recombination sites must not be interleaved. We thus generated only structures composed of well-balanced sequences of brackets and parentheses (corresponding to a

Dyck word, see methods and supplementary materials for details). Consequently, the identification of site pairs is unambiguous, and no annotation of parentheses and brackets is required (e.g. Fig S2).

Architecture generation

To implement logic, biological parts are placed between integrase sites conditioning their excision or inversion and subsequent effect on gene expression. Therefore, after having generated structures, we generated *functional structures* by placing a semantic in each available space between brackets or parentheses.

While various biological part concatenations can have the same semantic, we selected a single element to encode each of the 26 semantics (Fig1C-D). The choice of each element was guided by several criteria. As a general design rule, the chosen element had to be the simplest concatenation of parts capable of encoding the semantic of interest. More specifically, we aimed to use elements composed of the minimum number of parts, and respecting as much as possible the following two architectural constraints (e.g. Fig S3A). First, we prioritized avoiding promoters facing each other, a configuration known to generate interferences and unexpected transcriptional behavior (e.g. PR-PF is favored to PF-PR) [21,22]. Then, we chose elements having the minimum number of parts between a gene and the promoter controlling its transcription (e.g. GF-TR is favored to TR-GF), as gene expression generally decreases with the distance between the promoter and the gene [23].

We then generated architectures from functional structures by replacing semantics by their corresponding element (Fig2C).

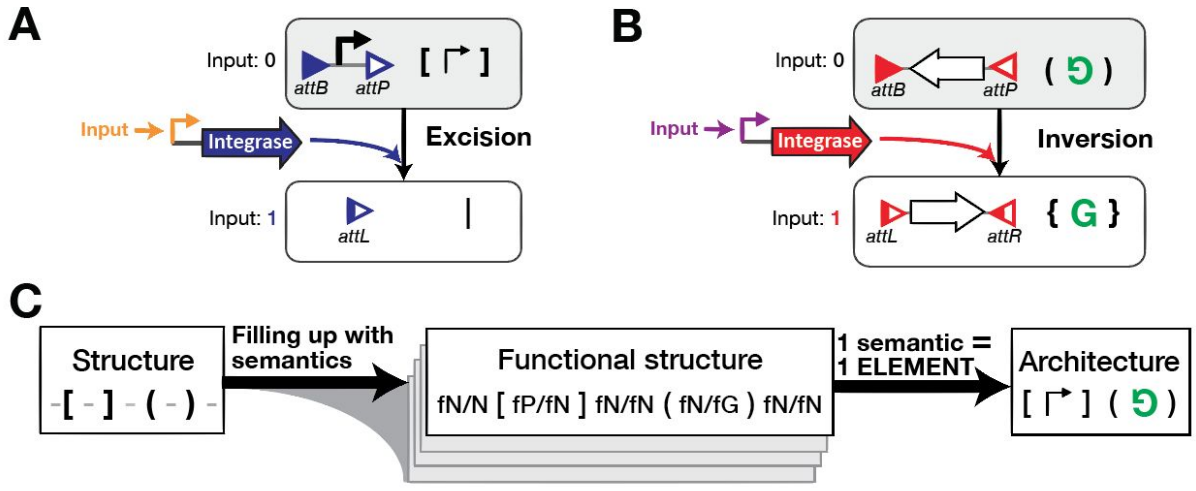


Figure 2: Formalization rules for generating a complete and finite design space. (A and B) Recombination mechanisms and its formalism. *Each integrase is expressed in the presence of a specific input signal. In A, the integrase sites (triangles) are in the same orientation, the integrase mediates excision of the DNA between the two sites (here a promoter). Brackets are used to represent integrase sites in excision orientation and a vertical bar for used site (either attL or attR)*

resulting from the excision. In B, the integrase sites are in opposite orientation, inversion of the DNA between the two sites is then performed by the integrase (here a gene). Parentheses are used to represent integrase sites in inversion orientation and curly braces for the resulting used sites (both attL and attR). ***(C) Workflow for generating an architecture****. Site structures are generated as functionalized Dyck words. Each space between sites, considered as variable, is filled with semantics using a backtracking algorithm to obtain functionalized structures. Each semantic is associated to one element, so that the architecture corresponding to each functionalized structure is obtained by direct replacement.*

2 - Algorithmic method for obtaining irreducible and non-redundant architectures.

We aimed at generating a recombinase device database composed of irreducible and non-redundant architectures and without chiral pairs of architectures. An irreducible architecture is an architecture in which no part can be removed without changing the logic implemented (e.g. Fig S3B). A non-redundant architecture is an architecture which implement a Boolean function which is not simplifiable into a Boolean function with a reduced number of inputs. A chiral pair of architectures corresponds to two architectures which are reverse-complement from each other (e.g. of chiral structures Fig S2C). We performed the generation and sorting at the level of functionalized structures, and the obtained set of irreducible and non-redundant functionalized structures was then converted in architectures (Fig S1).

From one architecture, we generated the derived architectures, corresponding to the different architectures resulting from integrase-mediated recombination (Fig 3A). A derived architecture corresponds to a specific input state, and from one architecture: 2^N-1 derived architectures are generated, with N being the number of integrase site pairs (equivalent to number of inputs). For each input state, the algorithm determines the gene expression status of the architecture. Determination of gene expression status was performed using a set of rules (Supplementary text3.3 and Table S2) that translate the total concatenation of semantics into a general semantic indicating its gene expression state, enabling the generation of the architecture's truth table.

Importantly, during the generation, we did not attributed inputs to integrases and recombination site pairs. All parenthesis and/or brackets can eventually be associated to a specific input. All logic functions corresponding to the various input permutations are thus implementable with one architecture (Fig3B). As logic functions are in the same permutation-class (P-class) if they are equivalent by permutation of inputs, a given architecture implements a complete P-class [24].

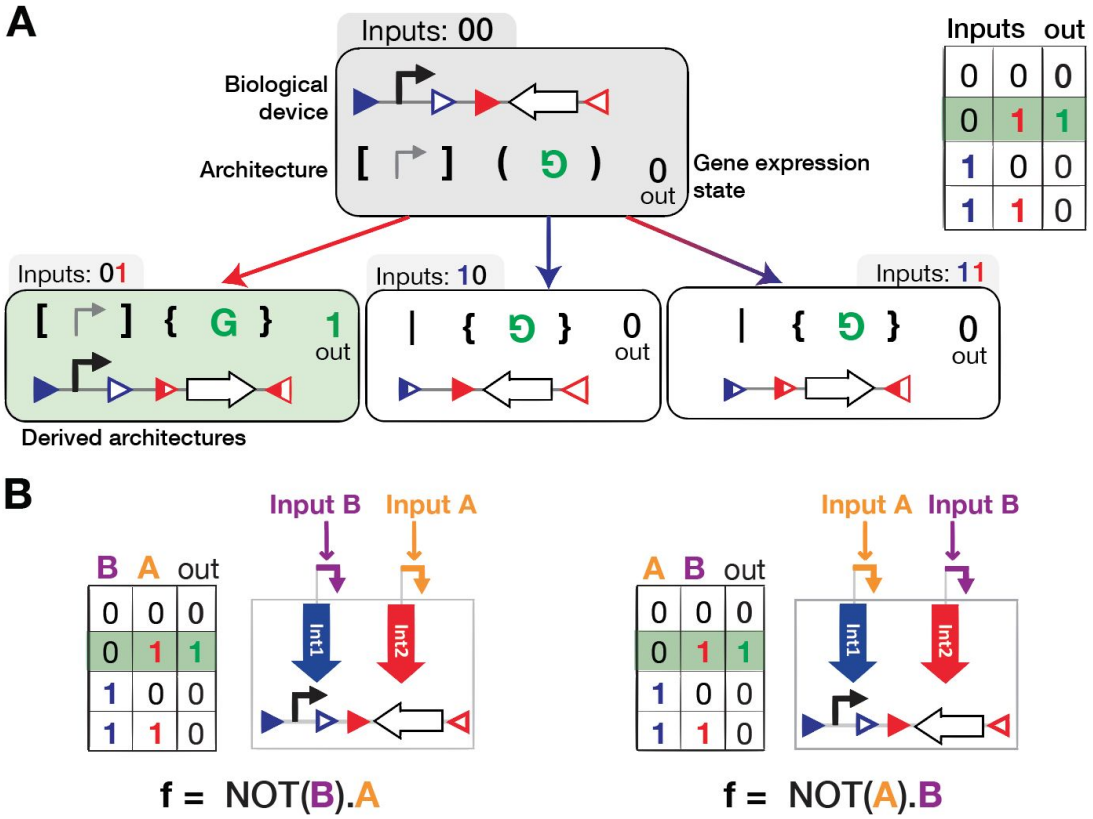


Figure 3: From architecture to truth table. (A) Generation of architecture derived states and of the corresponding truth table. For a given architecture, here also represented as a biological device, the derived architectures corresponding to the different input states are obtained by simulating integrase recombination. In this example, in the presence of one input (red) corresponding to the state 01, the gene is inverted, in the presence of the blue input (state 10), the promoter is excised and in the presence of the two inputs, the promoter is excised and gene is inverted. From each architecture and derived architecture, the gene expression state is obtained, and the truth table is generated. **(B) From one architecture, implementation of several logic functions belonging to the same P-class.** By differentially connecting inputs to integrases, various logic functions are obtained from the same architecture. For example, the function NOT(B).A realized by connecting input B to integrase 1 (blue) and input A to integrase 2 (red). The permutation function NOT(A).B is obtained by permuting the connections.

3 - Database analysis.

We performed the database generation for up to 4 inputs. We obtained 18,163,227 architectures implementing 3,608 P-classes corresponding to 59,820 logic functions. We found that using our specifications, all 1-, 2- and 3-input functions and P-classes were implementable (Table 1). On the other hand, 92.25% of the 4-input logic functions were implementable, corresponding to 90.4% of the 4-input P-classes (the discrepancy between the number of functions and P-classes arise from the fact that P-classes contain a different number of functions).

Number of inputs	Number of architectures	Number of function implemented	% of implemented functions	Number of P-class implemented	% of implemented P-classes
1	10	2	100%	2	100%
2	724	10	100%	8	100%
3	96,981	218	100%	68	100%
4	18,065,512	59,590	92.25%	3530	90.4%

Table 1: Database characteristics. The number of generated architectures and the corresponding functions/P-classes implemented are represented. Only non-redundant Boolean functions are represented. The percentages of implemented functions and of implemented P-classes were calculated as percentages of the total number of functions and P-classes corresponding to a given number of inputs (Table S3).

Implementability of logic functions

Over all the 3,904 4-input P-classes, 374 P-classes are not implementable with our circuit specifications. These P-classes are then the more complex ones to implement using recombinase logic. We sought to understand if non-implementable P-classes shared common properties. We started by searching if a correlation existed between the properties of the logic equations and P-class implementability.

We write logic functions as a sum of product of NOT or IDENTITY functions, corresponding to the minimal disjunctive form also called minimal SoP (sum of product) form [25]. A disjunctive form is called minimal if there exists no other equivalent expression involving fewer products, and no other equivalent expression involving the same number of products but a smaller number of literals [25].

We hypothesized that the most difficult P-classes to implement were the ones with the largest equation, such as while written in a minimal disjunctive form, these equations contained the highest number of terms, and the highest number of literals per term. Indeed, the non-implementable 4-input P-classes correspond to the P-class with the highest number of terms and highest number of literal per terms (Fig4A). While a total of 9.6% of 4-input P-classes are not implementable, this number reaches 23.7% for P-classes with more than 4 terms and more than 2.5 literals per term (Table S4).

However, we found some exceptions to this trend: some clusters of P-classes with a very high-number of terms and literals per term, such as 8 terms and more than 3 literals per term, and 7 terms and 4 literals per term, were totally implementable. These exceptions correspond to complete or partially symmetric functions (See supplementary text 3.4), such as functions which are entirely or partially dependent on the number of inputs present, e.g. a 2-input XOR function.This can be explained by the fact that, in comparison to other design strategy such as repressor-based logic, symmetric functions are easily implemented with recombinase logic using nested inversion ([10]; Supplementary text 3.4).

Number of architectures per P-class.

The number of architectures capable of implementing a particular P-class is highly variable, ranging from 1 to 1.4.10^6 (Fig4B) (6 P-classes with only one architecture: Supplementary text 3.5), and widely distributed (Fig4C). For implementable 4-input logic functions, the median number of architectures is of 252, providing a wide design space for most functions.

Similarly than for implementability, P-classes with the highest number of possible architectures are P-classes with the lowest number of terms and literals per term, corresponding to the simplest functions (Fig4B, 4C left panel). Reciprocally, the P-classes with the lowest number of possible architectures are the ones with the highest number of terms and literals per term, corresponding to the more complex functions (Fig4B, 4C, right panel). While the figure 4B represents the mean number of architectures for a cluster of P-classes with a specific number of terms and literals per term, we also plotted for each cluster the distribution of the number of architectures (Fig4C and FigS4). We found that the distribution of the number of architectures was quite homogeneous in each cluster and shifted from a high to a low number of architectures when P-classes complexity increased.

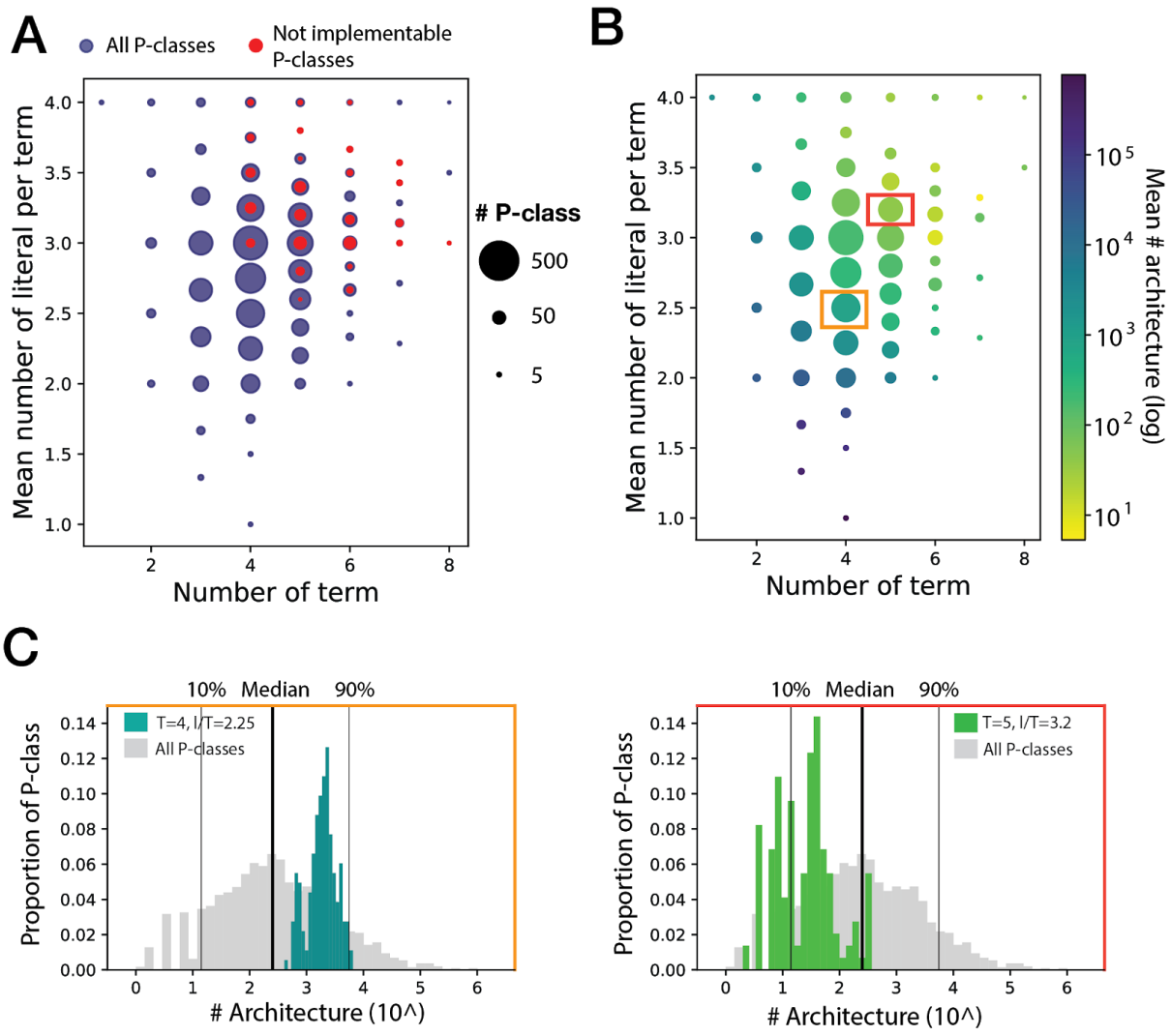


Figure 4: Implementability and number of possible architectures of P-classes according to their complexity. (A) and (B) P-classes with a specific number of terms and of literals per term are

represented in the corresponding x and y positions by a dot with a diameter proportional in logarithmic scale to the number of P-classes. **Representing in (A) the implementability of P-classes,** the blue dots correspond to all 4-input P-classes and the red ones to the not implementable 4-input P-classes. In (B), only the 4-input implementable P-classes are represented. **Representing the number of possible architectures according the P-class complexity,** the dot colors are associated to the mean number of possible architectures for each set of P-classes. Blue corresponding to a reduced number of architectures and yellow the highest number of architectures. 2 dots are surrounded by respectively an orange and red rectangle, corresponding to the two P-class sets represented in (C). Detailing (B), the distribution of the number of architectures for these two sets of P-classes is represented. For P-classes with 4 terms and 2.25 literals per term, the distribution is in teal (blue/green) and for 5 terms and 3.2 literals per term in light green (used bin is 20). The distribution for all 4-input P-classes is also represented in grey as reference (used bin is 40).

Influence of design specifications on P-classes implementability

Previously, various strategies have been defined for the design of recombinase-based devices using either only excision [11,12,26] or mainly inversion and terminator based elements [10]. In our database, we generated all possible designs using both excision and inversion, and a flexible use of the previously-defined biological parts. By sorting these architectures, we obtained the percentage of P-classes implementable with restrictive design criteria corresponding to the previous works implementing recombinase logic [10,11].

Excision vs inversion

We first analyzed the influence of the type of recombination reaction on P-classes implementability. We found that using only DNA excision highly reduces the number of implementable P-classes (82% of 3-input P-classes implementable and only 26% of 4-input P-classes) (Fig5). In contrast, almost all implementable P-classes can be realized using only inversion (100% of 3-input and 86% of 4-input P-classes) (Fig5). One explanation is that excision is a destructive mechanism leading from a specific semantic to a neutral semantic, reducing the semantic space of the device. On the other hand, inversion leads to a semantic change therefore permitting to implement more complex P-classes (Fig S5A-B). Very interestingly, while all 3-input P-classes are implementable with only inversion, we found that some 4-input P-classes (4.5% of implementable P-classes) require both inversion and excision to be implementable.These P-classes cluster in the medium complexity region of the plot (Fig S5C). Therefore, inversion only is not sufficient to implement the maximum range of Boolean functions.

Number of genes, promoters, and terminators.

A functional logic device requires at least one promoter and one gene, and can therefore be designed without a terminator. However, without terminator, 15% of 3-input P-classes are not implementable, as well as 30% of 4-input implementable P-classes (Fig5). Terminators are thus essential elements for recombinase logic implementation, in particular for the more complex P-classes (Fig S6A). Similarly, by limiting the number of genes or the number of promoters to one, the number of implementable P-classes is reduced by almost 50% for 4 inputs (Fig5). To conclude, using all gene-expression parts and letting flexible their respective numbers permit to maximize the number of implemented logic functions.

In our design, we considered terminators as asymmetric terminators (terminating transcription in a single orientation). Bidirectional terminators (semantic fT/fT) were obtained by combining two terminators in opposite orientations. A reduced number of asymmetric terminators has been characterized, and only in bacteria [27]; such parts might thus limit the portability of our devices to other organisms, particularly in eukaryotes. We wondered how many P-classes required strictly asymmetric terminators to be implementable. Excluding asymmetric terminators from the designs, we found that 93% of the 3-input P-classes and 54% of the 4-input implementable P-classes were implementable (Fig5 and FigS6B).

We previously designed 2-input logic gates in which a single output gene is placed in 3' of the device, and is easily interchangeable [10]. In addition, this design allows the logic device to be directly placed upstream of endogenous genes to add an externally-controlled layer of logic to their regulation. By filtering the database for architectures possessing a single gene in the 5' or 3' extremities, we found that all 3-input P-classes and 57% of 4-input implementable P-classes are implementable using this design.

Additionally, we tested the effect of combining criteria two by two, and found that for stringent criteria such as the use of only excision, the combination with another criteria drastically reduced the number of implementable P-classes (Fig S7-8).

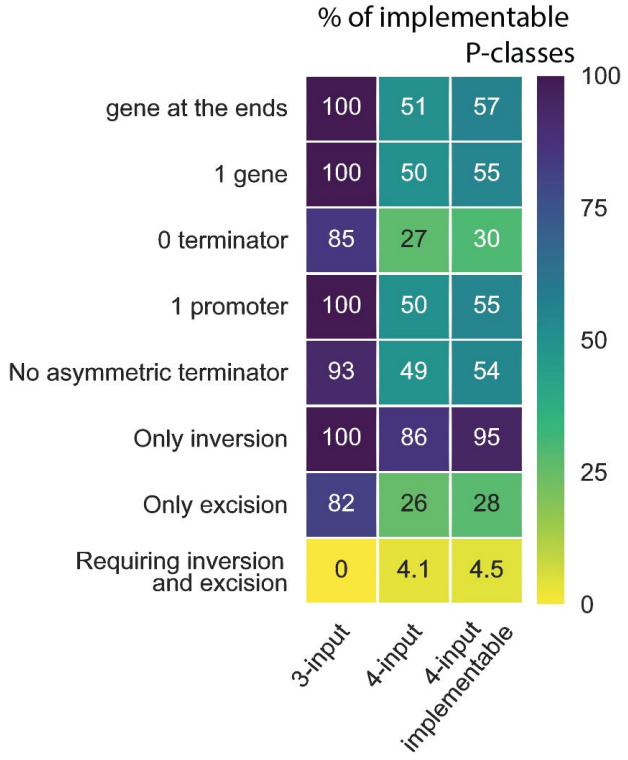


Figure 5: Percentage of P-classes implementable as a function of various restricting criteria. *The columns denote the different set of P-classes, such as 3-input, 4-input and 4-input implementable P-classes. Each line corresponds to the percentage of P-classes implementable with the corresponding design specification, the color scale represent their respective percentages, from yellow (0%) to dark blue (100%).*

Based on literature, we believe that the following criteria should be satisfied in order to obtain well-behaving logic devices:
1 - Not having two promoters facing each other, such as architectures respecting the no cross promotion constraint (see material and methods for criteria list).
2 - To have a reduced distance between the gene and the promoter leading to the expression of this gene.

In addition, in order to minimize differences in output level between the different input states, we recommend:
3 - To avoid having two genes expressed at the same time.
4 - To avoid having two promoters mediating the expression of one gene.

These criteria are not absolute, and sometimes cannot be all satisfied for implementing all logic functions. Experimental implementation and characterization of various architectures will ultimately be the only way to validate their functionality.

4 - A web-interface to sort architectures according to user-defined constraints.

Based on this large database, we created a user-friendly web-interface allowing the users to obtain all architectures implementing logic functions of their choice (Fig 6). For each architecture, the various specifications of the design are listed, such as the number of inputs, inversions, excisions, genes, terminators, and promoters, and the total number of parts. Additionally, the approximate length of the architecture is specified (calculated by considering as lengths: 20bp for a promoter, 40bp for a terminator and an integrase site, and 1 Kbp for a gene). We also specify if the architecture respects or not design criteria of interest, such as having the gene at the extremity of the construct (5' or 3'), the absence of asymmetric terminators, or the absence of promoters facing each other. Indeed, while we excluded facing promoters from element design, insertion of elements within functional structures can still result in architectures containing facing promoters. In all cases, the list of architectures can be re-ordered according to a criterion of choice. A filtering tool allows the user to selectively display specific architectures respecting a given set of criteria.

When the user selects a specific architecture, all the derived states of the architecture are also represented with the correspondence between sites and inputs. Additionally, all logic functions implementable with the same architecture (therefore P-equivalent functions) are accessible from this page.

A

List of architectures

http://recombinator.lirmm.fr

Showing 1-20 of 23 items.

#	Architecture	Dnf	Nb Genes	Nb Promoters	Nb Terminators	Nb Asymetric Terminators	Nb Parts	Gene At Ends	Cross Promotion Constraint	Length	Promoter-gene distance	Nb Inputs	Nb Inversions	Nb Excisions	Actions
1	((G))↵	0110	1	1	0	0	6	no	respected	1200	80	2	2	0	View
2	((↵))G	0110	1	1	0	0	6	yes	respected	1200	80	2	2	0	View
3	⊓((T))G	0110	1	1	1	1	7	yes	respected	1300	260	2	2	0	View
4	⊓([↵T]⊥)G	0110	1	2	2	2	9	yes	respected	1440	260	2	1	1	View
5	(G⊓)(↵Ɔ)	0110	2	2	0	0	8	no	respected	2240	1120	2	2	0	View
6	(↵Ɔ)(G⊓)	0110	2	2	0	0	8	no	respected	2240	1120	2	2	0	View
7	(Ɔ[G⊥])↵	0110	2	1	1	1	6	no	respected	2300	1080	2	1	1	View
8	(Ɔ(⊥)G)↵	0110	2	1	1	1	8	no	respected	2300	1220	2	2	0	View
9	(G⊓)[↵T]G	0110	2	2	1	1	9	no	respected	2340	220	2	1	1	View
10	⊓(⊥[TƆ])G	0110	2	1	2	2	9	no	respected	2400	1280	2	1	1	View
11	⊓((↵Ɔ)T⊥)G	0110	2	2	2	2	10	no	respected	2440	1320	2	2	0	View
12	Ɔ[⊥⊓][↵T]G	0110	2	2	2	2	10	yes	respected	2440	260	2	0	2	View
13	⊓(T⊥(↵Ɔ))G	0110	2	2	2	2	10	no	respected	2440	1320	2	2	0	View
14	(⊓(T)↵)G	0110	1	2	1	1	8	yes	violated	1340	260	2	2	0	View
15	(⊓[↵T])G	0110	1	2	1	1	6	yes	violated	1340	120	2	1	1	View
16	(↵Ɔ)[G⊥]↵	0110	2	2	1	1	9	no	violated	2340	1180	2	1	1	View
17	(G⊓)(T)G↵	0110	2	2	1	1	9	no	violated	2340	220	2	2	0	View
18	(↵Ɔ)(⊥)G↵	0110	2	2	1	1	9	no	violated	2340	220	2	2	0	View
19	⊓[TƆ][G⊥]↵	0110	2	2	2	2	10	no	violated	2440	1220	2	0	2	View
20	⊓Ɔ(⊥)[↵T]G	0110	2	2	2	2	10	no	violated	2440	360	2	1	1	View

B

Sequence	a	b	output
((G))↵	0	0	0
(◖Ɔ◗)↵	0	1	1
◖(Ɔ)◗↵	1	0	1
◖◖G◗◗↵	1	1	0

Figure 6: The recombinator web-interface. A. Example of search results on the recombinator web-interface *with as entry the XOR logic function, 23 architectures are found with here the 20 first architectures. This list can be sorted according to various criteria.* ***B.*** *The web-interface also allows the user to directly provide an architecture to obtain all its derived states and the implemented Boolean function.*

5 - A reduced set of 16 NP-equivalent architectures based on DNA inversion to implement all 3-input functions.

For most logic functions, the database provides many different possible architectures (with a median of 252, and a maximum of 1.4 million for the logic function in supplementary text 3.6). An important gap remains between theoretical designs and well-behaving biological devices, and experimentally testing all proposed architectures is challenging. We therefore wanted to identify a method enabling us to reduce the number of biological devices to be tested and optimized, while maximizing the number of implementable logic functions. We already used permutation classes to reduce the number of generated architectures by not assignating integrase sites to inputs. Here we built a minimization strategy based of NP-classes, which corresponds to class of logic functions which are equivalent by permutation and negation of inputs [24].

Permutation of inputs in integrase-based system is simply performed by permutation of the connection between integrase and inducible promoters. Importantly, negation is very easily performed in recombinase logic by using DNA inversion. For example, the NOT function

being the negation of the identity function, a NOT device in recombinase logic can be transformed into an IDENTITY by inverting the asymmetric terminator controlling the flow of RNA polymerase, and vice versa (Fig7A). In this simple case, the negation of the input in a logic function is performed by inversion of the part between integrase sites in inversion orientation. This approach is generalizable to more complex devices that use only DNA inversion. Consequently, starting from any all-inversion architecture, replacing LR sites by BP sites in the recombination intermediates leads to a new architecture implementing a specific P-class within the same NP-class (Fig7B).

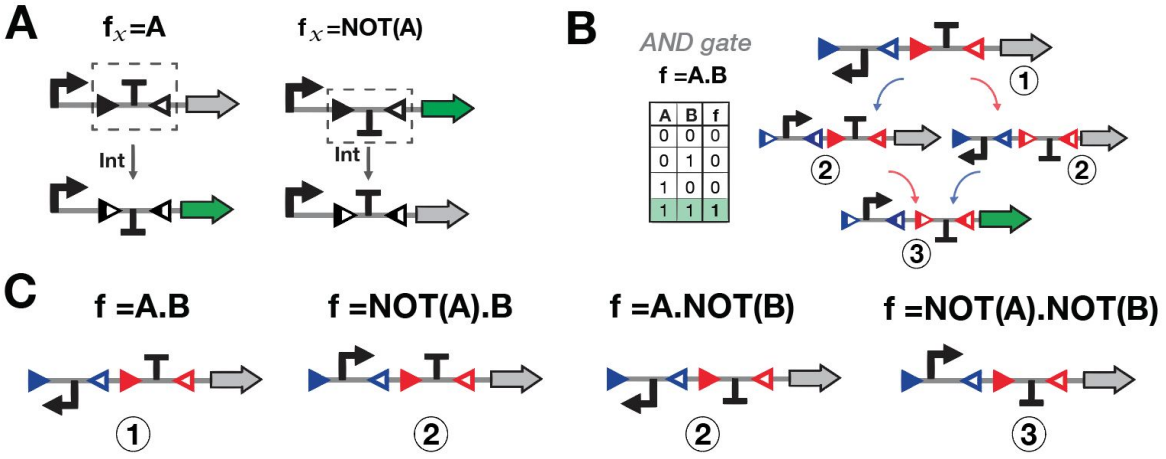


Figure 7: Implementation of NP-equivalent Boolean functions using inversion-based logic devices. (A) IDENTITY- and NOT- logic devices based on the inversion of an asymmetric terminator. (B) A 2-input AND gate based on the combination of promoter and terminator modules and its intermediate recombination states. (C) Implementation of all functions from the 2-input AND gate NP-class. Designs are based on the AND gate represented in B and correspond to the intermediate recombination states in which LR sites have been replaced by BP sites. The inversion of a single part of the device for A.B leads to either NOT(A).B or A.NOT(B), and the inversion of both parts to NOT(A).NOT(B) logic function.

We can thus reasonably assume that for a given NP-class, characterizing in all input states and optimizing a single logic device executing one function should provide a good estimate of the behavior of all derived logic devices implementing the NP-class. 16 and 380 NP-classes exists for 3-input and 4-input, respectively.

Because this approach is restricted to architectures using DNA-inversion, it is theoretically possible to optimize one architecture for each of the 16 three-input NP-classes and for 322 of the 380 four-input NP-classes.

Inversion-only architectures represent only 25.4% of the original set (4.6 million of architectures), thereby leading to a lower probability to find architectures respecting all the biological criteria listed above (Fig S9). We present in Figure 8 an example of architecture for each of the 16 3-input NP-classes, chosen according the our recommended criteria.

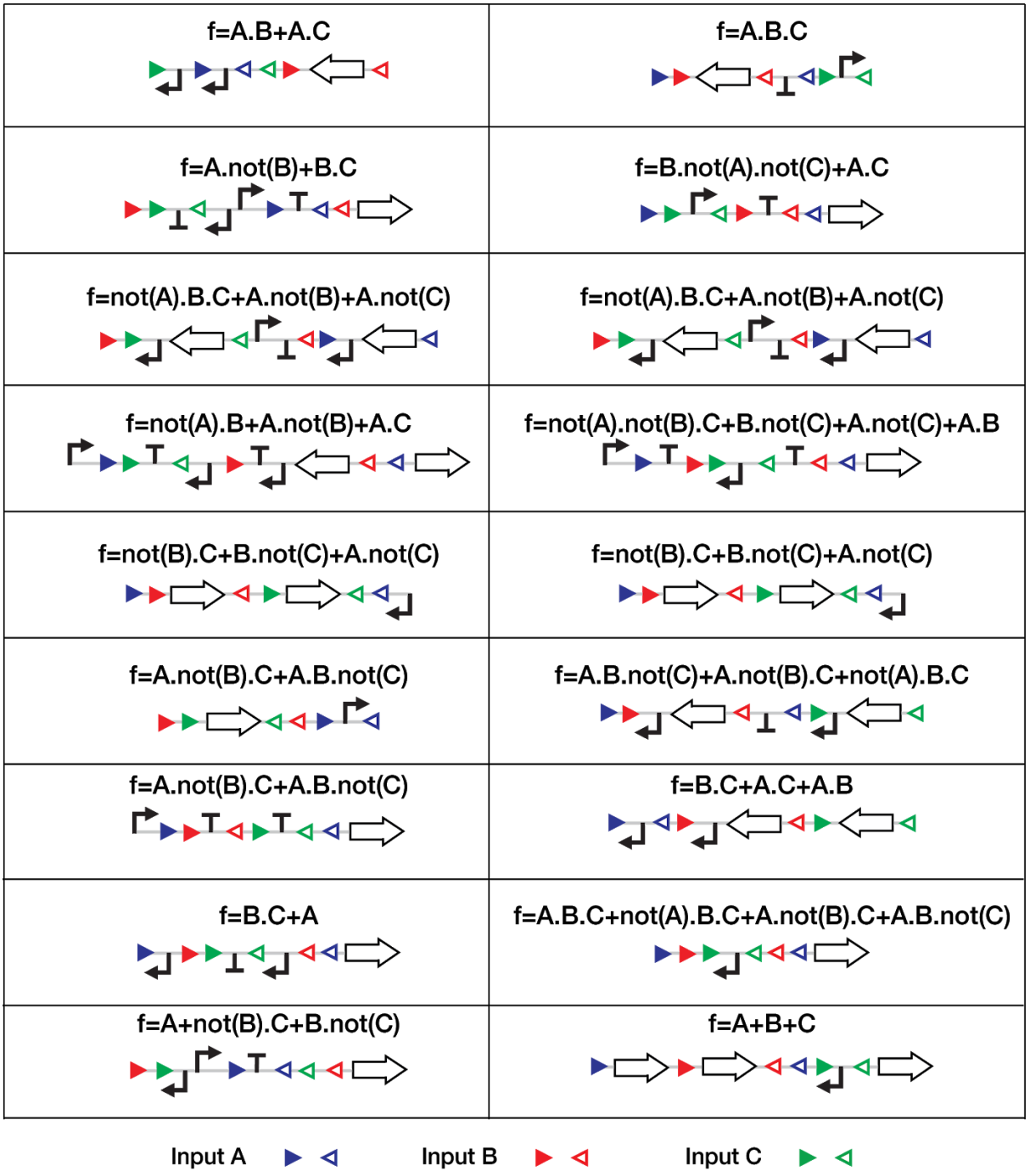


Figure 8: Selected architectures for the implementation of the 16 3-input NP-classes. Each box implements one NP-class. Here one Boolean function and a selected logic device implementing this Boolean function are depicted for each NP-class. Triangles correspond to integrase sites, blue is for variable A, red for B and green for C.

DISCUSSION

Here we took a systematic approach to explore the design space of recombinase logic devices. We devised a formal language to express and interpret the output state of genetic devices, along with several strategies allowing us to minimize the number of generated constructs to a non-redundant and irreducible set. We obtained a total of ~19 million architectures, to date the largest database of biological logic designs, implementing a total of 59,820 Boolean functions (corresponding to 3,610 P-classes).

We proved that all 3-input and 92% of 4-input Boolean functions are implementable in single cell using integrase-based devices within the constraints of using one integrase and one pair of recombination sites pair per input. Most P-classes can theoretically be implemented by hundreds of architectures. Not surprisingly, the number of possible architectures for a given P-class is inversely correlated with their complexity (i.e. the number of terms and literals per term).

We tested the impact of various design strategies on function implementability. Based on our design specifications, using DNA inversion is the most stringent requisite to reach the largest number of functions. We believe that this is due to the non-destructive nature of DNA inversion compared to DNA excision, and to the fact that DNA-inversion increases the semantic space accessible to a particular architecture. For example, symmetric functions (e.g. XOR), which are highly complex, are easily designed using DNA inversion and nesting pairs of sites. An important flexibility within the number of promoter, gene and terminator also allows to implement more functions.

To navigate within the millions of architectures, we constructed a web-interface allowing users to obtain all architectures implementing a logic function of interest with the possibility of sorting them according to specific design and biological constraints. This web interface will help biological logic designers to obtain minimized integrase-based logic circuits operating in single-cell. Yet, for each function, myriad architectures with highly-divergent designs are possible. Many will not be functional, and not all possible designs can be tested, making the obtention of working logic devices from the database challenging. To circumvent this issue, we reasoned that characterizing and optimizing a single inversion-based device per NP-class would approximate the characterization of all logic devices from this NP-class. This simplification reduces the number of devices to test to 16 for 3-input logic and to 380 for 4-input logic, of which 322 are implementable using only inversion. We provide an example set of the 16 architectures required for 3-input logic implementation.

For a given architecture, specific integrases, integrase site positions and orientations, and gene expression parts will have to be chosen. As highlighted in our previous work [12], integrase sites can affect the behavior of logic devices. Previous characterization of individual components coupled with mathematical modeling of part concatenations could be used to limit the number of trial-and-error circles.

A key contribution of our work is the establishment of a formal language for expressing and interpreting genetic construct functionality. The semantic we created, along with our

semantic concatenation rules, support unambiguous interpretation of genetic construct functionality. The reduced set of semantics proposed here could be extended to allow interpretation of larger set of genetic constructs, and support numerous synthetic biology applications.

While our minimization strategies allowed us to reduce the generation time to 1h30 for 4-input logic, this systematic generation method is unlikely to be applied for highest number of inputs due to the enormous computation time required (more than one year for 6 inputs). To obtain logic devices responding to a higher number of inputs, future work might focus on extracting from the database systematic design rules generalizable to N-inputs. Additionally, several characterization and optimization cycles could help determine rules for designing architectures responding to an increasing number of inputs. As an alternative approach, the 2-, 3- and 4-input designs proposed here could be concatenated to generate n-input designs.

This work lays the foundation for the systematic design of single-layer, compact recombinase logic devices operating in single-cell. By providing a user-friendly interface to navigate the design space of recombinase-based logic, we empower researchers and engineers to expand the use and design principles of this highly useful class of logic devices.

Acknowledgments:
We thank the synthetic biology group and members of the CBS and LIRMM for fruitful discussions.

Funding: Support was provided by an ERC Starting Grant “Compucell”, the INSERM Atip-Avenir program and the Bettencourt-Schueller Foundation. S.G. was supported by a Ph.D. fellowship from the French Ministry of Research and the French Foundation for Medical Research (FRM) FDT20170437282. The CBS acknowledges support from the French Infrastructure for Integrated Structural Biology (FRISBI) ANR-10-INSB-05-01.

Author contributions:
S.G. and J.B. designed the project. G.P., F.U. and M.L. designed the RECOMBINATOR algorithm. G.P. wrote the RECOMBINATOR algorithm and implemented the web server. S.G. and G.P. analyzed the data. S.G., G.P. and J.B. wrote the manuscript. All authors approved the manuscript.

Competing interests: The authors declare no competing interests.

REFERENCES

(1) Endy, D. Foundations for Engineering Biology. *Nature* **2005**, *438* (7067), 449–453.
(2) Chang, H.-J.; Voyvodic, P. L.; Zuniga, A.; Bonnet, J. Microbially Derived Biosensors for Diagnosis, Monitoring and Epidemiology. *Microbial biotechnology* **2017**. https://doi.org/10.1111/1751-7915.12791.
(3) Brophy, J. A. N.; Voigt, C. A. Principles of Genetic Circuit Design. *Nat. Methods* **2014**, *11* (5), 508–520.
(4) Nielsen, A. A. K.; Der, B. S.; Shin, J.; Vaidyanathan, P.; Paralanov, V.; Strychalski, E. A.; Ross, D.; Densmore, D.; Voigt, C. A. Genetic Circuit Design Automation. *Science* **2016**, *352* (6281), aac7341.
(5) Macia, J.; Manzoni, R.; Conde, N.; Urrios, A.; de Nadal, E.; Solé, R.; Posas, F. Implementation of Complex Biological Logic Circuits Using Spatially Distributed Multicellular Consortia. *PLoS Comput. Biol.* **2016**, *12* (2), e1004685.
(6) Win, M. N.; Smolke, C. D. Higher-Order Cellular Information Processing with Synthetic RNA Devices. *Science* **2008**, *322* (5900), 456–460.
(7) Green, A. A.; Kim, J.; Ma, D.; Silver, P. A.; Collins, J. J.; Yin, P. Complex Cellular Logic Computation Using Ribocomputing Devices. *Nature* **2017**, *548* (7665), 117–121.
(8) Miyamoto, T.; DeRose, R.; Suarez, A.; Ueno, T.; Chen, M.; Sun, T.-P.; Wolfgang, M. J.; Mukherjee, C.; Meyers, D. J.; Inoue, T. Rapid and Orthogonal Logic Gating with a Gibberellin-Induced Dimerization System. *Nat. Chem. Biol.* **2012**, *8* (5), 465–470.
(9) Dueber, J. E.; Yeh, B. J.; Chak, K.; Lim, W. A. Reprogramming Control of an Allosteric Signaling Switch through Modular Recombination. *Science* **2003**, *301* (5641), 1904–1908.
(10) Bonnet, J.; Yin, P.; Ortiz, M. E.; Subsoontorn, P.; Endy, D. Amplifying Genetic Logic Gates. *Science* **2013**, *340* (6132), 599–603.
(11) Weinberg, B. H.; Pham, N. T. H.; Caraballo, L. D.; Lozanoski, T.; Engel, A.; Bhatia, S.; Wong, W. W. Large-Scale Design of Robust Genetic Circuits with Multiple Inputs and Outputs for Mammalian Cells. *Nat. Biotechnol.* **2017**, *35* (5), 453–462.
(12) Guiziou, S.; Mayonove, P.; Bonnet, J. Hierarchical Composition of Reliable Recombinase Logic Devices. *Nat. Commun.* **2019**, *10* (1), 456.
(13) Podhajska, A. J.; Hasan, N.; Szybalski, W. Control of Cloned Gene Expression by Promoter Inversion in Vivo: Construction of the Heat-Pulse-Activated Att-nutL-P-Att-N Module. *Gene* **1985**, *40* (1), 163–168.
(14) Bonnet, J.; Subsoontorn, P.; Endy, D. Rewritable Digital Data Storage in Live Cells via Engineered Control of Recombination Directionality. *Proc. Natl. Acad. Sci. U. S. A.* **2012**, *109* (23), 8884–8889.
(15) Merrick, C. A.; Zhao, J.; Rosser, S. J. Serine Integrases: Advancing Synthetic Biology. *ACS Synth. Biol.* **2018**, *7* (2), 299–310.
(16) Bischof, J.; Maeda, R. K.; Hediger, M.; Karch, F.; Basler, K. An Optimized Transgenesis System for Drosophila Using Germ-Line-Specific φC31 Integrases. *Proc. Natl. Acad. Sci. U. S. A.* **2007**, *104* (9), 3312–3317.
(17) Hou, L.; Yau, Y.-Y.; Wei, J.; Han, Z.; Dong, Z.; Ow, D. W. An Open-Source System for in Planta Gene Stacking by Bxb1 and Cre Recombinases. *Mol. Plant* **2014**, *7* (12), 1756–1765.
(18) Lakso, M.; Sauer, B.; Mosinger, B., Jr; Lee, E. J.; Manning, R. W.; Yu, S. H.; Mulder, K. L.; Westphal, H. Targeted Oncogene Activation by Site-Specific Recombination in Transgenic Mice. *Proc. Natl. Acad. Sci. U. S. A.* **1992**, *89* (14), 6232–6236.
(19) Pichel, J. G.; Lakso, M.; Westphal, H. Timing of SV40 Oncogene Activation by Site-Specific Recombination Determines Subsequent Tumor Progression during Murine Lens Development. *Oncogene* **1993**, *8* (12), 3333–3342.

(20) Chiu, T.-Y.; Jiang, J.-H. R. Logic Synthesis of Recombinase-Based Genetic Circuits. *Sci. Rep.* **2017**, *7* (1), 12873.
(21) Boque-Sastre, R.; Soler, M.; Oliveira-Mateos, C.; Portela, A.; Moutinho, C.; Sayols, S.; Villanueva, A.; Esteller, M.; Guil, S. Head-to-Head Antisense Transcription and R-Loop Formation Promotes Transcriptional Activation. *Proc. Natl. Acad. Sci. U. S. A.* **2015**, *112* (18), 5785–5790.
(22) Uesaka, M.; Nishimura, O.; Go, Y.; Nakashima, K.; Agata, K.; Imamura, T. Bidirectional Promoters Are the Major Source of Gene Activation-Associated Non-Coding RNAs in Mammals. *BMC Genomics* **2014**, *15*, 35.
(23) Chizzolini, F.; Forlin, M.; Cecchi, D.; Mansy, S. S. Gene Position More Strongly Influences Cell-Free Protein Expression from Operons than T7 Transcriptional Promoter Strength. *ACS Synth. Biol.* **2014**, *3* (6), 363–371.
(24) Jaakko T. Astola, R. S. S. *Fundamentals of Switching Theory and Logic Design*; Springer, Ed.; 2006.
(25) Hill, F. J.; Peterson, G. R. *Introduction to Switching Theory and Logical Design*; philpapers.org, 1981.
(26) Guiziou, S.; Ulliana, F.; Moreau, V.; Leclere, M.; Bonnet, J. An Automated Design Framework for Multicellular Recombinase Logic. *ACS Synth. Biol.* **2018**, *7* (5), 1406–1412.
(27) Chen, Y.-J.; Liu, P.; Nielsen, A. A. K.; Brophy, J. A. N.; Clancy, K.; Peterson, T.; Voigt, C. A. Characterization of 582 Natural and Synthetic Terminators and Quantification of Their Design Constraints. *Nat. Methods* **2013**, *10* (7), 659–664.
(28) Brewka, G. Artificial Intelligence—a Modern Approach by Stuart Russell and Peter Norvig, Prentice Hall. Series in Artificial Intelligence, Englewood Cliffs, NJ. *The Knowledge Engineering Review*. 1996, p 78. https://doi.org/10.1017/s0269888900007724.
(29) Georges, G. Bases de Données: Objet et Relationnel. *Éditions Eyrolles* **1999**.

BOX1:

DEFINITIONS

Structure: a sequence of bracket and parenthesis corresponding to recombinase site arrays.
Semantic: gene expression function. The elementary semantics are: promote, terminate, encode, neutral. All sequences have a forward and reverse semantic, written as forward semantic / reverse semantic.
Functional structure: a structure with a semantic between each bracket and parenthesis. A functional structure is composed of 2N brackets and parentheses and 2N+1 semantics, with N the number of inputs.
Part: a promoter, gene or terminator.
Element: composition of parts enabling the “simplest” implementation of a specific forward and reverse semantic. These elements are selected to be composed from a reduced number of parts, to avoid facing promoters, and to reduce the space between a promoter and its transcribed gene.
Architecture: functional structure in which semantics have been replaced by their corresponding elements.
Biological device: an architecture with assignation of input to the parenthesis (integrase sites).
Boolean function: a function with binary variables and a binary output which can be expressed as a propositional formula.
P-class: permutation class, a set of Boolean functions equivalent by permutation of inputs.
NP-class: negation and permutation class, a set of Boolean functions equivalent by permutation and negation of inputs.
Non-redundant: a Boolean function is non-redundant if it cannot be simplified into a Boolean function with a reduced number of inputs. A non-redundant functionalized structure or architecture implement a Boolean function which cannot be simplified into a Boolean function with a number of inputs lower than the number of recombination pairs.
Symmetric: an object is symmetric if it is the reverse complement of itself.
Chiral: two objects (e.g. architectures, structures, functional structures, devices) are chirals if one object is the reverse complement of the other one.
Irreducible: a functionalized structure is irreducible if changing a semantic to a semantic implemented with a reduced number of parts changes the implemented Boolean function (or P-class).
Atomic recombination: an atomic recombination (either excision or inversion) corresponds to a recombination which do not contain other recombination inside it.

Methods

1 - Algorithm for architecture generation.

The generation algorithm is summarized in Fig S3. To generate the architecture database, we first generated all the structures. Structures correspond to well-balanced sequences of two sets of parentheses ([ ] and ( )) corresponding therefore to all the Dyck word of size 2N (with N the number of inputs) with the two sets of parentheses. For non symmetric structures, we removed one of the two structures of a chiral pair.

The next step of the algorithm consists of assigning a semantic between parentheses. We defined variables as the space between the parentheses, each structure is composed of 2N+1 variables. We defined a semantic domain for each variable, corresponding to a list of semantic which can be assigned to this variable. Domains are composed of either all possible 26 semantics or a reduced set defined to reduce the generation of reducible functionalized structures (see supplementary text 3.2).

For semantic assignation, we used a backtracking algorithm[28]. Briefly, a semantic is assigned to a variable, constraints are evaluated to determine if the semantic is useful. If a constraint is violated, a new track is followed, then the next semantic of the domain is assigned and tested. If no constraint is violated, the assignation algorithm pass to the following variables. All constraints are defined in supplementary text 3.2, the objective of these constraints being that each semantic in both forward and reverse orientations is useful in at least the functionalized structure or one of the derived structures.

When a complete assignation is obtained, the irreducibility of the functionalized structure is tested. To check it, each semantic is replaced with a less complex semantic (see supplementary text 3.2), as an example, fGP is replaced by fG, fP and fN. If the obtained structures implement the same logical function than previously, the initial functionalized structure is reducible.

The truth tables of irreducible functionalized structures are computed by generating all derived structures and computing the semantic of each structure and derived structure. By arbitrarily assigning each parenthesis to an input, one possible Boolean function is obtained and its redundancy can be tested. If the Boolean function cannot be simplified into a function with a reduced number of inputs, the functionalized structure as being non-redundant is saved with its associated Boolean function corresponding to one of the Boolean functions of the implemented P-class.

We implemented the generation algorithm in C++ 17 with Boost library (code available in git directory: https://bitbucket.org/Guigui_PL/genetixV2). We performed the generation for up to four inputs using a high performance computer with a processor Intel Xeon E5-2680 v4, 2.40GHz, 14 physical cores, and with 128Go RAM DDR4 2400Mhz. The generation took around 1h30 for 4 inputs.

2 - Database creation and Recombinator web-interface.

The database construction is decoupled from the generation as the used high performance computer does not have any database management system. From the text files containing the functionalized structures, we created a database satisfying the two normalization form[29] which permit to not have redundancy in the database. We decomposed the database in five tables: Boolean functions, P-classes, semantics, structures and functionalized structures (Fig S10). Each table contains in addition to its primary key the filtering criteria depending on this key, such as for the structure table, the number of excisions and number of inversions.

We implemented this database generation algorithm in C++ 14 with Boost and libpq++ libraries. The database management system is PostgreSQL (code available in git directory: https://bitbucket.org/Guigui_PL/genetix/src).

The web interface is built with the PHP Framework Yii2. In the web-interface, users fill in a Boolean function of interest either as formulae of propositional logic or as a binary number corresponding to the truth table. A list of architectures is obtained, architectures can be sorted and filtered according to a list of criteria defined in the following section. Users can select a specific architecture and obtain all corresponding information such as the derived architectures.

3 - Analyse of the architecture database using python scripts.

From the database of 19 millions of architectures, a file was generated for each P-class: an example of function, the total number of architectures implementing this P-class and the number of architectures respecting various criteria.
Following is the detailed list of criteria used in this work and their precise definition:

Filtering criterion	**Filtering type**	**Definition**
No cross promotion	Boolean	For derived architecture with fX semantic, no promoter in opposite orientation of the promoter transcribing the gene is positioned between the promoter and its transcribed gene and the sequence in 3' of the gene does not have a -/fP semantic.
No asymmetric terminator	Boolean	All fT/- and -/fT are replaceable by fT/fT without changing the semantic of the architecture and derived architectures
# functional asymmetric terminator	From 0 to # terminators	Number of terminator which are not replaceable by a double terminator without changing the semantic of the architecture and derived

		architectures
Gene at extremities	Boolean	True if fG/- semantic as last variable or -/fG semantic as first variable and no other fG semantic
# terminators	From 0 to 2^{N+1}	Number of terminator parts in the architecture, such as of fT/- and -/fT semantics
# inversion	From 0 to # of inputs	Number of inversion in the architecture, such as of pair of ( )
# excision	From 0 to # of inputs	Number of excision in the architecture, such as of pair of [ ]
# promoters	From 1 to 2^{N+1}	Number of promoter parts in the architecture, such as of fP/-, -/fP, fX, fGP/-, and -/fGP semantics
# genes	From 1 to 2^{N+1}	Number of gene parts in the architecture, such as of fG/-, -/fG, fX, fGP/-, and -/fGP semantics
Architecture size	Integer	Integer corresponding to an approximate size of the architecture considering a gene as 1kb, a promoter as 20bp, a site as 40bp, and a terminator as 20bp.
Distance P-G	Integer from 0 to (architecture size - 1020)	For one derived architecture with fX semantic (expression of a gene), the distance corresponds to the minimum distance between the promoter and its transcribed gene (as several promoter can promote the transcription of a gene). The distance of the architecture corresponds to the maximum distance between all derived architectures with fX semantic.
Low distance P-G	Boolean	Architecture with distance P-G lower than 1000

From this file, python scripts were used to extract the percentage table (Fig5) and generated P-class cluster and distribution plots (Fig4, FigS4-S5-S6). For distribution plots, histograms were generated with 20 bins for P-class clusters and 40 bins for histograms corresponding to all P-classes.

Additionally for the database, a file containing the full list of architectures and architecture characteristics was generated. This file was used to generate the two-by-two constraint tables (FigS7-S8-S9) using an automated python script.

4 - Selection of architectures for the 16 3-input NP-classes.

To select one architecture based on inversion per NP-class, the web interface was used. We first generated the list of NP-classes and associated P-classes with one Boolean function per P-classes. From this list, we randomly selected a single Boolean function per NP-class and searched in the database for architectures implementing this Boolean function. We sorted this list of architectures according to the number of excision in decreasing order to select architectures based on inversion only. We then sorted architectures according to the following two boolean constraints: no cross promotion and the low distance between the promoter and the gene. We selected one or several architectures following as much as possible these two criteria and having the lowest number of parts. This selection is not absolute and for most Boolean functions corresponding to one NP-class several architectures existand could be selected.

Supplementary Information: Exploring the design space of compacted recombinase logic circuits.

Sarah Guiziou*†1#, Guillaume Perution-Kihli†2, Federico Ulliana[2], Michel Leclere[2], and Jerome Bonnet*[1]

[1]Centre de Biochimie Structurale, INSERM U1054, CNRS UMR5048, University of Montpellier, France.
[2]Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier (LIRMM). CNRS UMR 5506, University of Montpellier, France.
†These authors contributed equally to this work
*To whom correspondence should be addressed: guiziou.sarah@gmail.com, jerome.bonnet@inserm.fr
#Current address: Department of Biology, University of Washington, Seattle, Washington 98195, USA

These supplementary materials contain:
-Supplementary Tables S1 to S4.
-Supplementary Figures S1 to S10.
-Supplementary Texts.

1 Supplementary Tables

Design Specifications	**Motivations**
One pair of sites/integrase	- Reduce problems of non-specific recombination. - Reduces genetic instability. - Reduces difficulties to synthesize. - Reduces the size of the circuit.
One integrase by input	- Reduce the number of orthogonal integrase needed. - Reduces metabolic load to the cell. - Reduces the size of the circuit.
Regulation of transcription using promoters and terminators	- Simple set of tools. - Two parts for opposite behaviors.

Supplementary Table 1: Specification of our logic design.

5' to 3'	**fN**	**fP**	**fT**	**fG**	**fGP**	**fX**
fN	fN	fP	fT	fG	fGP	fX
fP	fP	fP	fT	fX	fX	fX
fT	fT	fP	fT	fT	fP	fX
fG	fG	fGP	fG	fG	fGP	fX
fGP	fGP	fGP	fG	fX	fX	fX
fX	fX	fX	fX	fX	fX	fX

Supplementary Table 2: Simplification of all possible concatenations of the six semantics two by two, the row corresponds to the semantic in 5' and the column the semantic in 3'.

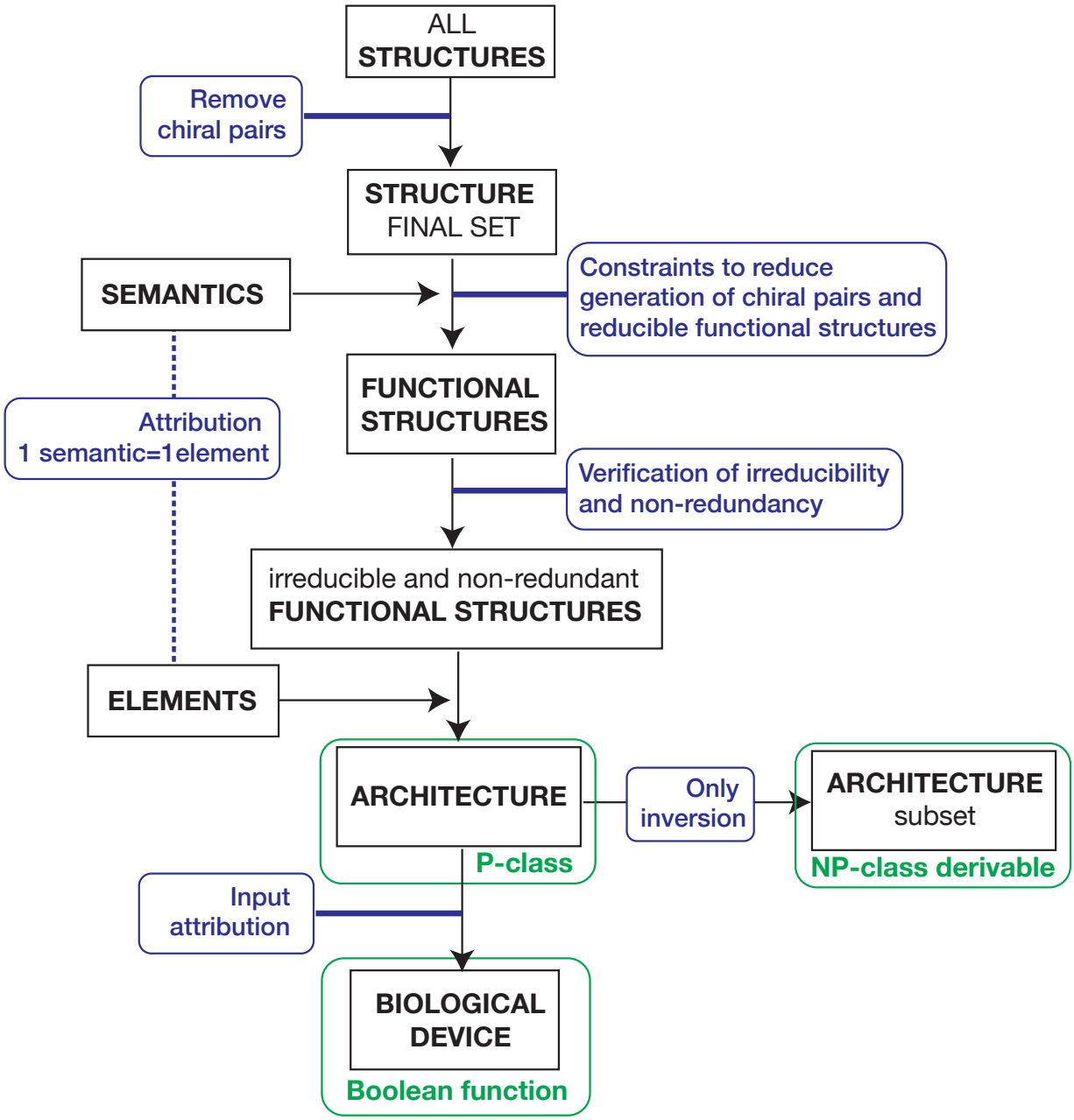
	1 input	**2 inputs**	**3 inputs**	**4 inputs**	**5 inputs**
# functions with strictly N inputs	2	10	218	64,594	$4.3\ 10^9$
# P-classes with strictly N inputs	2	8	68	3904	$3.7\ 10^7$
# NP-classes with strictly N inputs	1	5	16	380	1,227,756

Supplementary Table 3: Number of functions, P-classes, NP-classes for a given number of inputs.

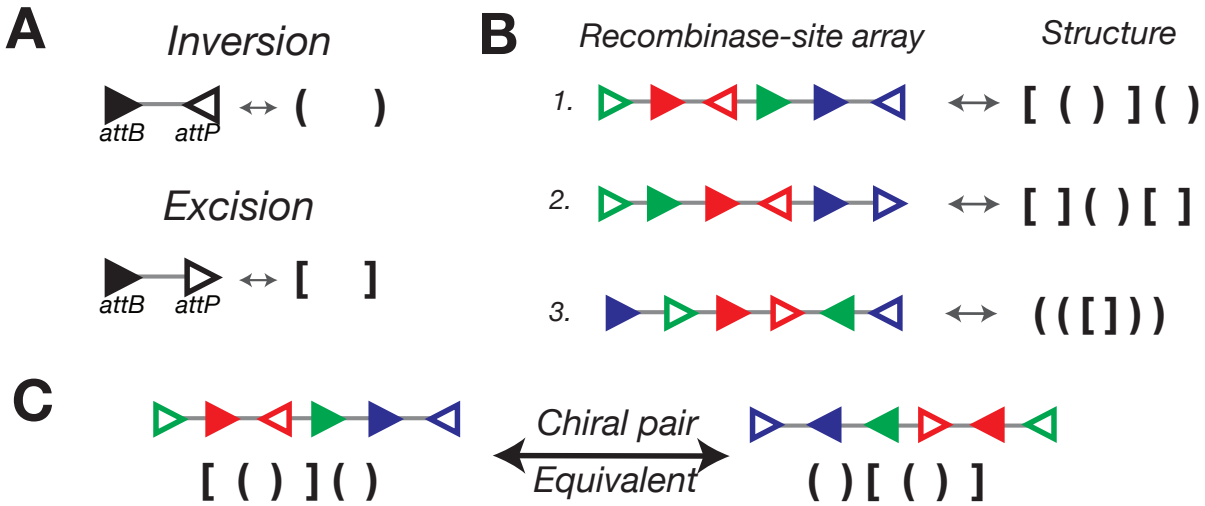
	> 1 term	**> 2 terms**	**> 3 terms**	**> 4 terms**	**> 5 terms**	**> 6 terms**	**> 7 terms**
> 3.5 literals	25.1% (223)	26.7% (210)	36.1% (155)	38.7% (93)	60.6% (33)	53.3% (15)	0% (2)
> 3 literals	20.7% (1098)	21.3% (1065)	25% (908)	29.6% (514)	37.2% (188)	51.8% (56)	0% (7)
> 2.5 literals	13.3% (2816)	13.6% (2750)	16.7% (2241)	23.7% (1181)	39.9% (336)	53.3% (75)	30% (10)
> 2 literals	10.4% (3599)	10.66% (3509)	13% (2875)	20.3% (1381)	36.7% (365)	50% (80)	30% (10)
> 1.5 literals	9.64% (3879)	9.9% (3775)	12.3% (3047)	19.7% (1418)	36.2% (370)	50% (80)	30% (10)
> 1 literal	9.6% (3894)	9.87% (3790)	12.3% (3053)	19.7% (1418)	36.2% (370)	50% (80)	30% (10)

Supplementary Table 4: Percentage of P-classes not implementable for P-classes of specific number of terms and mean number of literal per terms. Each cell corresponds to the P-classes with a number of term higher than the corresponding column value and a mean number of literal per terms higher than the corresponding line value. And in each cell is filled with the percentage of not implementable P-classes on these specific P-classes and bellow the number of not implementable P-classes.

2 Supplementary Figures

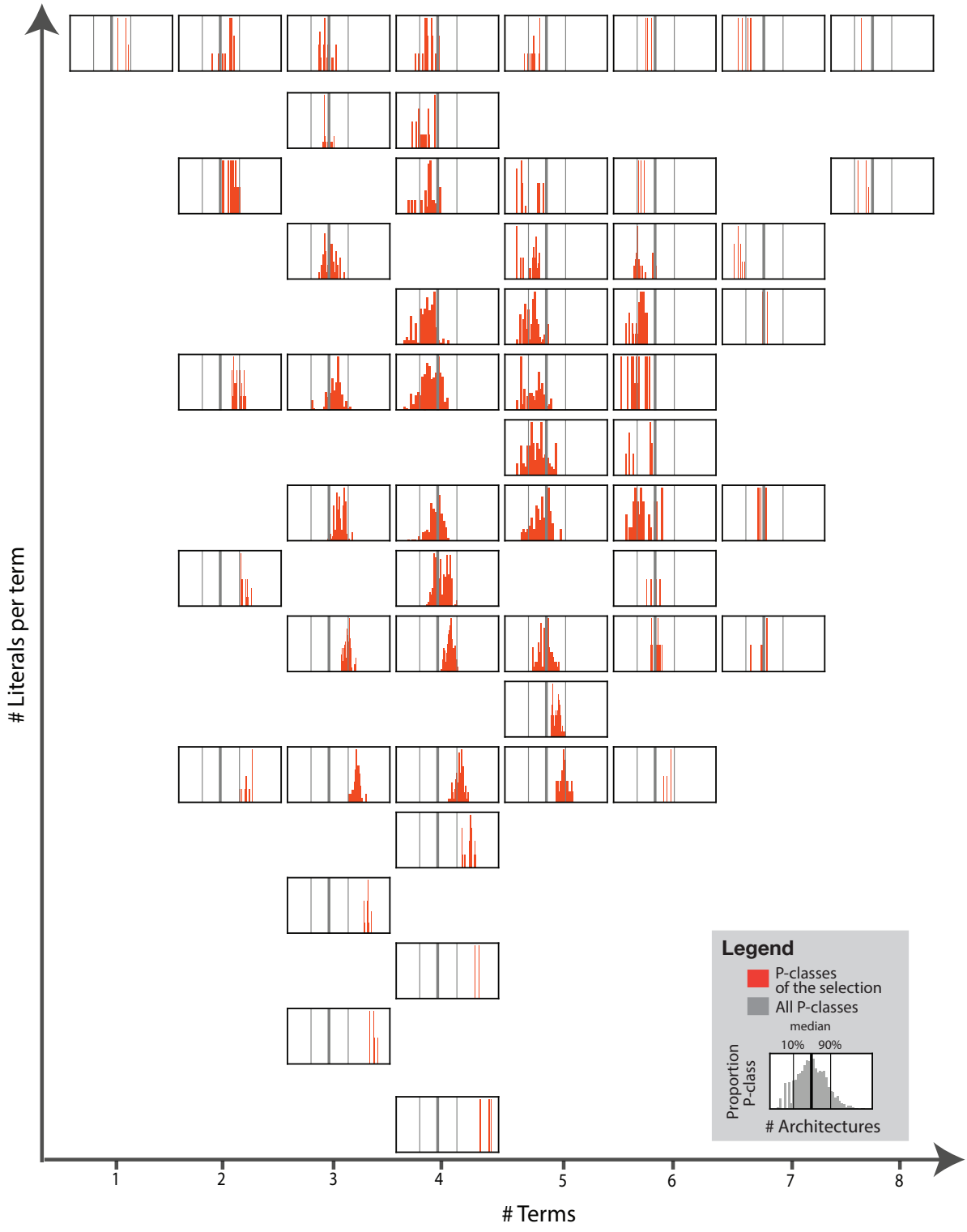


Supplementary Figure 1: General workflow of architecture generation.

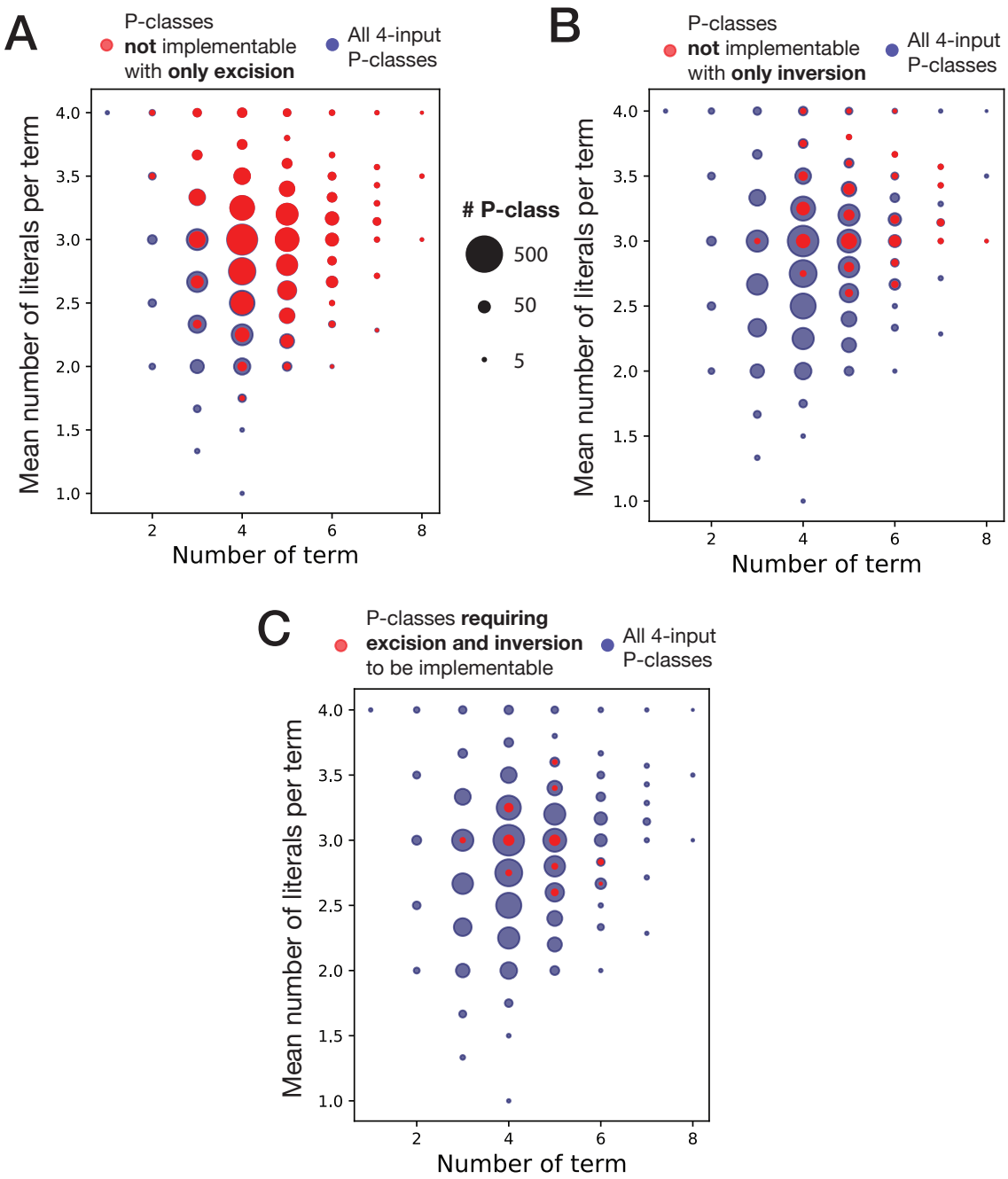


Supplementary Figure 2: **Correspondence between recombinase-site array and structure** A - Integrase sites in inversion orientation are represented by parenthesis while integrase sites in excision orientation are represented by brackets. B - Three examples of recombinase-site array and the corresponding structure. In recombinase-site array, each color corresponds to a different input. In the structure, correspondence between parenthesis and input is not specified. C - Example of two chiral structures and their corresponding recombinase-site arrays.

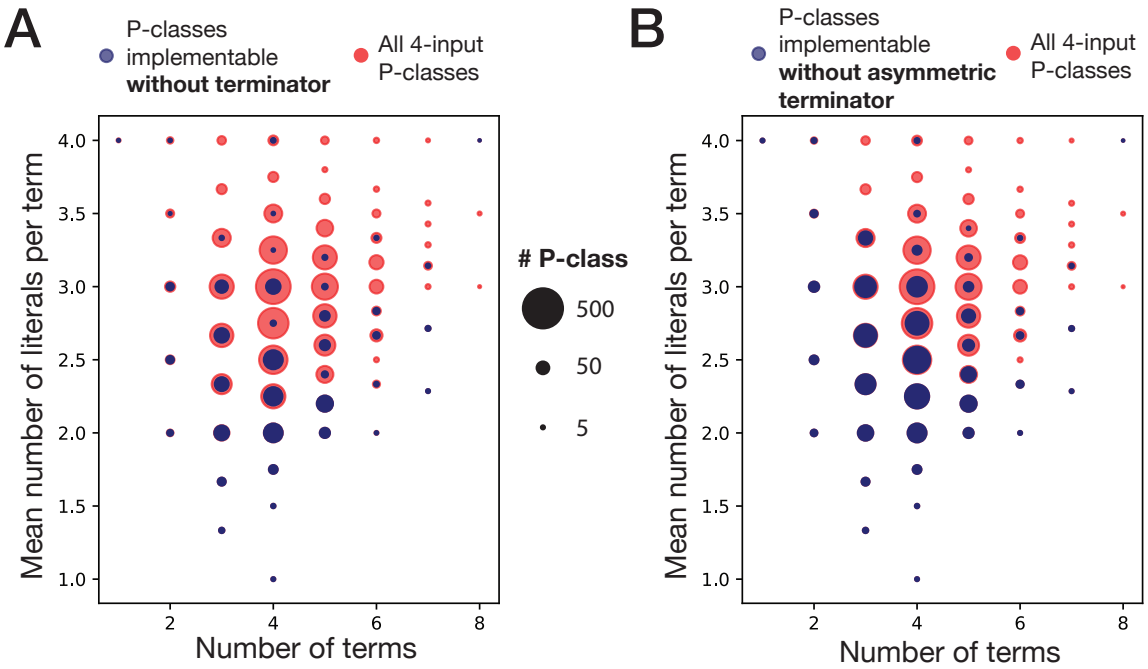
Supplementary Figure 3: **Irreducible architecture** A - Example for two semantics of the element selection, two examples of part composition are represented, the one selected as element is surrounded by a red rectangle. For the semantic fP/fN the selection is performed to reduce the number of biological parts while for the semantic fP/fG is performed to avoid cross promotion. B - Two architectures are represented, both implement the same P-class, one being reducible and the other one irreducible.



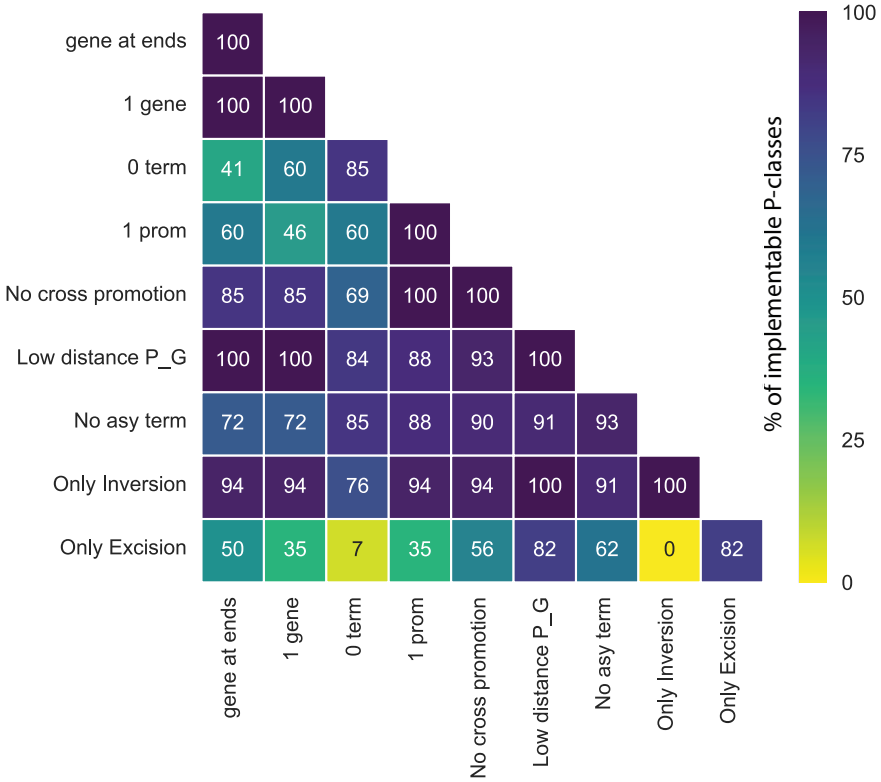
Supplementary Figure 4: **Distribution of the number of architecture for the implementation of P-classes with specific number of literal per terms and terms.** Each plot is the distribution of the number of possible architectures (in orange) for a specific set of P-classes; P-classes with a specific number of terms and literals per term (from the lowest in the bottom left to the highest in the top right). These distributions were obtain from a bin of 20. In the legend, the distribution for all P-classes is represented (with a bin of 40). In all graphs, the 10-percentile, the median and the 90-percentile of the distribution of all P-classes is shown in grey.



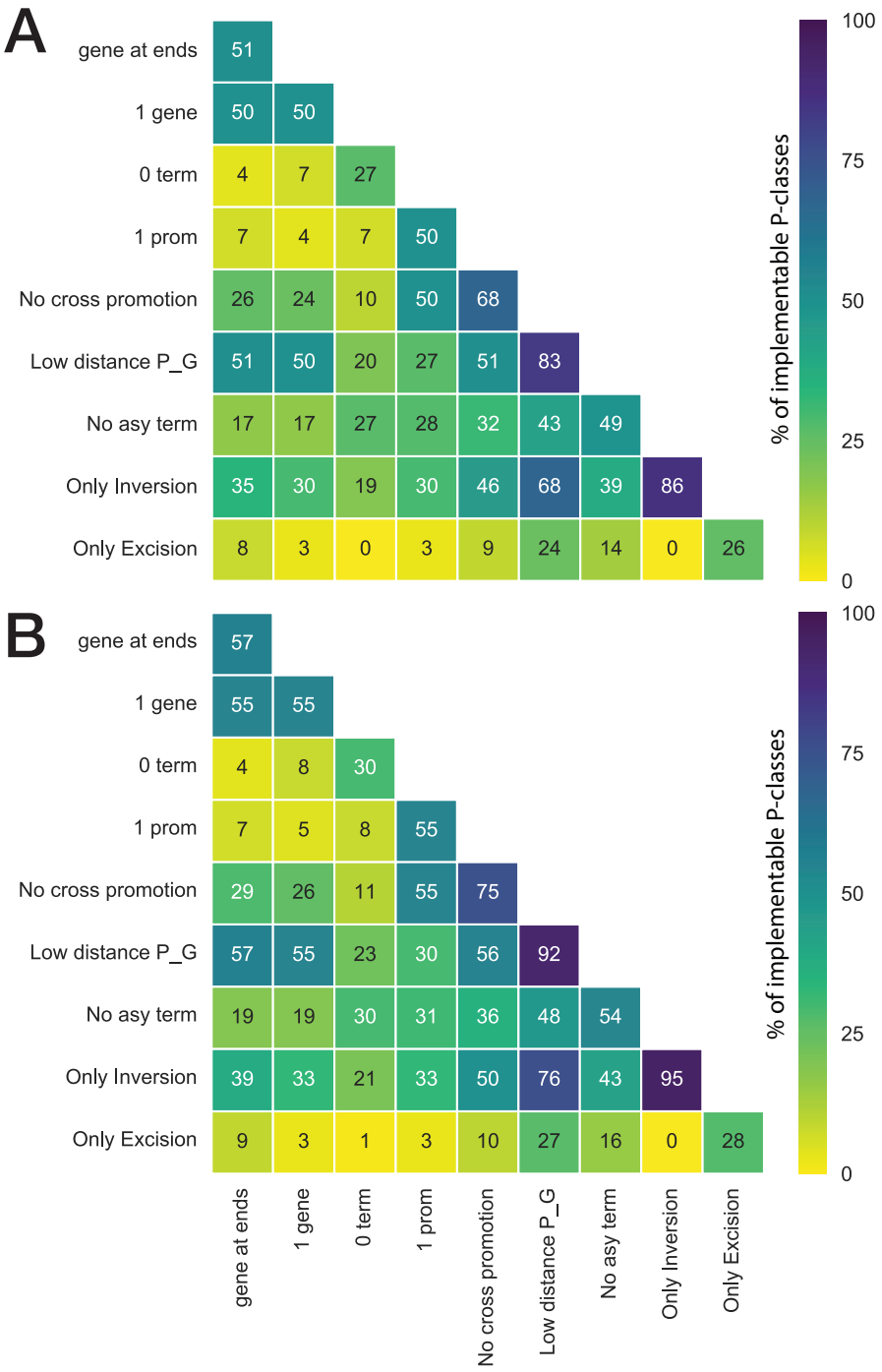
Supplementary Figure 5: **Implementability of P-classes with only excision and inversion according to their complexities.** In both plots, P-classes with a specific number of terms and of literals per term are represented in the corresponding position in the plot by a point with the diameter being proportional in logarithm to the number of P-classes. The blue points are for all 4-input P-classes and the red points are for P-classes not implementable with only excision in (A), with only inversion in (B), and P-classes requiring both inversion and excision to be implementable in (C).



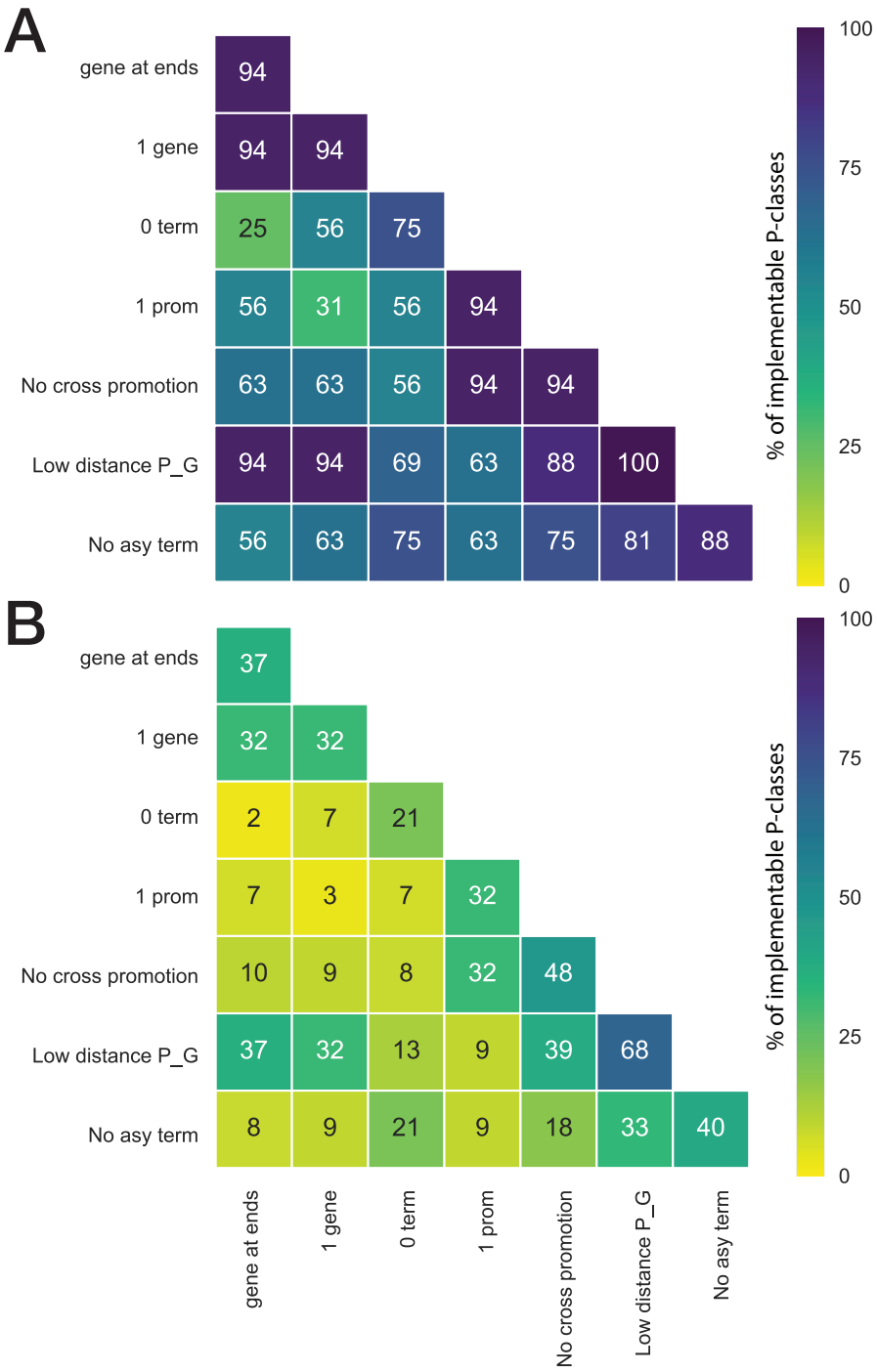
Supplementary Figure 6: **Implementability of P-classes without terminator or without asymmetric terminator.** In both plots, P-classes with a specific number of terms and of literals per term are represented in the corresponding position in the plot by a point with the diameter being proportional in logarithm to the number of P-classes. The blue points are for 4-input P-classes implementable without terminators in (A) and without asymmetric terminator in (B), and the orange points are for all 4-input P-classes.



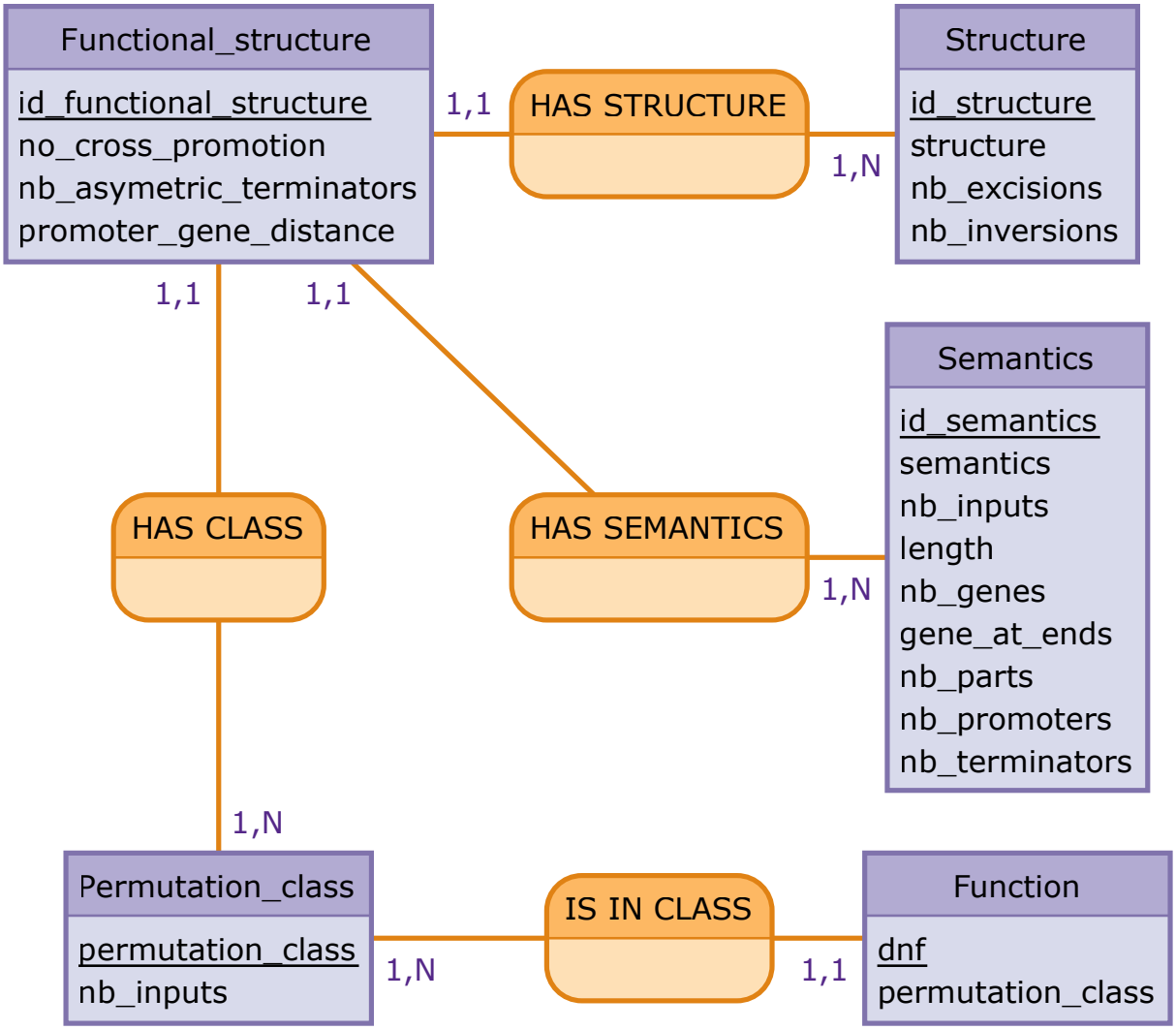
Supplementary Figure 7: **Percentage of 3-input P-classes implementable with each constraint and each two-by-two constraint composition.** Each cell corresponds to the percentage of 3-input P-classes implementable with the constraints corresponding to the specific line and column. The color of the cell is related to this percentage, from yellow with 0% to dark blue with 100%. The constraints are explained in the material and methods.



Supplementary Figure 8: **Percentage of 4-input P-classes implementable with each constraint and each two-by-two constraint composition.** (A) Percentage on all 4-input P-classes and (B) percentage on the 4-input implementable P-classes. Each cell corresponds to the percentage of implementable P-class with the constraints corresponding to the specific line and column. The color of the cell is related to this percentage, from yellow with 0% to dark blue with 100%. The constraints are explained in the material and methods.



Supplementary Figure 9: **Percentage of P-classes implementable with only inversion and with each two-by-two constraint composition.** (A) Percentage on all 3-input P-classes and (B) percentage on all 4-input P-classes. Each cell corresponds to the percentage of implementable P-class with only inversion and the constraints corresponding to the specific line and column. The color of the cell is related to this percentage, from yellow with 0% to dark blue with 100%. The constraints are explained in the material and methods.



Supplementary Figure 10: **Entity-relationship model of the database.** The database is composed of five tables: Boolean functions, P-class, semantics, structure and functionalized structures (corresponding to the purple rectangles). In each rectangle, all variables are listed with the primary key underlined. Their relations are also represented by line with a name specified and either a 1,1 or 1,N relationship.

3 Supplementary Texts

3.1 Mathematical formalization of the syntax and semantic of architectures

3.1.1 Syntax of an architecture

An architecture is an expression obtained by combining primitives by concatenation or by nesting an expression E between two marks of an excision or inversion reaction. The *excision* marks are denoted by a pair of square brackets $[E]$ while the *inversion* marks are denoted by a pair of round brackets $()$. The empty expression is denoted by ϵ and EE denotes the concatenation. The design language of such architecture is defined by the following grammar:

$$E \ ::= \ \epsilon \ | \ S \ | \ EE \ | \ [\ ] \ | \ (\ )$$

$$S \ ::= \ \mathcal{F} \ | \ \mathcal{L} \ | \ \top \ | \ \bot \ | \ G \ | \ \mathfrak{O}$$

An architecture without pair of sites is called an element.

3.1.2 Semantics of an architecture

Every biological part has a function, also called semantic. The function of a promoter denoted by fP, is the initiation of transcription. The function of a terminator is to terminate transcription, denoted by fT. The function of a gene is to encode for a RNA/protein, denoted by fG. Also, each part has an orientation which can be either forward or reverse (Fig1C).

The semantic of an architecture (that is, whether it expresses a gene) is a function whose input is a set of active recombinases and the output is a transcriptional value. The transcriptional value is a couple (fF,fR) where fF denotes the semantic of the device in the forward direction and fR denotes the semantic of the device in the reverse direction.

We start by defining the transcriptional semantic of biological parts and elements, as this is independent from any recombinase activation. The transcriptional semantic of a biological part in forward orientation is a couple (f,fN) where f denotes is its semantic (that is, fP or fT or fG) and fN denotes the neutral function meaning that no action is performed. For instance, (fP/fN) is the transcriptional semantic of an architecture reduced to a single forward promoter (Fig1D). In a similar way, the transcriptional semantic of a biological part in reverse orientation is a couple (fN,f) where again f is the function of the biological part.

An element is a concatenation of biological parts. The transcriptional semantic of an elementary sequence is thus given by the left-to-right composition of the semantics of its biological parts in forward orientation together with the right-to-left composition of the semantics of its biological parts in reverse orientation. The composition of semantics of biological parts yields two new semantics, namely the expression of a gene (denoted by fX) and the encapsulation of both encoding and promotion (denoted by fGP). Concretely, the semantic fX is given by the

concatenation of promoter with a gene, while the semantic fGP is given by the concatenation of a gene with a promoter.

Importantly, no other function is needed to express the transcriptional semantic of an elementary device. This means that, whatever the size of an element (i.e. the number of its biological parts), its transcriptional semantic is a couple of values (fF,fR) where both fF and fR are one of fN, fP, fT, fG, fGP, fX. Table1C presents the semantics of the composition of the biological parts functions. This means that any element has exactly one among 36 possible transcriptional semantic. Nevertheless, as our objective is too determine the gene expression state of an architecture, all couples (fF,fR) containing at least one fX function are considered as equivalent. In this regard, there are only 26 possible transcriptional semantic for an element.

The transcriptional behaviour of an architecture takes also into account the set of recombinases that are active. Every active pair of sites induces a transformation of the DNA sequence, which can be performed in any order. Once all transformations have been performed, the architecture cannot evolve anymore, and can therefore be seen as an element whose transcriptional semantic can be computed as described before.

The transcriptional semantic of an architecture is a set of couples (A,S) where A is a set of active inputs and S is the semantics of the elementary sequence obtained from the device after all DNA transformations have been performed.

The transcriptional semantic of an architecture can be put in correspondence with Boolean functions. Indeed, a transcriptional semantic can be seen as a truth table where each line corresponds to a couple (A,S). An input that belongs to the set A is active (1) and non active (0) otherwise. Then, the semantic S denotes either gene expression (1) if and only if it contains the behaviour fX.

3.2 Details on the constraints used to reduce the generation of irreducible architectures.

3.2.1 Utility of a sequence according to another one.

A semantic s_1 is useful after another semantic s_2 if there is no less complex semantic s'_1 of s_1 such that $s_2.s_1 \equiv s_2.s'_1$. And a semantic s_1 is useful before another semantic s_2 if there is no less complex semantic s'_1 of s_1 such that $s_1.s_2 \equiv s'_1.s_2$.

s_1 is less complex than s_2 if all the corresponding parts of s_1 are strictly included in the corresponding parts of s_2.

Example We defined $s_1 = fP/fN$ and $s_2 = fT/fN$. s_1 is useless before s_2 because $s_1.s_2 = fT/fN$ and therefore, it exists $s_{1'} = fN/fN$ such that $s_{1'}.s_2 = fT/fN$. But s_1 is useful after s_2 because there exists no $s_{1'}$ less complex that s_1 such that $s_2.s_1 \equiv s_2.s'_1$.

Constraints With this concept of utility, we create two constraints, one to check the utility of a semantic after another one, and a second one to check the utility of a semantic before another one, for each variable. A semantic is useful if, on one derived structure, there is an utility in the forward direction and on another one derived structure (not necessarily the same as before), there is an utility in the reverse direction.

All cases of utility are presented in the table S5 of this section. There is only the forward semantics : for the reverse semantics, the table must be read in the other direction.

To know if a semantic is useful before another one, we read the row and after the column. For example, if $s_1 = fP$ and $s_2 = fT$, to check if s_1 is useful before s_2, we read the row fP and the column fT : in the bottom-left corner of the cell, we can read fN, so there exists a $s_{1'} = fN$ such that $s_1.s_2 \equiv s_{1'}.s_2$. If the cell was blank, s_1 would be useful before s_2.

To know if a semantic is useful after another one, we read the row before the column. For example, if $s_1 = fGP$ and $s_2 = fP$, to check if s_1 is useful before s_2, we read the column fGP and the row fP : in the top-right corner of the cell, we can read fG, so there exists a $s_{1'} = fG$ such that $s_1.s_2 \equiv s_{1'}.s_2$. If the cell was blank, s_1 would be useful before s_2.

$s_1.s_2$	fN	fP	fT	fG	fGP	fX
fN						
fP		fN / fN	fN		fG	fG / fN
fT		fN	fN / fN	fN	fP	fN
fG			fN	fN / fN	fP / fN	fN
fGP		fN / fG	fG	fP	fG / fP	fG / fN
fX		fN	fN	fN / fP	fN / fP	fN / fN

Supplementary Table 5: utilities of a semantic concatenated to another one

3.2.2 Utility of prefix and suffix semantics.

The principle here is similar to the one described before. Instead of compare a semantic to a semantic before or after another one, we compare it to the prefix semantic or the suffix semantic of the variable in the derived functional structures.

The prefix semantic of a variable is the concatenation of all semantics before this variable. And the suffix semantic is the concatenation of all semantics after this variable.

The table S5 of this section must be read in the same way as the table S6.

$Sem_{column}.Sem_{row}$	fN	fP	fT	fG	fGP	fX
fN			fN	fN	fP	
fP	fN	fN / fN	fN		fG	fG / fN
fT	fN	fN	fN / fN	fN	fP	fN
fG			fN	fN / fN	fP / fN	fN
fGP	fG	fN / fG	fG	fP	fG / fP	fG / fN
fX		fN	fN	fN / fP	fN / fP	fN / fN

Supplementary Table 6: utilities prefix and suffix.

3.2.3 Domain definitions

By default, each variable has in its domain all the 26 semantics. But, in some cases, we can reduce the domain of some variables.

Atomic inversion and excision A pair of sites is atomic if there is no site between them.
In an atomic excision, we do not assign the fN/fN semantic because. Indeed, after excision, the semantic will remained identical (fN/fN) leading to the implementation of a redundant Boolean function.
Identically for atomic inversion to avoid redundant Boolean function, we do not assign symmetric semantics as they are not modified by inversion (fN/fN, fP/fP, fG/fG, fT/fT, fGP/fGP, fX).

Expressed semantic The semantic fX can only be put in an excision. Otherwise, the gene would be always expressed on all derived structures leading to the implementation of the True Boolean function which is redundant.

Semantics at the ends of a functional structures As a consequence of the prefix and suffix constraints, we only assign four semantics in the first variable (fN/fN, fP/fN, fN/fG, fP/fG) and four in the last variable (fN/fN, fN/fP, fG/fN, fG/fP). Indeed, the prefix of the first variable is always fN/fN to respect the previously defined constraints these are the only semantics that have a prefix utility after fN/fN. The same applies with the suffix of the last variable.

3.3 Rules permitting to simplify a concatenation of semantics, permitting therefore to determine the gene expression state.

Transcriptional parts are concatenated to form transcriptional sequences. We defined a set of rules to determine the semantics of sequences. As any forward and reverse semantics can be considered separately, the following properties are defined considering a single orientation of the construct. For simplification, the properties are written in the forward orientation, from 5' to 3'.

The semantic of a transcriptional sequence corresponds to the concatenation of the semantics of each part of the sequence. Indeed, to determine the semantics of a concatenation of transcriptional parts, we use a step-wise iterative process in which semantics are composed two by two.

This concatenation can be simplified with the following rules in a reduced set of six elementary semantics. These six semantics correspond to the four semantics described previously (fP, fT, fG, and fN) plus the semantics corresponding to the expression of a gene: fX and the composition of fG followed by fP: fGP. As the two-by-two concatenation of the four basic semantics leads to one of these six semantics, this set of semantics is complete (detailed below).

Rules:

(1) Non-commutativity: Concatenation of semantics is not commutative as parts concatenated in a different order do not lead to the same semantic. As an example, PF-GF permits expression of the gene, therefore encoding the semantic fX, which is not the case for GF-PF.

(2) Neutrality: A sequence without any activity in a particular orientation does not affect other sequences placed in the same orientation (i.e. fN is neutral to other semantics similarly oriented).
(3) Assimilation of fX: The semantic of gene expression, fX, assimilates all other semantics. In others words, the composition of the fX semantic with another semantic is simplifiable to fX. In this work, we aim at defining if a construct leads to expression of a gene or not and the composition of fX with another semantic does not affect the fX semantic.
(4) Idempotent: all semantics are idempotent (an operation has the same effect even if applied multiple times), as we consider that the concatenation of two similar parts is equivalent to a single part.

Others rules are due to the mechanism of gene expression. A gene is expressed if it is transcribed by RNA polymerase; consequently, a promoter needs to be positioned upstream without a terminator positioned between the promoter and the gene. This mechanism can be assimilated to a flow that is opened by the promoter, stopped by the terminator, and the system is ON when the flow is at a specific location: the gene.
(5) Expression occurs only if a promoter is placed upstream of a gene without a terminator in between. Such as, the concatenation of the semantic promotion with semantic gene is simplifiable to fX. i.e. fP-fG=fX
(6) A gene followed by a promoter leads to the semantic fGP, as the promoter can be active for a downstream gene and the gene can be expressed by an upstream promoter. Consequently, **fG-fP=fGP**. In this case, we have associativity of the semantics fG and fP.
(7) If a promoter is followed by a terminator, the RNA polymerase flux is blocked by the terminator, consequently, *fP-fT=fT*.
(8) If a terminator is followed by a promoter, the terminator will have no effect on the semantic of the sequence as the terminated transcription will be re-initiated by the promoter, consequently: **fT-fP=fP**.
(9) If a terminator is followed by a gene, the gene cannot be expressed as the RNA polymerase flux is blocked by the terminator; consequently, **fT-fG=fT**.
(10) If a gene is followed by a terminator, the terminator will have no effect on the semantic of the word; indeed, if the previous semantic is fP, it will result in the expression of the gene, consequently: **fG-fT=fG**.

Mathematical definition of the rules for semantic composition. For two semantics fA and fB, fA-fB is the concatenation of fA with fB, fA being in 5' and fB in 3'.
(1) Not commutative: fA-fB≠fB-fA
(2) Neutrality of fN: fN-fA=fA and fA-fN=fA
(3) Assimilation by fX: fX-fA=fX and fA-fX=fX
(4) Isomorphism: fA-fA=fA
(5) Condition of gene expression: fP-fA-fG=fX only if fA≠fT or fA=fP
(6) Composition of Gene-Promoter: fG-fP=fGP
(7) A terminator cancels promotion: fP-fT=fT
(8) A promoter cancels termination: fT-fP=fP

(9) A terminator block transcription of a following gene: fT-fG=fT
(10) A terminator cannot block transcription of a previous gene: fG-fT=fG

We concatenated the six previously defined semantics two by two. Using the previously defined rules, all concatenations of these six semantics are simplifiable to one of the six semantics (TableS2). Therefore, the set of semantics is complete and our rules are scalable to the concatenation of N transcriptional parts.

3.4 Symmetric functions are implementable with recombinases.

3.4.1 Definition of a symmetric function

A symmetric function of n variable is a function which is equivalent by any permutation of the n variable.
As examples:
(1) f(a,b,c)=a.b.c+!a.!b.!c (with !x being the negation of x) is a symmetric function which is equal to 1 if the number of one variable is zero or three. Therefore, it can be written as $S^3_{0,3}$(a,b,c), a,b and c being the variable of the symmetric function.
(2) f(a,b,c)=a.!b.c+!a.b.!c is also a symmetric function. Here the variables are a, !b, c and the function can be written as $S^3_{0,3}$(a,!b,c).

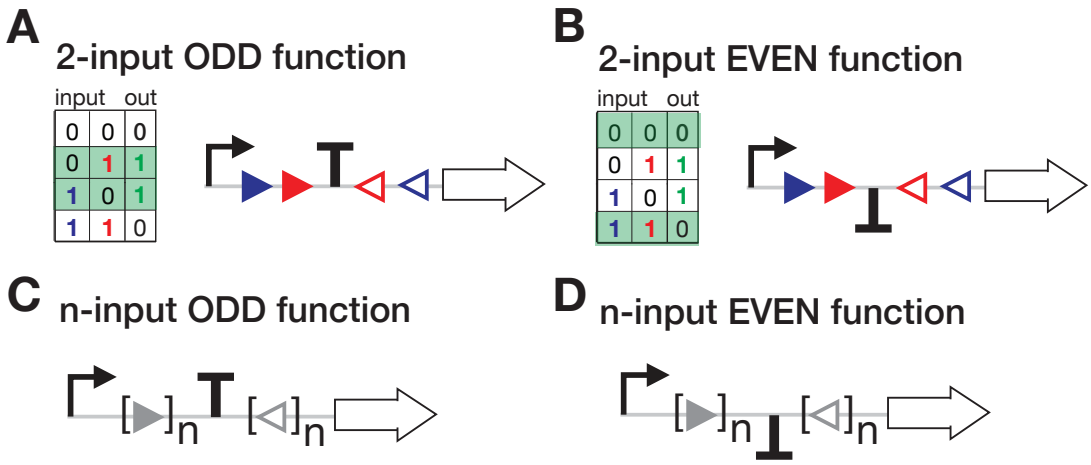
We note symmetric functions as: $S^n_{\{a_k\}}(x_1^{j_1}, x_2^{j_2}, ..., x_n^{j_m})$, where the j_i may take on only the values 0 and 1, and:

$$x_i^{j_i} = x_i \text{ if } j_i = 0 \text{ and } x_i^{j_i} = \neg x_i = !x_i \text{ if } j_i = 1$$

3.4.2 Formal implementation of symmetric functions with recombinases.

Symmetric functions are usually functions with a large number of terms and of literal per terms, so requiring large circuits for implementation if using second-order circuits based on sum-of-products or product-of-sums function expression. However, in electronic, in the 1960 century, methods have been developed to easily and economically realize completely symmetric functions using contact-type gating elements. As well in biology, symmetric functions are easily implementable using recombinases using nested inversion-based elements.

As example, the 2-input XOR gate were realized by Bonnet and colleagues using a single terminator nested inversion-based elements as represented in Figure bellow. In the presence of none of the input, the terminator is blocking the expression of the output gene. In presence of a single input, the terminator is inverted leading to expression of the output gene. In presence of the two input, the terminator is inverted back in its blocking orientation, the output gene is then not express. This design can be generalize for an n number of input for even and similarly odd symmetric functions (Figure S11).



Supplementary Figure 11: **Implementation of odd and even symmetric functions using recombinases.** (A) For the implementation of 2-input odd function also called XOR function, Bonnet and colleagues used a terminator nested inversion-based element. In more detail, the device is composed on an asymmetric terminator in blocking orientation surrounded by two nested pairs of integrase sites in inversion orientation. (B) Similarly, for the implementation of the 2-input even function also called NXOR function, Bonnet and colleagues used a terminator nested inversion-based element, however here the asymmetric terminator is in passing orientation in the off state. (C) and (D) These 2-input designs can be generalized to N-input designs by nested N-integrase site pairs in inversion orientation.

3.4.3 All fully implementable clusters of complex functions are composed of symmetric or partially symmetric Boolean functions.

List of all logic functions with 8 terms and 4 literals per terms:

ODD PARITY FUNCTION: $S^4_{1,3}$(a,b,c,d)=!a.!b.!c.d + !a.!b.c.!d + !a.b.!c.!d + a.!b.!c.!d + !a.b.c.d + a.!b.c.d + a.b.!c.d + a.b.c.!d

EVEN PARITY FUNCTION: $S^4_{0,2,4}$(a,b,c,d)=!a.!b.!c.!d + !a.!b.c.d + !a.b.!c.d + !a.b.c.!d + a.!b.!c.d + a.!b.c.!d + a.b.!c.!d + a.b.c.d

List of all logic functions with 7 terms and 4 literals per terms:

$S^4_{2,4}$(a,b,c,d)=!a.!b.c.d + !a.b.!c.d + !a.b.c.!d + a.!b.!c.d + a.!b.c.!d + a.b.!c.!d + a.b.c.d

$S^3_{1,3}$(a,b,c).!d+S^3_2(a,b,c).d=!a.!b.c.!d + !a.b.!c.!d + a.!b.!c.!d + !a.b.c.d + a.!b.c.d + a.b.!c.d + a.b.c.!d

S^4_1(a,b,c,d)+S^3_2(b,c,d).a==!a.!b.!c.d + !a.!b.c.!d + !a.b.!c.!d + a.!b.!c.!d + a.!b.c.d + a.b.!c.d + a.b.c.!d

$S^4_{0,4}$(a,b,c,d)+S^2_1(a,b).S^2_1(c,d)=!a.!b.!c.!d + !a.b.!c.d + !a.b.c.!d + a.!b.!c.d + a.!b.c.!d + a.b.!c.!d + a.b.c.d

$S^4_{0,2}$(a,b,c,d)=!a.!b.!c.!d + !a.!b.c.d + !a.b.!c.d + !a.b.c.!d + a.!b.!c.d + a.!b.c.!d + a.b.!c.!d

List of all logic functions with 8 terms and 3.5 literals per terms:

$S^4_{1,3,4}$(a,b,c,d)=!a.!b.!c.d + !a.!b.c.!d + !a.b.!c.!d + a.!b.!c.!d + b.c.d + a.c.d + a.b.d + a.b.c

S^2_1(c,d).!a.!b+c.d.S^2_1(a,b)+$S^2_{0,1}$(c,d).a.b+!c.!d.$S^2_{1,2}$(a,b)=!a.!b.!c.d + !a.!b.c.!d + !a.b.c.d + a.!b.c.d + b.!c.!d + a.!c.!d + a.b.!c + a.b.!d

$S^3_{0,2,3}$(a,b,c).!d+$S^3_{1,3}$(a,b,c).d=!a.!b.!c.!d + !a.!b.c.d + !a.b.!c.d + a.!b.!c.d + b.c.!d + a.c.!d + a.b.!d + a.b.c

$S^3_{0,2}$(b,c,d).!a+$S^3_{0,1,3}$(b,c,d).a =!a.!b.c.d + !a.b.!c.d + !a.b.c.!d + a.b.c.d + !b.!c.!d + a.!b.!c + a.!b.!d + a.!c.!d

$S^4_{0,1,3}$(a,b,c,d)=!a.b.c.d + a.!b.c.d + a.b.!c.d + a.b.c.!d + !a.!b.!c + !a.!b.!d + !a.!c.!d + !b.!c.!d

3.5 List of 4-input P-classes with a single possible architectures.

Six 4-input P-classes are implementable with a single possible architectures, following an example of logic function in a Quine McKluskey form for each of these 6 P-classes.

f1=!a.b.c.d + a.!b.!c + a.!b.!d + a.!c.!d

f2=a.b.c.d + !a.!b.!d + !a.!c.!d + !b.!c.!d

f3=a.!b.!c + a.!b.!d + a.!c.!d + b.c.d

f4=!a.!b.!d + !a.!c.!d + !b.!c.!d + a.b.c

f5=a.!b.!c + a.!b.!d + a.!c.!d + !a.c.d + !a.b.d + !a.b.c

f6=!a.!b.!d + !a.!c.!d + !b.!c.!d + b.c.d + a.c.d + a.b.d

3.6 4-input P-class with the maximum number of architectures.

Following an example of logic function of the P-class with the maximum number of architectures (1.4 millions).

f=a+b+!c+!d