# Constraint-Based Genetic Compilation

Christophe Ladroue and Sara Kalvala

Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Sara.Kalvala@warwick.ac.uk

**Abstract.** Synthetic biology aims at facilitating the design of new organisms via the standardization of biological parts and following engineering principles. We present ATGC (Assistant To Genetic Compilation), a software tool that automatically builds a functional sequence of DNA from a minimal set of requirements. Through a simple language, the user provides in-house knowledge about their construct (*e.g.* relative placement of parts, number of restriction enzymes). ATGC combines information from established biology, user knowledge and bioinformatics databases, and maps the problem to a constraint satisfaction setting. The solution is a functional DNA sequence ready to be assembled and transferred to a target organism.

**Keywords** synthetic biology, biocompilation, constraint satisfaction

## 1 Introduction

Synthetic biology [9] aims at reaching a level of control in biology that is traditionally found in mechanical or electrical engineering [5]. Its approach consists in considering biological *parts* and combining them to achieve a pre-defined task. Parts are fragments of DNA fulfilling a specific function: promoters, protein coding sequences (CDS), ribosome binding sites (RBS), terminators and so on. A *device* is an ordered collection of parts, that has a specific effect (*e.g.* producing a certain protein when sensing a signal). The resulting string of DNA can be thought of as a blueprint for a program to be run by the cell. The potential benefit for science and society is immense, from a better understanding of cellular processes[8], to drug manufacturing [16] to energy production [18].

Following this computer analogy, we present our approach to *biocompiling*[6]: deriving, from a high-level, human-readable encoding of desired behavior, the corresponding set of instructions at a machine (or a cell) level, in this case in terms of nucleotides (A, C, T and G) instead of zeroes and ones. In this paper we consider only simple forms of control, where the production of proteins is controlled by the choice of promoter that precedes the corresponding gene in the DNA sequence. Nature has evolved many more complex forms of control (such as the very powerful method known as RNA Interference [11]) but these are still beyond the scope of most of the projects developing practical design tools for synthetic biology.

From a conceptual point of view biocompilation *should* be a simple process: one can look up parts from the many comprehensive databases of biological parts available to find ones that provide the required functionality, and string the corresponding nucleotide sequences together. However, in practice there are many obstacles to obtaining functional sequences: the quantitative nature of genetic translation means that there may be many initial designs but they provide unacceptable reaction rates. Furthermore, there are local interactions that can occur between specific nucleotide sequences and the underlying cellular mechanisms which need to be considered. And there are usually some pragmatic constraints from the specific lab or experimental setting in which the actual biological experiments are to be performed.

Biocompilation has been addressed by a number of projects in recent years. GENOCAD [2] uses a context-free grammar for constraining sequence building and also verifying existing sequences. PROTO BIOCOMPILER [1] is based on the spatial language PROTO. An abstract regulatory network is derived from the behavioral specifications and then simplified and instantiated to a smaller genetic circuit. EUGENE [4] starts from a collection of parts and devices, and creates all possible combinations. It then prunes unwanted arrangements following simple rules. However, these rules are rarely under the control of the user, who is often unable to control the decisions made by the tools. Therefore, realistic automation of the biocompilation process has so far been quite elusive.

Our approach, described here and exemplified by our tool ATGC, also uses built-in rules informed from biology to construct a functional device from a collection of parts, but it also allows users to add simple, specific *directives* to control the decisions made. Potential inconsistencies are managed by using a constraint-based methodology that balances different constraints against each other. We make extensive use of a CSP solver, the JAVA library JACOP [14]. The advantage of this is that our decision making can easily accommodate more heuristics and user design strategies, without needing to re-structure the whole process. Our tool automatically completes unfinished devices, lowering the requirement for genetic knowledge from the user while accommodate user-defined constraints.

## 2 General Workflow

In the general workflow of ATGC, a user declares a list of parts to be include in the design, using parts of different types (promoters, CDS, terminators, etc.), and they can also add extra constraints on the construction of the final sequence, such as the relative position of parts or the direction of the CDS. The set of constraints offered may be informed by users' daily practice. These extra constraints are encoded in terms of a cost function to be minimized. The task of ATGC consists in 1) taking stock of what parts and devices are used, 2) finding a realistic arrangement of the parts 3) instantiating their exact nucleotide sequences and 4) creating an output file that can be fed into a DNA assembly process.

In the simplest use-case, users only specify promoters and CDSs, which provide enough information to check the desired functionality via simulation. But to be biologically functional, a device requires extra parts to be integrated into the design. The biocompiler automatically adds the missing necessary parts, such as RBS and terminators, as well as, if requested by the user, a number of cloning sites. Insertion of all these different components can raise several conflicts, and ATGC solves these conflicts in many cases. There are three main steps in the biocompilation process:

1. The set of parts specified by the user is expanded with other parts needed to complete the design and the various parts are placed in a coherent sequence, taking into consideration the fact that devices can be placed in *both* directions in the complementary DNA strands.
2. Ribosomal Binding Sites (RBSs) are tags that are translated along with CDSs and determine the translation rate for the CDSs. ATGC chooses the RBS sequences that implement the translation rate desired by the user.
3. Cloning sites are often desired by users and it is often very difficult to find appropriate cloning sites because these 4- and 6-mer sequences should not appear anywhere else in the DNA. ATGC has an extensive mechanism to try to find adequate cloning sites and their corresponding restriction enzymes, by re-generating nucleotide sequences for the rest of the DNA if required.

The solution found by ATGC can be exported in SBOL format [10], which can be simulated and which is compatible with a large a growing number of synthetic biology software, such as J5 [12].

In the next sections, we explain the various elements of user input and also explain the way that ATGC processes this information to produce viable designs at the nucleotide level.

## 3 User Input Language

The input language for ATGC is similar to that adopted by many other biocompiler tools, and consists mainly in ways in which parts can be accessed from various databases, and host organisms can be identified and the environment (such as concentrations of signaling molecules, etc.) can be specified.

What is unique in ATGC is the way *directives* can be specified. These are interspersed into the input file and are all preceded by the **ATGC** keyword. User expertise is often difficult to formalize as it contains *ad-hoc* rules and rules of thumb. The various directives which have been implemented so far will be shown in the next section alongside the algorithms we developed to process them. Here we explain how parts, devices, and cells are specified by users.

Parts are declared by the user by specifying their types and sequences, as both information will be required for building the final sequence. That is, a biological part in ATGC is just an ordinary variable. Four types of parts are available: promoter, gene, RBS and terminator. Each of these types take one argument: either an explicit DNA sequence, or a pointer to a database entry.

Users need to specify the set of parts that they may want to use in their designs: we believe this is more sensible than hard-wiring *our* favorite parts, which may unnecessarily increase the search space for designs. Fig. 1 shows the various ways in which parts can be defined: directly as sequences (PromD, GeneD), or as named parts in popular databases, for example PromB from BIOFAB [3], GeneP from the Parts Registry ([13], and PromV and GeneV from the Virtual Parts Repository [7], which we use in our running example. These repositories contain promoters, genes, RBSs and terminators that have been used extensively by the synthetic biology community.

```
PromD =
 PROMOTER(sequence = "TGTGCATGACAAATCAGATTAACAC")
GeneD =
 GENE(sequence = "ATGAGTCAGTTTCGATAATC")

PromB =
 PROMOTER(URI = "ATGC://biofab/part/PLTETo1")

PromP =
 PROMOTER(URI = "http://parts.igem.org/Part:BBa_I14033")
GeneP =
 GENE(URI = "http://parts.igem.org/Part:BBa_K592009")

PromV =
 PROMOTER(URI = "http://sbol.ncl.ac.uk:8081/part/BO_2689")
GeneV =
 GENE(URI = "http://sbol.ncl.ac.uk:8081/part/BO_28536")
```

**Fig. 1.** Declaring parts that can be used in the biocompiler

Genetic designs involve specifying the assembly of parts into devices and the placement of these devices into a context, such as a host cell. These structures are specified as given in Fig. 2. *Devices* are understood in synthetic biology as composite genetic units with functional parts such as promoters and CDSs. Devices are declared analogously to functions in a typical programming language, with a signature specifying the parts used. The actual placement of the parts is the goal of biocompilation. Thus, initially, this section may be empty, but is filled in by the biocompiler. Devices are used in cells, which in turn are placed in a region. The information about the cell and the region provide important information that is used to simulate the dynamics of the cell and thus verify the results of the biocompiler.

An important aspect for ATGC is that descriptions may be (and in fact are expected to be) incomplete: the goal of biocompilation is to fill in the design to make a feasible complete specification. This is done by automatically searching through a large space of genetic parts and configurations and finding consistent

```
DEVICE myDevice = new DEVICE(parts =
        [myPromoter, aGene, anotherGene])(){
                  // body of the device    }

define myCell typeof CELL(){
  (...)   // declare parts and devices    }

define myRegion typeof REGION(){
  CELL strain2015d = new myCell()         }
```

**Fig. 2.** Declaring devices, cells and regions

and good solutions, not only in terms of syntax but also in terms of quantitative analysis and analysis of the actual nucleotide sequences.

## 4 Biocompilation Step : Arranging the Bio-parts

The biocompiler will use the parts required for building the whole construct, following rules from known biology. There can be many equivalent ways to place the parts in the final piece of DNA, and some might be preferred by the user. For example, they might want to reproduce an existing construct that has been reported to work in previous studies.

ATGC finds the position of each part by using built-in biological knowledge as directives for placement. But some hard constraints, such as the notion that a device should start with a promoter and end with terminators, or that genes must be preceded by an RBS, cannot be violated. Extra constraints provided by the user, like relative positions of parts, are captured in a cost function.

The directive **ARRANGE**, followed by a list of parts, will force the compiler to favor an arrangement that matches the relative positions of the parts involved in the directive. This directive makes it easy to guide the biocompiler to generate solutions that fit a pre-determined, in-house design. For example, the directive:

```
ATGC ARRANGE nahR, Pnah, Psal, xysl2
```

directs the biocompiler to try to place these four parts in this particular order—for example, when such a component has already been produced in-house. **ARRANGE** is a 'soft' constraint, in that it favors solutions where it is satisfied, but not if it is in direct contradiction with hard-coded rules for genetic constructs.

The **DIRECTION** directive forces the direction of a device: forward or reverse. For example, the requirement that a device is to be read in a specific direction can be specified as:

```
ATGC myDevice DIRECTION: FORWARD
```

**Mapping to a Constraint Satisfaction Problem** The corresponding constraint satisfaction problem (CSP) has 2 types of objects: parts $p \in \{P, R, G, T, C\}$ and devices $i \in i^1, i^2, \ldots$, each of which is associated with a set $\sigma$ of parts. The goal is to assign locations $L() \in \{1, \ldots, \#parts\}$ to parts and to assign a direction $D() \in \{F(=0), B(=1)\}$ to devices.

The biocompiler adds a number of constraints in order to build a functional piece of DNA, such as the following:

- No two parts can be at the same location: $\forall p, p'.L(p) \neq L(p')$
- A device in either direction should start with a promoter:
  $\forall i.D(i) = F \rightarrow \min_{L(p).p \in \sigma(i)} \rightarrow p = P$
  $\forall i.D(i) = R \rightarrow \max_{L(p).p \in \sigma(i)} \rightarrow p = P$
- Each gene must be preceded by a RBS sequence:
  $\forall i.D(i) = F \rightarrow \forall G \in \sigma(i).\ \exists R \in \sigma(i).L(G) = L(R) + 1$
  $\forall i.D(i) = R \rightarrow \forall G \in \sigma(i).\ \exists R \in \sigma(i).L(R) = L(G) + 1$
- Devices do not overlap: Given the relative order $\sigma_.$ of the devices
  $\forall i.\forall p.\ p \notin \sigma(i) \rightarrow L(p) < min_{L(p).p \in \sigma(i)} \ \vee \ L(p) > max_{L(p).p \in \sigma(i)}$
- Terminators should be the last parts of a device:
  $\forall i.D(i) = F \rightarrow \max_{L(p).p \in \sigma(i)} \rightarrow p = T$
  $\forall i.D(i) = R \rightarrow \min_{L(p).p \in \sigma(i)} \rightarrow p = T$

To these and other in-built numerical constraints, the biocompiler then adds any user-specified constraints:

- The directive **ATGC ARRANGE** $P_1, P_2, ..., P_k$ is translated into the constraints: $L(P_1) < L(P_2) < \cdots < L(P_k)$.
- The directive **ATGC** *Dev* **DIRECTION** = REVERSE is translated into the constraint: $D(Dev) = R$.

These conditions are expressed as a system of numerical constraints that can be solved by the JaCoP CSP solver. Once the program is run, a solution (if it exists) is found, with all constraints satisfied and the positions of each parts assigned a particular value. Currently the tool only produces one (the best) solution, but we plan to extend it to export an arbitrary number of solutions, which can then all be tested through parallel wet-lab experiments. This extension is not conceptually difficult, but we need to ensure we provide solutions in a style that is easily integrated into lab protocols.

## 5 Biocompilation Step: RBS Selection

An RBS sequence needs to precede each gene in order to initiate translation. It is tailored for individual contexts: it depends on the CDS, the pre-sequence, and the type of host organism. The RBS sequences are computed via a stochastic process and can be very different even given the same initial conditions. The Salis RBS calculator [17] is well established as providing acceptable RBS sequences, so ATGC simply calls this tool with the specific parameters to generate this sequence which is then inserted into the emerging design.

In the Salis RBS Calculator, translation rates are specified in an arbitrary unit, and the default rate is set to 1000. ATGC also uses this default rate, but also allows it to be overridden by the user. To specify a particular initiation translation rate, users specify the value they would like to achieve:

```
ATGC TRANSLATION RATE: 50000
```

The RBS calculated for this device will have an initiation translation rate 50 times higher than the rate for a default RBS. Note that the sequence generated in this way may result in a conflict in the next step, so the RBS Calculator may be called several times during the biocompilation process.

## 6 Biocompilation Step: Cloning Sites Selection

ATGC also allows the user to ask for a number of *cloning sites*, non-coding fragments of DNA that can be cleaved with a *restriction enzyme* specific to each nucleotide sequence. Cloning sites are useful for *in vivo* use, *e.g.* by allowing reporters to be inserted at these locations. However, it is often difficult to find restriction enzymes that can be used because the cleavage sequences can appear in other places in the DNA, and cleaving the DNA at these sites can be catastrophic.

Users can insert a request for a number of cloning sites (say 5) within the specification of devices with a simple directive:

```
ATGC CLONING SITES: 5
```

**General Strategy** ATGC attempts to find a selection of restriction enzymes that cut only at the desired location in the final sequence. Since the restriction enzymes will cut the DNA string at any occurrence of their characteristic nucleotide sequence, they have to be chosen so as not to cut the DNA sequence anywhere else. Since restriction enzymes (and therefore cloning sequences) are in limited number, it might not be possible to find enough fitting restriction enzymes given a particular sequence. If this is the case, the algorithm is allowed to change the rest of the sequence: either RBS can be recalculated or codons can be changed to suit more restriction enzymes.

This work-flow is described in Fig. 3. From the whole sequence (with all devices) and the number of required cloning sites, we first build a list of potential restriction enzymes (line 5), *i.e.* with cloning sequences that do not appear in the whole sequence. In a first pass, in case of conflict with calculated RBSs, the RBS are re-calculated to achieve the previously set translation rates but with different sequences. This is done at most twice (to avoid infinite loops). In a second pass, if there is a conflict with some coding sequences, the biocompiler proceeds to find the optimal codon change that will both: 1) free up previously conflicting restriction enzymes, and 2) minimize the disruption to the CDS (by applying a cost dependent on the codon usage). This is done through the mapping to a constraint satisfaction problem described in the next subsection.

```
1  foreach device do
2  |  get the whole sequence (including all devices)
3  |  get the number of required restriction enzymes
4  |  while RBS have been updated at most twice do
5  |  |  build a list of potential restriction enzymes
6  |  |  attempt to find enough restriction enzymes (Fig. 4)
7  |  |  if successful then
8  |  |  |  replace the place holders by the actual sequences
9  |  |  |  exit
10 |  |  else
11 |  |  |  recompute the RBS's if it might free up some restriction enzymes
12 |  |  |  try again
13 |  end
14 |  if unsuccessful then
15 |  |  attempt codon optimization via CSP (Fig. 5)
16 end
```

**Fig. 3.** General workflow for finding restriction enzymes

```
1 use the list of non-cutting REs
2 repeat
3 |  replace a placeholder by an actual RE in the list
4 |  if it cuts more than once then
5 |  |  remove from the list and the placeholder
6 until the list of non-cutting RE is exhausted or enough RE's have been found
```

**Fig. 4.** Finding fitting restriction enzymes

Given a list of non-conflicting restriction enzymes and the whole sequence, the algorithm in Fig. 4 attempts to find a suitable selection. A restriction enzyme is selected from the list one at the time, and the corresponding cloning sequence is inserted into the sequence if there is no conflict. This ensures that the added sequences are not in conflict with the newly selected sequence. The algorithm either succeeds to fit the necessary number of cloning sites, in which case the search ends, or it fails, in which case the original CDS is modified if possible.

**Mapping to a Constraint Satisfaction Problem** Fig. 5 shows the algorithm used for updating the codons in CDSs in order to fit more restriction enzymes. It starts by considering the restriction enzymes that only have a conflict with the CDS, from the list of potential enzymes built in Fig. 3 (line 5). It then makes a list of the specific codons which cause conflicts, and for each such codon, finds possible alternatives according to the genetic code (*e.g.* Alanine can have 4 forms:GCT, GCC, GCA, GCG). A reified Boolean variable $RE_i$ is created for each candidate restriction enzymes (lines 3-6) that is true if and only if the codons each take a form such that the final sequence does not conflict with the enzyme.

1 Consider REs that cut in CDS only
2 Build the list of conflicting codons and their possible alternative forms
3 Create a reified Boolean variable for each restriction enzyme:
4 **foreach** $RE_i$ **do**
5 $\quad \mathrm{RE}_i := \bigvee_{\text{All fitting combinations}} (\mathrm{codon}_1 = \mathrm{form}_2 \wedge \cdots \wedge \mathrm{codon}_n = \mathrm{form}_4)$
6 **end**
7 Assign the cost of changing a codon to a particular form
8 $\mathrm{cost}(\mathrm{codon}_j, \mathrm{form}_k) =$

$$\begin{cases} -\log f_{j,k} & \text{with } f_{j,k} \text{ :natural frequency of form}_k \text{ for codon}_j \\ 0 & \text{if codon}_j \text{ has the current form: form}_k \end{cases}$$

9 Set that there must be at least enough fitting restriction enzymes:
10 $\quad \sum \mathrm{RE}_i \geq \#$ requested RE
11 And that codons are changed with minimal cost:
12 $\quad$ Minimise $\sum_{\text{all codons}} \mathrm{codon}_j$ takes form $k$

**Fig. 5.** Fitting restriction enzymes with codon modifications

In other words, they are all acceptable alternative codes for the aminoacids coded by the initial codons, but which do not give rise to forbidden sequences.

The program does not simply search for codon alternatives to free up at least a minimal number of enzymes (lines 9-10). It also does it in such a way that the original sequence is minimally disrupted: each codon change is associated with a cost (lines 7-8). If the original form is kept, the cost is 0. Otherwise, the cost is $-\log(f_k)$, where $f_k$ is the natural codon bias for the species. Thus, not changing a codon does not cost anything and changing a codon to a less common form is more expensive. The codon usage frequencies are found from the Codon Usage Database (`http://www.kazusa.or.jp/codon/` [15].

The overall goal of the algorithm is thus to change codons in order to free up a minimal number of restriction enzymes (lines 9-10) while minimizing the total cost of the changes (lines 11-12).

## 7 Example

In this section, we build a biological AND operator with two devices, following the approach taken in [19]. The aim is to make a cell produce a fluorescent protein (GFP) when two molecules IPTG and aTc are present. Fig. 6 shows the regulatory network designed to achieve this goal. The first device produces the molecules lacI and tetR. The production of GFP by the second device is inhibited by lacI and tetR. IPTG inhibits laclI and aTc inhibits tetR. As a result, GFP will be produced only when aTc and IPTG are present. The inhibition for the second device is achieved with two promoters PLacI and PTeR.

In Fig. 7 we show the directives coded by experimentalist colleagues in specifying some requirements. Typically, they wished to specify only the promoters and genes and leave other decisions to the tools. For experimental reasons, they

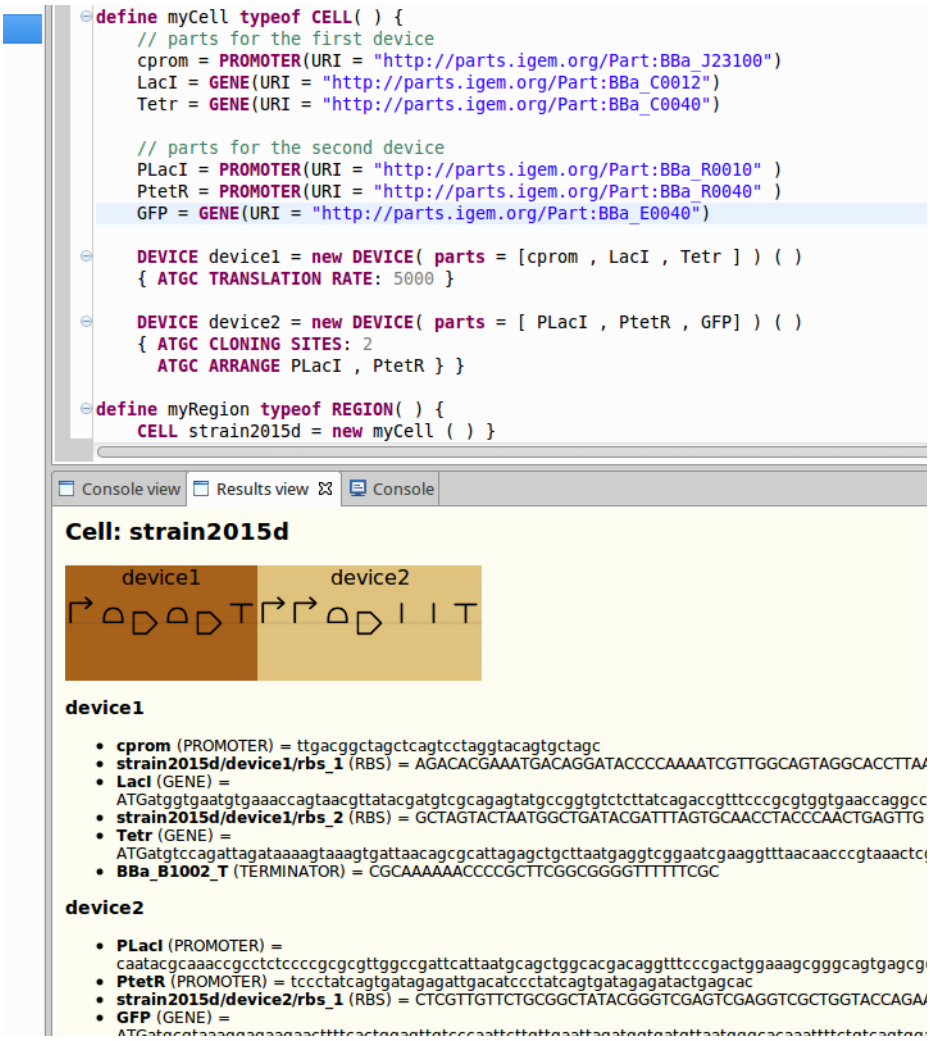


**Fig. 6.** Regulatory network and construct resulting from the biocompilation.

wished to enforce the order of the two promoters. They also wanted to add a couple of cloning sites to the second device for testing purposes.

The biocompiler automatically completed the devices with RBS and terminators and found a functional arrangement for the parts. The sequences for the parts were obtained in the Biobricks online database. The compiler also selected 2 non-cutting restriction enzymes once the rest of the sequence had been decided. In this case, the compiler found the following two restriction enzymes: BstI (GGATCC) and Bst6I (CTCTTC).

```
define myCell typeof CELL() {
        // parts for the first device
  cprom =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_J23100")
  LacI =
    GENE(URI = "http://parts.igem.org/Part:BBa_C0012")
  Tetr =
    GENE(URI = "http://parts.igem.org/Part:BBa_C0040")
        // parts for the second device
  PLacI =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_R0010")
  PtetR =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_R0040")
  GFP =
    GENE(URI = "http://parts.igem.org/Part:BBa_E0040")

  DEVICE device1 =
    new DEVICE(parts = [cprom, LacI, Tetr])()
   { ATGC TRANSLATION RATE: 5000 }

  DEVICE device2 =
    new DEVICE(parts = [PLacI, PtetR, GFP])()
   { ATGC CLONING SITES: 2
     ATGC ARRANGE PLacI, PtetR    }        }

define myRegion typeof REGION(){
  CELL strain2015d = new myCell()          }
```

**Fig. 7.** User directives for example device

## 8 Interface

ATGC's user interface is shown in Fig. 8. The central panel contains the editor. The left panel contains a project manager, where multiple files and folders can be created. The right hand-side panel shows the current understanding of the

biocompiler for the current model. The user updates its content by clicking on Refresh model. The panel then shows an overview of the construct and enables a second button labeled Compile. Pushing this button starts the biocompiler. Every step, as well as warnings and errors, are shown in the console (bottom panel). If the compilation is successful, the final construct, with the parts arranged and assigned a nucleotide sequence, is shown in the results panel. The corresponding SBOL file can be found the folder src-gen in the project manager.

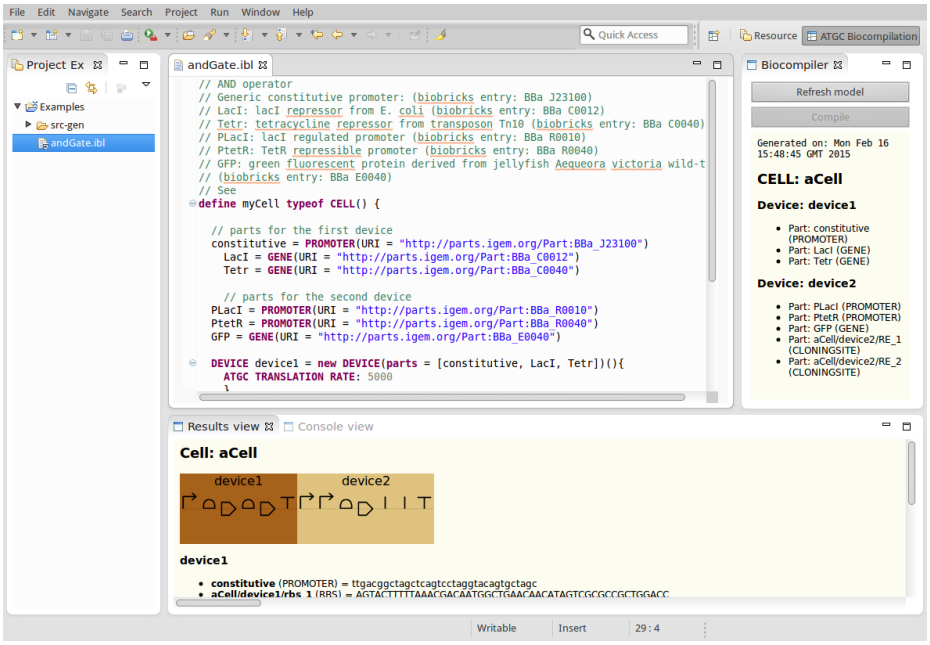


**Fig. 8.** User interface for ATGC with a fully-featured editor and project manager.

ATGC's interface is built on the Eclipse platform, and is compatible with Windows, Mac OS, and Linux. The user interface language is a simple domain-specific language developed with XTEXT (http://www.eclipse.org/Xtext), which provides a text editor with keyword completion and syntax coloring.

## 9 Conclusion

ATGC is a biocompiler that facilitates the building of a functional string of DNA from an initial specification of functionality, an initial collection of parts, and heuristics expressed through constraints. The compilation is done in three stages: initial parts placement, RBS optimisation, and insertion of cloning sites. It uses constraint solving for dealing with conflicts and choices, through the powerful, stable and well-established CSP solver JACOP.

The approach taken here is one that favors readability of the code (by using a domain-specific language), design automation (to facilitate the access to synthetic biology for non-specialists) and extensibility. By mapping the searches and the user-specified requirements to a constraint satisfaction program, ATGC leverages the tools and techniques developed by the CSP community; adding new types of constraint or new assumptions will not require the development of new *ad-hoc* algorithms but simply the addition of extra rules to the CSP basis.

In our experience and through discussion with biologists, there is a reluctance to use specialized software tools to complete genetic designs, as users often feel hindered by the lack of control in guiding the decision making. By using a constraint-based approach where user directives are injected directly into the decision process, we leave users in the driving seat. On the other hand, ATGC is very helpful in how it handles the low-level book-keeping issues, such as finding restriction enzymes that are compatible with the rest of the design.

ATGC is also part of an upcoming platform for synthetic biology, which will integrate the modeling, verification and biocompilation into a unified language and system. All three aspects will be seamlessly intertwined to produce a one-stop shop for designing new organisms *in silico*.

**Acknowledgments** This research was supported by EPSRC through grant EP/I03157X/1, *Towards Programmable Defensive Bacterial Coatings and Skins*. We are grateful for the collaboration within the Roadblock consortia.

## References

1. Beal, J., Lu, T., Weiss, R.: Automatic Compilation from High-Level Biologically-Oriented Programming Language to Genetic Regulatory Networks. PLoS ONE 6(8), e22490+ (Aug 2011), http://dx.doi.org/10.1371/journal.pone.0022490
2. Bilitchenko, L., et al.: Eugene, A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. PLoS ONE 6(4) (Apr 2011), http://dx.doi.org/10.1371/journal.pone.0018882
3. Biofab: Data Access Web Service. http://biofab.synberc.org/data (2015)
4. Cai, Y., Hartnett, B., Gustafsson, C., Peccoud, J.: A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. Bioinformatics 23(20), 2760–2767 (Oct 2007), http://dx.doi.org/10.1093/bioinformatics/btm446
5. Church, G.M., Elowitz, M.B., Smolke, C.D., Voigt, C.A., Weiss, R.: Realizing the potential of synthetic biology. Nature reviews. Molecular cell biology 15(4) (Apr 2014), http://dx.doi.org/10.1038/nrm3767
6. Clancy, K., Voigt, C.A.: Programming cells: towards an automated 'Genetic Compiler'. Current Opinion in Biotechnology 21(4), 572–581 (Aug 2010), http://dx.doi.org/10.1016/j.copbio.2010.07.005
7. Cooling, M.T., et al.: Standard virtual biological parts: a repository of modular modeling components for synthetic biology. Bioinformatics 26(7), 925–931 (2010), http://bioinformatics.oxfordjournals.org/content/26/7/925.abstract
8. Elowitz, M., Lim, W.A.: Build life to understand it. Nature 468(7326), 889–890 (Dec 2010), http://dx.doi.org/10.1038/468889a

9. Freemont, P.S., Kitney, R.I., Baldwin, G., Bayer, T., Dickinson, R., Ellis, T., Polizzi, K., Stan, G.B., Kitney, R.I.: Synthetic Biology - A Primer. World Scientific Publishing, 1 edn. (Jul 2012), http://www.worldcat.org/isbn/1848168632
10. Galdzicki, M., et al.: The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology. Nature Biotechnology 32(6), 545–550 (Jun 2014), http://dx.doi.org/10.1038/nbt.2891
11. Hannon, G.: RNA interference. Nature 418(6894) (2002)
12. Hillson, N.J., Rosengarten, R.D., Keasling, J.D.: j5 DNA Assembly Design Automation Software. ACS Synth. Biol. 1(1), 14–21 (Dec 2011), http://dx.doi.org/10.1021/sb2000116
13. iGem: Parts Registry. http://partsregistry.org/ (2015), http://partsregistry.org/
14. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Trans. Des. Autom. Electron. Syst. 8(3), 355–383 (Jul 2003), http://dx.doi.org/10.1145/785411.785416
15. Nakamura, Y., et al.: Codon usage tabulated from the international DNA sequence databases. Nucleic acids research 24(1) (Jan 1996), http://www.ncbi.nlm.nih.gov/pmc/articles/PMC145571/
16. Paddon, C.J., et al.: High-level semi-synthetic production of the potent antimalarial artemisinin. Nature advance online publication (Apr 2013), http://dx.doi.org/10.1038/nature12051
17. Salis, H.M.: The Ribosome Binding Site Calculator, vol. 498, chap. 2, pp. 19–42. Elsevier (2011), http://dx.doi.org/10.1016/b978-0-12-385120-8.00002-4
18. Schirmer, A., Rude, M.A., Li, X., Popova, E., del Cardayre, S.B.: Microbial Biosynthesis of Alkanes. Science 329(5991), 559–562 (Jul 2010), http://dx.doi.org/10.1126/science.1187936
19. Tamsir, A., Tabor, J.J., Voigt, C.A.: Robust multicellular computing using genetically encoded NOR gates and chemical /'wires/'. Nature 469(7329), 212–215 (Jan 2011), http://dx.doi.org/10.1038/nature09565