Scientific Annals of Computer Science vol. 23 (1), 2013, pp. 1–38
doi: 10.7561/SACS.2013.1.1

GUBS, a Behaviour-Based Language for Design in Synthetic Biology

Adrien BASSO-BLANDIN[1], Franck DELAPLACE[1]

Abstract

In this article, we propose a domain specific language, Gubs (Genomic Unified Behaviour Specification), dedicated to the behavioural specification of synthetic biological devices, viewed as discrete open dynamical systems. Gubs is a rule-based declarative language. In contrast to a closed system, a program is always a partial description of the behaviour of the system. The semantics of the language accounts the existence of some hidden non-specified actions that possibly alter the behaviour of the programmed devices. The compilation framework follows a scheme similar to automated theorem proving, aiming at improving synthetic biological design safety.

Keywords: Synthetic biology, CAD, Compilation

1 Introduction

Synthetic biology is an emerging scientific field combining the investigative nature of biology with the constructive nature of engineering [30] to design synthetic biological systems. The issue is to devise new functionality/behaviour that does not exist in nature. Then, the field of synthetic biology is looking for principles and tools to make the biological devices inter-operable and programmable [27]. Synthetic biology projects were first focusing on the design and the improvement of small genetic devices comparable to logical gates for electronic circuits [31, 12]. Recently, projects

[1]IBISC lab., Evry University, IBGBI, 23, boulevard de France, 91037 Evry, France. Email: {abasso,franck.delaplace}@ibisc.univ-evry.fr

have attempted to develop large bio-systems integrating different devices with as a long-term goal, the design of *de-novo* synthetic genome [23]. In this endeavour, computer-aided design (CAD) environments play a central role by providing the required features to engineer systems: specification, analysis, and tuning [5, 28, 38, 14]. Pioneer applications show the valuable potential of such environments in the Igem competition.

Currently, the design of synthetic genome specifies the structural assembly of Dna sequences (biobrick) as in Genocad [8]. Although this description is indispensable to provide a finalized specification of devices, the abstraction level seems inappropriate for tackling large bio-systems. The required size of programs for sequence description likely makes the task error-prone and infeasible. In the same way as large software cannot be programmed in binary, large biological systems cannot be described as a DNA sequence assembly. Then, scaling up the complexity of the synthetic biological systems needs to complete the structural description by an additional abstract programming layout based on a High-level programming languages and harness the automatic conversion of the design specification into a DNA sequence, like compilers. High-level programming languages for synthetic biology is announced as a key milestone for the second wave of synthetic biology to overcome the complexity of large synthetic system design [30]. Nonetheless, in this domain, language technology is still in its infancy and transforming this vision into reality remains a daunting challenge.

Such high-level language should describe the devices in term of functionalities, offering the ability to program the behaviour directly instead of the structure supporting this behaviour. Indeed, behaviour specification contributes to accurately document the device by adding its behavioural description, to assess its functionality automatically and formally, notably by generating test-benches from this specification, and to get a relative independence to technology because different biological structures can carry out the same functionality. In this framework, the components are selected and organized automatically or semi-automatically to generate a structural description of the device at compile phase whose behaviour complies with the specified function. One such approach has been already achieved in hardware by using languages as Vhdl [1] or Verilog [37] to overcome the growing complexity of electronic circuits. However, the major difference in synthetic biology relates to the openness of biological system. Hence, we propose to define a language dedicated to synthetic biology based on a behavioural specification that handles the openness of system.

More precisely, Gubs is a rule-based declarative language dedicated to the behavioural specification of *discrete open dynamical systems* for synthetic biology interacting with its environment. Gubs symbolically defines the behaviours to provide a relative independence from structures by postponing the biological component selection at compile phase. Within this framework, the compiler translates the behavioural specification to a structural description of a device whose behaviour carries the functional features defined by a program. The proposed compilation method is inspired by automated theorem proving.

After introducing related works(Section 2) on languages dedicated to systems and synthetic biology, we introduce the main features of Gubs language (Section 3), we define the semantics of Gubs based on hybrid logic. Then, we detail the proof-based principles governing the compilation (Section 5) illustrated with a complete example (Section 6).

2 Related Work

Several domain specific languages have been developed to model and simulate biological systems. Based on process calculus, seminally used to model process concurrency, several rule-based languages model protein interactions [29, 16, 11]. Another approach is based on logic, such as Biocham [9] that formalizes the temporal properties of a biological system. As these languages are dedicated to simulation, the objective is to close the systems because the simulations need to integrate all the characteristics of the analysed systems. By comparison, the purpose of Gubs is different since the issue is to represent the behaviour of a synthetic device in an organism, leading to translate the notion of the openness of biological systems by the semantics of the language.

In synthetic biology, structural description languages [14, 28, 5] allow to specify well-formed genome sequences by grammars modularly and hierarchically. Although the sequence description is necessary, the programmer must previously anticipate the behaviour of the device to conceive. Besides, the behavioural design is not included in the program while it initially motivates it. In Gubs, the design is driven by a behaviour description and sequence selection is postponed at compile phase. Moreover, the size of the structural description is also subject to a combinatorial explosion when the complexity of programmed systems increases.

Amorphous programming languages has been also investigated to specify the biological devices at the scale of cell colonies, here considered as a possible computing medium for amorphous program. J. Beal [4] demonstrates a proof of concept of this approach in Proto, showing the feasibility of an automatic compile chain. In Gubs, the compile chain is based on rewriting rules whose correctness have been formally proved with regards to a semantics describing the constraints of an open system.

Developing a language for biological systems actually involves to consider several unknowns due to their openness: lack of knowledge on all the interactions in biological circuits and imprecise definition of initial conditions. We only know the result of a chain of effects. Then, the major constraint for programming open systems seems to be: how to provide an expressive language to describe the dynamics of such systems, but simple enough to capture the essence of the biological questions in a small program in order to allow programming of large biological systems with a program humanly achievable.

In the future, the design in synthetic biology will certainly require different programming layouts based on different paradigms addressing the integration levels of biological systems. In a tower of languages, starting from a language with collective operations on cell colonies, using an amorphous programming language as Proto [4] or a language for dynamical systems with dynamical structures as MGS [22], and ending by a structural description programmed in a grammar based language, the Gubs language occupies the intermediary level dedicated to cell entity behavioural programming.

3 GUBS Language

In this section, we describe the main features of Gubs. Informally, a Gubs program describes the expected observed behaviour of a biological component. A sequence of observation must comply to a sequence of events related by a causal chain specified in a Gubs program.

Agents, attributes and states. The *agents* represent the biological objects. Hence their different observable *states* characterize their different *behaviours*. The behaviours define the different capacities for actions on the state of the other agents. It is worth noting that they are characterized symbolically by a set of *attributes* identifying these different capacities. The real significance of the attributes is a matter of convention depending on the targeted realization (*e.g.*, protein pathways, gene network) and will be addressed through examples. For instance, the regulatory activity of a gene is observationally related to thresholds of Rna transcripts concentration. At a given threshold, a gene regulates a given set of genes whereas at another one the regulation applies to another set of genes (See Figure 1). The different thresholds define the levels of gene activities leading to different regulatory activities. For example, if we identify three different kinds of regulatory activities for a gene G, the state of the gene will be defined by three different attributes $\{Low, Mid, High\}$ characterizing three possible behaviours symbolically. For example, $G(Low)$ expresses the fact that agent G is in state *Low* and then ready for the action corresponding to this attribute. In some cases, a single state is sufficient to qualify the capacity for the action of the agent. Hence, the agent is identified to its capacity. Then, G means that agent G is available.

By contrast, $G(\overline{Low})$ signifies that the state of the agent differs from *Low* ($\overline{G}$ when an agent has a single capacity). It is worth to point out that, not being in a state defined by an attribute, does not necessarily mean that the agent state is in another attribute. Indeed, for open systems the state of the agents could be of any sort that does not necessarily belong to the pre-defined attributes.

Two kinds of relations on attributes are defined: an order, $\prec$, meaning "less capacity than" and an inequality, $\not\approx$, meaning "different capacity than". Then $Low \prec Mid$ implies that the capacity for the action of *Mid* includes the capacity related to *Low*. Usually, in a gene regulatory model [17], the set of genes regulated at a given level will also be regulated at a higher level. By contrast, in signalling pathways, the phosphorylation of a protein induces a conformational change of the structure leading to a specific signalling potentiality not occurring in the unphosphosrylated conformation. Assuming that *Phos* and *UnPhos* respectively represents the phosphorylated and the unphosphorylated conformations of protein P, we have $Phos \not\approx UnPhos$. Then, $P(Phos)$ implies $P(\overline{UnPhos})$ implicitly. The attributes and the relation between attributes will be declared as follows: $G :: \{Low \prec Mid, Mid \prec High\}, P :: \{Phos \not\approx UnPhos\}$. A set of attributes replaces the relations if unknown and no specific relation is set between attributes.

Finally, the description of the agent state is extended to a collection of agent states as follows: $g_1 + \ldots + g_n$, meaning that all the agent states, g_i, are observed simultaneously.

Constant and variables. In GUBS, two kinds of agents are distinguished: the *constants* and the *variables*. The constants designate the real pre-defined objects in a corpus of knowledge. In biology, the constants may refer to proteins or genes of interest. For example, the agent *LacZ* refers to LacZ protein or gene. By convention, their name starts with a capital letter. The variables refer to an abstraction of these pre-defined objects and can be potentially replaced (substituted) by any constant. By convention, the variable names start with a lower-case letter.

Trace, event, and history. A Gubs program describes a behaviour, its interpretation is based on the observations of designed systems. Then, the issue is to formalize the notion of behaviour observation. To this end, we focus on the notion of a *trace* that symbolically represents the evolution of some quantities related to the agents of interest by the evolution of these agent states. A trace can be obtained from experiments by establishing a correspondence between measurements of some quantities (*e.g.*, Rna transcript concentration) and attributes of agents. Formally, a trace, $(T_t)_{1 \leq t \leq m}$, is a finite sequence of agent state sets where each set contains all observed agent states at a given instant. For instance, the evolution of a concentration evolving from *Low* to *High* for G may be described by the following trace of 6 instants[2]: $(\underset{1}{\{G(Low)\}},\underset{2}{\{G(Low)\}},\underset{3}{\{G(Mid)\}},\underset{4}{\{G(Mid)\}},\underset{5}{\{G(Mid)\}},\underset{6}{\{G(High)\}}),\underset{7}{\ }$. However, all the events in a trace are not necessarily relevant with regard to the behaviour description. For example, if we focus on the evolution from *Low* to *High* for G, we decide arbitrarily that only three events are relevant for the behaviour description: $G(Low)$ then $G(Mid)$ and finally $G(High)$; without accounting the intermediary evolution stages occurring between. Then, the behaviour recognition always emphasizes the key events in a trace entailing its contraction to show their succession. Such a contracted series is called a *consistent history* of the expected behaviour. Generally speaking, an history is related to a *chronological division* of a trace into periods where the events of a period represent all the agent states occurring at each instant. Then, an history is a sequence of these event sets. Given a trace $(T_t)_{1 \leq t \leq m}$, and a chronological division, $(d_i)_{1 \leq i \leq n}$ such that $d_i < d_{i+1}$, corresponding to a sequence of the starting dates for each period, the history is a sequence of agent states occurring in each period, $(H_i)_{1 \leq i < n}$, such that each $H_i = \bigcup_{d_i \leq t < d_{i+1}} T_t$. Hence, a consistent history is purposely made to point the characteristic event steps of a behaviour description out.

In the previous example, a chronological division of the trace leading to an history consistent with the expected evolution from *Low* to *High* for G is $(1, 3, 6, 7)$ which corresponds to following discrete time-intervals

[2]Step 7 is inserted as an extra step to comply with the definition of the chronological division.

$([1,2],[3,5],[6,6])$. The resulting history is: $(\{G(Low)\},\{G(Mid)\},\{G(High)\})$. Notice that $(1,2,4,7)$ also fits. However, the chronological division $(1,3,7)$ leads to an inconsistent history because the level *Mid* and *High* are not explicitly distinguished as too separate steps. Hence the history does not follow the expected progress from low to high. The formal definition of the consistency in the scope of the semantics will be given in Section 4.

Behavioural dependence and observation spot. A behavioural dependence identifies a relation between behaviours as a causal relation on events. Basically, the dependences should define the control of agents on each other. However, the definition of the causality also needs to tackle the openness of a system by adapting it to this context. An historical definition of the causality, proposed by Hume [24], is formulated in terms of regularity on events: "[we may define] a cause to be an object, followed by another, and where all the objects similar to the first are followed by objects similar to the second". Although this definition appropriately characterizes the notion of control, the openness of the system implies to account for the environment actions that possibly alter the causal dependence chain. For example, a programmed activation $G_1 \xrightarrow{+} G_2$ may be contradicted by an existing inhibition $G_3 \xrightarrow{-} G_2$ addressing the same target gene G_2. Hence, while G_1 is active, it may appear that G_2 will not be active because the regulatory strength of G_3 is greater than the regulatory strength of G_1, contradicting the expected activation by a hidden inhibition. Hence, pushed to the limit, this consideration prevents the ability to describe any behaviour causally because any programmed action can be unexpectedly preempted by an external one. Adapting the Hume's definition, we define a causality by the occur of its effect. If the effect is observed, the causal relation is effective. which is different from the basic approach considering : if the cause is observed, the causal relation is effective.

By ensuring that the design describes a new functionality which is not observed naturally, the observation of effects becomes the sole events indicating the trigger of causal dependences. Indeed, the observation of a cause cannot be considered as an indicator because its action could be preempted by external events. In other words, the proposed definition of the causality reflects the fact that the device may be not functional due to an external intervention. However, the functionality is still correctly specified because this eventually is accounted in the definition of causality validated by the observation of effects. Besides, as no cause external to the description is assumed to trigger the effects of dependences for the new functionality, the over-determination by unknown causes is supposed to be prevented, then ensuring that the program is the sole device entailing the expected effects in the biological system. Hence, the definition of the causal dependence will be governed by the effect leading to the following definition of the dependence: *"if effect e would occur then c occurs"*. Moreover, the scope of future (resp. past) is narrowed to a *closest future (resp. past) period*, representing the fact that a response is always expected in a given delay. Notice that, the proposed definition circumvents the afore mentioned problem illustrated by the hidden inhibition because if the effect does not occur the question of the existence of a cause is meaningless. This definition is somehow equivalent to the causal claims proposed by Lewis [26] in terms of counter-factual conditionals, *i.e.*, "If c had not occurred, e would not have occurred".

Three behavioural dependences are defined in GUBS: the *normal* denoted by $\bigcirc\!\!\rightarrow$, *persistent* by $\odot\!\!\rightarrow$, and *residual* by $\oplus\!\!\rightarrow$. These dependences are primitive in the sense that they cannot be expressed by the others without weakening there properties (see Table 5). Informally, for normal dependence the cause precedes the effect providing the effect is observed; for persistent dependence the cause still precedes the effect but it is maintained while the effect is observed; and for residual dependence, the effect is maintained despite the cause has disappeared. These dependences symbolize common biological interactions. For instance, in genetic engineering, a recombination enables the emergence of a regulated gene or an hereditary trait permanently. Such a mechanism typifies the residual dependence in biology. The relations between gene expressions at steady state are symbolized by persistent dependence. The behavioural dependences are defined as follows (see Section 4 for their formalization):

$c \bigcirc\!\!\rightarrow e$: if e occurs then c occurred in the closest past.

$c \odot\!\!\rightarrow e$: if e occurs then c occurred in the closest past and also currently.

$c \oplus\!\!\rightarrow e$: if e occurs then, either e occurred in the closest past or e does not occurs in the closest past and c necessary occurs.

Figure 1 exemplifies the correspondence between experimental traces, symbolic traces and the history for the causal dependences. All the dependences are extended to a set of causes and a set of consequences, *i.e.*,

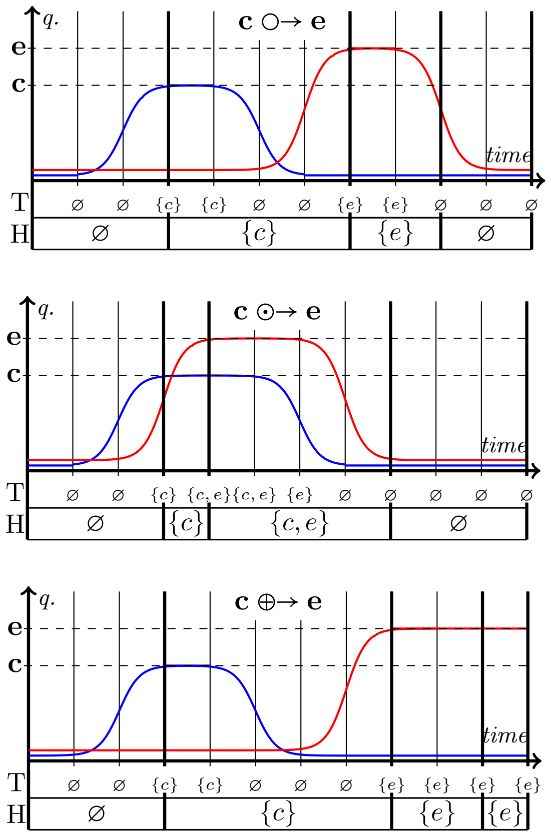

Figure 1: The curves represent the typical behaviours of the causal dependences based on the time evolution of a quantity (q) related to agents c and e (*e.g.*, Rna transcript for gene regulation). The symbolic agent states c and e are here both associated to the maximal threshold of the quantity. The symbolic trace (T) is issued from a periodic sampling of the evolution by identifying whether c or e occur. A consistent history (H) complying to a causal dependence definition is represented below the trace. The first graphic illustrates the normal causality: $c \bigcirc\!\!\rightarrow e$, the second the persistent: $c \odot\!\!\rightarrow e$ and the third the residual one: $c \oplus\!\!\rightarrow e$.

$c_1 + \ldots + c_n \circ\!\!\rightarrow e_1 + \ldots + e_m$. For example, let us define the activation and the inhibition as follows:
$g_1 \xrightarrow{+} g_2 \equiv g_1 \odot\!\!\rightarrow g_2, \overline{g}_1 \circ\!\!\rightarrow \overline{g}_2$ and $g_1 \xrightarrow{-} g_2 \equiv \overline{g}_1 \odot\!\!\rightarrow g_2, g_1 \circ\!\!\rightarrow \overline{g}_2$. Then, the program depicting a negative regulatory circuit with two genes, *i.e.*, $g_1 \xrightarrow{+} g_2, g_2 \xrightarrow{-} g_1$, is: $\{g_1 \odot\!\!\rightarrow g_2, \overline{g}_1 \circ\!\!\rightarrow \overline{g}_2, \overline{g}_2 \odot\!\!\rightarrow g_1, g_2 \circ\!\!\rightarrow \overline{g}_1\}$.

The *observation spots* describe the set of observations expected in a trace. For instance, observing that gene G is at level high is written $Obs{::}G(High)$. As the activation of a dependence lies on the observation of the effect, the observation spot is used to determine which effects must be necessarily observed. To some extends, observation spots can be assimilated to experimental requirements. For example, in the negative regulatory circuit, the characteristic observation spots are: $obs_1{::}(g_1 + \overline{g}_2), obs_2{::}\overline{(}g_1 + g_2)$.

Compartment & Context. A compartment encloses a set of dependences making them local to the compartment. For instance, $C\{g_1 \circ\!\!\rightarrow g_2\}$ describes a normal dependence occurring in compartment C. The compartments are hierarchically organized and all the compartments are included in another except for the outermost one. Although the compartments directly refer to the compartmentalized cellular organization (*e.g.*, nucleus, mitochondria), they are also used to emphasize the isolation of some interactions by syntactically enclosing the dependences into a compartment. $C.s$ refers to an agent state in compartment C.

A context refers to a stimulus acting on the system, as environmental conditions or external signalling. The application of a context c to a set of dependences b is written $[c]b$ where c is either a variable or a constant. This means that dependences of b are triggered when the context c is present. For instance, recently Ye et al. [39] explore the opto-genetics signalling to control the expression of target transgenes. The blue-light induces the expression of transgene (tg) via a signalling cascade leading to the binding of NFAT transcription factor to a specific promoter (PNFAT). The following program using a context summarizes the process: $[\text{BlueLight}]\{\text{NFAT} \odot\!\!\rightarrow tg\}$. A context can be decomposed to several contexts, $[k_1, \ldots, k_n]b$, meaning that all the conditions must be met to trigger the dependences of b. The interpretation is equivalent to a context cascading, $[k_1][k_2]\ldots[k_n]b$. Moreover, the observation spots and the attribute definition are context insensitive.

4 Semantics of GUBS

The interpretation of GUBS program is a formula of multi-modal hybrid logic with the "Always" operator, $\mathcal{H}(\mathbf{A}, @)$. Formally, a GUBS program defines a set of causal relations and observation spots. Notice that those sets can be empty. The program is translated into an hybrid logic formula.

Hybrid logic. In what follows, we recall the formal syntax and semantics of hybrid logic. The hybrid logic [6, 7] offers the possibility to denominate worlds by new symbols called *nominals*. They will be used in satisfaction of modal operators $@_a$; the formula $@_a\phi$ asserts that ϕ is satisfied at the unique point named by the nominal a identifying a particular truth value of a formula at this point. Given a set of propositional symbols, PROP, a set of relational symbols REL, and a set of nominals NOM disjoint to PROP, a set of well formed formula in the signature of $\langle \mathsf{PROP}, \mathsf{NOM}, \mathsf{REL} \rangle$ is defined as follows:

$$\phi ::= \top \mid p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid @_a\phi \mid \langle k \rangle\phi \mid \langle k \rangle^{-}\phi \mid \mathbf{A}\phi.$$

with $p \in \mathsf{PROP}, a \in \mathsf{NOM}$ and $k \in \mathsf{REL}$. Moreover, the syntax is extended to other logical operators [3]: $\bot, \vee, \rightarrow, [k], \mathbf{E}$, in the usual way.

The semantics of $\mathcal{H}(\mathbf{A}, @)$ is based on the Kripke model satisfaction (Table 1). $\mathcal{M}, w \Vdash \phi$ is interpreted as the satisfaction of a formula ϕ by a model $\mathcal{M}$ at world w where $\Vdash$ stands for the realizability relation (*i.e.*, "is a model of"). A model *validates* a formula, denoted by $\mathcal{M} \Vdash \phi$, if and only if it is satisfied for all the worlds of the model (*i.e.*, $\forall w \in \mathrm{Dom}\,\mathcal{M} : \mathcal{M}, w \Vdash \phi$).

Definition 1 (Kripke model)**.** *A Kripke model is defined as a structure :*

$$\mathcal{M} = \langle W, (R_k)_{k \in \tau}, V \rangle$$

where $W = Dom\,\mathcal{M}$ is a non-empty set of worlds, *$\tau \subseteq REL$ a subset of relational symbols denoting the* modalities*(i.e., label of edges), $R_k \subseteq W \times W, k \in \tau$ a relation of accessibility, $V : (PROP \cup NOM) \rightarrow 2^W$ an interpretation attributing to each nominal and propositional variable a set of worlds such that any nominal addresses at most one world(i.e., $\forall a \in NOM : |V(a)| \leq 1$). By convention, R stands for the union of the accessibility relation, $R = (\bigcup_{k \in \tau} R_k)$.*

[3] $\bot = \neg\top, \psi \vee \phi = \neg(\neg\psi \wedge \neg\phi), \psi \rightarrow \phi = \neg(\psi \wedge \neg\phi), [k]\phi = \neg\langle k \rangle\neg\phi, \mathbf{E}\phi = \neg\mathbf{A}\neg\phi.$

The *modal theory* of a model $\mathcal{M}$ with respect to a set of formulas F, $\mathrm{TH}_F(\mathcal{M})$, is the set of formulas of F validated by $\mathcal{M}$, *i.e.*, $\mathrm{TH}_F(\mathcal{M}) = \{\phi \in F \mid \mathcal{M} \Vdash \phi\}$. $\mathsf{KS}(\phi)$ denotes the set of all models validating ϕ, *i.e.*, $\mathsf{KS}(\phi) = \{\mathcal{M} \mid \mathcal{M} \Vdash \phi\}$.

$\mathcal{M}, w \Vdash \top$	iff *true*
$\mathcal{M}, w \Vdash a$	iff $w \in V(a)$, $a \in \mathsf{NOM} \cup \mathsf{PROP}$
$\mathcal{M}, w \Vdash \neg\phi$	iff $\mathcal{M}, w \nVdash \phi$
$\mathcal{M}, w \Vdash \phi_1 \wedge \phi_2$	iff $\mathcal{M}, w \Vdash \phi_1$ and $\mathcal{M}, w \Vdash \phi_2$
$\mathcal{M}, w \Vdash @_a\phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and $\{w'\} = V(a)$
$\mathcal{M}, w \Vdash \langle k \rangle\phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and wR_kw'
$\mathcal{M}, w \Vdash \langle k \rangle^-\phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and $w'R_kw$
$\mathcal{M}, w \Vdash \mathbf{A}\phi$	iff $\forall w' \in W : \mathcal{M}, w' \Vdash \phi$

Table 1: Hybrid logic interpretation.

Semantics. A Gubs program is interpreted by an hybrid logic formula. Hence, it is considered as observable if and only if its corresponding formula is valid. The validity/satisfiability is defined from a Kripke model (Definition 1) gathering different possible histories. A world in a Kripke model represents an event defined by a set of agent states at a given point in the history 1.

Operator $[\ ]$ means *"observed in all possible closest futures"* and $\langle\ \rangle$ means *"observed in a possible closest future at least"* (resp. $\langle\ \rangle^-, [\ ]^-$ for the closest past). Besides, accessibility relations, $(R_k)_{k\in\tau}$, represents a "temporal evolution" in regard to some contexts. Thus, they are indexed by the non-empty parts of the set of all contexts of a program P, denoted by K_P (*i.e.*, $\tau = \mathbf{2}^{K_P} \setminus \{\varnothing\}$). A non-empty set of contexts $\varnothing \subset K \subseteq K_P$, is then a modality, *i.e.*, $\langle K \rangle, [K]$ with $\langle\ \rangle = \langle \varnothing \rangle$ by convention. Agent states are variables of the formulas and observation spots are interpreted by nominals used to identify worlds.

Let $\langle \mathsf{W}, \bullet, \Lambda \rangle$ be the set of worlds W with the concatenation operation and the neutral element, the empty world Λ and $\mathsf{F}_{\mathcal{H}}$ the set of well-formed formulas of $\mathcal{H}(\mathbf{A}, @)$, the denotational semantics is defined by four functions: $[\![.]\!] : \mathsf{P} \to \mathsf{F}_{\mathcal{H}}, [\![.]\!]_P : \mathsf{P} \to \mathsf{W} \to \mathbf{2}^{\mathsf{W}} \to \mathsf{F}_{\mathcal{H}}, [\![.]\!]_B : \mathsf{B} \to \mathsf{W} \to \mathsf{F}_{\mathcal{H}}, [\![.]\!]_R : \mathsf{R} \to \mathsf{W} \to \mathsf{F}_{\mathcal{H}}$, where $\mathsf{P}, \mathsf{B}, \mathsf{R}$ respectively stand for the set of Gubs programs, the set of agent state sets and the set of relations on attributes. $[\![.]\!]$ is the main function initiating the interpretation. $[\![.]\!]_P$ provides an interpretation for behaviours: causal relation, compartment, context and observation spot. $[\![.]\!]_B$

$[\![\{b\}]\!]$	$= \mathbf{A}\left([\![b]\!]_P(\Lambda)(\varnothing)\right)$
$[\![\epsilon]\!]_P(C)(K)$	$= \top$
$[\![b_1, b_2]\!]_P(C)(K)$	$= [\![b_1]\!]_P(C)(K) \wedge [\![b_2]\!]_P(C)(K)$
$[\![s_1 \circ\!\!\rightarrow s_2]\!]_P(C)(K)$	$= [\![s_2]\!]_B(C) \rightarrow \langle K\rangle^{-}\left([\![s_1]\!]_B(C)\right)$
$[\![s_1 \odot\!\!\rightarrow s_2]\!]_P(C)(K)$	$= [\![s_2]\!]_B(C) \rightarrow \left([\![s_1]\!]_B(C) \wedge \langle K\rangle^{-}\left([\![s_1]\!]_B(C)\right)\right)$
$[\![s_1 \oplus\!\!\rightarrow s_2]\!]_P(C)(K)$	$= [\![s_2]\!]_B(C) \rightarrow \left((\langle\ \rangle^{-} [\![s_2]\!]_B(C)) \vee (\langle K\rangle^{-} [\![s_1]\!]_B(C))\right)$
$[\![g_1, \cdots, g_n : \{r_1, \cdots, r_m\}]\!]_P(C)(K)$	$= \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} [\![r_j]\!]_R(C.g_i)$
$[\![l{::}s]\!]_P(C)(K)$	$= @_l\, [\![s]\!]_B(C)$
$[\![C'\{b\}]\!]_P(C)(K)$	$= [\![b]\!]_P(C.C')(K)$
$[\![[K]\{b\}]\!]_P(C)(K')$	$= [\![b]\!]_P(C)(K \cup K')$
$[\![s_1 + \ldots + s_n]\!]_B(C)$	$= \bigwedge_{i=1}^{n} [\![s_i]\!]_B(C)$
$[\![C'.s]\!]_B(C)$	$= [\![s]\!]_B(C.C')$
$[\![g(a)]\!]_B(C)$	$= C.g_a$
$[\![g(\overline{a})]\!]_B(C)$	$= \neg C.g_a$
$[\![g]\!]_B(C)$	$= C.g$
$[\![\overline{g}]\!]_B(C)$	$= \neg C.g$
$[\![a_1 \prec a_2]\!]_R(g)$	$= g_{a_2} \rightarrow g_{a_1}$
$[\![a_1 \not\approx a_2]\!]_R(g)$	$= g_{a_1} \rightarrow \neg g_{a_2} \wedge g_{a_2} \rightarrow \neg g_{a_1}$
$[\![a]\!]_R(g)$	$= \top$

Table 2: Semantics of Gubs. In the definition, a represents an attribute, b a behaviour, g an agent, s a set of agent states or an agent state, r a relation on attributes, C a compartment, K a set of contexts and b a set of behaviours (*i.e.*, contexts, compartments, dependences, attributes, observation spots).

defines the interpretation of agent and agent set. Finally, $[\![.]\!]_R$ corresponds to the interpretation of attribute relation. Table 2 defines these functions. Table 3 describes two interpretations of Gubs program : the first program is a negative cycle with two genes, the other one is a part of band detector pattern used in Section 6.

The observability is based on the interpretation of a program translating it to an hybrid logic formula. An observable program corresponds to a valid formula. Hence we use tableau method for hybrid logic which is proved decidable for hybrid logic fragments without the binder. For this fragment, tableau method is proved exp-time with a logarithm bottom floor[15]. According to the semantics (Table 2), the resulting formulas are in conjunctive form with at most 3 disjunctive clauses for persistent causes. Each application of the disjunction rule will create a new branch in the tree formed by

GUBS program		Hybrid logic interpretation	
$\{$		$\mathbf{A}($	
$g_1 \odot\!\!\rightarrow g_2$	,	$g_2 \rightarrow ((\langle\ \rangle^{-} g_1) \wedge g_1)$	$\wedge$
$\overline{g}_1 \circ\!\!\rightarrow \overline{g}_2$	,	$\neg g_2 \rightarrow (\langle\ \rangle^{-} \neg g_1)$	$\wedge$
$g_2 \odot\!\!\rightarrow \overline{g}_1$	,	$g_1 \rightarrow ((\langle\ \rangle^{-} \neg g_2) \wedge \neg g_2)$	$\wedge$
$\overline{g}_2 \circ\!\!\rightarrow g_1$	,	$\neg g_1 \rightarrow (\langle\ \rangle^{-} g_2)$	$\wedge$
$obs_1 :: g_1 + \overline{g}_2$	,	$@_{obs_1}(g_1 \wedge \neg g_2)$	$\wedge$
$obs_2 :: \overline{g}_1 + g_2$		$@_{obs_2}(\neg g_1 \wedge g_2)$	
$\}$		$)$	

GUBS program		Hybrid logic interpretation	
$\{$		$\mathbf{A}($	
$\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}$	,	$AHL\_high \rightarrow AHL\_mid$	$\wedge$
		$AHL\_mid \rightarrow AHL\_low$	$\wedge$
$[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(low)\}$	,	$AHL\_low \rightarrow ((\langle Light\rangle^{-} detect)$	$\wedge$
$[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(mid)\}$	,	$AHL\_mid \rightarrow ((\langle Light\rangle^{-} detect)$	$\wedge$
$[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(high)\}$	,	$AHL\_high \rightarrow ((\langle Light\rangle^{-} detect)$	$\wedge$
$\}$		$)$	

Table 3: Interpretation of GUBS program into hybrid logic.

the tableau resolution, so the complexity resulting of those formulas will be in $\mathcal{O}(3^n)$ where n is the number of lines in the normalized program.

Consistent history. Now, we formally define the consistency of the history with regards to models.

Definition 2 (Consistent history)**.** *Let $P_n(\mathcal{M})$ be the set of path ending with an observable spot on the last world for a model $\mathcal{M}$, a consistent history, $\mathcal{M}_H$, with regards to a program P is defined as follows:*

1. *$\mathcal{M}_H$ is a model.*
2. *$P_N(\mathcal{M}_H) = \{\mathcal{M}_H\}$.*
3. *$\mathcal{M}_H \vDash [\![P]\!]$*

Remark 1. *Note that, if $P_n(\mathcal{M})$ is the set of all paths in $\mathcal{M}$ such that the final point is named, $\forall \mathcal{M}_H \in P(\mathcal{M})$, $\mathcal{M} \Vdash [\![P]\!] \Rightarrow \mathcal{M}_H \Vdash [\![P]\!]$.*

Notice that, if a program is validated by a model, all the histories are validated.

5 Compilation

At compile time, a program is transformed to a structure (*e.g.*, a DNA sequence) while inserted in a vector cell (such as a bacteria), that should behave according to the programmed specification. The structure will result in an assembly of several devices stored in a library of components (*e.g.*, parts registry). As the design relates here to a behavioural/functional description, we need to bridge the gap between structural and functional description. This stage is called the *functional synthesis*. The issue is to select a set of components whose assembly preserves the behaviour of the program. To achieve this goal, a GUBS program is associated to each component to describe its behaviour. Thereby, the component assembly corresponds to a program assembly preserving the behaviour of the compiled program. Preserving a behaviour is captured by a property called the *behavioural inclusion* formalizing the fact that the characteristic observational traits of the specified function must be recognized in traces related to the device experiments. In other words, we can construct histories consistent with the programmed behaviour from histories consistent with the device behaviour description. The behavioral inclusion is defined from the interpretation of the programs, as a logical consequence (Definition 3).

Definition 3 (Behavioral inclusion)**.** *A program Q* behaviourally includes *another program P, if and only if the interpretation of the latter is a logical consequence of the interpretation of the former:*

$$P \sqsubseteq Q \triangleq \forall \mathcal{M} : \mathcal{M} \Vdash [\![Q]\!] \implies \mathcal{M} \Vdash [\![P]\!] .$$

The behavioral inclusion is a pre-order[4] such that the empty program, denoted by ϵ, is a minimum; meaning that a program with no expected behaviour can be observed in all traces. A program whose interpretation equals $\bot$, is a maximum. Figure 2 illustrates the behavioural inclusion on a particular model P.

Observability. It may arise that no history will be consistent with a programmed behaviour. For example, the program $\{Obs :: g, \overline{g} \odot\!\!\rightarrow g\}$ is not observable in a trace. Indeed, its interpretation yields the following formula: $\mathbf{A}((@_{Obs} g) \wedge (g \rightarrow ((\langle\ \rangle^{-} \neg g) \wedge \neg g)))$, false in all models because world Obs must both satisfies g and $\neg g$ by definition of the persistent dependence. A

[4]A reflexive and transitive relation.

P	**Q**
$\{g_1 \circ\!\!\rightarrow g_3,$	$\{[k_1]\{g_0 \circ\!\!\rightarrow g_1\},$
$[k_3]\{g_3 \circ\!\!\rightarrow g_4\},$	$[k_2]\{g_0 \circ\!\!\rightarrow g_2\},$
$[k_4]\{g_3 \circ\!\!\rightarrow g_5\},$	$g_1 \circ\!\!\rightarrow g_3,$
$[k_6]\{g_8 \circ\!\!\rightarrow g_9\},$	$g_2 \circ\!\!\rightarrow g_5,$
$[k_7]\{g_8 \circ\!\!\rightarrow g_{10}\},$	$[k_3]\{g_3 \circ\!\!\rightarrow g_4\},$
$g_9 \circ\!\!\rightarrow g_{11},$	$[k_4]\{g_3 \circ\!\!\rightarrow g_5\},$
$g_{10} \circ\!\!\rightarrow g_{11},$	$g_6 \circ\!\!\rightarrow g_8,$
$a :: g_4\}$	$[k_5]\{g_6 \circ\!\!\rightarrow g_7\},$
	$[k_6]\{g_8 \circ\!\!\rightarrow g_9\},$
	$[k_7]\{g_8 \circ\!\!\rightarrow g_{10}\},$
	$g_9 \circ\!\!\rightarrow g_{11},$
	$g_{10} \circ\!\!\rightarrow g_{11},$
	$a :: g_4, b :: g_5, c : g_{11}\}$

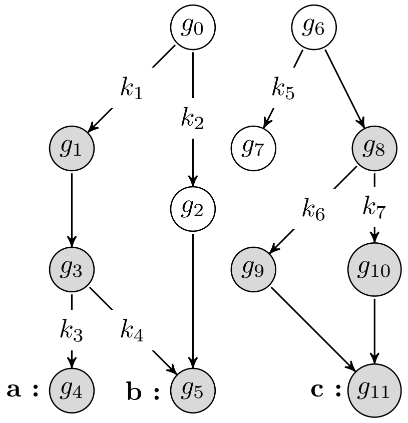

Figure 2: Behavioural inclusion example. Consistent histories of P necessary contains worlds coloured in grey. From the a,b,c observation spots, the model corresponding to worlds in grey validate the original model. Hence, the behaviour of P is included in the model of Q represented by the entire graph.

Gubs program is said to be *observable* if and only if the formula resulting from its interpretation is validated by one model at least. Hence, the interpretation of an unobservable program is a contradiction. An unobservable program can be assimilated to a programming error. The detection of such errors can be carried out at compile time using tableaux method [10] that automatically determines whether a formula is satisfiable in a model. Indeed Gubs uses a fragment of $\mathcal{H}(\mathbf{A}, @)$ named *HL(@)* logic which is decidable. The observation of the behaviour is essential to validate a program to ensure its safety. Hence, the assembled components must be always observable because a program behaviourally included in an observable program is also observable (Proposition 1).

Proposition 1. *A program behaviourally included in an observable program, **obs P**, is observable:* $\forall P, Q \in \mathcal{P} : (\boldsymbol{obs}\, Q) \wedge (P \sqsubseteq Q) \implies \boldsymbol{obs}\, P$.

5.1 Functional Synthesis

Functional synthesis is the operation whereby biological components of a library are selected and assembled to generate a device behaviourally including the designed function. The behaviour of each component is described by a GUBS program. At its simplest, the functional synthesis could be considered as a proper substitution of variables by constants. For example, in the following activation $\{G_1 \xrightarrow{+} g_2\}$, g_2 will be substituted by gene G_2, providing that component Q describes the activation $\{G_1 \xrightarrow{+} G_2\}$. However, more complex situations may arise during component selection. For example, if the activation $G_1 \xrightarrow{+} G_2$ occurs with another regulation only *i.e.*, $Q = \{G_1 \xrightarrow{+} G_2, G_3 \xrightarrow{+} G_4\}$ then the selection of Q adds a supplementary regulation.

Formally, a finite substitution is a set of mappings, $\sigma = \{v_i \mapsto b_i\}_i$, on variables and constants such that a variable can be substituted by a variable or a constant, and a constant can only substituted by itself[5]. For instance, we have: $\{\mathit{Obs}{::}G(l) + b_2, b_1 \circ\!\!\rightarrow G(l)\}[\{b_1 \mapsto B_1, b_2 \mapsto B_2, l \mapsto \mathit{Low}\}] = \{\mathit{Obs}{::}G(\mathit{Low}) + B_2, B_1 \circ\!\!\rightarrow G(\mathit{Low})\}$.

Functional synthesis rules. Functional synthesis is defined by rules (Table 4) governing the component assembly. Only the dependences and the attributes will be functionally synthesized. The observation spots are considered as annotations used for the compilation process. To ensure the correctness, each transform must preserved the original behaviour. Hence, each program resulting from the application of a rule must behaviourally include the previous one. Formally, the functional synthesis is modelled by a relation on programs denoted by $\leftarrow$, *i.e.*, $Q \leftarrow_\sigma P$ where P is the initial program and Q the transformed one, such that each rule insures that: *$Q \leftarrow_\sigma P$ is correct in regard to a substitution σ, that is $P[\sigma] \sqsubseteq Q[\sigma]$ and $Q[\sigma]$ is observable.* Also notice that the behavioural inclusion is preserved by substitution (Proposition 2).

Proposition 2. *For all substitutions σ, we have: $P \sqsubseteq Q \implies P[\sigma] \sqsubseteq Q[\sigma]$.*

Table 4 describes the functional synthesis rules[6]. Γ is a set of components representing the library. $P \subseteq_{\mathrm{Asm}} Q$ denotes the fact that program Q corresponds to an assembly including P *i.e.*, $Q = (Q_1, P, Q_2)$ where Q_1 or Q_2 may be an empty program. Rule (Inst.) describes the fact that an observable instance of a part of a component in the library is functionally synthesized.

[5] $P\sigma$ or $P[\sigma]$ represents its application on program P and identity substitutions are omitted.

[6] Rules are of the form: $\dfrac{\text{hypothesis}}{\text{conclusion}}$.

- Instantiation -

$$\frac{Q[\sigma] \subseteq_{\mathrm{Asm}} P[\sigma] \qquad \mathbf{obs}\,(Q[\sigma]) \qquad Q \in \Gamma}{Q \hookleftarrow_{\sigma} P}\ \text{(Inst.)}$$

- Commutativity, Contraction -

$$\frac{Q \hookleftarrow_{\sigma} P, P'}{Q \hookleftarrow_{\sigma} P', P}\ \text{(Com.)} \qquad \frac{Q \hookleftarrow_{\sigma} P}{Q \hookleftarrow_{\sigma} P, P}\ \text{(Cont.)}$$

- Assembly -

$$\frac{Q \hookleftarrow_{\sigma} P \qquad Q' \hookleftarrow_{\sigma'} P' \qquad \sigma|_{\mathrm{VA}(P)\cap\mathrm{VA}(P')} = \sigma'|_{\mathrm{VA}(P)\cap\mathrm{VA}(P')} \qquad \mathbf{obs}\,(Q[\sigma], Q'[\sigma'])}{Q, Q' \hookleftarrow_{\sigma\cup\sigma'} P, P'}\ \text{(Asm.)}$$

Table 4: Functional synthesis rules

Rule (Com.) expresses the commutativity of the assembly. Rule (Cont.) contracts the redundant formulation of programs. Finally, Rule (Asm.) details the conditions for an assembly of two components, each representing a functional synthesis of a part of the designed function. A detailed example of their use on a real case is given in Section 6.

Theorem 1. *The functional synthesis rules (Table 4) are correct.*

Another set of rules, more specifically devoted to dependences (Table 5), defines the alternate possibilities to express similar behaviours. The table also includes the rules for agent sets. Rule (Trans.) expands the chain of the persistent dependences ($S_1 \odot\!\!\rightarrow S_3$) by adding intermediary dependence (S_2) to refine a pathway. Rule (N2P.) weaken a normal dependence ($S_1 \circ\!\!\rightarrow S_2$) to

- Dependences -

$$\frac{Q \hookleftarrow_{\sigma} S_1 \odot\!\!\rightarrow S_2, S_2 \odot\!\!\rightarrow S_3, \Delta}{Q \hookleftarrow_{\sigma} S_1 \odot\!\!\rightarrow S_3, \Delta}\ \text{(Trans.)} \qquad \frac{Q \hookleftarrow_{\sigma} S_1 \odot\!\!\rightarrow S_2, \Delta}{Q \hookleftarrow_{\sigma} S_1 \circ\!\!\rightarrow S_2, \Delta}\ \text{(N2P.)}$$

$$\frac{Q \hookleftarrow_{\sigma} S_1 \circ\!\!\rightarrow S_2, \Delta}{Q \hookleftarrow_{\sigma} S_1 \oplus\!\!\rightarrow S_2, \Delta}\ \text{(R2N.)}$$

- Agent states -

$$\frac{S_1 + S_2}{S_2 + S_1}\ \text{(SCom.)} \qquad \frac{S + s}{S + s + s}\ \text{(SCont.)} \qquad \frac{S + s}{S}\ \text{(Incl.)}$$

Table 5: Rules for the dependences and the agent states. S_i stands for a collection, $s_1 + \ldots + s_n$, of agent states, including negation, and Δ stands for the rest of the program.

a persistent one ($S_1 \odot\!\!\rightarrow S_2$) since the latter is a normal dependence with an additional property. And Rule (R2N.) weaken a residual dependence($S_1 \oplus\!\!\rightarrow S_2$) to a normal dependence ($S_1 \circ\!\!\rightarrow S_2$), since normal dependence is also residual dependence with a repetition of the effect restricted to one step. According to these rules, all the dependence chains can be implemented with persistent dependences. Final rules are devoted to agent states. Rule (SCom.) and (SCont.) describe the propriety of + operator which is a logical $\wedge$. Finally, (Incl.) specify that a behaviour can be extended with another unless the original one still present.

Theorem 2. *Dependences rules are correct according to the model specification, and agent states rules are correct according to logic operators (Table 5).*

A possible algorithm for the assembly could be based on a combinatorial application of the rules. However, such algorithm may reveal inefficient in practice. The conditions for an efficient algorithm of compilation should be based on an internal representation of a program, as a set of contextualized dependences with attributes, $\{\{A, [K]S_1 \circledast\!\!\rightarrow S_2\}\}$,where $\circledast\!\!\rightarrow$ stands for any kind of dependence, such that A, K, S_1, S_2 are respectively: a set of attributes specification related to the agent involved in the dependency, a set of contexts and sets of agent states. Any program can be encoded under this representation from a normal form of the program (not detailed here). Accordingly, the problem solved by the compilation algorithm can be defined as follows (Definition 4):

Definition 4 (Functional Synthesis Problem (FSP))**.** *Let $\Gamma = \{Q_i\}_{1 \le i \le n}$ be set where each Q_i is a set of contextualized dependences with attributes and P a set of contextualized dependences with attributes, can we find the smallest observable subset of components $C \subseteq \Gamma$, such that there exists a substitution σ so that its application on the components of C form a cover of $P[\sigma]$,i.e., $\exists \sigma : P[\sigma] \subseteq \bigcup_{Q_j \in C} Q_j[\sigma] \wedge \boldsymbol{obs}\, C$.*

As the set cover problem is reducible to this problem, the problem is NP-complete (Proposition 3).

Proposition 3. *The Functional Synthesis Problem is NP-Complete.*

5.2 Compilation Steps

In this section, we detail the main steps of the compilation performing the functional synthesis. The result of the compilation is composed of a component list and a substitution list attributing a constant to each variable of the compiled program. The resolution is oriented towards a heuristic algorithm aiming at finding a minimal set of components covering the behavior of a program. Besides, extension capabilities are also considered for facilitating further software developments. Mainly, these developments would improve the component selection, notably by integrating biological compatibility between components without being necessary mentioned explicitly in the program in order to ease the programmer task. Hence, the design of the compiler must take the both requirements in consideration: the functional synthesis and the software sustainability. These requirements orient the development towards the use of a meta optimization heuristic, and more precisely an evolutionary algorithm providing a suitable framework for the resolution of the functional synthesis while facilitating the further developments..

Evolutionary algorithm [13, 21] is a class of meta heuristic optimization algorithms inspired by Darwinian evolution principles mimicking the biological evolution process: evolutionary algorithm selects candidate solutions and stochastically makes them evolve by recombination and mutation leading to improve their quality quantified by a fitness function. Generally speaking, evolutionary algorithm solves a multi-objective optimization problem specified as follows [40]:

$$\text{minimize}\, F(x) = (f_1(x), \ldots, f_n(x)) \text{ such that } x \in X,$$

where X is a set of viable solutions/individuals chosen in a domain X' and validated by a predicate p (*i.e.*, $X = \{x \in X' \mid p(x)\}$) and F is a sequence of objective/fitness functions, $f_i : X \to \mathbb{R}$.

Accordingly, the application of evolutionary algorithm requires to specify the three elements (the encoding of individual $x \in X$, the viability constraint p and the fitness functions f_i) in accordance with the concerned problem, related here to FSP.

Individual. An individual stands for a proposal for solving FSP. It represents a subset of components $C = \{Q_i\}_i$ chosen in database Γ. Then, as individuals correspond to finite subsets of a reference set (database), they are implemented by boolean vectors of size $|\Gamma|$ such that 1 identifies the selected elements and 0 for the others.

Fitness functions. The fitness functions guide the selection of viable individuals to improve the synthesis quality. By definition of FSP, the number of components (*i.e.*, the number of elements equal to 1 in a vector) is necessary a fitness function since we aim at minimizing it. However, other fitness functions may be added for a better component selection guidance, notably by accounting biological aspects. As evolutionary algorithm deals with multiple objectives, their addition is technically easy. The focus is then rather puts on the ability to properly model biological constraints quantitatively. This challenging problem is not studied in the article but considered as a working perspective.

Viability constraints. The viability constraints are related to the observability of an individual on one hand, and the ability to determine whether an individual behaviorally includes the program to compile on the other hand. A possible approach to verify the observability can be achieved by translating the program into formula and then by applying a tableau method to verify the satisfiability of the resulting formula. However, the exponential complexity of the algorithm would make its use impractical for some cases. To circumvent this potential problem, we orient the validation of the observability to another method, called the *strong observability* (**Obs**), based on the syntax of the program determining a necessary condition for the observability (*i.e.*, $\mathbf{Obs}(P) \implies \mathbf{obs}(P)$). Basically, a program is not observable if the formula describing its semantics is an unsatisfiable formula such as a variable and its negation. Hence, no Kripke model validates such formula. In the context of Gubs, a such situation comes from the simultaneous occurrence of incompatible agent states. An incompatible pair of agent states corresponds to: an agent state and its negation (*e.g.*, $g, \bar{g}$), agent states with mutually excluded attributes (*e.g.*, $g(\mathit{Phos}), g(\mathit{UnPhos})$ with $\mathit{Phos} \not\approx \mathit{UnPhos}$), or an agent state expressed by an attribute and the negation of another agent state by an attribute with less capacity than the first one (*e.g.*, $g(\mathit{High}), \overline{g(\mathit{Low})}$ with $\mathit{Low} < \mathit{High}$). An incompatible pair of agent states arises in the following cases: either an incompatible pair occur in left or right side of a dependence, if there exists an incompatible pair in the agent states of a chain of persistent dependences since the cause always occur with the effect by definition of the persistence. Otherwise, the program is observable.

Therefore, the strong observability consists of checking whether these cases appear in the program that can be achieved in polynomial time from its text and the definition of attributes by analyzing: the agent states of the dependences and the agent states for pair of persistent dependences in a chain of persistent dependences.

The behavioral inclusion implies to "match" each causal dependence of the program with a causal dependence of an individual while respecting the nature and the structure of the dependences. The matching is a pure syntactic process proceeding on text of programs that is assimilated to an unification of terms. For FSP, the unification algorithm is applied on associative commutative and idempotent function (ACI-unification). Indeed, in Table 4 and 5, Rules (Com.), (SCom.), and Rules (Cont.), (SCont.) identify the respective role of the commutativity and the idempotency in the synthesis whereas Rule (Inst.) (Table 4) characterizes the outcome of unification. ACI-unification [2] solves equation of terms using associative, commutative and idempotent operators by determining a list of substitution (unifier). In our context, the objective is to find a substitution σ and a part Q of an individual such that $Q = P\sigma$ where P is the program to compile. Thus, a program is viewed as a term representing an union of causal dependences set such that a causal relation stands for a symbolic (non commutative) binary function applied on two sets containing variables and constants as parameters. ACI-unification problem is NP-Complete [25, 2]. However, in literature, efficient heuristics covering the different variations of ACI-unification problem such as set unification have been proposed [35, 18, 36] and can be adapted to our case.

Figure 3 describes the compilation process of a Gubs program. Additionally, several CAD environments for synthetic biology design use meta optimization algorithms in a similar way than genetic programming for the combinatorial logic function generation where the design of logical circuits complying to an expected input/output profile specifying their behavior, is automatically achieved by an evolutionary algorithm. Pioneer work on synthetic biology has been undertaken in [19] for the design of a bistable oscillatory circuit, by using a genetic algorithm. In [34], the authors apply a Monte-Carlo algorithm to synthesize a *de-novo* ribosome binding site. In [33, 32], the authors propose an evolutionary algorithm based environment to automatically produce small regulatory networks from a library of biological components that match with a behavior represented by an evolution profile of RNA concentration for some target genes. These works evidence the applicability of evolutionary algorithm for automatic design in synthetic biology. Although the use of evolutionary algorithm in Gubs differs, it supports the same objective and opens up the possibility of an integration based on the same optimization framework.

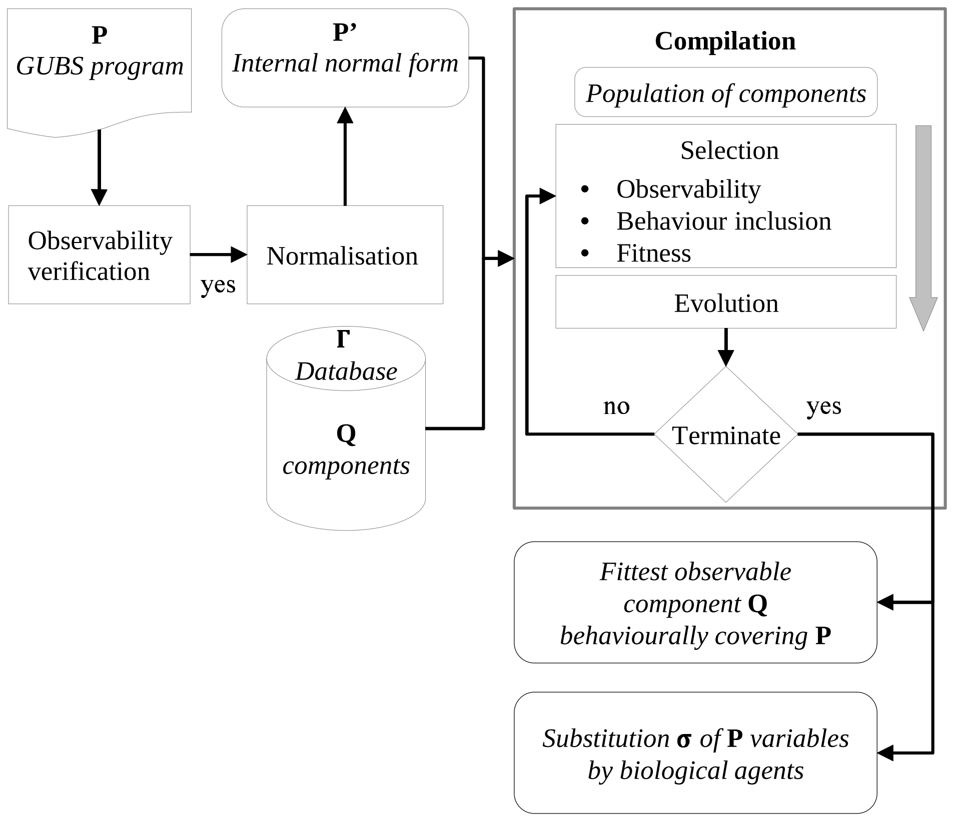

Figure 3: Overview of compilation process of GUBS.

6 Example

The application of rules underlying the compilation process is here described in a real case for the design of the Band Detector proposed in [3]. This example explains how from a simple abstract definition of the functionality a complex design can be synthesized. Accordingly, GUBS may be used to describe a behaviour with a high-level of abstraction as well as a low-level, detailing the components involved in the design. We introduce each step of the different transforms from the high-level program to the low-level one in the example. Each is ruled by the application of rewriting rules defined in Tables 4, 5, ensuring its correctness and so, its functional safety in the context of open system. The design of the example aims at forming patterns of different colours in a population of bacteria exploiting the quorum sensing

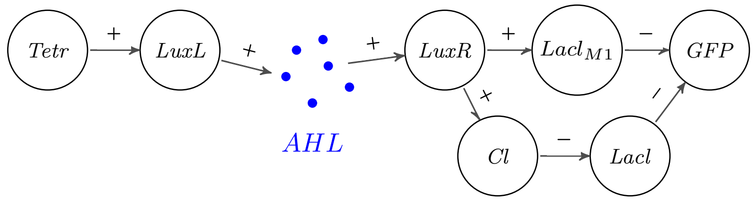

Figure 4: The band detector regulatory circuit.

phenomenon by staining with fluorescent protein (GFP). The amount of molecules of interest a cell receives depends on its relative position to the cell diffusing the molecule of interest controlled by an external event: the more the cell is far from the source, the fewer is the amount of molecules received. The activation or inhibition of the fluorescent protein due to the concentration will distinguish the bands surrounding the source. In the original design, the protein does not fluoresce in an intermediary band.

From a computing standpoint, we can assimilate the design to a message transmission coupled to a sensor/actuator responsible for fluorescence, then leading to a concise Gubs program presented below: the diffusive molecule is *AHL* which production is controlled by a context and the observation is applied on *GFP*. Two categories of cells are defined: the *Sender* and the *Receiver*. Therefore, two Gubs programs identify the two cell types.

$$\begin{aligned} \mathit{Sender} = \quad & \{\ \mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}, \\ & [Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(low), detect \circ\!\!\rightarrow \mathsf{AHL}(mid), detect \circ\!\!\rightarrow \mathsf{AHL}(high)\}\} \\ \mathit{Receiver} = \quad & \{\ \mathsf{AHL}(low) \circ\!\!\rightarrow \overline{\mathsf{GFP}}, \mathsf{AHL}(mid) \circ\!\!\rightarrow \mathsf{GFP}, \mathsf{AHL}(high) \circ\!\!\rightarrow \overline{\mathsf{GFP}}, obs_1{::}\mathsf{GFP}, obs_2{::}\overline{\mathsf{GFP}}\} \end{aligned}$$

Figure 4 describes the original genetic circuit used in the article. The diffusible molecule is the constant *AHL*. *detect* is a variable used to represent the initial action of Light activating *AHL* diffusion. The gene *LuxR* has three activation thresholds: at Level 2, it activates both *LaclM1* and *Cl*, at level 1, the amount of *AHL* only allows activation of *Cl*, and finally, at level 0, none are activated.

To ease compilation follow-up, we label each dependency of the sender-receiver program (Table 6). We show that from the sender-receiver program, we obtain the original design by applying the aforementioned rules with an appropriate selection of components (Table 7). The regulations of Figure 4 are described in a Gubs program translating in terms of dependences and relations on their attributes their regulatory action. We focus here on some illustrative steps of the sender program compilation. The complete functional synthesis is given next. The compilation consists in finding the appropriate components whose assembly behaviourally includes the sender-receiver program, with the particularity that the diffusive molecule must be the same in both programs.

Sender	*Receiver*
$P_{11} = \{[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(low)\}\}$	$P_{21} = \{\mathsf{AHL}(low) \circ\!\!\rightarrow \overline{\mathsf{GFP}}\}$
$P_{12} = \{[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(mid)\}\}$	$P_{22} = \{\mathsf{AHL}(mid) \circ\!\!\rightarrow \mathsf{GFP}\}$
$P_{13} = \{[Light]\{detect \circ\!\!\rightarrow \mathsf{AHL}(high)\}\}$	$P_{23} = \{\mathsf{AHL}(high) \circ\!\!\rightarrow \overline{\mathsf{GFP}}\}$

with $\{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}\}$ as attributes of *AHL*.

Table 6: Separation of the dependences.

$$\begin{aligned}
Q_1 &= \{[\text{Light}]\{detect \circ\!\!\rightarrow \mathsf{Tetr}\}\}\\
Q_2 &= \{\mathsf{Tetr} \xrightarrow{+} \mathsf{LuxI}\}\\
Q_3 &= \{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}, \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(low), \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(mid), \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(high)\}\\
Q_4 &= \{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}, \mathsf{LuxR}{:}\{low \not\approx \{mid \prec high\}\},\\
&\quad \mathsf{AHL}(mid) \circ\!\!\rightarrow \mathsf{LuxR}(mid), \mathsf{AHL}(high) \circ\!\!\rightarrow \mathsf{LuxR}(high)\}\\
Q_5 &= \{\mathsf{LuxR}{:}\{low \not\approx \{mid \prec high\}\}, \mathsf{LuxR}(mid) \xrightarrow{+} \mathsf{CI}, \mathsf{LuxR}(high) \xrightarrow{+} \mathsf{CI} + \mathsf{LacIM1}\}\\
Q_6 &= \{\mathsf{CI} \xrightarrow{-} \mathsf{LacI}\}\\
Q_7 &= \{\mathsf{LacIM1} \xrightarrow{-} \mathsf{GFP}\}\\
Q_8 &= \{\mathsf{LacI} \xrightarrow{-} \mathsf{GFP}\}
\end{aligned}$$

Table 7: Part of the database dedicated to the Band Detector.

In the sequel, P_{ij} refers to j^{th} normalized causal relation of the program P_i where P_1 is the Sender and P_2 is the Receiver. Let us consider P_{11} whose compilation is close to P_{12} and P_{13}. Notice that P_{11} cannot be directly instantiated with any component because, on the one hand, the component Q_1 contains a context like P_{11} but applied on gene *Tetr* instead of *AHL*, and on the other hand Q_3 has the *AHL* molecule but no context is defined. So, to match P_{11} with the components Q_1, Q_2 and Q_3, first, the normal dependence is converted to persistent one (Rule (N2P.)).

$$\frac{Q_1, Q_2, Q_3 \leftarrow_\sigma \{[light]\{detect \odot\!\!\rightarrow AHL(low)\}\}}{Q_1, Q_2, Q_3 \leftarrow_\sigma P_{11}}\ \text{(N2P.)}$$

Thereby, the resulting dependence can also be separated to match the assembly Q_1, Q_2, Q_3 by applying (Trans.) rule twice. v_1 and v_2 are fresh variables.

$$\dfrac{\dfrac{Q_1, Q_2, Q_3 \leftarrow_\sigma P'_{11} = \{[light]\{detect \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow AHL(low)\}}{Q_1, Q_2, Q_3 \leftarrow_\sigma [light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow AHL(low)\}}\text{(Trans.)}}{Q_1, Q_2, Q_3 \leftarrow_\sigma [light]\{detect \odot\!\!\rightarrow AHL(low)\}}\text{(Trans.)}$$

Finally, we obtain a new program program P'_{11} compatible with Q_1, Q_2, Q_3, and each variable is substituted by a constant (biological element) with the application of Rule (Inst.). For P'_{11} we have:

$$\dfrac{Q_1, Q_2, Q_3[\sigma = \{light/Light, v_1/Tetr, v_2/Luxl\}] \subseteq_{Asm} P'_{11}[\sigma] \qquad obs(Q_1, Q_2, Q_3[\sigma])}{Q_1, Q_2, Q_3 \leftarrow_\sigma [light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow AHL(low)\}}\text{(Inst.)}$$

By following this scheme for P_{12} and P_{13}, we respectively obtain P'_{12} and P'_{13}. The final assembly corresponds to the functional synthesis of *Sender* program.

$$\dfrac{\begin{array}{ccc} Q_1, Q_2, Q_3 \leftarrow_\sigma P'_{11} & Q_1, Q_2, Q_3 \leftarrow_\sigma P'_{12} & Q_1, Q_2, Q_3 \leftarrow_\sigma P'_{13} \\ \vdots & \vdots & \vdots \\ Q_1, Q_2, Q_3 \leftarrow_\sigma P_{11} & Q_1, Q_2, Q_3 \leftarrow_{\sigma'} P_{12} & Q_1, Q_2, Q_3 \leftarrow_{\sigma''} P_{13} \end{array}}{Q_1, Q_2, Q_3 \leftarrow_{\sigma \cup \sigma' \cup \sigma''} P_{11}, P_{12}, P_{13}}\text{(Asm.)}$$

In conclusion, the functional synthesis generates the original genetic circuit (Figure 4) from the sender program. A similar approach can be also applied to obtain the receiver program (see the complete proof below 6).

$$\begin{array}{rcl} Sender & = & \{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\}, [\mathsf{Light}]\{detect \odot\!\!\rightarrow \mathsf{Tetr}\}, \\ & & \mathsf{Tetr} \xrightarrow{+} \mathsf{Luxl}, \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(low), \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(mid), \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(high)\} \end{array}$$

Complete Compilation of the Band Detector

This section details step by step the application of the rules to perform the functional synthesis of the Band Detector example Tables 8,9 and 10.

- Sender -

$$\frac{Q_1, Q_2, Q_3[\sigma = \{detect/Detect, light/Light, v_1/Tetr, v_2/Luxl\}] \subseteq_{Asm} P'_{11}[\sigma] \qquad obs(Q_1, Q_2, Q_3[\sigma])}{Q_1, Q_2, Q_3 \vdash_{\sigma} P'_{11}\}}\ (\text{Inst.}) \qquad \mathbf{3}$$

$$\frac{Q_1, Q_2, Q_3[\sigma' = \{detect/Detect, light/Light, v_3/Tetr, v_4/Luxl\}] \subseteq_{Asm} P'_{12}[\sigma'] \qquad obs(Q_1, Q_2, Q_3[\sigma'])}{Q_1, Q_2, Q_3 \vdash_{\sigma'} P'_{12}}\ (\text{Inst.}) \qquad \mathbf{3}$$

$$\frac{Q_1, Q_2, Q_3[\sigma'' = \{detect/Detect, light/Light, v_5/Tetr, v_6/Luxl\}] \subseteq_{Asm} P'_{13}[\sigma''] \qquad obs(Q_1, Q_2, Q_3[\sigma''])}{Q_1, Q_2, Q_3 \vdash_{\sigma''} P'_{13}}\ (\text{Inst.}) \qquad \mathbf{3}$$

$$\cfrac{\cfrac{\cfrac{P'_{11} = [light]\{detect \odot\!\!\to v_1, v_1 \odot\!\!\to v_2, v_2 \odot\!\!\to AHL(low)\}}{[light]\{detect \odot\!\!\to v_1, v_1 \odot\!\!\to AHL(low)\}}\ (\text{Trans.})}{[light]\{detect \odot\!\!\to AHL(low)\}}\ (\text{Trans.})}{P_{11}}\ (\text{N2P.}) \qquad \mathbf{2}$$

$$\cfrac{\cfrac{\cfrac{P'_{12} = [light]\{detect \odot\!\!\to v_3, v_3 \odot\!\!\to v_4, v_4 \odot\!\!\to AHL(mid)\}}{[light]\{detect \odot\!\!\to v_3, v_3 \odot\!\!\to AHL(mid)\}}\ (\text{Trans.})}{[light]\{detect \odot\!\!\to AHL(mid)\}}\ (\text{Trans.})}{P_{12}}\ (\text{N2P.}) \qquad \mathbf{2}$$

$$\cfrac{\cfrac{\cfrac{P'_{13} = [light]\{detect \odot\!\!\to v_5, v_5 \odot\!\!\to v_6, v_6 \odot\!\!\to AHL(high)\}}{[light]\{detect \odot\!\!\to v_5, v_5 \odot\!\!\to AHL(high)\}}\ (\text{Trans.})}{[light]\{detect \odot\!\!\to AHL(high)\}}\ (\text{Trans.})}{P_{13}}\ (\text{N2P.}) \qquad \mathbf{2}$$

$$\frac{Q_1, Q_2, Q_3 \vdash_{\sigma} P_{11} \qquad Q_1, Q_2, Q_3 \vdash_{\sigma'} P_{12} \qquad Q_1, Q_2, Q_3 \vdash_{\sigma''} P_{13}}{Q_1, Q_2, Q_3 \vdash_{\sigma \cup \sigma' \cup \sigma''} P_{11}, P_{12}, P_{13}}\ (\text{Asm.}) \qquad \mathbf{1}$$

1. Firstly, we split the sender program in three sub programs P_{11}, P_{12}, and P_{13}, each corresponding to a causal relation.

2. Initially, P_{11}, P_{12} and P_{13} don't match with any component of the database, so we extend them (P'_{11}, P'_{12} and P'_{13}) to find a matching.

3. Finally, we can match P'_{11}, P'_{12} and P'_{13} with components Q_1, Q_2, Q_3.

Table 8: Sender compilation.

- Receiver -

$$\frac{Q_4, Q_5, Q_6, Q_8[\sigma = \{v_1/LuxR, v_2/Cl, v_3/Lacl\}] \subseteq_{Asm} P'_{21}[\sigma] \qquad obs(Q_4, Q_5, Q_6, Q_8[\sigma])}{Q_4, Q_5, Q_6, Q_8 \hookleftarrow_{\sigma} P'_{21}}\text{(Inst.)} \qquad \mathbf{3}$$

$$\frac{Q_4, Q_5, Q_6, Q_8[\sigma' = \{v_4/LuxR, v_5/Cl, v_6/Lacl\}] \subseteq_{Asm} P'_{22}[\sigma'] \qquad obs(Q_4, Q_5, Q_6, Q_8[\sigma'])}{Q_4, Q_5, Q_6, Q_8 \hookleftarrow'_{\sigma} P'_{22}}\text{(Inst.)} \qquad \mathbf{3}$$

$$\frac{Q_4, Q_5, Q_7[\sigma'' = \{v_7/LuxR, v_8/LacM1\}] \subseteq_{Asm} P'_{23}[\sigma''] \qquad obs(Q_4, Q_5, Q_7[\sigma''])}{Q_4, Q_5, Q_7 \hookleftarrow''_{\sigma} P'_{23}}\text{(Inst.)} \qquad \mathbf{3}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{P'_{21} = AHL(low) \odot\!\!\to v_1, v_1 \odot\!\!\to v_2, v_2 \odot\!\!\to v_3, v_3 \odot\!\!\to \overline{GFP}}{AHL(low) \odot\!\!\to v_1, v_1 \odot\!\!\to v_2, v_2 \odot\!\!\to \overline{GFP}}\text{(Trans.)}}{AHL(low) \odot\!\!\to v_1, v_1 \odot\!\!\to \overline{GFP}}\text{(Trans.)}}{AHL(low) \odot\!\!\to \overline{GFP}}\text{(Trans.)}}{P_{21}}\text{(N2P.)} \qquad \mathbf{2}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{P'_{22} = AHL(mid) \odot\!\!\to v_4, v_4 \odot\!\!\to v_5, v_5 \odot\!\!\to v_6, v_6 \odot\!\!\to GFP}{AHL(mid) \odot\!\!\to v_4, v_4 \odot\!\!\to v_5, v_5 \odot\!\!\to GFP}\text{(Trans.)}}{AHL(mid) \odot\!\!\to v_4, v_4 \odot\!\!\to GFP}\text{(Trans.)}}{AHL(mid) \odot\!\!\to GFP}\text{(Trans.)}}{P_{22}}\text{(N2P.)} \qquad \mathbf{2}$$

$$\cfrac{\cfrac{\cfrac{P'_{23} = AHL(high) \odot\!\!\to v_7, v_7 \odot\!\!\to v_8, v_8 \odot\!\!\to \overline{GFP}}{AHL(high) \odot\!\!\to v_7, v_7 \odot\!\!\to \overline{GFP}}\text{(Trans.)}}{AHL(high) \odot\!\!\to \overline{GFP}}\text{(Trans.)}}{P_{23}}\text{(N2P.)} \qquad \mathbf{2}$$

$$\frac{Q_4, Q_5, Q_6, Q_8 \hookleftarrow_{\sigma} P_{21} \qquad Q_4, Q_5, Q_6, Q_8 \hookleftarrow_{\sigma'} P_{22} \qquad Q_4, Q_5, Q_7 \hookleftarrow_{\sigma''} P_{23}}{Q_4, Q_5, Q_6, Q_7, Q_8 \hookleftarrow_{\sigma \cup \sigma' \cup \sigma''} P_{21}, P_{22}, P_{23}}\text{(Asm.)} \qquad \mathbf{1}$$

1. As for the sender program, the receiver program is split in three subprograms P_{21}, P_{22} and P_{23}, each corresponding to a cause differing by their AHL concentration.

2. P_{21}, P_{22} and P_{23} initially do not match with any component in the database. So, we extend them (P'_{21}, P'_{22} and P'_{23}) by applying extension rule (Ext.).

3. P'_{21} and P'_{23} describe the same behaviour for two different AHL concentrations. Thus, their respective variable (v_1 and v_7) is substituted by the same constant LuxR. P'_{22} describes the presence of GFP matching with the components $\{Q_4, Q_5, Q_6, Q_8\}$. Finally, P'_{21}, P'_{22} and P'_{23} match with components $\{Q_4, Q_5, Q_6, Q_7, Q_8\}$.

Table 9: Receiver compilation.

- Final design -

Sender	
$\{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\},$	$[\text{Light}]\{detect \circ\!\!\rightarrow \mathsf{Tetr}\},$
$\mathsf{Tetr} \xrightarrow{+} \mathsf{LuxL},$	$\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(low),$
$\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(mid),$	$\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(high)\}$
Receiver	
$\{\mathsf{AHL}{:}\{low \not\approx mid \not\approx high\},$	$\mathsf{LuxR}{:}\{low \not\approx \{mid < high\}\},$
$\mathsf{AHL}(mid) \circ\!\!\rightarrow \mathsf{LuxR}(mid),$	$\mathsf{AHL}(high) \circ\!\!\rightarrow \mathsf{LuxR}(high),$
$\mathsf{LuxR}(mid) \xrightarrow{+} \mathsf{CI},$	$\mathsf{LuxR}(high) \xrightarrow{+} \mathsf{LacIM1},$
$\mathsf{CI} \xrightarrow{-} \mathsf{LacI},$ $\mathsf{LacIM1} \xrightarrow{-} \mathsf{GFP},$	$\mathsf{LacI} \xrightarrow{-} \mathsf{GFP}\}$

Table 10: Complete band detector compilation.

7 Conclusion

In Gubs language, we propose to characterize a programming paradigm abstracting the molecular interactions in the context of open system, that differs to an approach dedicated to biological system modelling. Accordingly, the interactions are symbolized by causal dependences whose interpretation is driven by effect. We have demonstrated the proof-of-concept of the compilation based on rewriting rules, and illustrated it on a realistic example. A perspective of this work is to improve the component selection by identifying the biological parameters and define the appropriate fitness function for a selection also accounting quantitative biological constraints.

Acknowledgements

The funding for most of this work is granted by the Anr synbiotic (Anr blanc 0307 01) and we would like to thank the colleagues of this project for their fruitful discussions and, Serena Cerrito and Amelie David for their helpful comments.

Appendix

program	::= {behaviour}
behaviour	::= behaviour, behaviour | behaviour
behaviour	::= compartment | dependence | context | observation | defattributes
compartment	::= varconstant {behaviour}
observation	::= varconstant**::**words
context	::= [varconstants] {behaviour}
dependence	::= words $\circ\!\!\rightarrow$ words | words $\odot\!\!\rightarrow$ words | words $\oplus\!\!\rightarrow$ words
word	::= attribute | varconstant(attribute) | varconstant.word
words	::= words + word | word
attribute	::= varconstant | $\overline{\text{varconstant}}$
defattribute	::= varconstants : attspec
attspec	::= defspec{varconstants} | {attrels}
defspec	::= exclusion | inclusion
attrels	::= attrels, attrel | attrel
attrel	::= varconstant < varconstant | varconstant $\not\approx$ varconstant | varconstant
varconstant	::= *word* | *Word*
varconstants	::= varconstants, varconstant | varconstant

Table 11: Syntax of Gubs program

Proofs

Proposition 1. By contradiction, assume that P is unobservable, then there does not exist a model satisfying the formula. As Q is observable, we deduce that there exists models satisfying Q, but no restricted model must satisfy P, that contradicts the definition of the behavioural consequence. $\square$

Proposition 4. *Let $\psi \in F_{\mathcal{H}}$ be a formula, let $\sigma : (\mathsf{NOM} \cup \mathsf{PROP} \cup \mathsf{REL}) \rightarrow (\mathsf{NOM} \cup \mathsf{PROP} \cup \mathsf{REL})$ be a substitution on nominals, variables and relational symbols, let $\mathcal{M} = \langle W, (R_k)_{k \in \tau}, V\rangle$ be a model, we define the model $\tilde{\mathcal{M}} = \langle W, (\tilde{R}_k)_{k \in \tilde{\tau}}, \tilde{V}\rangle$ from $\mathcal{M}$ as follows:*

1. $\forall a \in \mathsf{NOM} \cup \mathsf{PROP}, \forall w \in W : w \in V(a\sigma) \iff w \in \tilde{V}(a)$
2. $\forall k \in \tilde{\tau} : wR_{k\sigma}w' \iff w\tilde{R}_k w'$;

we have: $\mathcal{M}, w \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w \Vdash \psi$.

Proof. The proof is defined by induction on the formula:

without loss of generality, we assume that ψ is in Negation Normal Form where negation occurs only immediately before variables only. Recall that every formula can be set in Negation Normal Form.

$\mathcal{M}, w \Vdash a \iff \tilde{\mathcal{M}}, w \Vdash a, a \in \mathsf{PROP} \cup \mathsf{NOM}$. By (1), we have $w \in V(a\sigma) \iff w \in \tilde{V}(a)$ leading to the equivalence.

$\mathcal{M}, w \Vdash \neg a \iff \tilde{\mathcal{M}}, w \Vdash \neg a$. By definition of the realizability relation, this is equivalent to: $\tilde{\mathcal{M}}, w \nVdash a \iff \tilde{\mathcal{M}}, w \nVdash a$. By (1), this equivalence holds.

$\mathcal{M}, w \Vdash (\psi_1 \wedge \psi_2)\sigma \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \wedge \psi_2)$. By definition of the substitution, we have to prove: $\mathcal{M}, w \Vdash (\psi_1\sigma) \wedge (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \wedge \psi_2)$. By definition of the realizability relation we can formulate the property equivalently as follows:

$$\mathcal{M}, w \Vdash (\psi_1\sigma) \wedge \tilde{\mathcal{M}}, w \Vdash (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_1 \wedge \tilde{\mathcal{M}}, w \Vdash \psi_2.$$

By induction hypothesis, we have: $\tilde{\mathcal{M}}, w \Vdash (\psi_1\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_1$ and $\tilde{\mathcal{M}}, w \Vdash (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_2$, implying the previous condition.

$\mathcal{M}, w \Vdash (\psi_1 \vee \psi_2)\sigma \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \vee \psi_2)$. The proof is similar to the proof of the previous item ($\wedge$).

$\mathcal{M}, w \Vdash (@_a\psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash @_a\psi$. By definition of the substitution we have to prove that: $\mathcal{M}, w \Vdash (@_{a\sigma}\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash @_a\psi$ By definition of the realizability relation, this is equivalent to:

$$\exists w' \in W : w \in V(a\sigma) \wedge \mathcal{M}, w' \Vdash \psi\sigma \iff \exists w'' \in W : w'' \in \tilde{V}(a)\sigma \wedge \tilde{\mathcal{M}}, w'' \Vdash \psi.$$

By setting $w' = w''$, from (1) we have: $w' \in V(a\sigma) \iff w' \in V(a)$. By induction hypothesis, we have: $\mathcal{M}, w' \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w' \Vdash \psi$. The both last properties imply that:

$$\exists w' \in W : w \in V(a\sigma) \wedge \mathcal{M}, w' \Vdash \psi\sigma \iff \exists w' \in W : w' \in \tilde{V}(a)\sigma \wedge \tilde{\mathcal{M}}, w'' \Vdash \psi,$$

which implies the initial property.

$\mathcal{M}, w \Vdash (\langle k \rangle\psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash \langle k \rangle\psi$. By definition of the substitution we prove that: $\mathcal{M}, w \Vdash \langle k\sigma \rangle\psi\sigma \iff \tilde{\mathcal{M}}, w \Vdash \langle k \rangle\psi$.

By definition of the realizability relation the condition is equivalent to:

$$\exists w' \in W : \mathcal{M}, w' \Vdash \psi\sigma \wedge w R_{k\sigma} w' \iff \exists w'' \in W : \tilde{\mathcal{M}}, w'' \Vdash \psi \wedge w\tilde{R}_k w''.$$

By setting $w' = w''$, the following equivalence holds from (2): $wR_{k\sigma}w' \iff w\tilde{R}_k w'$. By induction hypothesis, we have: $\mathcal{M}, w' \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w' \Vdash \psi$. The both last properties imply that:

$$\exists w' \in W : \mathcal{M}, w' \Vdash \psi\sigma \wedge wR_{k\sigma}w' \iff \tilde{\mathcal{M}}, w' \Vdash \psi \wedge w\tilde{R}_k w'$$

which implies the initial property.

$\mathcal{M}, w \Vdash ([k]\psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash [k]\psi$. The proof is similar to the previous item.

$\mathcal{M} \Vdash (\mathbf{E}\psi)\sigma \iff \tilde{\mathcal{M}} \Vdash \mathbf{E}\psi$. By definition of the substitution we prove that: $\mathcal{M}, w \Vdash \mathbf{E}(\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \mathbf{E}\psi$.

By definition of the realizability relation, we have:

$$\exists w \in W : \mathcal{M}, w \Vdash (\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi,$$

which is directly verified by induction hypothesis.

$\mathcal{M} \Vdash (\mathbf{A}\psi)\sigma \iff \tilde{\mathcal{M}} \Vdash \mathbf{A}\psi$. The proof is similar to the previous item.

$\square$

Proposition 2. First, let us remark that when $P \not\sqsubseteq Q$, the property is trivially verified. Besides, under the assumption $P \sqsubseteq Q$, if $Q[\sigma]$ is not observable the property is also verified because an unobservable program includes all programs behaviourally (Definition 3).

In the rest of the proof, we assume that P is behaviourally included in Q and $Q[\sigma]$ is observable (*i.e.*, $P \sqsubseteq Q$ and $\mathbf{obs}\, Q[\sigma]$). Hence, by definition of the observability there exists a model $\mathcal{M}$ such that $\mathcal{M} \Vdash [\![Q[\sigma]]\!]$. By proposition 4, we deduce that there exists a model $\tilde{\mathcal{M}}$ such that: $\tilde{\mathcal{M}} \Vdash [\![Q]\!]$. Moreover, as $P \sqsubseteq Q$ by hypothesis, there exists $\tilde{S} \subseteq \mathrm{Dom}\, \tilde{\mathcal{M}}$ such that: $\tilde{\mathcal{M}}_{\tilde{S}} \Vdash [\![P]\!]$. By construction of $\tilde{\mathcal{M}}$ we deduce that there exists a sub model of $\mathcal{M}$, denoted by $\mathcal{M}'$, complying to the properties, (1) and (2) of Proposition 4 which corresponds to $\tilde{\mathcal{M}}_{\tilde{S}}$. Moreover, we have $\mathcal{M}' \Vdash P[\sigma]$ by Proposition 4. Then we conclude that: $P[\sigma] \sqsubseteq Q[\sigma]$. $\square$

Proposition 3. By reduction to the minimum covering problem (SP5 in [20]).
The problem is in NP. Assume we have a substitution σ and $Q = \{Q_i\}_i$ a set of components, checking whether $P[\sigma]$ is included in $\bigcup_{Q_j \in C} Q_j[\sigma]$ is performed in polynomial time.
The problem is NP-complete. The reduction is performed on minimum covering problem based on an encoding of elements by dependences.
Instance: Collection X of subsets of a finite set S, and a positive integer $k < |C|$.
Question: Does X contain a cover for S of size k or less,*i.e.*, a subset $X' \subseteq X$ with $|X'| \leq k$ such that every element of S belongs to at least one member of X'?

Reduction. Each element $A \in S$ is assimilated to a constant and translated into a dependence $A \circ\!\!\rightarrow A$. Therefore, the substitution is trivially the identity. The database is X(*i.e.*, $X = \Gamma$). Finally, the result of the functional synthesis is X' (*i.e.*, $X' = C$). □

Theorem 1. First, let us remark that $P \sqsubseteq Q$ is true whenever $\mathcal{M} \not\Vdash Q$ by definition of the behavioural inclusion (Definition 3). Hence, the proof doesn't consider the trivial verified case but rather the case where $\mathcal{M} \Vdash Q$.

Inst. Directly from the definition of the behavioural inclusion (Definition 3).

Com. By definition of the semantics $[\![P, P']\!] = \mathbf{A}(\phi \wedge \phi') = \mathbf{A}(\phi' \wedge \phi) = [\![P', P]\!]$ with $[\![P]\!]_P = \phi$ and $[\![P']\!]_P = \phi'$. Thus, for all $\mathcal{M}$ we have: $\mathcal{M} \Vdash [\![P, P']\!] \iff \mathcal{M} \Vdash [\![P', P]\!]$. Hence, if $Q \sqsubseteq P, P'$ we conclude that: $Q \sqsubseteq P', P$.

Cont. Similar to the proof of (Com.).

Asm. First let us remark that $\sigma|_{\mathrm{VA}(P) \cap \mathrm{VA}(P')} = \sigma'|_{\mathrm{VA}(P) \cap \mathrm{VA}(P')}$ means that the substitution of the common variables are the same for σ and σ', leading to, $Q[\sigma \cup \sigma'] = Q[\sigma]$ and $Q'[\sigma \cup \sigma'] = Q'[\sigma']$. Let $\sigma'' = \sigma \cup \sigma'$. Then, we have the following property by definition of the semantics (Table 2) and σ''.

$$\forall \mathcal{M} \in \mathsf{KS}(\left[\!\left[(Q, Q')[\sigma'']\right]\!\right]) : \mathcal{M} \Vdash [\![Q[\sigma]]\!] \wedge \mathcal{M} \Vdash \left[\!\left[Q'[\sigma']\right]\!\right].$$

Notice that the set of models, $\mathsf{KS}([\![(Q, Q')[\sigma'']]\!])$, is not empty since, by hypothesis,
obs $(Q[\sigma], Q'[\sigma'])$ holds. As $Q \leftarrow_\sigma P$ and $Q' \leftarrow_{\sigma'} P'$, any model

validating Q (resp. Q') also validates P, (resp. P') by definition of the functional synthesis. Then, we deduce that:

$$\forall \mathcal{M} \in \mathsf{KS}(\llbracket (Q,Q')[\sigma''] \rrbracket) : \mathcal{M} \Vdash \llbracket P[\sigma] \rrbracket \wedge \mathcal{M} \Vdash \llbracket P'[\sigma'] \rrbracket \,.$$

Then, we conclude that:

$$\forall \mathcal{M} \in \mathsf{KS}(\llbracket (Q,Q')[\sigma''] \rrbracket) : \mathcal{M} \Vdash \llbracket (P,P')[\sigma''] \rrbracket \,.$$

□

References

[1] P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2008.

[2] F. Baader and W. Snyder. Unification Theory. In Andrei Robinson, Alan and Voronkov, editor, *Handbook of automated reasoning*, chapter 8, pages 441–533. The MIT Press, 2001. doi:10.1016/B978-044450813-3/50010-2.

[3] S. Basu, Y. Gerchman, C. H. Collins, F. H. Arnold, and R. Weiss. A Synthetic Multicellular System for Programmed Pattern Formation. *Nature*, 434(7037):1130–1134, April 2005. doi:10.1038/nature03461.

[4] J. Beal, T. Lu, and R. Weiss. Automatic Compilation from High-Level Biologically-Oriented Programming Language to Genetic Regulatory Networks. *PLoS ONE*, 6(8):e22490, August 2011. doi:10.1371/journal.pone.0022490.

[5] L. Bilitchenko, A. Liu, S. Cheung, E. Weeding, B. Xia, M. Leguia, J C. Anderson, and D. Densmore. Eugene–A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. *PloS one*, 6(4):e18882, January 2011. doi:10.1371/journal.pone.0018882.

[6] P. Blackburn, J. F. A. K. van Benthem, and F. Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., December 2006. doi:10.1016/S1570-2464(07)80017-6.

[7] T. Braüner. *Hybrid Logic and Its Proof-Theory.* Springer, 2011. doi:10.1007/978-94-007-0002-4.

[8] Y. Cai, M. W Lux, L. Adam, and J. Peccoud. Modeling Structure-function Relationships in Synthetic DNA Sequences Using Attribute Grammars. *PLoS Computational Biology*, 5(10):e1000529, 2009. doi:10.1371/journal.pcbi.1000529.

[9] L. Calzone, F. Fages, and S. Soliman. BIOCHAM: An Environment for Modeling Biological Systems and Formalizing Experimental Knowledge. *Bioinformatics*, 22(14):1805–1807, July 2006. doi:10.1093/bioinformatics/btl172.

[10] S. Cerrito and M. C. Mayer. A tableaux based decision procedure for a broad class of hybrid formulae with binders. In *Proceedings of the 20th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX'11, pages 104–118. Springer-Verlag, 2011. doi:10.1007/978-3-642-22119-4_10.

[11] F. Ciocchetta and J. Hillston. Bio-PEPA: A Framework for the Modelling and Analysis of Biological Systems. *Theoretical Computer Science*, 410(33-34):3065–3084, August 2009. doi:10.1016/j.tcs.2009.02.037.

[12] K. Clancy and C. A Voigt. Programming Cells: Towards an Automated Genetic Compiler. *Current Opinion in Biotechnology*, 21(4):572–581, August 2010. doi:10.1016/j.copbio.2010.07.005.

[13] C. A. Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems*, 1(3):269–308, 1999. doi:10.1007/BF03325101.

[14] M. J Czar, Y. Cai, and J. Peccoud. Writing DNA with GenoCAD. *Nucleic Acids Research*, 37(Web Server issue):W40–W47, July 2009. doi:10.1093/nar/gkp361.

[15] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of Tableau Methods.* Springer Netherlands, Dordrecht, 1999. doi:10.1007/978-94-017-1754-0.

[16] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *Proceedings of CONCUR'07*, pages 17–41, 2007. doi:10.1007/978-3-540-74407-8_3.

[17] F. Delaplace, H. Klaudel, and A. Cartier-Michaud. Discrete Causal ModelView of Biological Networks. In *Proceedings of the 8th International Conference on Computational Methods in Systems Biology - CMSB '10*, pages 4–13, New York, New York, USA, September 2010. ACM Press. doi:10.1145/1839764.1839767.

[18] F. Fages. Associative-commutative unification. *J. Symb. Comput.*, 3(3):257–275, June 1987. doi:10.1016/S0747-7171(87)80004-4.

[19] P. Francois and V. Hakim. Design of genetic networks with specified functions by evolution in silico. *PNAS*, 101(2):580–585, 2004. doi:10.1073/pnas.0304532101.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[21] A. Ghosh and S. Dehuri. Evolutionary algorithms for multi-criterion optimization: A survey. *International Journal of Computing & Information Sciences*, 2(1):38–57, 2004.

[22] J.L. Giavitto, O. Michel, J. Cohen, and A. Spicher. Computations in space and space in computations. In *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 137–152. Springer Berlin / Heidelberg, 2005. doi:10.1007/11527800_11.

[23] D.G. Gibson, J.I. Glass, C. Lartigue, V.N. Noskov, R.Y. Chuang, M.A. Algire, G.A. Benders, M.G. Montague, L. Ma, M.M. Moodie, and al. Creation of a Bacterial Cell Controlled by a Chemically Synthesized Genome. *Science*, 329(5987):52–56, May 2010. doi:10.1126/science.1190719.

[24] D. Hume. *A Treatise of Human Nature, Being an Attempt to Introduce the Experimental Method of Reasoning into Moral Subjects.* Clarendon Press, 1739. doi:10.1037/12868-000.

[25] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In *8th International Conference On Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495, 1986. doi:10.1007/3-540-16780-3_113.

[26] D. Lewis. Causation as Influence. *The Journal of Philosophy*, 97(4):182–197, 2000. doi:10.2307/2678389.

[27] T. K Lu, A. S Khalil, and J. J Collins. Next-generation Synthetic Gene Networks. *Nature Biotechnology*, 27(12):1139–1150, 2009. doi:10.1038/nbt.1591.

[28] M. P. Pedersen. Towards Programming Languages for Genetic Engineering of Living Cells. *Journal of the Royal Society, Interface*, 6 Suppl 4:S437–S450, 2009. doi:10.1098/rsif.2008.0516.focus.

[29] C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a Stochastic Name-passing Calculus to Representation and Simulation of Molecular Processes. *Information Processing Letters*, 80(1):25–31, October 2001. doi:10.1016/S0020-0190(01)00214-9.

[30] P. E M Purnick and R. Weiss. The Second Wave of Synthetic Biology: From Modules to Systems. *Nature Reviews. Molecular Cell Biology*, 10(6):410–422, 2009. doi:10.1038/nrm2698.

[31] S. Regot, J. Macia, N. Conde, K. Furukawa, J. Kjellén, T. Peeters, S. Hohmann, E. de Nadal, F. Posas, and R. Solé. Distributed Biological Computation with Multicellular Engineered Networks. *Nature*, 469(7329):207–211, 2011. doi:10.1038/nature09679.

[32] G. Rodrigo, J. Carrera, T. E. Landrain, and A. Jaramillo. Perspectives on the automatic design of regulatory systems for synthetic biology. *FEBS letters*, 586(15):2037–2042, July 2012. doi:10.1016/j.febslet.2012.02.031.

[33] G. Rodrigo and A. Jaramillo. AutoBioCAD: full biodesign automation of genetic circuits. *ACS synthetic biology*, 2(5):230–236, May 2013. doi:10.1021/sb300084h.

[34] H. M. Salis, E. Mirsky, and C. Voigt. Automated design of synthetic ribosome binding sites to control protein expression. *Nature biotechnology*, 27(10):946–950, 2009. doi:10.1038/nbt.1568.

[35] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, 1981. doi:10.1145/322261.322262.

[36] F. Stolzenburg. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning*, 22(1):45–63, 1999. doi:10.1023/A:1006019227709.

[37] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1998. doi:10.1007/978-1-4615-3992-6.

[38] P. Umesh, F. Naveen, C.U.M. Rao, and S.A. Nair. Programming languages for synthetic biology. *Systems and Synthetic Biology*, 4(4):265–269, 2010. doi:10.1007/s11693-011-9070-y.

[39] H. Ye, M. Daoud-El Baba, R-W. Peng, and M. Fussenegger. A Synthetic Optogenetic Transcription Device Enhances Blood-glucose Homeostasis in Mice. *Science*, 332(6037):1565–1568, 2011. doi:10.1126/science.1203535.

[40] A. Zhou, B.Y. Qu, H. Li, S.Z. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49, 2011. doi:10.1016/j.swevo.2011.03.001.