View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E IMPLEMENTAZIONE DI UN DOMAIN SPECIFIC LANGUAGE PER LA COSTRUZIONE DI RETI GENICHE

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Presentata da
LUCA GIULIANI

Correlatori
Ing. DANILO PIANINI
Ing. MARILISA CORTESI

Seconda Sessione di Laurea
Anno Accademico 2018-2019

Let's think the unthinkable, let's do the undoable.
Let us prepare to grapple with the ineffable itself,
and see if we may not eff it after all.
- **Douglas Adams**

a tutti coloro che
mi hanno reso
ciò che sono

Sommario

Il lavoro esposto in questa trattazione si colloca all'interno dell'ambito della biologia sintetica, settore interdisciplinare fondato sulla biologia e sull'ingegneria, i cui obiettivi consistono nella progettazione e costruzione di sistemi biologici spesso finalizzati alla sintesi di particolari elementi chimici difficilmente reperibili in natura.

Preso atto della complessità di realizzazione di tali sistemi, quindi, si è deciso di sfruttare le potenzialità dei nuovi linguaggi di programmazione, sempre più indirizzati verso un connubio fra il paradigma ad oggetti e quello funzionale, per sviluppare un tool adatto alla loro descrizione e, successivamente, alla simulazione dei loro comportamenti.

In questo contesto, pertanto, si presenta *Another Genetic Circuit Transcriber*, un linguaggio dominio-specifico per la costruzione di reti geniche, nato con l'intento di essere quanto più possibile affine al linguaggio naturale e dotato di una serie – espandibile – di generatori in grado di esportare il contenuto della rete verso software esterni.

Indice

Sommario **iii**

Introduzione **1**

1 Background **5**

1.1 Reti Geniche 5

1.1.1 Componenti delle Reti Geniche 6

1.1.2 Applicazioni delle Reti Geniche Sintetiche 8

1.1.3 Rappresentazioni e Formalismi 11

1.1.4 Linguaggi e Compilatori Esistenti 13

1.1.5 Esempi di Circuiti Genici Noti 17

1.2 Domain-Specific Languages 20

1.2.1 Linguaggi Adatti alla Creazione di DSL 21

1.2.2 Kotlin 22

2 Another Genetic Circuit Transcriber **27**

2.1 Analisi 28

2.1.1 Analisi del Modello del Dominio 28

2.1.2 Analisi dei Requisiti 35

2.2 Design del Modello 38

2.2.1 Design delle Variabili 38

2.2.2 Design delle Entità 39

2.2.3 Design delle Reazioni 42

2.2.4 Design della Rete Genica 46

2.3 Design del Linguaggio 47
2.3.1 Scelta della Sintassi 47
2.3.2 Analisi Lessicale della Sintassi 51
2.3.3 Mapping degli Elementi di Modello 52
2.3.4 Livelli e Wrapper 54
2.4 Organizzazione del Progetto Software 59
2.4.1 Divisione in Packages 59
2.4.2 Testing Automatizzato 60
2.4.3 Tool di Sviluppo 61

3 Utilizzo di AGCT **63**
3.1 Guida alla Sintassi 63
3.1.1 Descrizione delle Entità 64
3.1.2 Assegnamento dei Parametri 67
3.1.3 Parametri di Default 71
3.1.4 Reazioni Personalizzate 72
3.1.5 Export dei Dati 73
3.2 Creazione di un Generatore 74
3.2.1 Esempi di Generatori Implementati 75
3.3 Esempi e Risultati 82
3.3.1 Repressilator 82
3.3.2 Brusselator 86
3.3.3 Genetic Toggle Switch 88

4 Conclusioni **95**
4.1 Sviluppi Futuri 95

Bibliografia **97**

Ringraziamenti **101**

Introduzione

A meno di due secoli di distanza dalla formulazione della *Legge di Dominanza*, di fatto considerata il punto d'avvio della genetica, tale disciplina vanta ad oggi un campo di ricerca denso di scoperte e innovazioni che, più di una volta, è riuscito a trovare ampio respiro anche nel dibattito pubblico, accendendo discussioni complesse e perfino creando, in numerose occasioni, una serie di controversie esterne alla scienza in sé, ma piuttosto relative alle implicazioni etiche e sociali di questa. Ciò nondimeno, le attività di analisi e sperimentazione proseguono giornalmente nei laboratori di tutto il mondo – seppur con differenze tanto ovvie quanto sostanziali rispetto al periodo in cui del DNA non si presagiva neanche l'esistenza – grazie alle innovazioni tecnologiche caratteristiche di questa terza rivoluzione industriale, iniziata nel secondo dopoguerra e ancora attiva ai nostri giorni. È dal 1953 infatti, anno in cui Watson e Crick – basandosi sui lavori svolti negli anni precedenti da altri team di biologi come Avery[1] e Hershey[2] e avvalendosi della cristallografia a raggi X di Rosalind Franklin – riescono a immortalare la doppia elica del DNA[3], che l'intera ricerca genetica vira dal *fenotipo* al *genotipo*[1], andando ad analizzare gli attori interni alle cellule che, attraverso meccanismi perlopiù ancora ignoti, determinano il nostro funzionamento interno e anche le nostre caratteristiche esteriori.

Se, tuttavia, nella metà dell'Ottocento a Mendel spettava l'onere di attendere un intero anno la fioritura delle piante di pisello odoroso prima di poter ottenere i risultati dei suoi esperimenti[4], questo non solo risulta inconcepibile al giorno d'oggi, ma fortunatamente non è in alcun modo necessario grazie, appunto, alle novità che la tecnologia ha introdotto

[1] Si intende con *fenotipo* l'insieme di caratteristiche macroscopiche di un organismo vivente, come possono essere il colore del pelo di un cane o il gruppo sanguigno di un essere umano, mentre con *genotipo* si intende più strettamente il corredo genetico, ovvero ciò che contribuisce a determinare i fattori fenotipici.

negli ultimi decenni. In misura sempre maggiore, infatti, la genetica – e, più in generale, l'intera biologia – si avvale dell'informatica come strumento per catalogare, analizzare e simulare sistemi di varia complessità al fine di ottimizzare i tempi e le risorse senza perdere di accuratezza. Non a caso, la *Bioinformatica* si è ormai guadagnata una salda posizione fra i trend più in crescita negli ultimi anni, con corsi di laurea interamente dedicati e risultati scientifici di enorme scala quali, ad esempio, il *Progetto Genoma Umano*, conclusosi con esito positivo nel 2003 – due anni in anticipo rispetto a quanto previsto – e con un investimento di denaro inferiore alle aspettative[5].

Quello dell'*Ingegneria delle Reti Geniche*, infine, è uno dei vari settori del più vasto ambito della *Biologia Sintetica*, e si fonda sopra una serie di concetti matematici come le *equazioni differenziali ordinarie* (*ODE*), le *equazioni differenziali stocastiche* (*EDS*) e le *reti booleane*. Proprio sulle *ODE* si basano i lavori di molti ingegneri genetici che studiano nuovi metodi per una veloce individuazione di parametri e valori che, a causa di limiti fisici, sono raramente quantificabili in maniera diretta durante gli esperimenti effettuati nei laboratori[2], rendendo di fatto inconcludenti gli sforzi effettuati[6]; tuttavia, anche la matematica analitica è soggetta a un ampio numero di limitazioni, e risulta quindi, per questo e per altri motivi, più semplice ed efficace affidare l'esecuzione di tali esperimenti – almeno in una prima fase di testing – a dei simulatori biologici salvo poi ripeterli in vitro in modo da verificarne la correttezza attraverso delle concrete osservazioni fisiche.

In direzione di tale obiettivo, pertanto, è scaturito il punto di partenza di questo elaborato di tesi, finalizzato alla progettazione e alla realizzazione di un *Domain Specific Language* semplice e coerente per la descrizione di reti geniche, con l'aspirazione di poter, in seguito a futuri sviluppi, esportare i dati della rete in ogni forma necessaria così da riuscire a simulare anche in ambienti diversi il comportamento della rete stessa, basandosi sulle reazioni che la caratterizzano. Le prospettive sono numerose; una fra tutte, quella che ha dato il via al progetto – pur rimanendone distaccata proprio al fine di rendere il linguaggio generico e adatto a qualsiasi tipo di situazione e simulatore –, è la possibilità di individuare i parametri ignoti di una certa rete attraverso l'uso del simulatore stocastico *Alchemist*[7].

[2] Negli ultimi anni si sono visti alcuni miglioramenti riguardo la fattibilità di tali misure, le quali tuttavia dipendono ancora strettamente dal parametro preso in considerazione.

Per la creazione del *DSL* si è deciso di usare il linguaggio di programmazione *Kotlin*, particolarmente adatto allo scopo per via della sua flessibilità ed estensibilità. Di questo tema, come di quello riguardante le reti geniche, verrà fornita una breve panoramica nel primo capitolo, mentre nei successivi verranno rispettivamente descritte la struttura e il percorso compiuto per la realizzazione di *AGCT* – a partire dall'analisi del modello fino alla descrizione della fase di testing, passando inevitabilmente attraverso una spiegazione delle principali scelte di design effettuate – e, infine, mostrati alcuni esempi di sue applicazioni pratiche, complete anche dei risultati conseguibili.

Capitolo 1

Background

Al fine di garantire una corretta, seppur minimale, padronanza del contesto, si presenterà ora una panoramica generale all'ingegneria genetica e ai linguaggi dominio-specifici, indubbiamente necessaria alla comprensione dei capitoli successivi.

In particolare, nella prima sezione verranno mostrate, contestualmente ad alcuni esempi pratici, la struttura e le applicazioni delle reti geniche, mentre nella seconda verranno elencati alcuni linguaggi di programmazione adatti allo sviluppo di DSL interni, riservando particolare attenzione a *Kotlin*, quello utilizzato per lo sviluppo del software presentato in questa tesi.

1.1 Reti Geniche

Così come un circuito elettronico è costituito da un insieme di componenti elettroniche – quali ad esempio gli induttori, i condensatori, i transistor, ecc... –, allo stesso modo una rete genica è costituita da elementi biochimici come i geni e le proteine[1]. L'analogia, inoltre, risulta essere anche più efficace se si tiene in considerazione l'idea che il design di una rete genica ha come scopo pratico proprio quello di ottenere in output un risultato; non a caso si parla infatti di *programmazione* di reti geniche, enfatizzando come il fine

[1] Il termine *Rete Genica*, o *GNR* (*Genetic Regulation Network*) è in questo caso non propriamente il più adatto; sarebbe meglio infatti parlare di *Circuito Genetico*, ovvero il singolo modulo di una rete, analoga a un singolo circuito elettronico posto all'interno di una rete più grande, come ad esempio un calcolatore. In questo testo, tuttavia, i due termini saranno usati come sinonimi.

ultimo sia quello di restituire sempre uno specifico output se posti di fronte alle stesse condizioni iniziali.

Differentemente da un calcolatore, tuttavia, una cellula risulta meno adatta – o, in ogni caso, più difficilmente adattabile – alla computazione ma, come verrà più ampiamente discusso nelle sezioni a seguire, essa ha una caratteristica che la rende atipica rispetto a qualsiasi macchina artificiale: è in grado, grazie ai meccanismi di regolazione genica, feedback positivo e catalizzazione che avvengono al suo interno, di sintetizzare elementi biochimici con moderati costi di tempo ed erogazione energetica[8].

Prima di descrivere nello specifico gli utilizzi delle reti geniche e mostrare alcuni semplici esempi di circuiti noti nel mondo della biologia sintetica, è tuttavia necessario elencarne dapprima le entità principali e le reazioni che avvengono fra di esse.

1.1.1 Componenti delle Reti Geniche

È il 1958 quando viene formulata la prima versione del *Dogma Centrale della Biologia Molecolare*[9], la legge che descrive gli stati attraverso cui passa l'informazione genetica, contenuta all'interno di ogni cellula, che andrà poi a definire le caratteristiche macroscopiche della cellula stessa.

Sebbene, per correttezza, sia necessario sottolineare come negli anni siano stati identificati diversi organismi che hanno espanso e invalidato, almeno parzialmente, l'originale definizione del dogma, in questa sede è sufficiente considerare il flusso di informazione così come inizialmente descritto da Francis Crick, ovvero **unidirezionale**. Si ammette quindi che l'informazione possa transitare dal *DNA* alle *proteine* – passando attraverso l'*RNA messaggero* – soltanto in questo verso, e che non vi sia quindi la possibilità di seguire il percorso contrario.

Il dogma può essere quindi ricondotto a queste tre reazioni principali – alle quali, per una completa descrizione delle reti geniche, va aggiunta quella della **regolazione**[2] –:

[2] La *regolazione genica* verrà ampiamente discussa nel capitolo seguente poiché, pur non essendo parte del *Dogma Centrale della Biologia Molecolare* risulta di fondamentale importanza nell'ambito dell'ingegneria genetica in quanto permette, tramite delle molecole regolatrici, un controllo esterno sulla capacità di un gene di codificare per una specifica proteina. In particolare, in caso di incremento della produzione di parla di **attivazione**, mentre in caso di decremento si parla di **inibizione**.

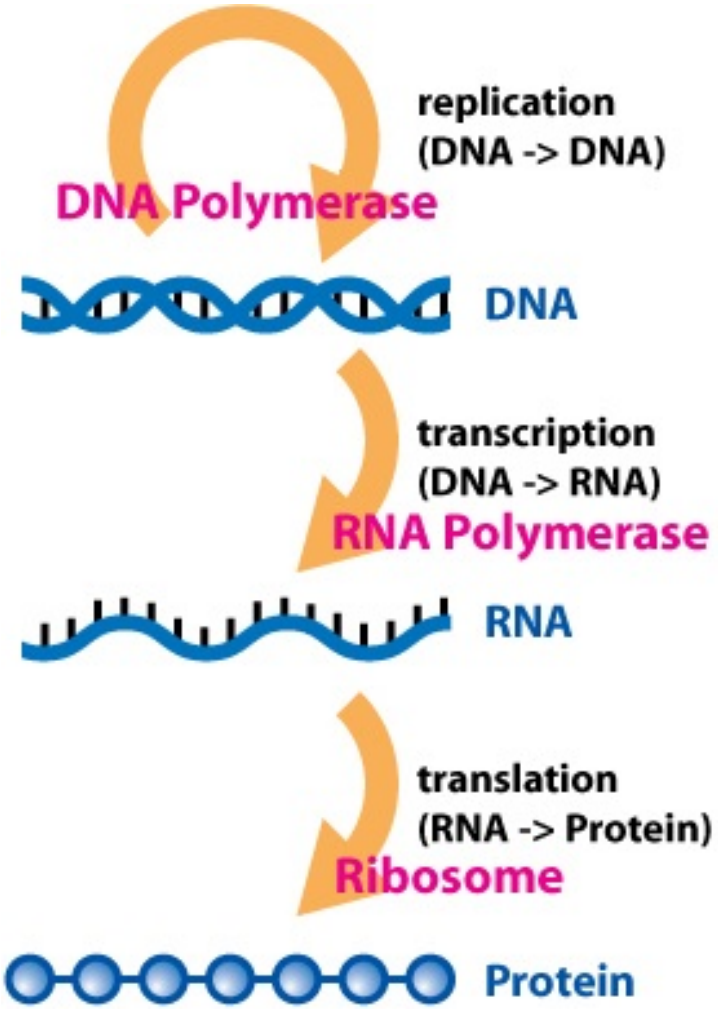


Figura 1.1: Il passaggio di informazione genetica all'interno di una cellula.
(fonte: `www.wikipedia.com`)

Duplicazione ovvero la reazione che vede l'informazione genetica, conservata nel DNA, venire *duplicata* al fine di essere propagato.

Trascrizione ovvero la reazione che vede il DNA venire *trascritto* sotto forma di RNA al fine di essere espresso nella cellula.

Traduzione ovvero la reazione che vede l'RNA codificante venire *tradotto* nelle relative proteine, le quali assumono la funzione di contenitori terminali dell'informazione genetica.

Va sottolineato che tutte queste reazioni accadono principalmente nel nucleo della cellula[3]. Infatti, sebbene una cellula sia molto complessa e presenti una serie di diverse zone e organuli, ognuno impiegato in differenti attività, è nella parte centrale che viene contenuta l'informazione genetica al suo stadio primordiale – ovvero il **DNA** –, essendo esso il punto d'avvio per la costruzione dell'intero organismo. Soltanto al termine delle operazioni di decodifica dell'informazione, infine, i prodotti – ovvero l'**RNA** e, a seguito, le **proteine** – saranno distribuiti prima nel citoplasma e poi, al termine del processo, alle rispettive zone di competenza.

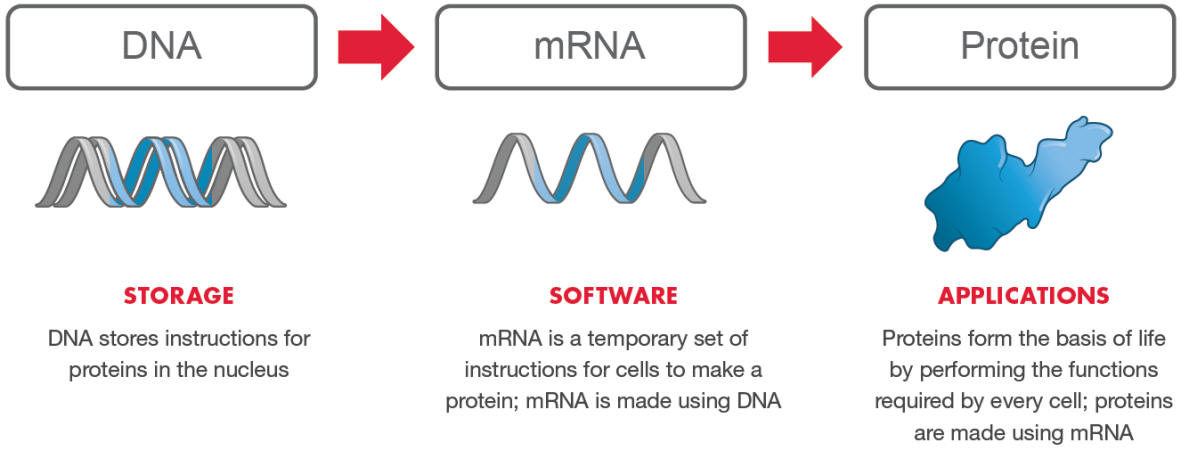


Figura 1.2: Le principali entità in gioco nella sintesi proteica.
(fonte: `www.modernatx.com`)

1.1.2 Applicazioni delle Reti Geniche Sintetiche

Lo scopo dei progettisti di reti geniche è quello di utilizzare le cellule, o parti di esse, per eseguire operazioni logiche simili a quelle che si osservano nei circuiti elettronici, così da trovare strade alternative – più semplici, più ecologiche e meno costose – alla

[3] In questa trattazione si farà riferimento unicamente alle cellule eucarioti, provviste sia di un nucleo che della maggior parte delle entità nominate ed esposte nelle sezioni e nei capitoli successivi quali cromosomi, alleli, introni ed esoni. Differentemente, negli organismi procarioti – come ad esempio i batteri – non si hanno né un nucleo né un'organizzazione strutturale complessa del genoma che, invece, è presente in copia singola e direttamente tradotto in una forma già matura di mRNA, non essendoci introni ed esoni. .

produzione di determinati elementi biochimici. Molti di questi elementi, infatti, vengono già sintetizzati quotidianamente da organismi più o meno complessi presenti in natura, ma sia per motivi di quantità – di solito un microrganismo produce dosi corrispettive alle sue dimensioni –, sia per evitare lo sfruttamento di tali sistemi, si è iniziato a ricreare in vitro le stesse condizioni in modo da gestire artificialmente la produzione del materiale.

Per fare questo, tuttavia, è necessario creare una serie di componenti a sé stanti e meccanismi di controllo – ne sono un esempio le porte logiche mostrate nella figura 1.3 – che possano essere introdotti nelle cellule così da permetterne la programmazione nello stesso modo in cui si potrebbe programmare un calcolatore, utilizzando gli organismi biologici come fucine attraverso le quali eseguire compiti e generare certi output[8]. Le applicazioni di ciò possono variare da semplici "programmi" per indurre la produzione di materiale sintetico alla creazione di nuovi sistemi che possono essere visti come vere e proprie **macchine biologiche**, ma fondamentalmente ognuna di esse comprende la sintesi di elementi che potranno poi essere utilizzati nella fabbricazione di tinture, fragranze o insetticidi, puuttosto che di medicine, alternative ai combustibili fossili o altri prodotti di reazioni metaboliche[10].

Al fine di giustificare quanto più possibile le motivazioni che hanno ispirato questo progetto di tesi, pertanto, vengono di seguito illustrati alcuni esempi di utilizzo delle reti geniche sintetiche e dei meccanismi di ingegneria genetica per la sintesi di biomolecole altrimenti estraibili per vie naturali con enormi difficoltà.

Produzione di Insulina Sintetica

Una delle prime operazioni di ingegneria genetica è stata la produzione di insulina sintetica verso la fine degli anni '80[11], ormone le cui quantità necessarie a soddisfare la domanda annua di medicinali da essa derivati venivano in precedenza tratte da più di cinquanta milioni di pancreas suini[4],

Da quel momento, si è iniziato ad applicare lo stesso meccanismo ad altri ormoni umani, quali l'ormone della crescita o gli ormoni relativi ai trattamenti per l'infertilità, per poi arrivare anche alla generazione di componenti necessarie alla produzione di vaccini

[4] `https://www.gene.com/stories/cloning-insulin`.

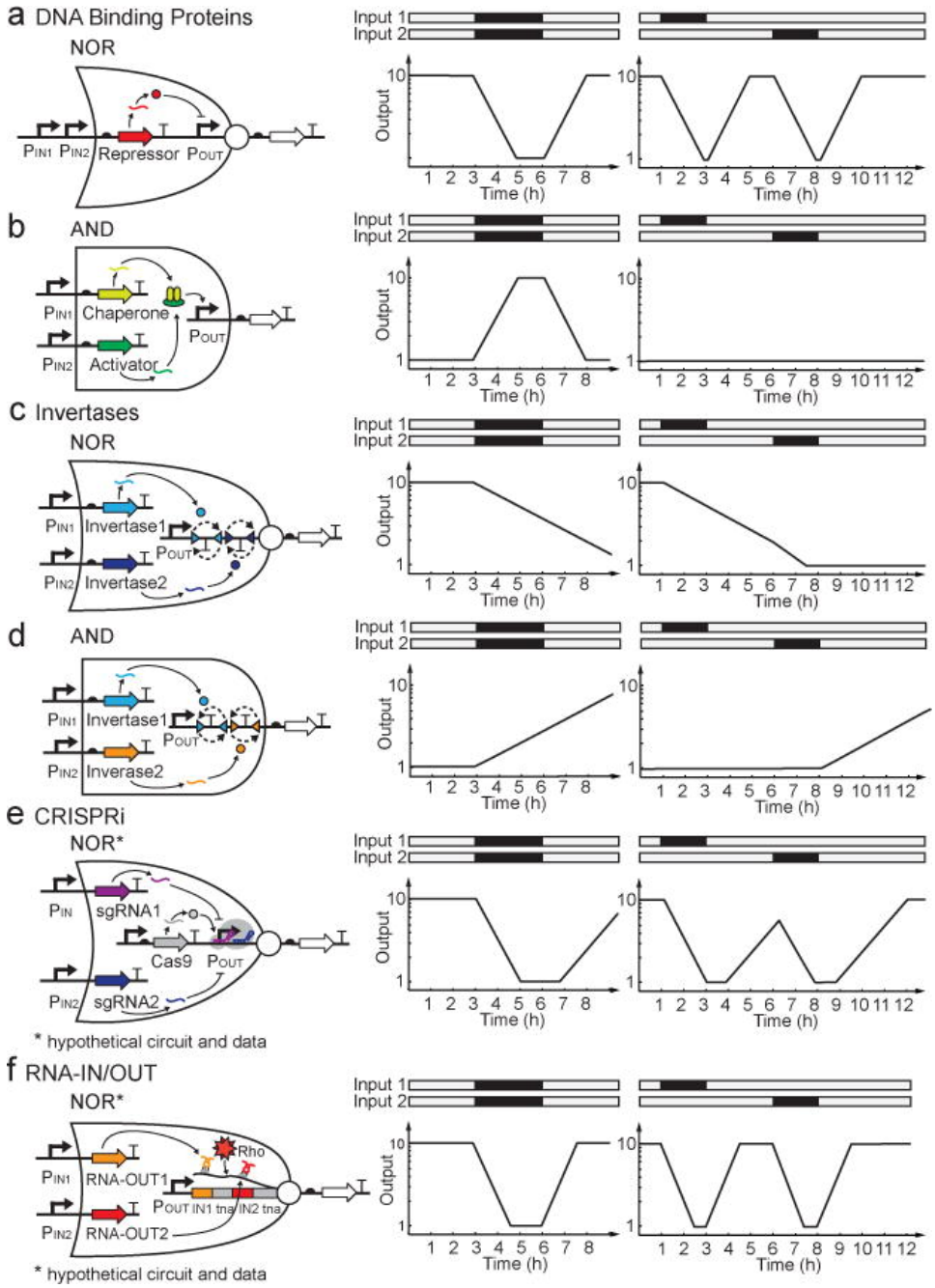


Figura 1.3: Alcune porte logiche ricostruite attraverso delle reti geniche.
(fonte: *Principles of Genetic Circuit Design* [8])

e altre medicine. Recentemente, la ricerca si sta addirittura espandendo verso la sintesi di biomolecole utili per trattamenti contro il cancro e la leucemia[12].

Il Caso dell'Artemisinina

Analogamente, le reti geniche sintetiche sono state utilizzate per la sintesi dell'artemisinina, un principio attivo scoperto nel 1972 dal Premio Nobel per la Medicina *Tu Youyou*[13]. Utilizzato come farmaco per contrastare la malaria, esso può essere ricavato dalle piante di *Artemisia Annua*, con tempi di estrazione e lavorazione di più di un anno. Nel 2004, tuttavia, un gruppo di ricercatori è riuscito, tramite un lievito geneticamente modificato, a sintetizzare un precursore dell'artemisina, minimizzando così i costi e i tempi di produzione della molecola e permettendo una maggiore diffusione del medicinale da essa derivato[14].

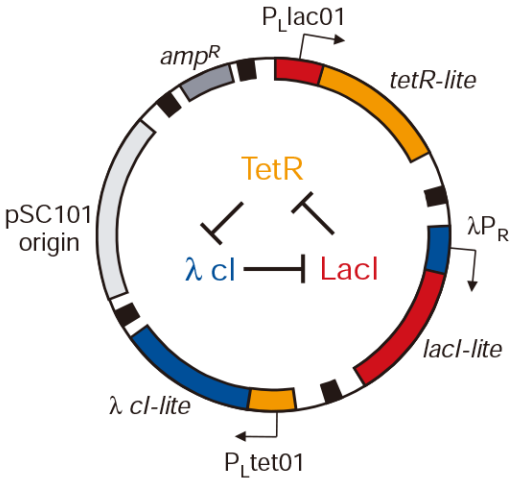
1.1.3 Rappresentazioni e Formalismi

Nonostante l'ampia crescita vista dall'intero settore dell'ingegneria genetica negli ultimi anni, non solo non esiste ancora uno standard *de iure* per la rappresentazione di reti geniche, ma neanche uno *de facto*, con l'inevitabile conseguenza di descrizioni arbitrarie, non uniformi e spesso ridondanti di informazioni. Al momento, infatti, molte delle scelte compiute nel descrivere una rete sono dettate dalla volontà del progettista di enfatizzare alcuni aspetti maggiormente rilevanti o, addirittura, nel caso di rappresentazioni grafiche, da motivazioni prettamente estetiche che, seppur ammirevoli, dovrebbero entrare in gioco soltanto in un secondo momento.

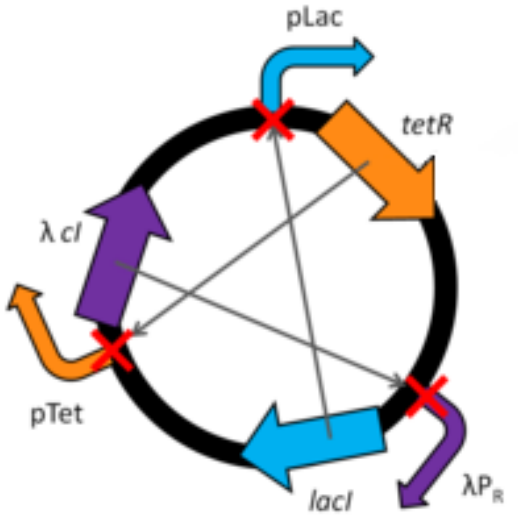
Il risultato è quindi una grande varietà di schematizzazioni che, però, esprimono la stessa identica rete genica – come mostrato nella figura 1.4 –, e dal quale è possibile estrapolare giusto alcune convenzioni grafiche, ormai divenute comuni, quali ad esempio[5]:

- un *gene* è rappresentato da un rettangolo o da una freccia di grandi dimensioni;
- un *promotore* può essere o meno esplicitamente rappresentato – nel primo caso lo si considera parte del gene –, e anche quando rappresentato lo si può indicare con un rettangolo di piccole dimensioni posto prima del gene, con una freccia sottile posta prima del gene o, in certi casi, con entrambe le notazioni contemporaneamente;

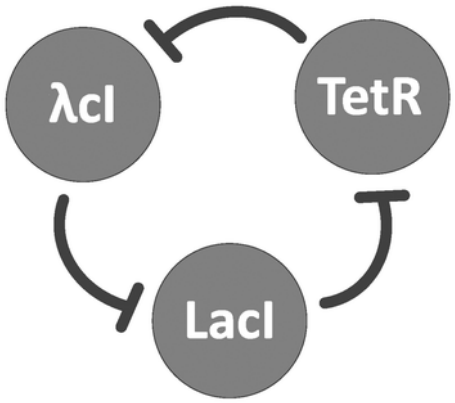
[5] Per una discussione più dettagliata dei termini utilizzati qui di seguito si rimanda alla sezione 2.1.1.



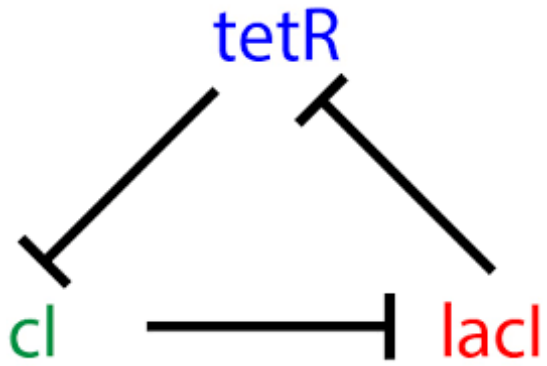
(fonte: www.igem.com)



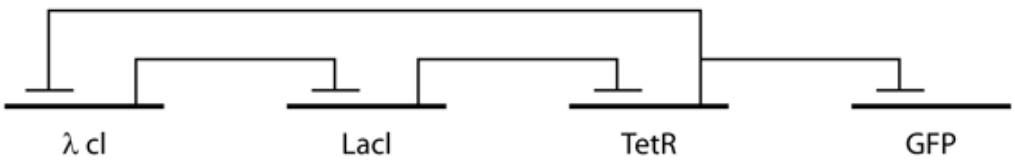
(fonte: www.nature.com)



(fonte: pubs.rsc.org)



(fonte: www.yschaerli.com)



(fonte: www.wikipedia.com)

Figura 1.4: Cinque rappresentazioni analoghe di un Repressilator.

- una *proteina* può essere o meno esplicitamente indicata e, nel caso in cui lo si faccia, non esiste una sintassi univoca per la sua rappresentazione, mentre nel caso in cui non lo si faccia il gene codificante diventa espressione della proteina stessa;
- le *frecce di regolazione* escono dalle proteine – nel caso in cui queste siano indicate – o dai geni – in caso contrario – e analogamente entrano nei promotori o nei geni; in ogni modo, la regolazione è rappresentata, in caso di attivazione, con delle frecce sottili, mentre in caso di inibizione con delle frecce barrate o con una croce davanti.

1.1.4 Linguaggi e Compilatori Esistenti

Come visto nella sezione precedente, per quanto labili e incostanti, delle convenzioni grafiche esistono e sono riconosciute dalla comunità scientifica. Ciò non accade invece per le rappresentazioni descrittive, in quanto non vi è alcuna sintassi standard che permetta di descrivere una rete genica al di fuori del linguaggio naturale – notoriamente inadatto a utilizzi formali e rigidi – o, alternativamente, l'elenco di tutte le reazioni chimiche coinvolte nella rete — che resta, tuttavia, un approccio molto *di basso livello*, e non è quindi particolarmente utile né espressivo.

In ogni caso, negli ultimi anni sono stati fatti alcuni sforzi per creare dei linguaggi standardizzati i quali potessero essere usati in primo luogo per la descrizione e, in secondo luogo, per la simulazione di reti geniche. Di seguito, quindi, ne verranno mostrati alcuni esempi.

SBOL

SBOL[6] (*Synthetic Biology Open Language*) è uno standard aperto per la rappresentazione di sistemi biologici. Esso fornisce un insieme di termini appartenenti al vocabolario proprio della genetica, detto *SBOL Data*, necessari alla costruzione di una rete genica. Inoltre, esso mette a disposizione anche un insieme di simboli – come quelli mostrati in figura 1.5 –, detto *SBOL Visual*, per la raffigurazione della suddetta rete[15].

[6] `https://sbolstandard.org`.

Sviluppato dal 2008, *SBOL* è ormai giunto ormai alla versione 2.3.0 e comprende una vasta gamma di aspetti biologici, dai classici DNA, RNA e proteine fino alla possibilità di descrivere l'interno genoma di un organismo.

Essendo uno standard aperto, inoltre, e supportando il suo Data Model in vari linguaggi di programmazione – fra cui C/C++, Python, Java e Javascript –, è possibile contribuire sia al sviluppo interno del linguaggio sia alla creazione di framework e applicazioni esterne basate su di esso, come ad esempio i vari software grafici che si appoggiano sullo *SBOL Visual*.

GCDL

GCDL (*Genetic Circuit Description Language*)[16] è un linguaggio presentato per la prima volta nel 2018 che permette la descrizione di circuiti genetici attraverso una sintassi semplice e basilare — è molto lontano dal linguaggio naturale, ma risulta in ogni modo molto chiaro ed esplicativo poiché adotta la nomenclatura standard delle entità biologiche senza tuttavia entrare nei cavilli, come invece capita a SBOL, più complesso anche per via della sua longevità.

Alla possibilità di descrizione, inoltre, è affiancata la possibilità di compilazione attraverso il compilatore **GCC** (*Genetic Circuit Compiler*)[7], che restituisce un eseguibile in grado di simulare la rete genica.

Motivazioni per la Creazione di un Nuovo Linguaggio

È evidente come l'assenza di un formalismo generi confusione, soprattutto a chi, non esperto del dominio, è impossibilitato a inferire delle regole generali da un modello che non ne prevede. La situazione, tuttavia, non è delle migliori neanche per i ricercatori del settore, in quanto, come risulta chiaro dalla figura 1.4, tutte le rappresentazioni mancano di un aspetto fondamentale: i rate delle reazioni e le concentrazioni delle molecole, che in certi casi – come si potrà notare nel Brusselator, un noto circuito genico descritto a pagina 18 – risultano fondamentali se non alla comprensione della rete, di certo alla loro esecuzione.

[7] Da non confondere con il ben più noto *GNU Compiler Collection*.

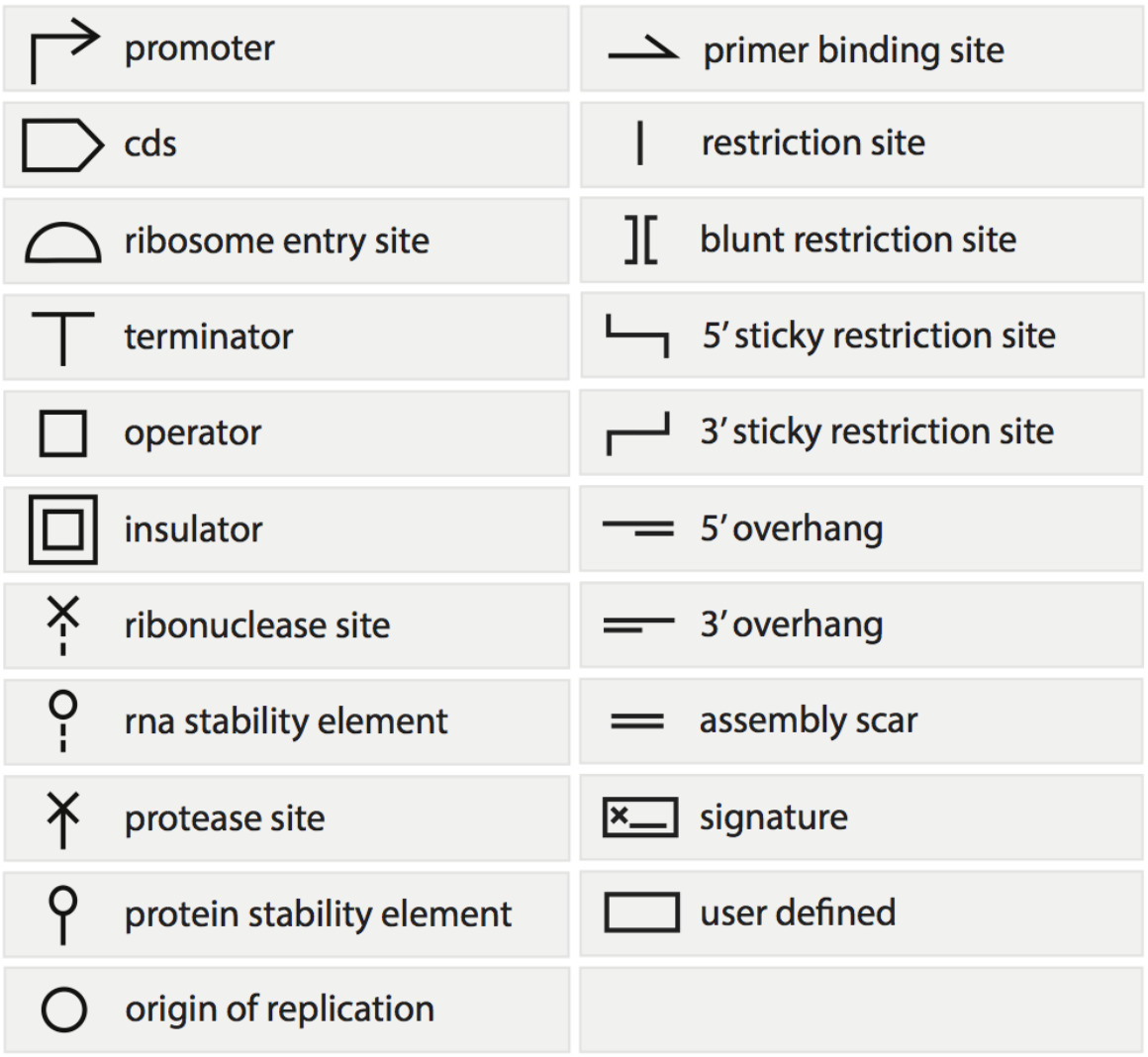


Figura 1.5: Alcuni simboli dello *SBOL Visual*.
(fonte: `www.wikipedia.org`)

L'obiettivo di questa tesi è quindi quello di fornire un tool che in un primo momento permetta di descrivere *a parole* una rete genica, attraverso una sintassi comprensibile sia dagli esperti di dominio che dai non esperti, e che in un secondo momento fornisca degli strumenti di esportazione sia verso i software squisitamente descrittivi – ovvero quei tool grafici generalmente di tipo *drag & drop*[8] – sia verso quelli il cui scopo è la simulazione

[8] Con software *drag & drop* si intende tutta quella categoria di programmi per computer in cui

classes	
gcc:Part	Generic biological part
gcc:Operator	Operator
gcc:Promoter	Promoter
gcc:RibosomeBindingSite	Ribosome Binding Site
gcc:CodingSequence	Coding Sequence
gcc:Terminator	Terminator
gcc:Token	Token or symbol in a template

(a) Classi.

predicates	
gcc:include	Include a low-level model fragment
gcc:prefix	The prefix to use for generated annotations
gcc:init	Specifies initial copy numbers
gcc:part	Links a part to its token or symbol
gcc:overlaps	Indicates that two parts overlap (symmetric)
gcc:linear	Linear circuit type
gcc:circular	Circular circuit type
gcc:transcriptionFactor	Relates an operator to its transcription factor
gcc:transcriptionFactorBindingRate	Various rates
gcc:transcriptionFactorUnbindingRate	
gcc:rnapBindingRate	
gcc:rnapUnbindingRate	

(b) Predicati.

Figura 1.6: Alcuni elementi presenti in *GCDL*
(fonte: *A Genetic Circuit Compiler* [16])

della rete – ovvero quei linguaggi di programmazione, come *GCDL*, generalmente di basso livello e con un'impronta molto chimica –, creando di fatto un ponte fra le due categorie.

1.1.5 Esempi di Circuiti Genici Noti

Come ultimo argomento di questa introduzione alle reti geniche e all'ingegneria genetica, vengono mostrati tre esempi di circuiti noti allo stato dell'arte, spiegandone brevemente le funzionalità.

Repressilator

Il *Repressilator* è un circuito genetico costituito da tre geni che si inibiscono a vicenda attraverso un meccanismo circolare in cui ogni gene esprime la proteina che blocca l'espressione della proteina codificata dal gene successivo. Macroscopicamente, questo comporta un susseguirsi di fasi analoghe di oscillazione delle concentrazioni delle tre proteine, con picchi regolari che si alternano circolarmente e un andamento simil-sinusoidale come mostrato nella figura 1.8.

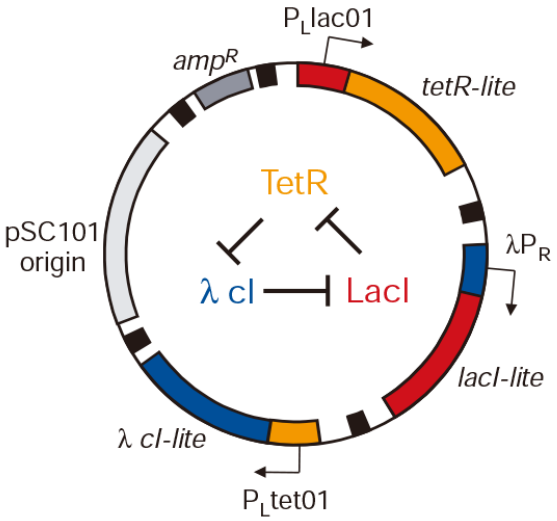


Figura 1.7: Schema di un Repressilator.
(fonte: `www.igem.org`)

Il primo repressilator artificiale è stato completato nel 2000 da Michael Elowitz e Stanislas Leibler[17]. Successivamente, sono stati trovati esempi naturali di repressilator utilizzati da organismi animali e vegetali per controllare i propri ritmi circadiani[9][18].

gli elementi grafici vengono presi e trascinati tramite il mouse per essere inseriti, spostati o rimossi dall'ambiente grafico.

[9] Il *ritmo circadiano* è un ritmo intrinseco in molti esseri viventi che, solitamente, dura circa 24 ore e

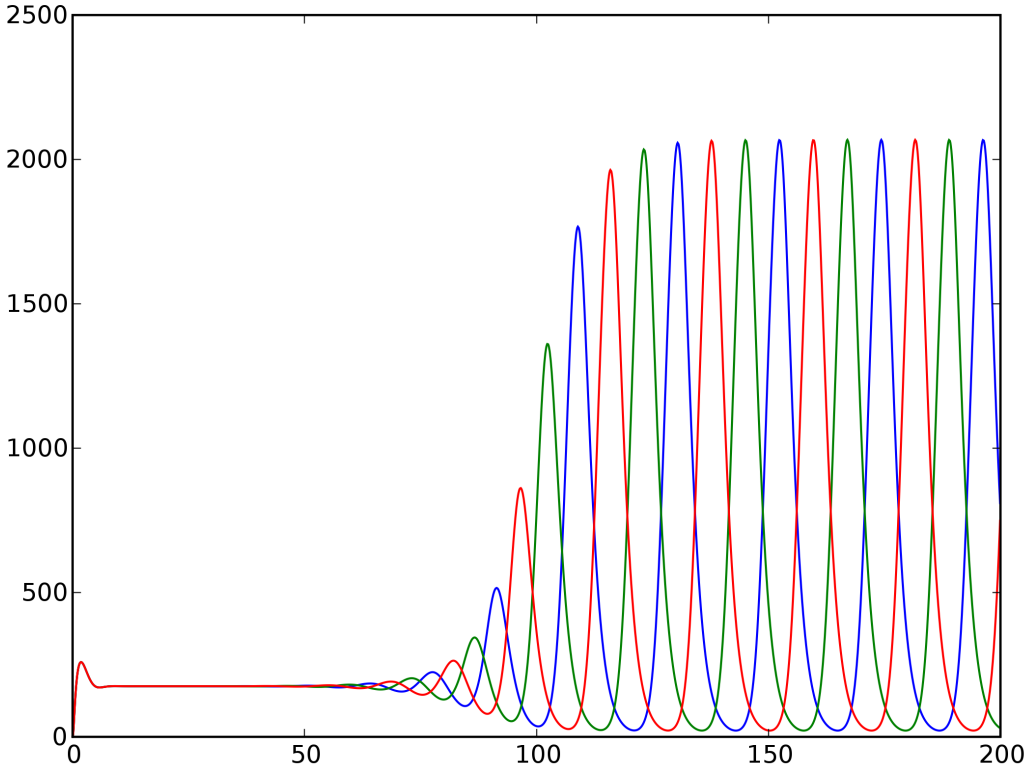


Figura 1.8: Grafico con le concentrazioni delle tre proteine del repressilator.
(fonte: `physics.cornell.edu`)

Brusselator

Il *Brusselator* è un modello teorico di reazione autocatalitica, ovvero in cui uno dei prodotti della reazione rappresenta il catalizzatore della reazione stessa, presentato per la prima volta da *Ilya Prigogine* e dai suoi collaboratori all'università di Bruxelles[19] — da qui il nome, composto dalle parole *Brussels* e *Oscillator*.

Esso è caratterizzato dalle seguenti reazioni biochimiche:

- $\mathrm{A} \longrightarrow \mathrm{X}$
- $2\,\mathrm{X} + \mathrm{Y} \longrightarrow 3\,\mathrm{X}$

permette agli organismi di regolare i cicli di sonno/veglia anche in assenza di luce.

- $B + X \longrightarrow Y + D$
- $X \longrightarrow E$

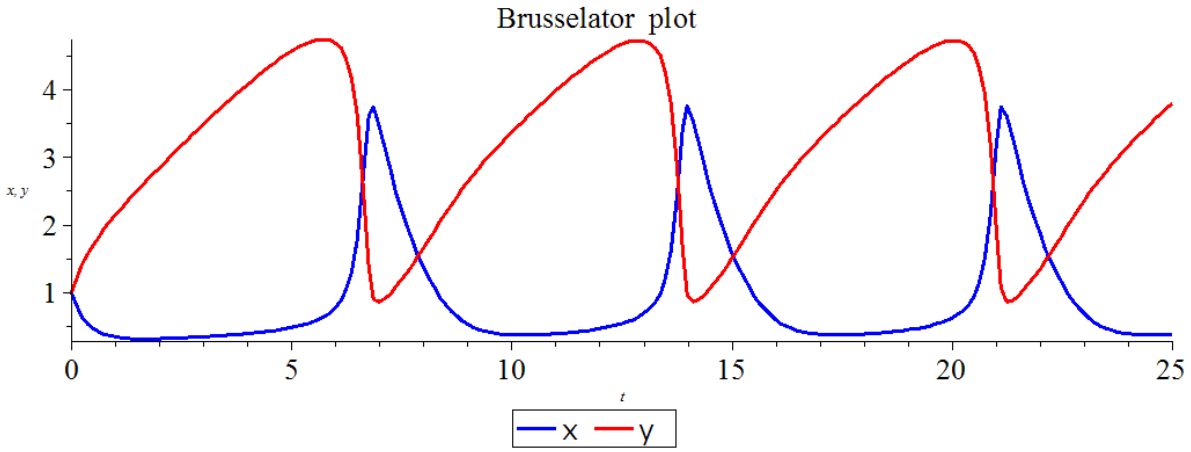


Figura 1.9: Grafico con le concentrazioni delle due proteine del brusselator.
(fonte: `www.wikipedia.org`)

Date concentrazioni unitarie delle molecole X e Y, il Brusselator raggiunge un punto di equilibrio soltanto se $B < 1 + A^2$: in tal caso, infatti, le concentrazioni di X e Y convergono rispettivamente ai valori di A e B, mentre in caso contrario il sistema permane in uno stato ciclico di oscillazione, come si può notare dalla figura 1.9.

Genetic Toggle Switch

Come ultimo esempio, viene mostrato un *Genetic Toggle Switch*.

In elettronica, un Toggle Switch, noto in italiano come *Deviatore*, è un tipo di interruttore che, invece di interrompere il flusso elettrico, lo ridistribuisce su una porzione differente di circuito; allo stesso modo la versione genetica del Toggle Switch riesce a controllare il metabolismo di una cellula attraverso una funzione che permette di commutare due processi.

Il primo Toggle Switch Genetico è stato costruito nel 2000 da Gardgner, Cantor e Collins all'interno di un batterio di *Escherichia Coli* usando meccanismi di reazione a catena da parte di varie unità di controllo[20].

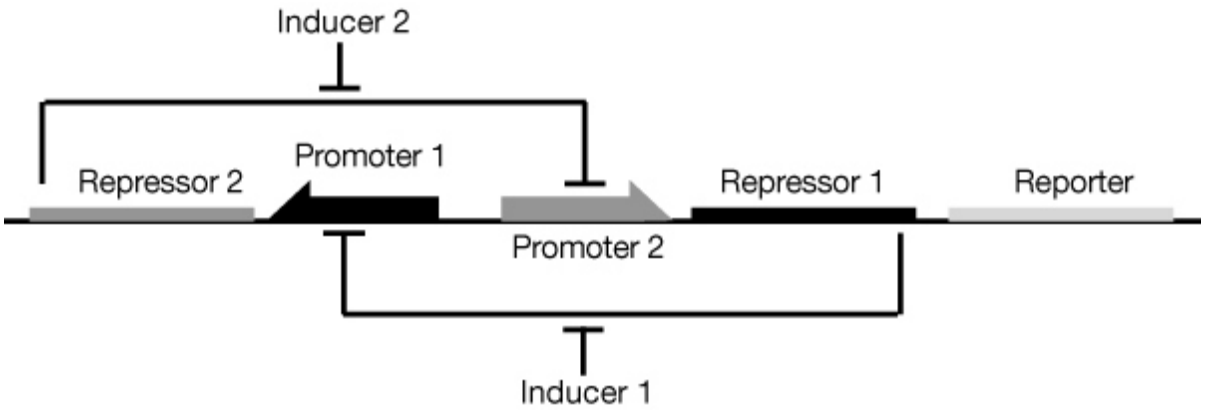


Figura 1.10: Schema di un Genetic Toggle Switch.
(fonte: `www.nature.com`)

1.2 Domain-Specific Languages

Spiegato quindi il funzionamento delle reti geniche, si fornirà ora una breve panoramica sui *Domain-Specific Languages*, analizzandone le caratteristiche e le applicazioni, fino a mostrare come sia possibile creare un proprio DSL per agevolare la risoluzione di particolari problemi. A differenza dei *General-Purpose Language* (*GPL*), infatti, ovvero quell'insieme di linguaggi che, essendo privi di specificità verso un particolare dominio, possono essere utilizzati trasversalmente per la risoluzione di problemi appartenenti a ogni tipo di categoria, un *Domain-Specific Language* è strutturato appositamente per adattarsi a un certo dominio applicativo, permettendo quindi di risolvere in modo mirato problemi relativi a una categoria specifica.

Esempi di *GPL* possono essere i linguaggi di programmazione come C/C++, Java o Python, ma anche linguaggi di modellazione come UML o linguaggi di markup come XML e YAML. Al contrario, sono esempi di *DSL* linguaggi come SQL, HTML o lo Shell Script dei sistemi Unix, tutti utilizzati per funzioni molto specifiche quali, rispettivamente, il reperimento dei dati da un insieme di tabelle che popolano un database, la formattazione e l'impaginazione di pagine web e l'impartizione di specifiche direttive al sistema operativo.

1.2.1 Linguaggi Adatti alla Creazione di DSL

Con il recente spostamento dei paradigmi di programmazione da OOP verso un ibrido di oggetti e funzionale[10], molti linguaggi di alto livello hanno iniziato a fornire un vero e proprio supporto per la progettazione e la costruzione di Domain-Specific Languages interni. Di seguito, quindi, verranno mostrati due noti linguaggi di programmazione – Groovy e Scala – particolarmente adatti alla creazione di DSL, per poi parlare infine di Kotlin, a cui è stata dedicata una sezione a parte essendo esso il linguaggio adottato per lo sviluppo di AGCT.

Groovy è un linguaggio di programmazione, basato su Java e sulla Java Virtual Machine, caratterizzato da una sintassi concisa e semplice da imparare. Esso può integrarsi facilmente con del codice Java già esistente, oltre a fornire supporto per la programmazione funzionale e per la costruzione di DSL[11].

Scala è anch'esso un linguaggio di programmazione basato su Java e sulla Java Virtual Machine che combina programmazione funzionale e orientate a oggetti, ma, diversamente da Groovy, si focalizza principalmente sul fornire una sintassi di alto livello che permetta di evitare le insidie tipiche dei vecchi linguaggi e, soprattutto, sul garantire alte performance anche al crescere della complessità del software. Come Groovy, inoltre, anche Scala risulta particolarmente adatto alla creazione di DSL interni grazie alla sua flessibilità sintattica.

[10] Con *Programmazione Orientata agli Oggetti* (*OOP*) si intende, in informatica, quel paradigma di programmazione che permette di lavorare – definendoli, creandoli e utilizzandoli – con degli oggetti software che incapsulano le proprie logiche e che possono interagire con altri oggetti tramite scambio di messaggi, cosa che non succede invece nel *Paradigma Funzionale* dove, al contrario, gli oggetti vengono messi in secondo piano per concentrarsi sulle funzioni esistenti fra di essi. Dall'avvento dell'esecuzione parallela e delle problematiche ad essa legate, la programmazione funzionale ha quindi ricominciato a guadagnare sempre più popolarità, e la quasi totalità dei linguaggi avanzati forniscono ormai un paradigma ibrido che garantisce sia la possibilità di creare oggetti, sia la possibilità di controllare e delegare il flusso del programma a dei blocchi funzionali.

[11] Per l'ultimo punto in particolare si rimanda alla pagina: `http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html`.

1.2.2 Kotlin

Sviluppato da un team guidato da *Andrey Breslav* e *Dmitry Jemerov* nei laboratori dell'azienda *JetBrains*, Kotlin, la cui prima versione risale al 2011, è un linguaggio di programmazione concepito come un miglioramento di Java interoperabile con esso. Fortemente ispirato da Scala – già disponibile dal 2004 –, uno dei principali obiettivi di Kotlin è sempre stato quello di superare il problema, notoriamente presente nell'altro linguaggio, dei lunghi tempi di compilazione, assicurando invece la velocità tipica di un compilatore Java[12]. Inoltre, Kotlin si ispira anche a buona parte dei linguaggi di programmazione correntemente utilizzati – quali, fra tutti, C#, Javascript, e Python –, e anzi è stato ideato proprio con l'intento di racchiudere in un unico linguaggio tutte le migliori caratteristiche presenti negli altri garantendo, allo stesso tempo, una completa portabilità su ogni tipo di piattaforma.

Concetti Fondamentali in Kotlin

Prima di proseguire, si fornisce una breve panoramica sui termini e i concetti fondamentali presenti in Kotlin, necessari per una corretta comprensione delle sezioni successive:

- si indica con **classe** un qualsiasi tipo di dato astratto che può essere utilizzato come modello per la creazione di **oggetti**, comunemente detti *istanze della classe*;
- una *classe* può contenere delle **proprietà**, ovvero l'insieme dei dati interni che ne descrivono lo stato, e dei **metodi**, ovvero delle funzionalità applicabili ad essa che, a partire da una serie di input, restituiscono un certo output;
- con **funzioni top-level** si indicano invece tutte quelle operazioni non applicabili ad alcuna classe in particolare che però, analogamente ai *metodi*, restituiscono un certo output a partire da una serie di input.

[12] `www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html`.

Supporto per la Definizione di DSL Interni

L'adeguatezza di Kotlin per la creazione di DSL interni deriva principalmente dalla possibilità di scrivere una sintassi pulita e fortemente espressiva grazie ai meccanismi tipici della programmazione funzionale, prime fra tutti le *Funzioni di Ordine Superiore*[13], e ad altre funzionalità avanzate relative al parsing.

Di seguito verranno quindi mostrate e spiegate le principali caratteristiche presenti in Kotlin utili allo sviluppo di DSL interni, le quali hanno indirizzato verso di esso la scelta del linguaggio in cui sviluppare AGCT:

Extension Functions sono funzioni – o proprietà – che estendono direttamente il comportamento di una classe senza il bisogno di crearne una nuova, figlia di quella in questione. L'oggetto sul quale si invoca il metodo è detto *receiver* e può essere richiamato all'interno della funzione tramite la keyword *this*.

Infix Function sono funzioni unarie applicate a un receiver – possono infatti essere dei metodi implementati all'interno della classe o anche delle extension function – a cui è possibile rimuovere i punti e le parentesi tonde durante l'invocazione. Degna di particolare nota, inoltre, è la possibilità di concatenare più infix functions fra loro, nel rispetto delle regole di priorità imposte da Kotlin.

Operator Overloading permette, data una certa classe, di utilizzare gli operatori standard di Kotlin attribuendo loro il significato che si desidera.

Convenzione per il metodo "get" è un tipo specifico di *Operator Overloading* applicato all'operatore *get*, che permette l'invocazione tramite parentesi quadre.

Lambda Esterne alle Parentesi se l'ultimo parametro di una funzione è una *higher-order function* – le cosiddette *lambda* – allora questo può essere passato fuori dalle parentesi tonde. In particolare, ciò risulta utile se combinato con l'operatore **invoke**, che permettere di utilizzare gli oggetti come se fossero delle funzioni, ovvero invocandoli tramite parentesi tonde.

[13] Le Funzioni di Ordine Superiore, o *Higher-Order Functions*, sono funzioni che accettano altre funzioni come parametri.

Sintassi Normale	Sintassi Moderna	Funzionalità
StringUtils.capitalize(s)	s.capitalize()	Extension Function
1.to("one")	1 to "one"	Infix Function
set.add(2)	set += 2	Operator Overloading
map.get("key")	map["key"]	Convention For "get" Method
file.use({ it -> it.read() })	file.use { it.read() }	Lambda Outside of Parenthesis
sb.append("yes") sb.append("no")	with(sb){append("yes") append("no")}	Lambda with Receiver

Tabella 1.1: Esempi tratti dalla Tabella 11.1 del libro *Kotlin in Action* [21, Capitolo 11]

Lambda with Receiver permette di utilizzare la lambda come se fosse una extension function invocata su un *receiver* il quale, all'interno del corpo della funzione, potrà essere richiamato tramite la keyword *this*.

Kotlin DSL per Gradle

Prima di concludere, si mostra quindi un esempio pratico di DSL sviluppato in Kotlin, ovvero *Gradle Kotlin DSL*[14] — con molta probabilità il più utilizzato in ambito di sviluppo software[15]. Se infatti, inizialmente, i file di configurazione Gradle venivano scritti attraverso un DSL integrato basato su Groovy, ormai in molti hanno deciso di migrare verso il DSL Kotlin, più sicuro ed espressivo grazie al *type checking statico*[16], come si può notare dal confronto fra le listings 1.1 e 1.2.

```
1 // build.gradle
2
3 plugins {
4   id 'java'
```

[14] https://docs.gradle.org/current/userguide/kotlin_dsl.html.

[15] *Gradle* è un *build automation tool*, ovvero un software che permette di compilare l'intero software che si sta sviluppando e di eseguire la fase di build e altre azioni specifiche – pulizia della cache, testing del sistema, generazione degli eseguibili o della documentazione, ... – in maniera automatica.

[16] Con *type checking statico* si intende il processo di verifica del tipo di un oggetto a tempo di compilazione, ovvero a priori rispetto all'esecuzione del programma.

```
  id 'org.springframework.boot' version '2.0.2.RELEASE'
}

bootJar {
  archiveName = 'app.jar'
  mainClassName = 'com.example.demo.Demo'
}

bootRun {
  main = 'com.example.demo.Demo'
  args '--spring.profiles.active=demo'
}
```

Listing 1.1: Un esempio di file di configurazione per Gradle espresso con la sintassi di Groovy.
Fonte: `https://guides.gradle.org/migrating-build-logic-from-groovy-to-kotlin`

```
// build.gradle.kts

import org.springframework.boot.gradle.tasks.bundling.BootJar
import org.springframework.boot.gradle.tasks.run.BootRun

val mainClass = "com.example.demo.Demo"

plugins {
  java
  id("org.springframework.boot") version "2.0.2.RELEASE"
}

tasks.named<BootJar>("bootJar") {
  archiveName = "app.jar"
  mainClassName = mainClass
}

tasks.named<BootRun>("bootRun") {
  main = mainClass
  args("--spring.profiles.active=demo")
```

```
21 }
```

Listing 1.2: Lo stesso esempio della listing 1.1 espresso con la sintassi di Kotlin.
Fonte: `https://guides.gradle.org/migrating-build-logic-from-groovy-to-kotlin`

Capitolo 2

Another Genetic Circuit Transcriber

Il ciclo di vita di un software di medie o grandi dimensioni, generalmente, prevede il susseguirsi di una serie di stadi preliminari atti allo studio e alla comprensione del problema che si andrà a risolvere al fine di identificare e, auspicabilmente, di minimizzare fin da subito il numero di complicazioni che potranno emergere durante la fase di implementazione. In questi primi periodi di analisi e di design, pertanto, è stato necessario compiere varie scelte – alcune delle quali si sono rivelate in seguito non del tutto corrette, imponendo una parziale inversione del flusso dello sviluppo che ha riportato il progetto alla fase organizzativa – le quali sono andate a definire la struttura di AGCT, orientando l'intero applicativo verso certe direzioni e certe possibilità di utilizzo e, contestualmente, limitandone altre.

Nel presente capitolo si riporterà quindi un resoconto conclusivo delle decisioni prese per poi mostrare, attraverso descrizioni e diagrammi, ciò che ne è scaturito. Al contempo, si tenterà di spiegare quanto più dettagliatamente possibile le motivazioni che hanno condotto verso alcune scelte piuttosto che altre, pur esponendo, ove presenti, anche le strade che si è deciso di non intraprendere. In particolare, ognuna delle sezioni sottostanti andrà a trattare una specifica fase del ciclo di vita di un software, a partire dall'analisi fino all'ispezione e al testing del sistema.

2.1 Analisi

Quella dell'*Analisi*, come appena detto, è un'attività che si svolge anteriormente all'effettivo sviluppo di un sistema software, la quale si compone di diverse sotto-attività fra cui, le più importanti, l'*Analisi dei Requisiti* e l'*Analisi del Dominio*.

La prima mira a chiarire l'insieme di funzionalità e caratteristiche macroscopiche che il software dovrà avere una volta completato, con l'evidente scopo di rendere noto a priori ciò che ci si aspetta dal risultato finale così da poterlo valutare in maniera critica e oggettiva una volta concluso. La seconda, invece, ha come finalità quella di descrivere formalmente – ovvero in termini informatici – gli attori facenti parte del modello di dominio, così da eliminare il gap presente fra chi commissiona il software, tendenzialmente esperto del dominio applicativo, e chi invece lo realizza.

Essendo AGCT formato da due componenti principali, una strettamente legata all'implementazione del DSL mentre l'altra predisposta a modellare i fenomeni biologici in maniera più classica attraverso soluzioni tipiche della *Programmazione Orientata agli Oggetti* – da qui in poi chiameremo queste due parti rispettivamente di **linguaggio** e di **modello** –, si è ritenuto opportuno analizzare tali componenti separatamente.

In particolare, nel caso del *modello*, essendo esso privo di veri e propri requisiti di funzionalità e operatività in quanto sviluppato unicamente per un'esigenza interna, si è preferito improntare l'analisi sulla comprensione del dominio; al contrario, nel caso del *linguaggio*, ovvero l'effettivo software ipoteticamente dotato di utilizzatori esterni, si è ereditata la parte di dominio e ci si è concentrati maggiormente sulla comprensione dei requisiti, obbligatori e opzionali, che esso avrebbe dovuto avere.

2.1.1 Analisi del Modello del Dominio

Tenendo a mente le entità già nominate nella sezione 1.1.1, si procederà ora a una descrizione più approfondita di queste ultime, tracciando ove possibile dei paralleli con le scienze informatiche al fine di consolidare ulteriormente il concetto di *programmazione di reti geniche*. Analogamente, verranno poi esposti i dettagli più rilevanti delle singole reazioni, focalizzandosi e sulle generiche funzioni che queste possono assumere all'interno

di una rete genica e, più specificatamente, sul rilievo da esse assunto all'interno di questa trattazione.

Entità

Geni Come prima cosa, è necessario chiarire la differenza fra i termini *DNA*, *gene*, *allele* e *cromosoma*, illustrando quali siano le mansioni e le peculiarità di ognuno di questi elementi e, soprattutto, quale gerarchia esista fra di essi.

Si definisce **gene** ogni frammento di DNA incaricato di archiviare l'informazione riguardante una certa espressione genetica, dove con *DNA* (*Acido Desossiribonucleico*) si intende il polimero, formato da una sequenza ordinata di monomeri detti nucleotidi, contenente le informazioni necessarie alla sintesi di RNA e proteine.

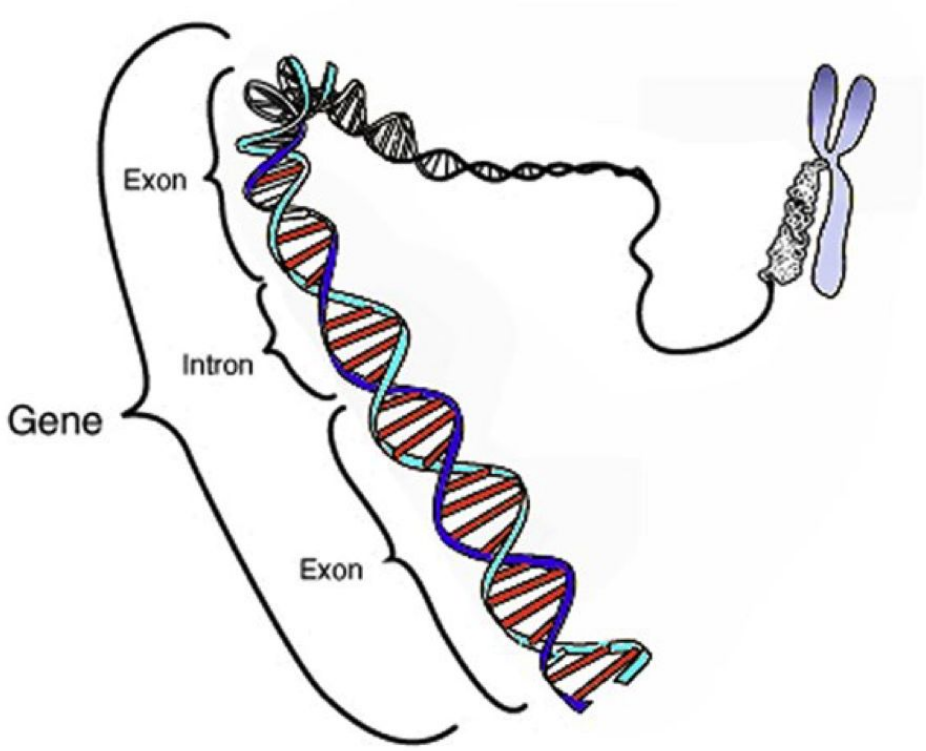


Figura 2.1: Un gene è formato da filamenti di DNA ed è contenuto nei cromosomi.
(fonte: www.wikipedia.org)

Ogni organismo appartenente alla medesima specie biologica possiede lo stesso numero di geni, organizzati in *coppie di cromosomi* – generalmente rappresentati a forma di X, come nella figura 2.1 –, anch'essi ovviamente in egual numero per individui di specie identiche. Dei due cromosomi formanti la coppia, entrambi contengono gli stessi geni che, però, pur avendo la medesima funzione, possono assumere diverse forme in base alla sequenza nucleotidica in essi contenuta, la quale viene definita *allele*[1]. Gli alleli, quindi, diversi nei due rispettivi cromosomi, rappresentano l'effettiva sequenza di DNA che andrà poi a plasmare le caratteristiche macroscopiche dell'organismo; in particolare, dei due alleli se ne può avere uno *dominante* e uno *recessivo* dove, in caso di compresenza, il primo sarà l'unico dei due a caratterizzare l'espressione fenotipica di quello specifico gene all'interno dell'organismo[2].

Ai fini di questa trattazione sarà usato solamente il termine *gene* – ignorando quindi i termini *DNA*, *allele* e *cromosoma* –, senza fare distinzione fra i vari contenitori e l'effettivo contenuto della sequenza nucleotidica per focalizzarsi invece sul concetto più astratto di entità codificante una proteina.

Promotori Affinché il processo di trascrizione inizi, è necessario che delle specifiche proteine, dette *fattori di trascrizione*, si leghino a degli specifici siti, posti all'inizio di ogni sequenza genetica contenuta all'interno di una catena di DNA, detti promotori.

Sebbene i *promotori* siano indicati in buona parte delle reti geniche, tuttavia, in AGCT si è preferito ignorarli, almeno in questa prima versione, e pertanto non verranno più trattati.

RNA L'*Acido Ribonucleico*, o *RNA*, è una molecola soggetta a degradazione ottenuta della trascrizione di un gene. Esso può essere di tre tipologie:

[1] Utilizzando un linguaggio prettamente informatico, si può vedere un allele come un oggetto istanza della classe gene.

[2] Questa regola, nota come *Legge di Dominanza*, è stata poi rivisitata negli anni ammettendo casi di *Codominanza* e *Dominanza Incompleta*.

- l'**mRNA**, o *RNA messaggero*, ovvero il temporaneo contenitore dell'informazione genetica che verrà, in un secondo momento, usata per la sintesi delle proteine.
- il **tRNA**, o *RNA transfer*, ovvero quello che si occupa di trasferire l'informazione genetica letta dalla sequenza di mRNA – più precisamente da ogni insieme di tre nucleotidi consecutivi, detti *codoni* – nel relativo *polipeptide*, ovvero il componente base delle proteine.
- l'**rRNA**, o *RNA ribosomiale*, ovvero quello che si trova all'interno dei ribosomi, i complessi macromolecolari nel quale viene effettuata la traduzione dell'mRNA, e che compone circa l'80% di tutto l'RNA cellulare.

Da qui in avanti, si utilizzerà il termine *RNA* come sinonimo di *mRNA*, ignorando le altre due tipologie in quanto non necessarie alla descrizione di una rete genica.

Proteine Una proteina è una molecola soggetta a degradazione costituita da una serie di *amminoacidi* ottenuti dalla traduzione dell'RNA messaggero e connessi fra loro attraverso dei legami peptidici. Essa rappresenta il risultato terminale dei processi di trasformazione dell'informazione genetica, il quale verrà poi inviato ai relativi organuli di competenza andando, di fatto, a determinare il fenotipo dell'organismo.

Regolatori Si può definire regolatore una qualsiasi molecola – generalmente una proteina prodotta da un altro gene, detto *gene regolatore* – che interviene in meccanismi di *regolazione genica*. Tale molecola andrà quindi a legarsi a un gene, influenzandone la produttività, con un certa probabilità detta *binding rate*, per poi slegarsi con un'altra frequenza detta *unbinding rate*.

I regolatori sono, pertanto, degli attori fondamentali all'interno di una rete genica, e possono essere divisi, a seconda della loro funzionalità, in *attivatori* e *inibitori*, i quali rispettivamente incrementano o decrementano la produzione di una proteina da parte del gene bersaglio della regolazione.

Non essendoci sostanziali differenze fra le due categorie, in questo testo verrà usato il termine più generale di *regolatore*.

Reazioni

Duplicazione Come specificato nel primo punto del *Dogma Centrale della Biologia Molecolare*, la duplicazione ha come scopo quello di propagare l'informazione genetica all'interno della cellula, così che più frammenti di DNA possano essere trascritti parallelamente al fine di diminuire il tempo necessario alla sintesi delle proteine. In ogni caso, essendo questo un fattore non essenziale nella programmazione delle reti geniche – esso diviene infatti necessario soltanto a posteriori, in un sistema reale, qualora si volesse aumentare il rate di codifica di una proteina –, la sua modellazione non rientra negli scopi di questo lavoro.

Trascrizione La trascrizione è un processo che avviene con una frequenza detta *rate basale* e che permette il trasferimento dell'informazione genetica dal DNA all'RNA. In particolare, nel nostro caso in cui il DNA è finalizzato alla codifica di una proteina, la molecola restituita sarà un filamento di mRNA o, per essere ancora più precisi, un filamento di *pre-mRNA*, il quale deve essere sottoposto ad ulteriori passaggi – fra i quali lo *splicing*, che rimuove dal pre-mRNA le porzioni di nucleotidi non codificanti – prima di maturare e raggiungere una forma adeguata alla traduzione. Come si nota nella figura 2.1, infatti, il DNA contiene delle regioni delle *introni* e delle regioni dette *esoni*, delle quali soltanto le seconde sono utili alla codifica delle proteine, mentre le prime contengono sequenze "cuscinetto"[3].

La trascrizione è quindi il passo fondamentale che permette di utilizzare le reti geniche così come ci si prefigge di fare, e ha inizio nel momento in cui i fattori di trascrizione si legano al DNA srotolando i due filamenti in porzioni singole. A seguire, dei complessi molecolare detti *DNA-polimerasi* si occupano di leggerne le sequenze nucleotidiche e creare il relativo filamento unico di RNA fin quando non incontrano specifiche sequenze dette *terminatori*.

[3] Lungi dall'essere inutili, le sequenze introniche sono impiegate in varie funzioni – molte delle quali non ancora completamente note – fra cui la possibilità di regolare il cosiddetto *splicing alternativo* e la possibilità, attraverso la presenza di *enhancer*, di aumentare la velocità di polimerizzazione delle RNA-polimerasi. Per una trattazione più esaustiva si rimanda a: `https://it.wikipedia.org/wiki/Introne`.

Sebbene la capacità da parte del DNA di codificare direttamente una proteina, evitando lo stadio intermedio dell'mRNA, sia stata provata soltanto in vitro[22], per semplicità sia di progettazione che di utilizzo si è deciso di includere questa opportunità nel modello.

Traduzione Nel caso di DNA finalizzato alla trascrizione di mRNA, la traduzione è il passaggio successivo – e conclusivo –, anch'esso occorrente con un certo *rate basale*, che permette la sintesi della relativa proteina a partire dall'informazione genica trasferita. Analogamente alla trascrizione, dove il processo è eseguito dalle *DNA-polimerasi*, così anche in questo caso si ha un complesso molecolare a cui sono delegate le operazioni di lettura delle sequenze e creazione delle relative catene polipeptidiche, ovvero il *Ribosoma*.

Regolazione La regolazione genica è il processo che permette a una cellula di incrementare o ridurre la probabilità di trascrizione di un certo gruppo di geni al fine di aumentare o diminuire la produzione delle relative proteine tramite una modifica del *rate basale*. Nell'ambito dell'ingegneria delle reti geniche, questo aspetto è di estrema rilevanza in quanto permette un controllo esterno sulla produttività della cellula, che può essere regolata e con l'introduzione forzata delle molecole regolatrici e con meccanismi automatici di inibizione e intensificazione, come ad esempio i feedback positivi e negativi, i quali rendono la rete in costante evoluzione nel tempo, dando origine a comportamenti tanto interessanti quanto, in certi casi, imprevedibili.

Come per i *regolatori*, anche in questo caso si utilizzerà il termine *regolazione* per identificare sia i meccanismi di attivazione che quelli di inibizione, non essendovi particolari differenze fra i due al di fuori dei rate di reazione.

Vincoli Generali

Oltre agli ovvi vincoli di cardinalità delle reazioni descritte sopra e all'obbligo di impostare dei valori maggiori di zero per i rate e le concentrazioni, una rete genica ha anche ulteriori limitazioni. In particolare:

- una molecola che degrada deve avere obbligatoriamente un rate di degradazione, mentre una molecola che non degrada non può averlo.
- un gene può codificare un numero arbitrario di proteine, ma una proteina può essere codificata da un solo gene.
- analogamente, un gene può codificare un numero arbitrario di filamenti di mRNA, ma un filamento può essere codificato da un solo gene.
- il rapporto fra le cardinalità di mRNA e proteine nella trascrizione, invece, è di uno a uno.
- al contrario, non vi sono restrizioni sul numero di reazioni regolabili da un regolatore, e sul numero di regolatori di una reazione.

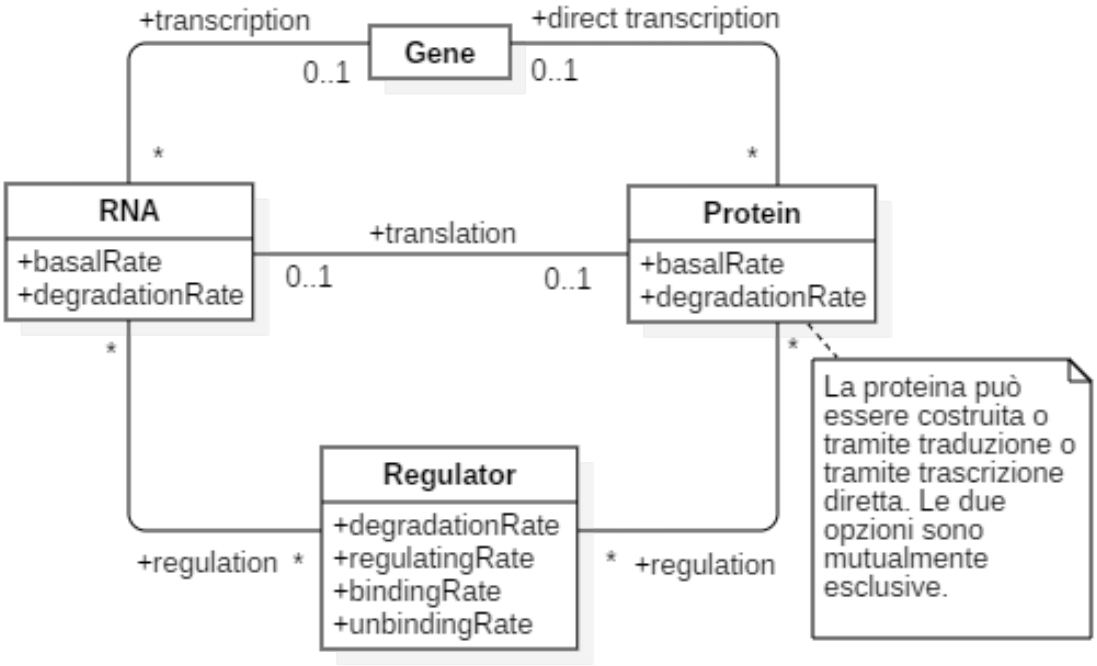


Figura 2.2: Bozza del Modello del Dominio, Prima Versione.

Considerato quindi questo set di regole, e basandosi sulle informazioni collezionate nelle sezioni precedenti, si può ottenere un modello come quello mostrato in figura 2.2.

Tuttavia, nonostante tale modello abbia il pregio di esprimere sia i vincoli di trascrizione/traduzione, sia quelli relativi al rate di degradazione, esso non solo risulta poco estendibile, in quanto l'aggiunta di un ulteriore vincolo, così come la rimozione di uno esistente[4], potrebbero comportare un pesante riorganizzazione del sistema, ma anche poco aderente alla realtà, essendo le entità stesse, infatti, a contenere delle informazioni che, al contrario, dovrebbero essere proprie delle reazioni.

Per questi motivi, quindi, si è preferito reificare le associazioni, andando a creare un modello più denso in cui le reazioni venissero considerate delle entità a sé stanti, come indicato dallo schema in figura 2.3.

2.1.2 Analisi dei Requisiti

Dopo aver illustrato l'ambito entro il quale si andrà a collocare il progetto, ne verranno ora esposti i principali requisiti, suddivisi in due categorie: quelli *funzionali*, ovvero le features che dovranno essere garantite agli utilizzatori del software, e quelli *non funzionali*, ovvero i vincoli e le regole che il sistema dovrà rispettare in termini di qualità, metodi di sviluppo, facilità d'uso, ecc...

Prima di farlo, tuttavia, va chiarito verso che genere di targer andrà indirizzato l'applicativo, distinguendo fra due tipologie di possibili utilizzatori, ovvero:

- gli **utenti**, cioè coloro che usufruiranno di AGCT per descrivere le proprie reti geniche e, successivamente, esportarle.
- i **programmatori**, cioè coloro che vorranno creare delle estensioni per AGCT.

[4] Da sempre siamo abituati al fatto che nuove scoperte scientifiche mostrino come le leggi che, al momento, pensiamo essere inviolabili, possano in realtà ammettere delle eccezioni. Se ciò non bastasse a giustificare un cambiamento di modello, in quanto accadimento poco probabile, si fa comunque notare che il fascino della simulazione di un sistema attraverso un calcolatore è dato anche dalla possibilità di studiarne il comportamento quando sottoposto a vincoli differenti, siano essi validati o non dei modelli teorici.

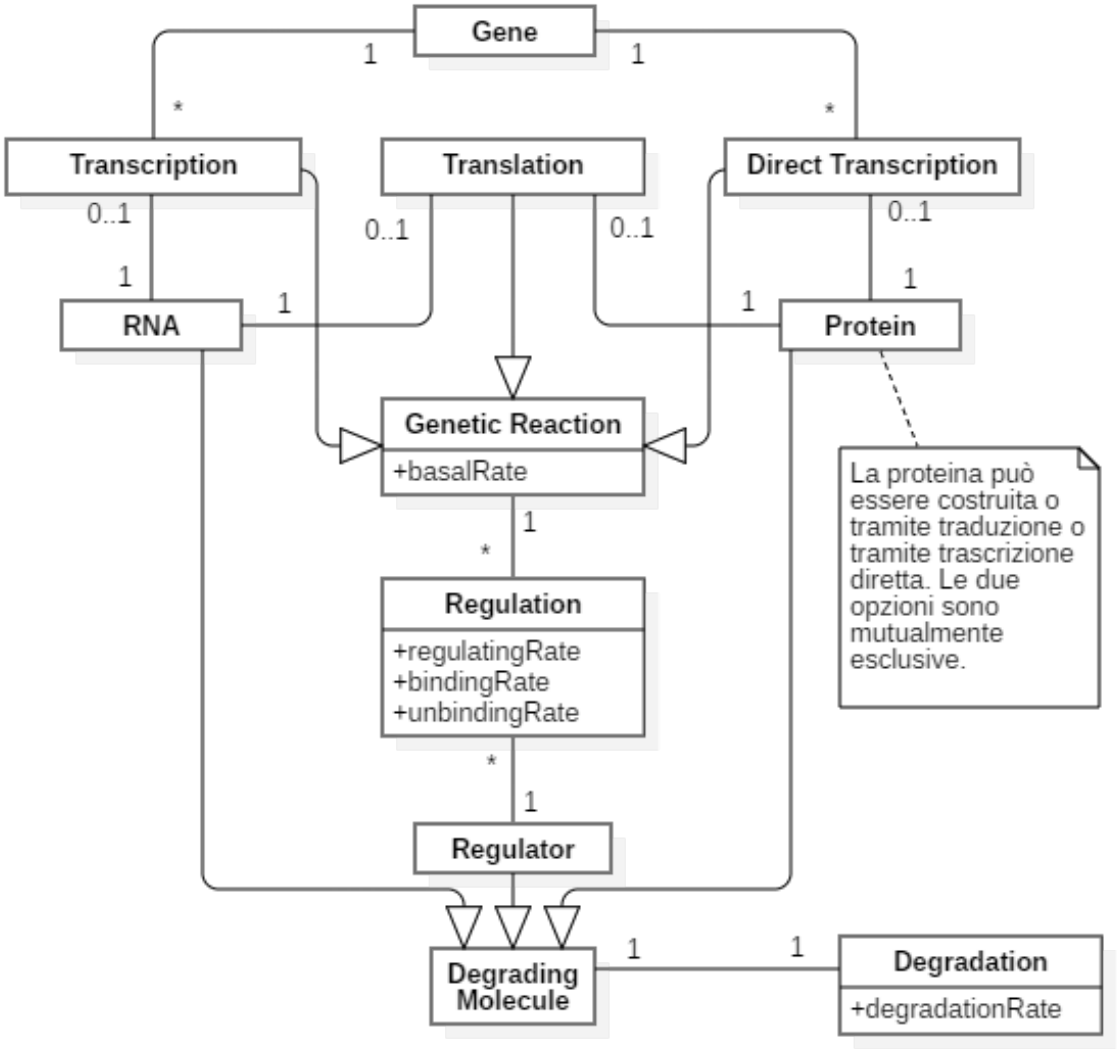


Figura 2.3: Bozza del Modello del Dominio, Seconda Versione.

Requisiti Funzionali

Inserimento Dati un *utente* deve poter descrivere un circuito attraverso le direttive del domain-specific language, inserendovi le entità e le reazioni che ne fanno parte. Tale circuito, inoltre, andrà mappato sulle classi presenti nel modello di AGCT.

Setting dei Parametri un *utente* deve poter settare i rate delle reazioni e le concentra-

zioni iniziali delle entità. Dev'essere inoltre possibile non esplicitare tutti i valori; in quei casi, il software provvederà automaticamente a utilizzare un insieme di valori di default.

Export del Circuito il circuito descritto da un *utente* deve poter essere esportato verso uno o più formati supportati.

Estensione del Software a un *programmatore* deve essere garantita la possibilità di implementare nuovi generatori di export verso formati non ancora supportati dal software.

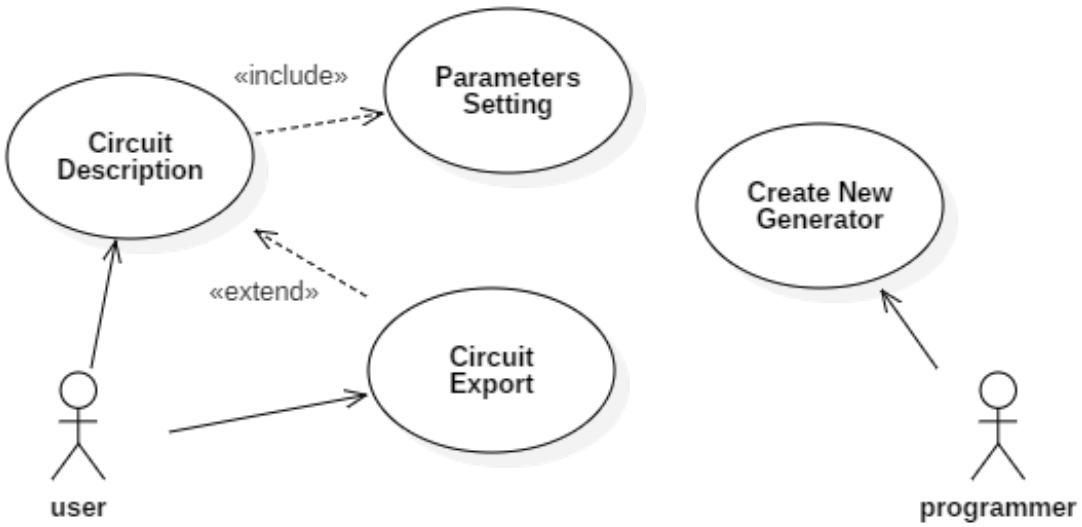


Figura 2.4: Diagramma dei Casi d'Uso di AGCT.

Requisiti Non Funzionali

Conformità al Linguaggio Naturale il DSL deve essere quanto più possibile vicino al linguaggio naturale, cercando di evitare le formalità tipiche dei linguaggi di programmazioni – punti, parantesi e altri simboli non necessari – e garantendo una sintassi ad alta comprensibilità.

Estensibilità il software deve essere pensato e realizzato con l'idea di future aggiunte ed estensioni, sia interne che esterne.

Separazione delle Funzionalità le componenti di modello, linguaggio e generazione dei dati da esportare devono rimanere separate[5], e la dipendenza dell'una dalle altre minimizzata.

Incapsulamento del Modello il modello ha come unico scopo quello di fornire un appoggio concreto sul quale basare il DSL. Non essendo quindi utile come software a se stante, esso va nascosto agli utilizzatori esterni, così da impedire agli utenti di mischiare alle direttive del linguaggio dei blocchi di codice procedurale che si interfaccino direttamente con il modello sottostante.

2.2 Design del Modello

Partendo dall'analisi fatta nella sezione precedente, in questa verranno spiegate – e illustrate attraverso dei diagrammi UML – le principali scelte di design riguardanti il modello interno di AGCT, le quali hanno posto la base per la sua successiva implementazione.

2.2.1 Design delle Variabili

Prima di addentrarsi nel design delle entità e delle reazioni, è necessario esporre brevemente i metodi con cui si è deciso di modellare i valori di concentrazione e rate.

Banalmente, infatti, si potrebbe pensare di memorizzare tali dati all'interno di semplici variabili intere o a virgola mobile, cosa che, tuttavia, risulterebbe molto limitante nel momento in cui si volesse eseguire un set di simulazioni della stessa rete genica sottoposta a condizioni iniziali differenti — si ricordi, ad esempio, il già discusso caso dell'individuazione dei parametri non misurabili, in cui fra un insieme di valori si va a scegliere quello che meglio approssima i risultati sperimentali ottenuti, a sostegno del fatto che quello dei valori multipli è uno scenario fin troppo usuale per poter essere ignorato .

Affinché le variabili potessero accettare anche più di un valore, quindi, si è pensato quindi di creare due classi *Concentration* e *Rate*, entrambe figlie dell'entità generica

[5] Si veda, a riguardo, la sezione 2.4.1 a pagina 59.

UnsignedVariable come si può evincere dalla figura 2.5, le quali ricoprissero il ruolo di wrapper di un insieme immutabile di oggetti dello stesso tipo.

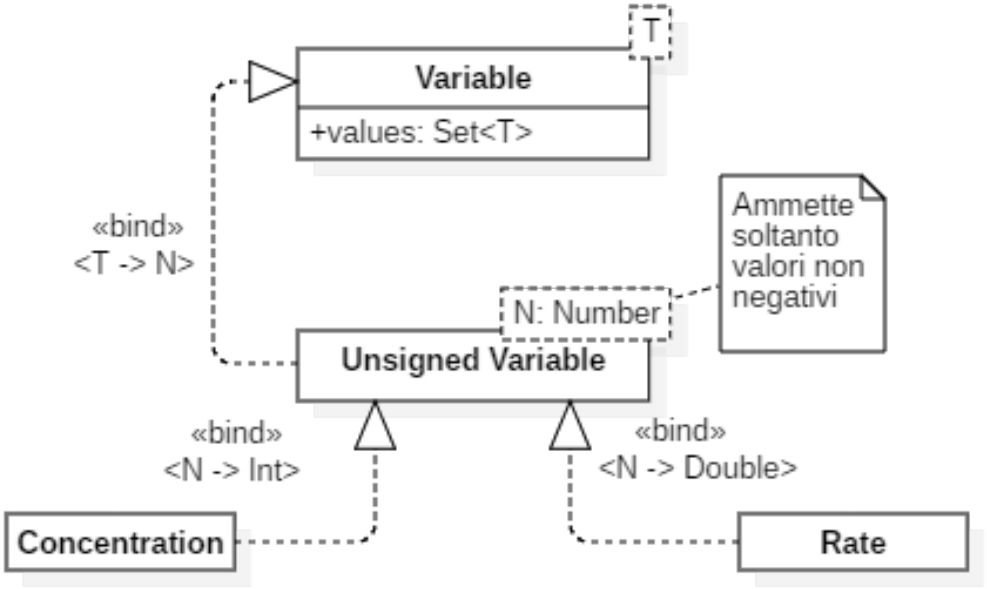


Figura 2.5: Tipi di Variabile presenti in AGCT.

2.2.2 Design delle Entità

Siano **Geni**, **mRNA**, **Proteine** e **Regolatori**, come già indicato nella sezione 2.1.1, le quattro entità da descrivere in AGCT, risulta palese che esse presentino alcune caratteristiche collettive, quali:

- un **nome** che le identifica — ad esempio, nei vari repressilator visti a pagina 12, apparivano i geni *TetR*, *LacI* e λcI.
- eventuali **alias** per indicare il nome completo della molecola, in una o più nomenclature[6], o anche altre sue abbreviazioni.

[6] Ad oggi la nomenclatura standard utilizzata in chimica è definita dalla *IUPAC* (*International Union for Pure and Applied Chemistry*), tuttavia, per ragioni storiche, alcune molecole ancora mantengono il loro nome della nomenclatura tradizionale, oppure questi vengono usati liberamente come sinonimi.

- una **concentrazione iniziale** che permetterà, poi, l'esecuzione della rete genica.

Tali caratteristiche, quindi, dovranno essere modellate in un'interfaccia comune, ragionevolmente chiamata *Entity*[7], mentre altre, che di seguito verranno descritte, dovranno essere specifiche soltanto per singole entità o gruppi di esse.

Prima di procedere, tuttavia, va fatto notare che i nomi utilizzati da qui in poi non vanno presi come termini tecnici del linguaggio biochimico quanto, piuttosto, come termini specifici relativi a questa trattazione, i quali andranno, successivamente, a identificare i nomi delle interfacce e delle classi presenti all'interno di AGCT.

Rate di Degradazione Definiamo *entità degradante* una qualsiasi molecola avente un rate di degradazione, al quale è associata la reazione chimica: $E \longrightarrow \emptyset$.

Sono entità degradanti l'*mRNA* e le *proteine*, mentre i regolatori possono o possono non degradare a seconda dei casi — si ricorda infatti che un regolatore non è un effettivo tipo di molecola come invece lo sono l'mRNA o le proteina, quanto piuttosto una qualsiasi entità avente uno specifico ruolo all'interno di una reazione di regolazione, pertanto se l'entità regolante dovesse essere una proteina essa avrà il proprio rate di degradazione, se invece dovesse essere una qualsiasi altra molecola non degradante il regolatore risulterà privo di tale proprietà.

Possibilità di Regolare Le entità in grado di prendere parte a una reazione di regolazione genica in qualità di regolatori, da qui in poi indicate come *entità regolanti*, sono l'*mRNA*, le *proteine* o anche qualsiasi altro tipo di molecola generica. Non lo sono invece i *geni* — non direttamente, almeno; un gene infatti può regolarne un altro soltanto attraverso la proteina per cui esso codifica.

Va chiarito tuttavia che l'essere un'entità regolante non comporta automaticamente l'essere un regolatore: un'entità regolante, infatti, può o può non ricoprire il ruolo

[7] Per garantire una migliore entensibilità del modello, al momento dell'implementazione si è creata una classe aggiuntiva, chiamata *Entity Parameters*, contenente tutti i parametri comuni delle entità, la quale andrà poi passata al costruttore delle entità, con una modalità simile a quella del pattern *Builder*. Contestualmente, si è anche fornita una funzione di primo livello nominata *entity* che prendesse in ingresso l'id dell'entità, il suo tipo e un eventuale routine per il setting dei parametri, così da poter istanziare un'entità – che, internamente, verrà creata tramite reflection – con una sintassi del tipo: entity<Type>(id) ... .

di regolatore; ciò che è certo, invece, è che a ricoprire il ruolo di regolatore sarà sicuramente un'entità regolante.

Capacità di Codificare o Essere Codificata Nelle reazioni di trascrizione e traduzione intervengono sempre due entità: una, detta *coder*, che è la parte codificante, e l'altra, detta *target*, che è la parte codificata.

Le uniche due entità in grado di codificare sono quindi i *geni* – da cui si possono ottenere mRNA o proteine – e i filmenti di *mRNA* — da cui si possono ottenere soltanto le proteine. Va da sé che, per gli stessi motivi, a poter essere codificate saranno soltanto le *proteine* e l'*mRNA*.

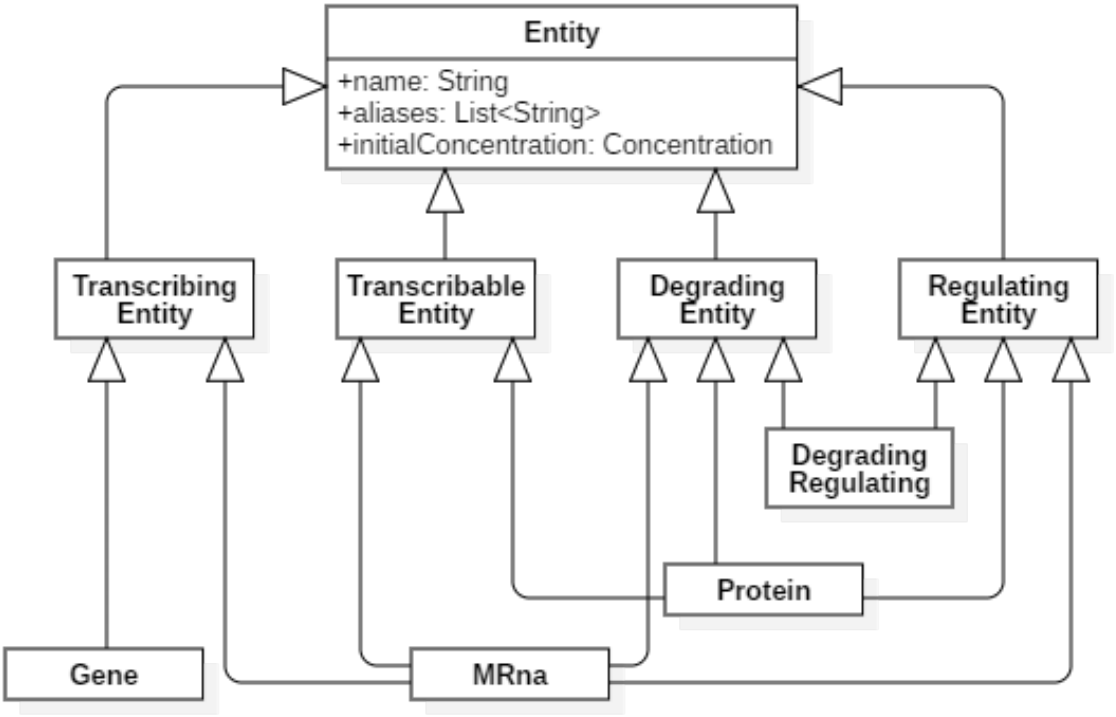


Figura 2.6: Schema della Gerarchia presente fra le Entità di AGCT.

Entità Composte Un'ultima considerazione va fatta per le entità formate da due o più componenti singole. Esse entrano in gioco nei meccanismi di regolazione, dove il regolatore si lega al gene formando una nuova entità, anch'essa codificante — quindi

in tutto e per tutto un gene, almeno da un punto di vista biologico. Tali entità vengono create a una certa velocità detta **binding rate** e si slegano a un'altra velocità detta **unbinding rate**, scindendosi nelle due componenti originarie senza avervi apportato alcuna modifica.

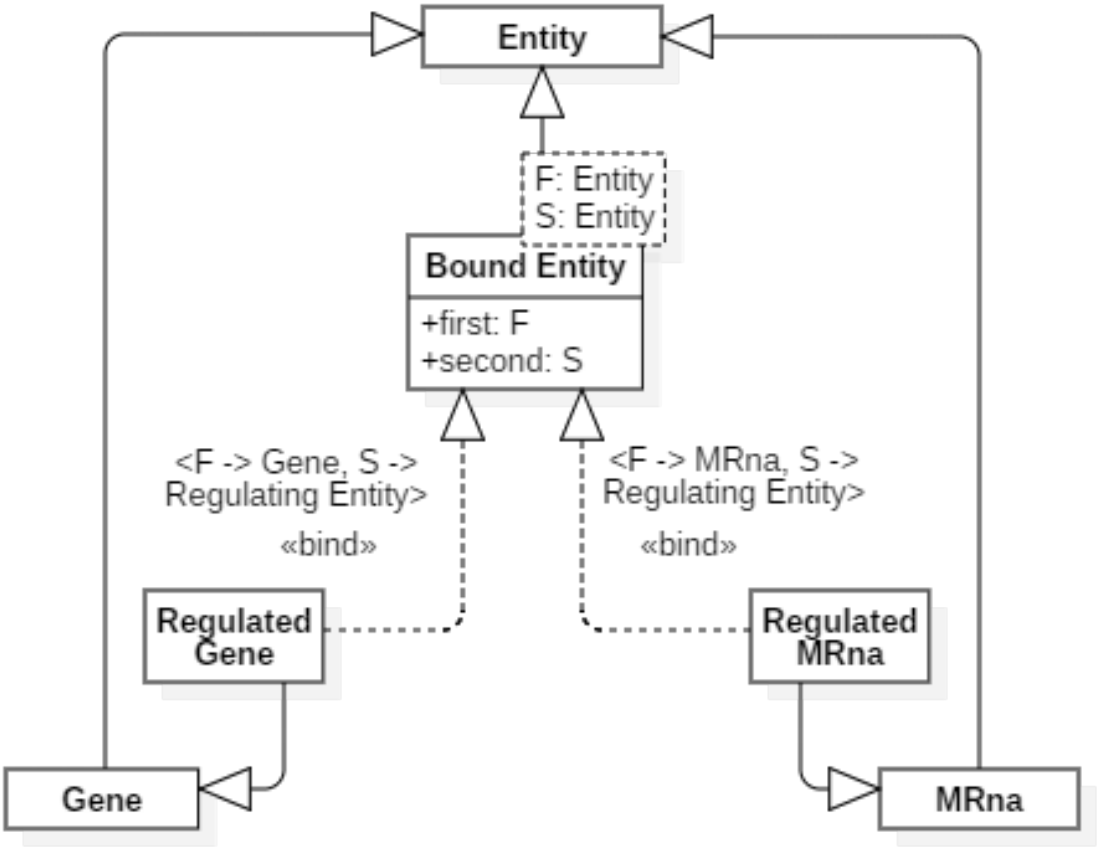


Figura 2.7: Design delle Entità Composte. Da notare la presenza dei *generici*, più volte utilizzati all'interno di tutto il progetto.

2.2.3 Design delle Reazioni

Se stabilire una gerarchia fra le entità è stato semplice per via della presenza di attributi e caratteristiche comuni, lo stesso non vale per le reazioni, dove i ruoli delle entità in gioco, così come il nome e il numero di rate, variano di caso in caso.

Esistono infatti delle reazioni come la *regolazione* che, di fatto, rappresentano tre diversi passaggi, ovvero: il *binding* del gene con il regolatore, l'*unbinding* del gene regolato e l'effettiva *trascrizione* regolata. Per tale motivo è necessario, prima di tutto, distinguere i concetti di **reazione** e **reazione singola**, così come intesi all'interno di AGCT:

- con *reazione singola* si intende un qualsiasi processo chimico che, oltre ad avere un *nome*, è identificato univocamente dai suoi *reagenti*, dai suoi *prodotti* e dal *rate* con cui essa si verifica.
- con *reazione* si intende invece una di quelle descritte nella sezione 2.1.1 – ovvero quelle tipiche delle reti geniche – oppure delle reazioni chimiche "personalizzate", le quali possono essere composte da una o più reazioni semplici.

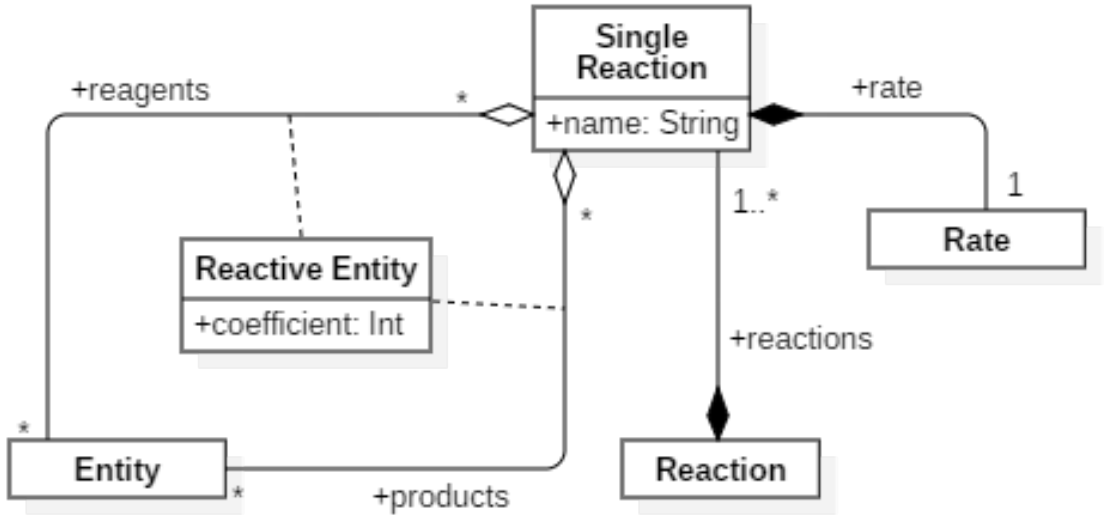


Figura 2.8: Schema della relazione fra la classe *Single Reaction* e la classe *Reaction*.

A questo punto, quindi, tutte le reazioni della rete genica saranno figlie della classe *Reaction* e, come deducibile dallo schema in figura 2.8, sarà possibile ottenere la lista di effettivi processi chimici che la caratterizzano attraverso il campo *reactions*. Ognuna di esse, infine, sarà caratterizzata da una serie di proprietà specifiche. Se ne riporta, di seguito, l'elenco.

Degradazione per poter creare internamente la reazione chimica corrispondente, la classe *Degradation* dovrà avere un proprio *rate di degradazione* e le dovrà essere assegnata un'*entità degradante* come mostrato in figura 2.9.

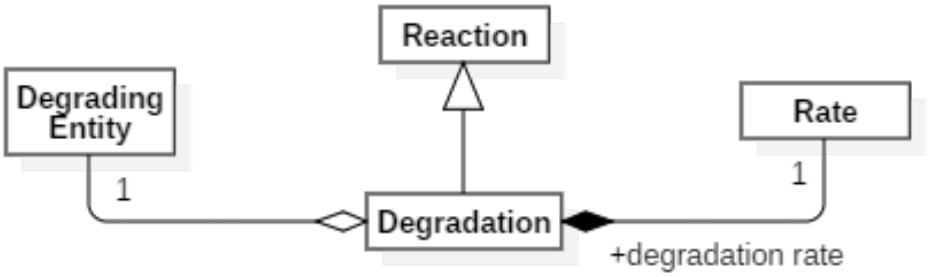


Figura 2.9: Schema delle reazioni di degradazione.

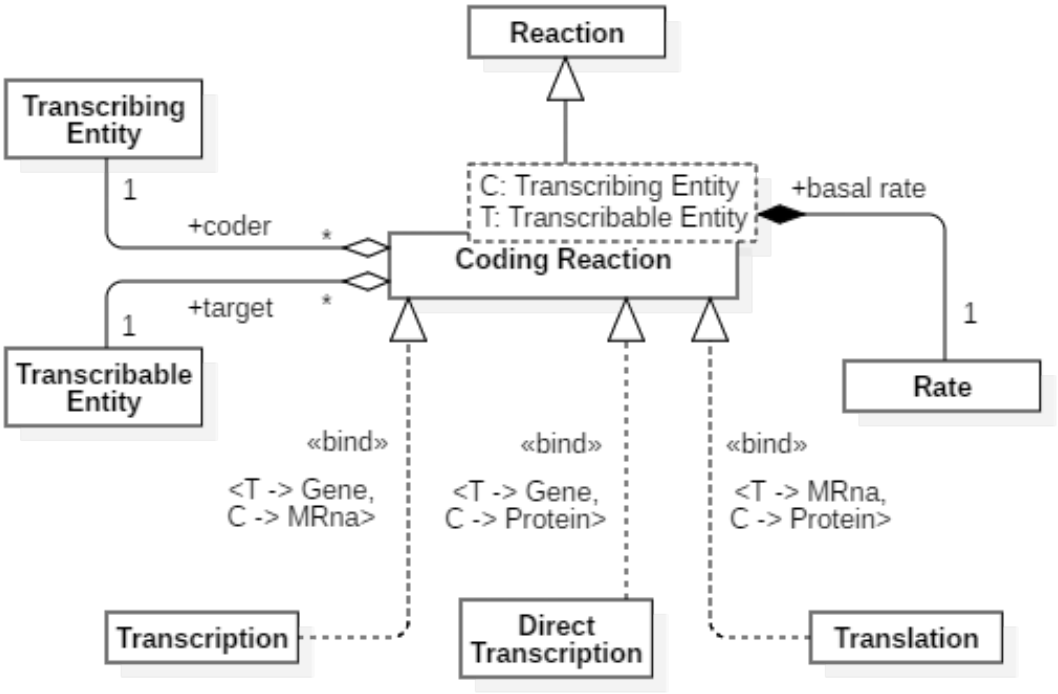


Figura 2.10: Schema delle reazioni di trascrizione e traduzione.

Trascrizione e Traduzione entrambe queste reazioni sono state considerate come caso specifico della più generica *reazione di codifica – Coding Reaction* in figura 2.10 –. Ognuna di essa conterrà quindi un *coder*, l'entità trascrivente, e un *target*, l'entità trascritta, entrambi di tipo generico nella classe padre.

Regolazione la regolazione viene effettuata su una *Coding Reaction* da passare come parametro, alla quale andranno aggiunti anche il *regolatore* e i vari *rate* caratteristici della reazione.

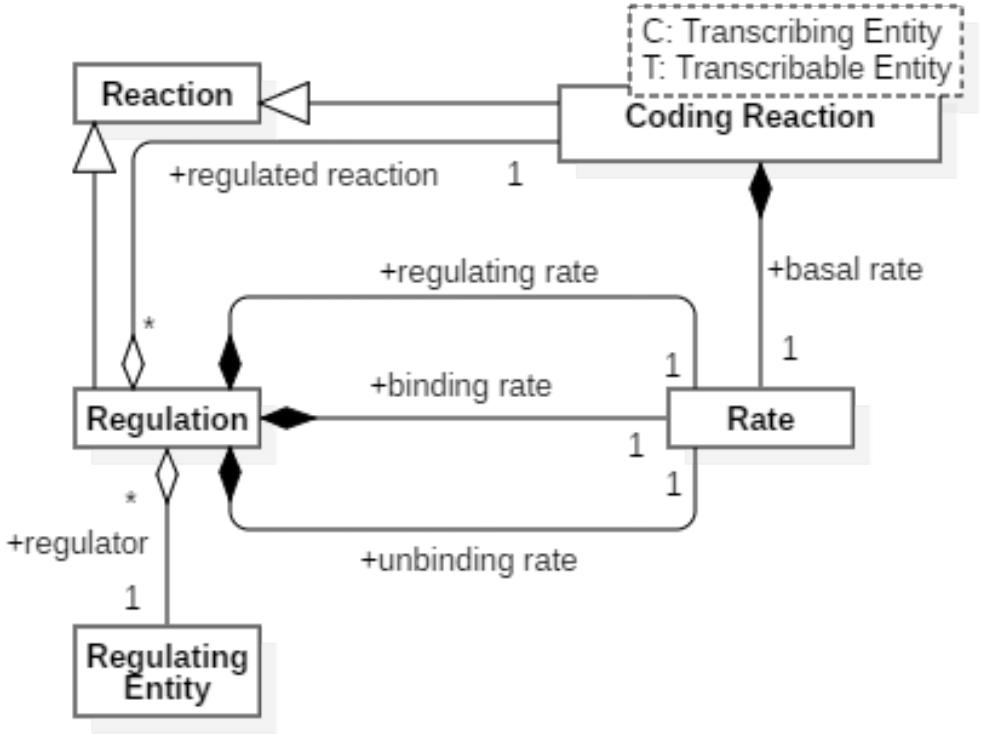


Figura 2.11: Schema della reazione di regolazione.

Reazione Chimica le reazioni chimiche garantiscono la possibilità di personalizzazione della rete genica inserendo processi non direttamente derivati dalle reazioni precedenti ma, ad esempio, relativi al comportamento di due proteine o di molecole esterne che interagiscono con le entità della rete in maniera particolare.

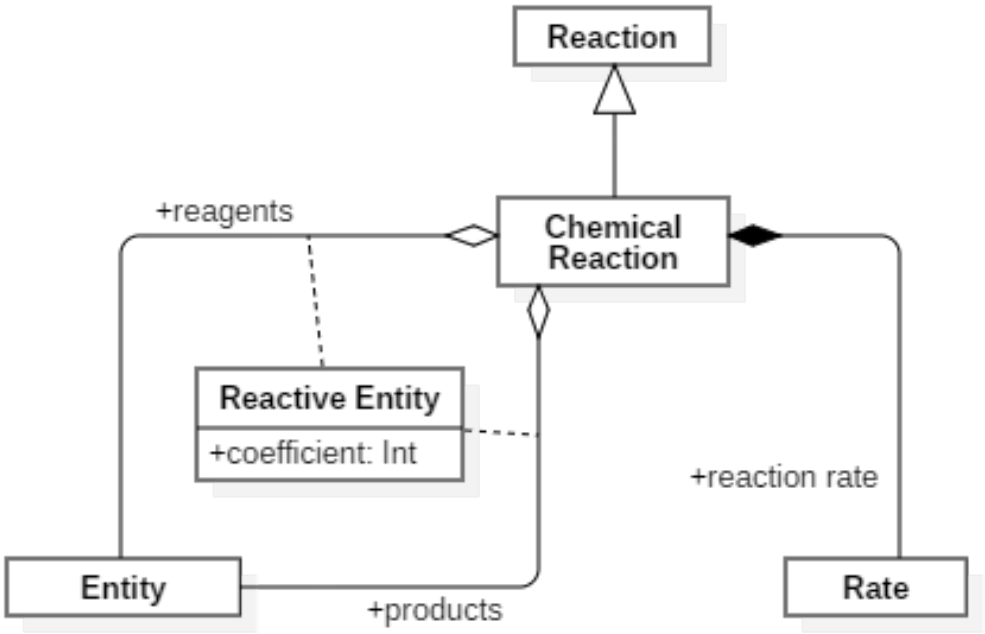


Figura 2.12: Schema delle reazioni chimiche personalizzate.

2.2.4 Design della Rete Genica

Prima di concludere questa sezione riguardante il design del modello, va fatta una precisazione sulle modalità con cui le varie entità e reazioni andranno collegate insieme in modo da poter descrivere l'intera rete genica e, al contempo, assicurarsi che al suo interno vengano rispettate tutte le regole elencate nella sezione 2.1.1 — è facile notare infatti che, dagli schemi sopra esposti, non vi sia garanzia alcuna che i vincoli di codifica delle proteine e dell'mRNA, così come il fatto che a ogni entità degradante venga assegnata una propria reazione di degradazione, siano rispettati.

In figura 2.13 viene quindi mostrata la classe *Genetic Circuit*, alla quale è possibile impostare un insieme di regole[8] – o *strategie*, per utilizzare la terminologia tipica del pattern in questione, ovvero lo *Strategy* – le quali dovranno essere rispettate sia nel

[8] Da un punto di vista implementativo, le regole sono delle *funzioni di ordine superiore* che si occupano di lanciare un'eccezione nel caso in cui non vengano rispettate. Esse vengono passate nel costruttore della rete genica.

momento dell'aggiunta di una nuova entità o reazione, sia nel momento in cui si andrà a validare il circuito prima dell'export.

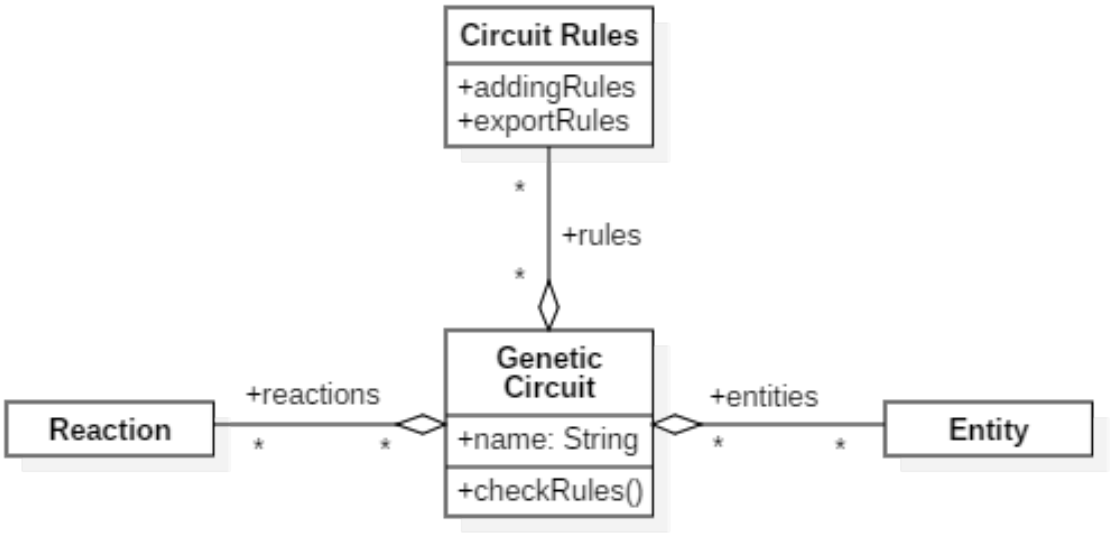


Figura 2.13: Design di una rete genica.

2.3 Design del Linguaggio

Presentato il design della componente di modello di AGCT, verrà ora mostrata la sua controparte, ovvero il linguaggio. Dopo aver stabilito quale tipologia di sintassi utilizzare fra le alternative possibili, quindi, si andranno ad analizzare i meccanismi che permettono al DSL di poggiarsi sul modella senza interferire con esso né in maniera diretta, né attraverso la mediazione di un utente esterno.

2.3.1 Scelta della Sintassi

Nell'ideare e, successivamente, scegliere una sintassi adatta per AGCT, si è tenuto in considerazione il requisito riguardante l'aderenza al linguaggio naturale e l'alta leggibilità. Prima di giungere a una forma conclusiva, pertanto, si è passati attraverso una serie di possibili versioni del DSL, molte delle quali non abbastanza soddisfacenti. Di seguito se

ne mostrano due esempi – i più rilevanti – per poi spiegare, analizzandone le differenze, i motivi per il quale è preferito scegliere la seconda sintassi alla prima.

Sintassi a Blocchi

Come si può notare dalla listing 2.1, la sintassi a blocchi risulta particolarmente conforme al linguaggio naturale, soprattutto per via della totale assenza di segni di interpunzione — fatta eccezione per la descrizione degli spazi di variabili, ovvero *logspace* e *linspace*, dove la loro presenza è giustificata. Tuttavia, si è deciso di abbandonarla in favore della sintassi mostrata nella listing 2.2 proprio perché, a causa della larga presenza di parentesi quadre e, appunto, blocchi indentati, non rimane di facile lettura.

```
1  Create circuit "Feedbacks" with {
2    the gene "gA" having {
3      an initial concentration of 10
4    } that codes {
5      the protein "pA" with {
6        a basal rate of 100
7      } being regulated by {
8        the protein "pA" with {
9          a regulating rate of 50
10         a binding rate of 10
11         an unbinding rate of 10
12       }
13
14       the protein "pB" with {
15         a regulating rate of 200
16       }
17     }
18   }
19
20   the gene "gB" having {
21     an initial concentration of 10
22   } that codes {
23     the protein "pB" with {
24       a basal rate of 100
25     } being regulated by {
```

```
26        the protein "pB" with {
27          a regulating rate of 50
28          a binding rate of 10
29          an unbinding rate of 10
30        }
31
32        the protein "pA" with {
33          a regulating rate of 200
34        }
35      }
36    }
37  } having {
38    default initial concentration of 0
39    default degradation rate into logspace(0, 4, 5)
40    default binding rate into linspace(5, 15, 1)
41    default unbinding rate into linspace(5, 15, 1)
42  } then export to AGCT
```

Listing 2.1: Esempio di rete genica – un semplice sistema fatto da due geni che producono ognuno una proteina che attiva il gene opposto e inibisce sé stesso – descritta attraverso la prima sintassi ammissibile di AGCT (*Sintassi a Blocchi*)

Sintassi Sequenziale

Sebbene il livello di indentazione sia pressoché lo stesso dell'esempio precedente, nella listing 2.2 è evidente la maggior semplicità di lettura, di certo non disturbata dalla presenza di un punto fra le parole ogni tanto e dalle parentesi intorno ai nomi delle entità — che, in ogni caso, possono essere evitate tramite l'utilizzo della keyword *the*, supportata come nella *Sintsssi a Blocchi*. Inoltre, come nota tecnica, va aggiunto che l'implementazione di questa sintassi e, soprattutto, la sua manutenzione, è di gran lunga più semplice rispetto alla precedente, in quanto è sufficiente lavorare all'interno dei singoli livelli senza dover creare una notevole mole di classi che ne facciano da wrapper. Per questi motivi si è preferito utilizzare la sintassi sequenziale rispetto a quella a blocchi.

```
1  Create circuit "Feedbacks" containing {
2    gene("gA") that {
3      has an initial.concentration of 10
```

```
4     codes {
5       protein("pA")
6       with a basal.rate of 100
7       regulated by {
8         protein("pA")
9         with a regulating.rate of 50
10        with a binding.rate of 10
11        with an unbinding.rate of 10
12      } and {
13        protein("pB")
14        with a regulating.rate of 200
15      }
16    }
17  }
18
19  gene("gB") that {
20    has an initial.concentration of 10
21    codes {
22      protein("pB")
23      with a basal.rate of 100
24      regulated by {
25        protein("pB")
26        with a regulating.rate of 50
27        with a binding.rate of 10
28        with an unbinding.rate of 10
29      } and {
30        protein("pA")
31        with a regulating.rate of 200
32      }
33    }
34  }
35 } with {
36  a default initial.concentration of 0
37  a default degradation.rate into logspace(0, 4, 5)
38  a default binding.rate into linspace(5, 15, 10)
39  a default unbinding.rate into linspace(5, 15, 10)
```

```
40 } then export to AGCT
```

Listing 2.2: La stessa rete genica della listing 2.1 descritta però attraverso la seconda sintassi ammissibile di AGCT (*Sintassi Sequenziale*)

2.3.2 Analisi Lessicale della Sintassi

Prima di passare di addentrarsi nel vero e proprio design delle classi del linguaggio, è necessario effettuare un'attenta analisi lessicale della sintassi scelta al fine di individuare il ruolo che ognuna delle parole presenti dovrà assumere all'interno del DSL.

In particolare, tenendo in considerazione quanto detto riguardo alle *infix functions* nella sezione 1.2.2, va notato che le parole **circuit**, **containing**, **that**, **by**, **a**, **an**, **of**, **into**, **default**, **export** e **to** indicheranno appunto delle funzioni infisse chiamate su un oggetto *receiver* e aventi come parametro certi tipi di oggetto posto dopo di esse sui quali, invece, vanno fatte le specifiche distinzioni, poiché:

- parole come **the**, **has**, **codes**, **with**, **regulated** rappresentano degli elementi la cui funzione principale è quella di permettere l'utilizzo della notazione infissa. È probabile pertanto che, al momento dell'implementazione, tali campi andranno semplicemente ad eseguire un *override* del valore *this* o, comunque, a reindirizzare verso una specifica classe che contenga le funzioni descritte in precedenza.

- gli oggetti *numerici* e di tipo **stringa** non vanno ad intaccare la progettazione del DSL. Essi possono essere considerati dei parametri normali da passare alle funzioni infisse, i quali andranno salvati all'interno delle rispettive classi di competenza.

- parole come **Create**, **export**, **initial.concentration**, tutti i vari **rate** e gli **spazi di variabili** rappresentano invece oggetti – o funzioni di primo livello che restituiscono oggetti – utili alla concatenazione di due funzioni infisse oppure alla terminazione di esse attraverso un valore da assegnare alle proprietà interne delle classi — come accade ad esempio per i vari *logspace* e *linspace*, il cui valore restituito verrò poi assegnato alle variabili indicate. Tali oggetti, presumibilmente, andranno implementati tramite pattern *singleton* – figura 2.14 – diventando quelle che potrebbero essere definite come le **keywords** del linguaggio.

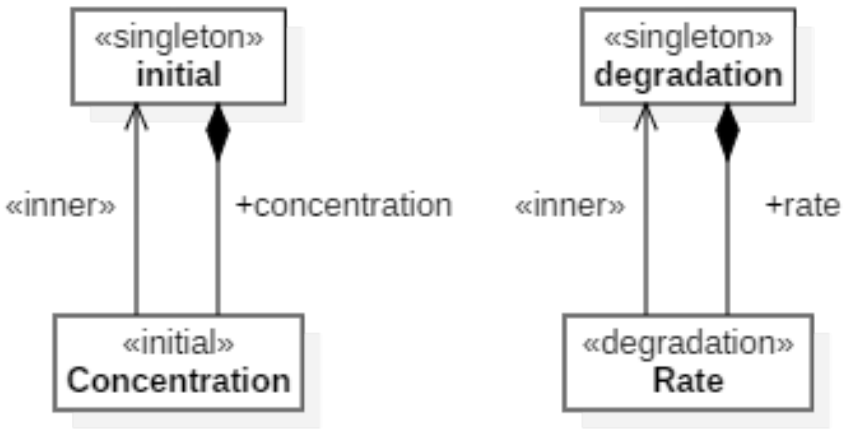


Figura 2.14: Modellazione delle keywords che rappresentano le variabili attraverso un oggetto unico – singleton – e una inner class. Lo stesso modello è applicato anche alle keywords *basal*, *regulating*, *binding* e *unbinding*.

2.3.3 Mapping degli Elementi di Modello

Da un punto di vista architetturale, il linguaggio di AGCT si appoggia sul modello permettendo, tuttavia, l'inserimento dei dati in una forma più flessibile, con la possibilità di cambiarne valori e di avere, almeno in una situazione iniziale, alcune proprietà nulle. Per rendere possibile ciò, è necessario costruire delle nuove classi e interfacce che contengano[9] le classi del modello o, se non altro, che possano essere mappate verso di esse in modo da restituire, al termine della descrizione di una rete, un oggetto di tipo *Genetic Circuit* dal quale poi esportare i dati.

Un esempio di questa modellazione, applicato alle variabili, è indicato in figura 2.15 dove, come si può notare, la gerarchia delle classi di modello è stata di fatto ricreata – in maniera leggermente semplificata, in quanto già con variabile si intende una numerica non negativa –. Lo stesso procedimento, poi, andrà intrapreso anche con le *Entità*, con le *Reazioni* e con l'intera *Rete Genica*, ottenendo dei diagrammi analoghi, salvo piccole

[9] Come indicato anche nel testo *Effective Java* di *Joshua Block*[23], si è preferita alla generalizzazione l'aggregazione, più adatta allo scopo e più facilmente estendibile/manutenibile.

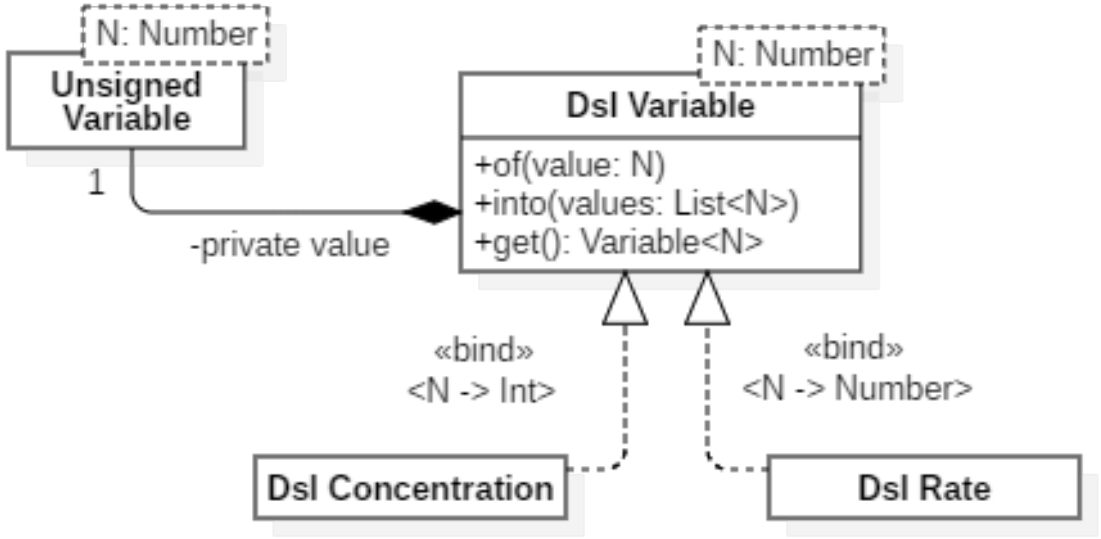


Figura 2.15: Schema delle variabili del DSL.

semplificazioni, a quelli del modello, i quali perciò non verranno mostrati ma ne sarà soltanto data una breve descrizione testuale qui di seguito.

Dsl Entity rappresenta un'entità genetica all'interno del DSL.

Dsl Degradable rappresenta un'entità che potrebbe degradare — ma non necessariamente lo fa. Ne sono un esempio le *Dsl Protein*, obbligate a degradare, ma anche le *Dsl Molecule*, le quali possono o non possono degradare a seconda che il loro rate di degradazione venga settato o meno. Oltre ai soliti parametri *id* e *concentrazione iniziale*, tali entità avranno anche un parametro *rate di degradazione*, essendo la reazione di degradazione non esplicitamente modellata.

Dsl Regulating rappresenta un'entità che potrebbe svolgere il ruolo di regolatore — anche qui, *Dsl Protein* e *Dsl Molecule*.

Dsl Gene rappresenta un gene.

Dsl Protein rappresenta una proteina.

Dsl Molecule rappresenta una generica molecola biochimica, che, per fare un esempio pratico, potrà indicare una molecola regolatrice immessa dall'esterno.

Dsl Reaction rappresenta una reazione genetica all'interno del DSL.

Dsl Transcription rappresenta una reazione di trascrizione.

Dsl Regulation rappresenta una reazione di regolazione.

Dsl Chemical Reaction rappresenta una reazione chimica personalizzata.

Dsl Circuit rappresenta una rete genica all'interno del DSL, la quale dovrà sia fornire i metodi *putReaction* e *getOrPutEntity*, per ottenere o inserire entità e reazioni all'interno del circuito, che contenere un insieme di valori di default per tutte le concentrazioni e i rate, memorizzato all'interno della classe **Dsl Defaults**.

Nonostante l'overhead che essa comporta, questo tipo di architettura garantisce l'incapsulamento delle componenti, permettendo quindi alla parte di modello di rimanere separata da quella del linguaggio. Azzardando un paragone, infatti, si può vedere questa divisione analoga a quella del noto pattern architetturale *MVC*, con il modello a fare appunto da *Model* e il DSL a fare da *View/Controller*.

2.3.4 Livelli e Wrapper

Definite ora le entità, ci si appresta quindi a illustrare il nucleo del linguaggio di AGCT, composto principalmente da *Livelli*, ovvero le porzioni di codice racchiuse dentro le parentesi graffe entro i quali settare i valori delle proprietà, e da *Wrapper di Livelli*, ovvero le porzioni di codice fra una parentesi graffa e l'altra, utili al collegamento dei livelli stessi.

Livello di Entità

Si consideri il seguente blocco di codice:

```
1 gene("gene") that { // Gene Level
2   has an initial.concentration of 10
3   codes {
```

```
4       protein("pro1") that { // Protein Level
5         has an initial.concentration of 10
6         has a degradation.rate of 100
7       }
8     } and {
9       protein("pro2") that { // Protein Level
10        has an initial.concentration of 10
11        has a degradation.rate of 100
12      }
13    }
14  }
```

Definiamo *Livelli di Entità* tutti quei blocchi indicati dal commento a fianco, i quali dovranno fornire una serie di funzioni utili all'inserimento della concentrazione iniziale e del rate di degradazione dell'entità indicata. Al contrario, ciò che è fuori da quelle graffe – la parola *that*, ad esempio –, non va considerato dominio del livello quanto, piuttosto, del suo *wrapper*.

Livello di Reazione

Si consideri il seguente blocco di codice:

```
1  gene("gene") that {
2    codes { // Transcription Level
3      protein("pro")
4      with a basal.rate of 20
5      regulated by { // Regulation Level
6        protein("reg1")
7        with a regulating.rate of 50
8        with a binding.rate of 20
9        with an unbinding.rate of 30
10     } and { // Regulation Level
11       molecule("reg2")
12     }
13   }
14 }
```

A differenza dei livelli di entità, i *Livelli di Reazione* non hanno alcuna caratteristica macroscopica comune. Tuttavia, dovendo entrambi contenere la reazione genetica che al

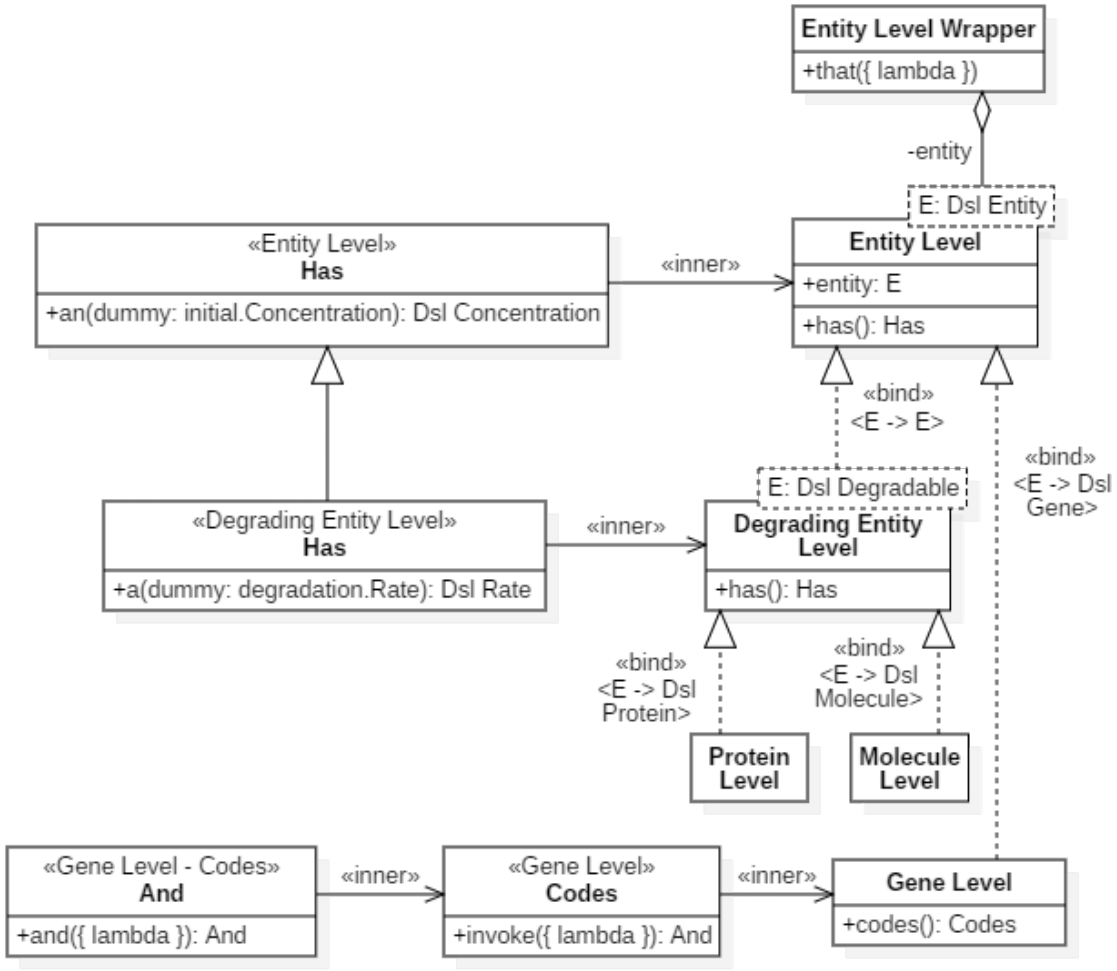


Figura 2.16: Diagramma delle classi *Entity Level* e *Entity Level Wrapper*. Va specificato che, nonostante le parole *has* e *codes*, durante l'analisi della sintassi, non fossero state incluse fra le funzioni ma fra gli oggetti, in Kotlin sarà possibile trattarle come tali – evitando quindi l'utilizzo delle parentesi tonde – dal momento che esse non necessitano di parametri.

loro interno si va a descrivere, sia il *Transcription Level* che il *Regulation Level* sono stati modellati come specializzazioni della classe padre *Reaction Level*.

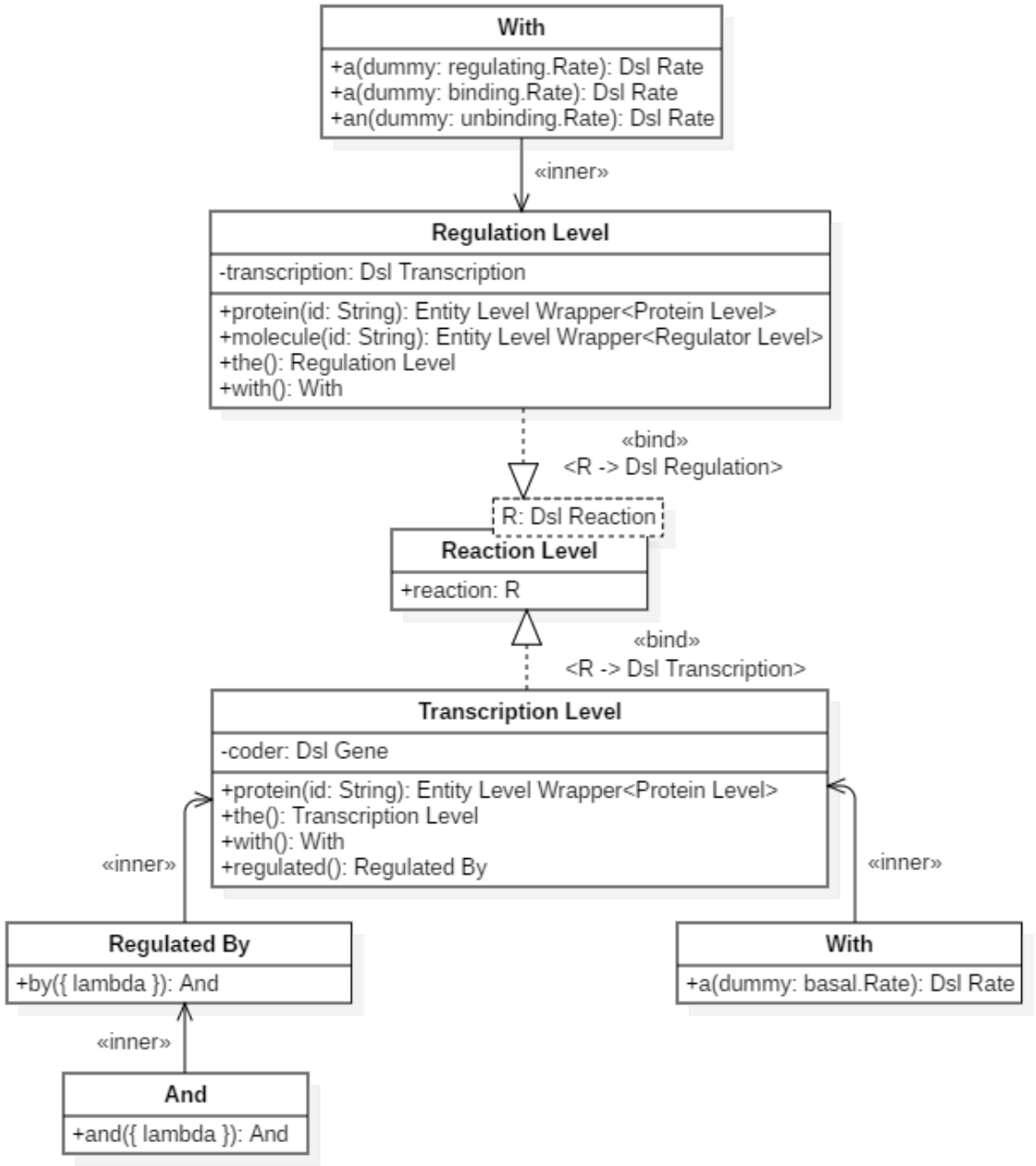


Figura 2.17: Schemi delle classi *Transcription Level* e *Regulation Level*.

Livello di Reazioni Personalizzate

Dato che nel modello è presente la possibilità di inserire reazioni esterne alle classiche *trascrizione*, *traduzione* e *regolazione*, tale possibilità viene estesa anche al linguaggio.

Si consideri pertanto il seguente blocco di codice:

```
Create circuit "Custom" containing {
  chemical reactions {
    "2X + Y" to "3X" having reaction.rate of 1
  }
}
```

Tale livello non è troppo dissimile dai precedenti, ma fornisce un semplice metodo per scrivere una reazione chimica indicandone reagenti, prodotti e, opzionalmente, il rate.

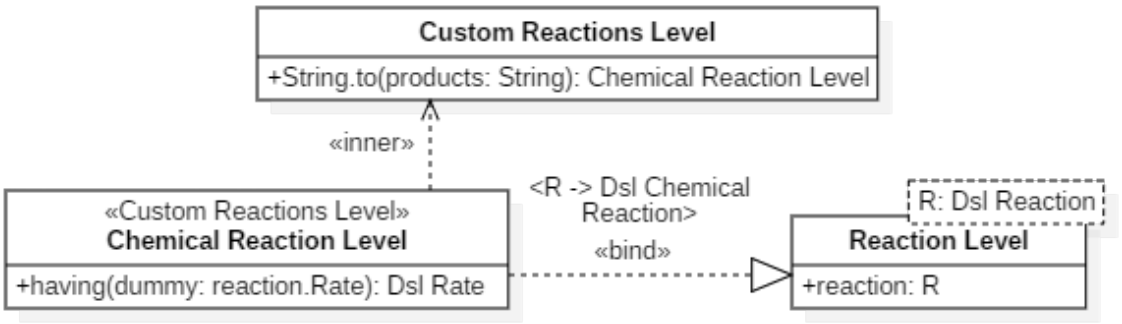


Figura 2.18: Schemi delle classi *Custom Reactions Level* e *Chemical Reaction Level*.

Livello Principale

Si consideri il seguente blocco di codice:

```
Create circuit "My Circuit" containing { // Containing Level
  gene("gen")
} with { // Defaults Level
  a default initial.concentration of 100
  a default degradation.rate of 1
} then export to AGCT
```

I livelli principali – *Containing Level* e *Defaults Level* – e il loro wrapper fanno da contorno all'effettivo circuito, permettendone la descrizione e il settaggio dei parametri attraverso delle *lambda* che prendono in ingresso i livelli stessi e l'esportazione attraverso la keyword *export* e la funzione *to*, alla quale andranno passati come parametro uno o

più istanze della classe *Generator*, fornite di un metodo che, preso un circuito genetico, lo esporta con le modalità indicate.

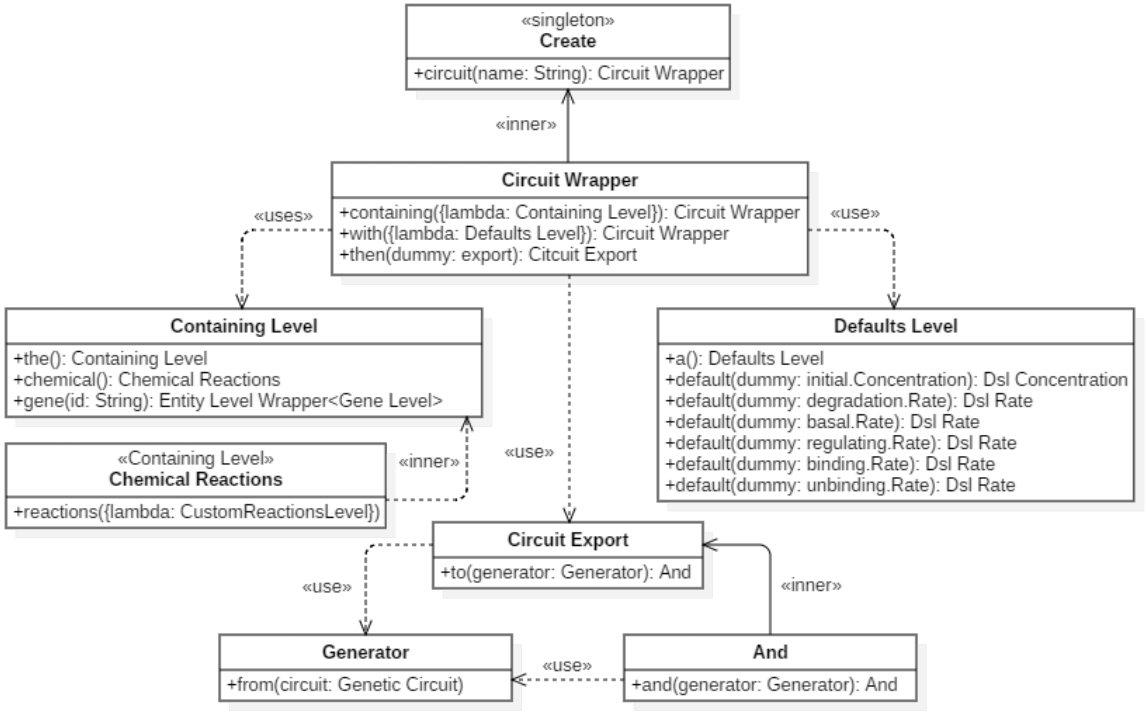


Figura 2.19: Schemi delle classi top level.

2.4 Organizzazione del Progetto Software

Come ultimo argomento, in questa sezione si descriveranno le caratteristiche architetturali dell'intero applicativo, mostrando come questo sia organizzato al suo interno e spiegando brevemente su quali tool e framework esterni esso si appoggia al fine di facilitarne sia lo sviluppo che la manutenzione.

2.4.1 Divisione in Packages

La volontà di tenere divise le classi di modello da quelle di linguaggio è già stata esposta durante l'enumerazione dei requisiti non funzionali nella sezione 2.1.2. Nel momento

dell'organizzazione architetturale, perciò, si è deciso di enfatizzare ulteriormente tale richiesta separando le due componenti in packages differenti, chiamati `model` e `dsl`.

Il package `model` contiene pertanto tutte e sole le classi e le interfacce descritte nella sezione 2.2 — quella relativa al proprio design. In particolare, al fine di celare agli utilizzatori del software il contenuto del modello, si è deciso tenere a visibilità *pubblica* soltanto le interfacce – indispensabili per garantire la possibilità di estendere il codice in maniera corretta e completa –, mentre tutte a le implementazioni è stato imposto il livello di visibilità *interna*. Un utente esterno, quindi, potrà sì utilizzare le direttive del modello anche all'interno del DSL – cosa che viola parzialmente i requisiti –, ma soltanto per operazioni di lettura, evitando così che vi siano delle interferenze esterne in grado di compromettere la coerenza dei dati.

Analogamente, anche il package *dsl* garantisce il corretto incapsulamento dei dati, impostando a visibilità interna, se non l'intera classe, almeno l'insieme delle proprietà e delle funzioni che l'utente non deve poter invocare. Va inoltre fatto notare che si è fatto ricorso a un piccolo workaround per semplificare l'utilizzo esterno del software: nonostante, all'interno del package *dsl*, si sia effettuata un'ulteriore divisione fisica dei file in una serie distinta di directory, si è sempre utilizzata l'intestazione "`package agct`" all'interno di ognuno di essi affinché l'utente potesse includere tutte le componenti necessarie con una sola istruzione di import come mostrato qui di seguito:

```
1 import agct.* // istruzione unica di import
2
3 Create circuit "example" containing {
4   ...
5 }
```

Oltre ai due package appena indicati, infine, ne è stato aggiunto un terzo nominato `generation`, contenente, oltre alla classe *Generator* e ad alcune sue implementazioni standard, delle utils pubbliche per agevolare l'export del circuito.

2.4.2 Testing Automatizzato

Per garantire la costante validazione e il corretto funzionamento del software, sono stati creati dei *test case* atti alla verifica del comportamento di ognuna delle sue componenti o

di un insieme di esse. La stesura dei test è stata eseguita affidandosi a *KotlinTest*[10], un framework per lo sviluppo di test automatizzati in Kotlin basato sul più famoso *jUnit*[11] e, al termine, i file prodotti sono stati collocati all'interno di un apposito modulo chiamato **test** – separato dal modulo **src** contenente invece il codice sorgente – così da permetterne l'esecuzione automatica tramite i tool di sviluppo presentati nella sezione successiva.

2.4.3 Tool di Sviluppo

Al fine di facilitare il processo di sviluppo dell'intero progetto, sono stati utilizzati vari tool ormai divenuti popolari nell'ambito dell'ingegneria dei sistemi software e del loro collaudo.

In particolare, per la stesura del codice si è fatto ricorso a *Git*[12], il noto DVCS (*Distributed Version Control System*) creato da Linus Torvalds. Tale repository git è poi stato caricato online, tramite il servizio di hosting offerto da *Github*, al seguente link: `www.github.com/AGCT-DSL/AGCT`.

Infine, affinché il processo di importazione delle dipendenze e di testing fosse automatizzato, ci si è affidati al noto sistema di automazione dello sviluppo e di compilazione automatica del codice *Gradle*[13] – già trattato nella sezione 1.2.2 per presentare il suo DSL Kotlin, con cui sono stati scritti anche i file di configurazione per AGCT –, e il servizio di hosting per la continuous integration *Travis CI*, grazie al quale è possibile vedere il risultato delle ultime build del progetto al link: `www.travis-ci.org/AGCT-DSL/AGCT`.

[10] Di fatto, *KotlinTest*, rappresenta un esempio di per Kotlin specifico per il dominio del testing di sistemi o componenti software. Ulteriori informazioni a riguardo possono essere trovate nella guida presente su: `https://github.com/kotlintest/kotlintest`.

[11] `https://junit.org/`.

[12] `https://git-scm.com/`.

[13] `https://gradle.org/`.

Capitolo 3

Utilizzo di AGCT

Dopo aver spiegato il processo di sviluppo e la struttura di AGCT, in questo ultimo capitolo si presenta una piccola guida al suo utilizzo sia in qualità di *utenti* che in qualità *programmatori*[1]. In particolare, si farà una breve panoramica generale della sintassi – in maniera leggermente più dettagliata di quanto compiuto nella sezione 2.3 –, si mostrerà come sia possibile estendere AGCT creando un nuovo generatore e, infine, si esibiranno degli esempi di reti geniche descritte con AGCT, compresi dei relativi dati ottenuti dall'export in Alchemist.

3.1 Guida alla Sintassi

Come più volte ribadito nel capitolo precedente, la sintassi di AGCT permette di descrivere una rete genica indicandone, attraverso dei livelli innestati, le entità che la compongono – geni, proteine e regolatori – e il valore assunto dai parametri di queste ultime e delle reazioni fra esse intercorrenti.

Prima di proseguire con gli esempi, tuttavia, va specificato che ognuno di essi sottostanti è da considerarsi non come file a se stante, ma come blocco di codice interno alla funzione main di un file con estensione *.kt* strutturato nel seguente modo:

```
1 // mycircuit.kt
2
```

[1] Il significato di questi due termini va inteso con il significato esposto nella sezione 2.1.2.

```
3  import agct.*
4
5  fun main() {
6    // codice AGCT
7  }
```

3.1.1 Descrizione delle Entità

In AGCT, le entità devono necessariamente essere descritte in maniera sequenziale, nel senso che un gene codifica una proteina, e la reazione di trascrizione generata può essere regolata da una molecola generica o, più specificatamente, da un'altra proteina — anche dalla stessa, in realtà. Questo genere di descrizione, quindi, va a formare una serie di livelli innestati, racchiusi dentro delle parentesi graffe, come nell'esempio mostrato qui sotto:

```
1  Create circuit "My Circuit" containing {
2    gene("g1") that codes {  // transcription info
3      protein("p1")
4      regulated by {                // regulation info
5        molecule("m1")
6      } and {                       // regulation info
7        molecule("m2")
8      } and {                       // regulation info
9        protein("p2")
10     }
11   } and {                         // transcription info
12     protein("p2")
13     regulated by {                // regulation info
14       molecule("m1")
15     } and {                       // regulation info
16       molecule("m2")
17     } and {                       // regulation info
18       protein("p1")
19     }
20   }
21
22   gene("g2") that codes {
```

```
23      // ...
24    }
25  }
```

Come si può notare, dopo l'istruzione iniziale '*Create circuit "My Circuit" containing*', si avvia il primo blocco vero e proprio, all'interno del quale sarà possibile definire solamente i geni. Le proteine e le altre molecole, infatti, potranno essere dichiarate solo successivamente: le seconde all'interno di un blocco di regolazione invocato attraverso l'istruzione *regulated by* mentre le prime, rispettivamente in qualità di target o di regolatori, sia nei blocchi di trascrizione che in quelli di regolazione.

L'istruzione *codes*

L'istruzione *codes* permette, dato un gene, di entrare nel relativo livello di trascrizione e indicare, fra le altre cose, il nome della proteina trascritta dal gene in questione. Questa istruzione può essere affiancata al gene stesso o, alternativamente, inserita all'interno del suo blocco — l'utilità di questo secondo caso rimarrà più chiara quando, nella sezione 3.1.2 si spiegherà come impostare il valore delle concentrazioni e dei rate.

```
1  Create circuit "Syntax One" containing {
2    gene("gen") that codes {
3      protein("pro")
4    }
5  }
6
7  Create circuit "Syntax Two" containing {
8    gene("gen") that {
9      codes {
10       protein("pro")
11     }
12   }
13 }
```

L'istruzione *and*

L'istruzione *and* permette di concatenare due o più blocchi simili che, se contenenti una sola riga di codice, possono anche essere posti uno di fianco all'altro. Nel caso in

cui tale sintassi dovesse risultare troppo confusionaria, inoltre, è anche possibile ripetere l'istruzione iniziale – *codes For* o *regulated by* – senza modificare il risultato ottenuto.

```
Create circuit "Syntax One" containing {
  gene("g1") that codes {
    protein("p1")
  } and {
    protein("p2")
  } and {
    protein("p3")
  }
}

Create circuit "Syntax Two" containing {
  gene("g1") that codes { protein("p1") } and { protein("p2") } and {
      protein("p3") }
}

Create circuit "Syntax Three" containing {
  gene("g1") that {
    codes { protein("p1") }
    codes { protein("p2") }
    codes { protein("p3") }
  }
}
```

L'istruzione *the*

L'istruzione *the* permette di creare o richiamare una certa entità appartenente alla rete genica esplicitandone la tipologia e il nome. In particolare, nel caso in cui essa non fosse già stata nominata[2], il sistema provvederà a crearla *ex novo* e aggiungerla alla rete; in caso contrario, invece, verrà restituita l'entità con lo stesso nome dopo aver verificato che sia della tipologia indicata — ovvero, se si vuole utilizzare la proteina *"gen"* ma nella rete quel nome identifica un gene, il software lancerà un'eccezione a tempo di esecuzione.

[2] Si raccomanda di evitare l'utilizzo di segni di interpunzione, simboli matematici o altri caratteri speciali per evitare che l'export verso quale sistema non in grado di supportarli possa produrre errori.

Alternativamente all'istruzione *the*, è possibile invocare un'entità con una sintassi più simile a quella dei linguaggi di programmazione standard, ovvero chiamando la funzione relativa alla sua tipologia e passando il nome come parametro fra parentesi tonde.

```
1  Create circuit "Syntax One" containing {
2    gene("gen") that codes For {
3      protein("pro")
4      regulated by { molecule("mol") }
5    }
6  }
7
8  Create circuit "Syntax Two" containing {
9    the gene "gen" that codes For {
10     the protein "pro"
11     regulated by { the molecule "mol" }
12   }
13 }
```

3.1.2 Assegnamento dei Parametri

Oltre alla mera descrizione di una rete genica, in AGCT è possibile impostare le concentrazioni iniziali delle entità e i rate delle reazioni che compongono quest'ultima in modo tale da poterne simulare il comportamento attraverso software esterni.

Parametri delle Entità

Oltre al proprio nome, un'entità è provvista di una *concentrazione iniziale* e, nel caso si tratti di una molecola degradante, di un *rate di degradazione*. In particolare:

- un gene non degrada, perciò non sarà possibile impostarne il rate di degradazione.
- una proteina degrada sempre, perciò sarà obbligatorio per essa avere un rate di degradazione che, se impostato, avrà il valore indicato, altrimenti assumerà un valore di default.
- una molecola generica può o può non degradare, per cui se il rate verrà impostato la molecola verrà considerata degradante, altrimenti no.

Data un'entità, il valore dei suoi parametri può essere impostato all'interno del proprio blocco o nel momento della creazione, o in qualunque punto in cui essa venga utilizzata oppure, come ultima alternativa, richiamandola per nome senza specificarne la tipologia. Va da sé che, in quest'ultimo caso, è necessario che l'entità sia stata precedente definita, altrimenti il programma produrrà un errore a tempo di esecuzione. Inoltre, mancando la tipologia, viene fornita la possibilità di impostare all'entità una caratteristica qualsiasi, non ottenendo quindi alcun richiamo durante la fase di compilazione se, ad esempio, si tentasse di assegnare a un gene un rate di degradazione. Nel momento dell'esecuzione però, ovvero appena il programma potrà rendersi conto dell'errore commesso, verrà lanciata un'eccezione, bloccando di conseguenza l'utilizzo del software.

Per evitare confusione, infine, è proibito assegnare valori multipli al medesimo parametro di un'entità — lo stesso vale anche per le reazioni. Utilizzando più volte un'istruzione come *initial.concentration*, sia nello stesso blocco che in blocchi diversi relativi alla stessa entità, si incorrerà infatti un errore a tempo di esecuzione.

```
1  Create circuit "Syntax One" containing {
2    gene("gen") that {
3      has an initial.concentration of 10
4      codes {
5        protein("pro") that {
6          has an initial.concentration of 20
7          has a degradation.rate of 50
8        }
9      }
10   }
11 }
12
13 Create circuit "Syntax Two" containing {
14   gene("gen") that codes { protein("pro") }
15
16   "gen" {
17     has an initial.concentration of 10
18   }
19
20   "pro" {
21     has an initial.concentration of 20
```

```
22      has a degradation.rate of 50
23    }
24  }
```

Parametri delle Reazioni

Diversamente dalle entità, a cui è possibile impostare il valore dei parametri in punti arbitrari, alle reazioni vanno necessariamente assegnati nel blocco in cui vengono create, essendo esse sprovviste di un nome e, quindi, impossibili da richiamare nuovamente all'interno della rete genica.

Tali parametri, in particolare, potranno essere il *basal rate* per le reazioni di trascrizione e il *regulating rate*, il *binding rate* e l'*unbinding rate* per le reazioni di regolazione.

```
1  Create circuit "Unique Syntax" containing {
2    gene("gen") that codes {
3      protein("pro")
4      with a basal.rate of 10
5      regulated by {
6        molecule("reg")
7        with a regulating.rate of 10
8        with a binding.rate of 20
9        with an unbinding.rate of 20
10     }
11   }
12 }
```

Gli Spazi di Variabili

Si è visto, nei blocchi di codice mostrati in precedenza, come sia possibile assegnare il valore di un parametro attraverso l'istruzione *of*; esiste però un'altra modalità per eseguire questa operazione, ovvero tramite l'istruzione *into*.

Mentre *of* sottintende l'univocità del valore impostato, *into* permette di definire *una serie di valori* entro cui è ammissibile che si trovi quello corretto, il quale, essendo però ignoto, andrà identificato a seguito della simulazione della rete.

A supporto di *into*, pertanto, entrano in gioco i cosiddetti *spazi di variabile*, ovvero:

values che denota un insieme di valori arbitrari scelti dall'utente.

range che denota un insieme di valori compresi fra un *inizio* e una *fine*, ognuno dei quali intervallato da un dato *step*.

linspace che denota un insieme di n valori compresi fra un *minimo* e un *massimo*, equispaziati su una scala lineare.

geomspace che denota un insieme di n valori compresi fra un *minimo* e un *massimo*, equispaziati su una scala logaritmica a base 10 — ovvero una progressione geometrica.

logspace che denota un insieme di n valori equispaziati su una scala logaritmica a base 10, dove però come parametri andranno inseriti, al posto del valore minimo e del valore massimo, il risultato dei relativi *logaritmi*.

```
1  Create circuit "Variables Spaces" containing {
2    gene("gen") that codes {
3      protein("pro")
4      regulated by { molecule("reg") }
5    }
6
7    "gen" {
8      // (1, 5, 10, 42)
9      has an initial.concentration into values(1, 5, 10, 42)
10   }
11
12   "pro" {
13     // (3, 6, 9)
14     has an initial.concentration into range(3, 10, 3)
15
16     // (3.0, 6.5, 10.0)
17     has a degradation.rate into linspace(3, 10, 3)
18   }
19
20   "reg" {
21     // (0, 10, 1000, 100000)
```

```
22      has an initial.concentration into geomspace(0.1, 100000, 4)
23
24      // (0.1, 10.0, 1000.0, 100000.0)
25      has a degradation.rate into logspace(-1, 5, 4)
26    }
27  }
```

Come si può notare, infine, potendo le concentrazioni assumere soltanto valori interi, nel caso in cui dagli spazi di variabili si dovessero ottenere dei numeri a virgola mobile, verrà effettuato un casting automatico.

3.1.3 Parametri di Default

Oltre a quelli relativi ai parametri delle singole entità o reazioni, per ogni rete è possibile impostare una serie di valori di default che verranno utilizzati nel caso di mancata dichiarazione degli altri. Tali *parametri di default* potranno essere quindi fissati, prima o dopo l'effettiva descrizione della rete, tramite l'istruzione *with*.

```
1  Create circuit "Syntax One" containing {
2    gene("gen") that codes {
3      protein("pro")
4      regulated by { molecule("reg") }
5    }
6  } with {
7    a default initial.concentration of 4
8    a default degradation.rate of 8
9    a default basal.rate of 15
10   a default regulating.rate of 16
11   a default binding.rate of 23
12   a default unbinding.rate of 42
13 }
14
15 Create circuit "Syntax Two" with {
16   a default initial.concentration of 4
17   a default degradation.rate of 8
18   a default basal.rate of 15
19   a default regulating.rate of 16
20   a default binding.rate of 23
```

```
21   a default unbinding.rate of 42
22 } containing {
23   gene("gen") that codes {
24     protein("pro")
25     regulated by { molecule("reg") }
26   }
27 }
28
29 Create circuit "Syntax Three" containing {
30   gene("gen") that {
31     has an initial.concentration of 4
32     codes {
33       protein("pro") that {
34         has an initial.concentration of 4
35         has a degradation.rate of 8
36       }
37       with a basal.rate of 15
38       regulated by {
39         molecule("reg") that {
40           has an initial.concentration of 4
41         }
42         with a regulating.rate of 16
43         with a binding.rate of 23
44         with an unbinding.rate of 42
45       }
46     }
47   }
48 }
```

3.1.4 Reazioni Personalizzate

Un'ulteriore possibilità offerta da AGCT è quella di inserire delle reazioni che, a differenza della *trascrizione* e della *regolazione*, abbiano dei comportamenti personalizzati. Ne è un esempio la reazione $2\,X + Y \longrightarrow 3\,X$ presente nel Brusselator, circuito genico già visto a pagina 18, non altrimenti modellabile.

```
1 Create circuit "Custom Reactions" containing {
```

```
2     gene("gen") that codes {
3       protein("pro")
4       regulated by { molecule("mol") }
5     }
6
7     chemical reactions {
8       "2 pro + sth" to "sthElse" having reaction.rate of 5
9     }
10  }
```

Poiché, come in questo caso, le *custom reactions* potrebbero coinvolgere entità già presenti nella, è consigliato invocare tale blocco soltanto al termine della sua descrizione. Così facendo, le entità già definite saranno del tipo indicato al momento della loro creazione, mentre alle altre sarà assegnato il tipo *Molecule*.

3.1.5 Export dei Dati

Come ultimo step prima di ottenere in output i dati della rete nei formati prediletti, è necessario invocare le direttive di esportazione attraverso le istruzioni *then export to* o *then export to each one into*.

```
1 Create circuit "Syntax One" containing {
2   // ...
3 } then export to entities and reactions and AGCT and Alchemist
4
5 Create circuit "Syntax Two" containing {
6   // ...
7 } then export to each one into (entities, reactions, AGCT, Alchemist)
```

Al momento, è possibile esportare la rete nei seguenti formati:

Entities ottenendo un file *.txt* contenente la lista delle entità.

Reactions ottenendo un file *.txt* contenente la lista delle reazioni.

AGCT ottenendo un file *.kt* eseguibile che descrive la rete attraverso una versione standardizzata della sintassi di AGCT.

Alchemist ottenendo un file *.yaml* da utilizzare per configurare il simulatore *Alchemist* in modo tale che possa eseguire la rete.

3.2 Creazione di un Generatore

La possibilità di esportare una rete genica descritta tramite AGCT è garantita dall'interfaccia *Generator*[3], contenente un unico metodo chiamato *from* il quale, preso un oggetto di tipo *Genetic Circuit*, lo mappa nel formato desiderato. A un *programmatore* desideroso di estendere il software per supportare l'export verso applicazioni terze, quindi, sarà sufficiente estendere l'interfaccia *Generator* e implementarne il relativo metodo[4] come nell'esempio mostrato di seguito:

```
1  object SingletonGenerator : Generator {
2    override fun from(circuit: GeneticCircuit) {
3      // ...
4    }
5  }
6
7  class ClassGenerator : Generator {
8    override fun from(circuit: GeneticCircuit) {
9      // ...
10   }
11 }
12
13 class ParametricGenerator(vararg parameters: Any) : Generator {
14   override fun from(circuit: GeneticCircuit) {
15     // ..
16   }
17 }
18
19 fun main() {
20   Create circuit "My Circuit" containing {
21     // ...
```

[3] Per capire come essa sia collegata collegata al resto del software si faccia riferimento alla figura 2.19.

[4] Si consiglia, inoltre, la creazione di una o più istanze del generatore stesso così da evitare la ridondante presenza di parentesi tonde all'interno del DSL.

```
22   } then export to SingletonGenerator and ClassGenerator() and
         ParametricGenerator("insert", "parameters", "here")
23 }
```

3.2.1 Esempi di Generatori Implementati

Al fine di agevolarne l'implementazione, oltre all'interfaccia *Generator* sono messe a disposizione delle utilities pubbliche quali la classe *Level*, che fornisce un'astrazione per esportare la rete verso formati testuali organizzati in livelli – come accade sia, banalmente, per AGCT, che per Alchemist, essendo la struttura di un file *.yaml* di natura gerarchica –, e la classe astratta *Abstract Generator*, contenente un campo *files* il quale, dato un circuito, restituisce una mappa di stringhe a stringhe di cui la chiave rappresenta il path del file mentre il valore il suo contenuto.

Di seguito vengono quindi mostrati alcuni esempi di generatori, dai più semplici ai più complessi, attraverso i quali è possibile comprendere il funzionamento delle classi di utilità sopra descritte.

Generatore per Entità e Reazioni

Da questo semplice esempio è facile comprendere la struttura della classe *Abstract Generator*, la quale ammette un costruttore che accetta un singolo parametro di tipo *GeneticCircuit.(MutableMap<String, String>) -> Unit* grazie al quale è possibile indicare il path e il contenuto dei file utilizzando la classica sintassi dell'accesso a mappa tramite parentesi quadre in Kotlin. Automaticamente, quindi, la funzione *from* è implementata affinché ogni file indicato nella mappa venga esportato con il relativo contenuto.

```
1  val GeneticCircuit.defaultDirectory
2    get() = "export/${name.toLowerCase()}"
3
4  object entities : EntitiesGenerator(
5    { "$defaultDirectory/entities.txt" }
6  )
7
8  open class EntitiesGenerator(
9    filename: GeneticCircuit.() -> String
10 ) : AbstractGenerator({ file ->
11   file[filename()] =
```

```
12      entities.joinToString("\n", "${name.toUpperCase()}\n\n")
13  })
14
15  object reactions : ReactionsGenerator(
16    { "$defaultDirectory/reactions.txt" }
17  )
18
19  open class ReactionsGenerator(filename: GeneticCircuit.() -> String) : AbstractGenerator
        ({ file ->
20    file[filename()] =
21      reactions.map { it.reactions }
22        .flatten()
23        .joinToString("\n", "${name.toUpperCase()}\n\n")
24  })
```

Da notare, inoltre, la duplice definizione sia delle classi *EntitiesGenerator* e *ReactionGenerator*, così da renderne possibile l'estensione anche grazie alla clausola *open*, sia degli oggetti *entities* e *reactions*, da utilizzare nel DSL per una sintassi più scorrevole.

Generatore per AGCT

Molto più complesso è invece il generatore per AGCT, il quale utilizza la classe di utilità *Level* – richiamata tramite il metodo *start* – per costruire una stringa che rappresenti l'intero circuito attraverso una sintassi il più possibile simile a quella del DSL stesso.

```
1   object AGCT : AGCTGenerator(
2     { "$defaultDirectory/agct.kt" }
3   )
4
5   open class AGCTGenerator(
6     filename: GeneticCircuit.() -> String
7   ) : AbstractGenerator({ file ->
8     context = this
9
10    file[filename()] =
11      start(prefix = " {", postfix = "}") { // Level
12        "import agct.*"()
13        line()
14        "fun main()" { // Level
15          "Create circuit ${name.string} containing" { // Level
16            genes.forEach { gene ->
17              gene.transcriptions(
18                line = "the ${gene.string} that codes",
19                innerLine = " and",
```

```
20              spacings = -1
21            ) { transcription ->
22              "the ${transcription.target.string}"()
23              "with a basal.rate ${transcription.basalRate.string}"()
24              transcription.regulations(
25                line = "regulated by",
26                innerLine = " and",
27                spacings = -1
28              ) { regulation ->
29                "the ${regulation.regulator.string}"()
30                "with a regulating.rate ${regulation.regulatingRate.string}"()
31                "with a binding.rate ${regulation.bindingRate.string}"()
32                "with an unbinding.rate ${regulation.unbindingRate.string}"()
33              }
34            }
35            line()
36          }
37          chemicalReactions { reaction ->
38            reaction.line()
39          }
40          line()
41          agctEntities(
42            line = { "\"${it.id}\"" },
43            spacings = 1
44          ) { entity ->
45            "has an initial.concentration ${entity.initialConcentration.string}"()
46            if (entity is DegradingEntity) {
47              "has a degradation.rate ${entity.degradation.degradationRate.string}"()
48            }
49          }
50        }
51      }
52    }.toString()
53
54  context = null
55 })
56
57 private var context: GeneticCircuit? = null
58
59 private val GeneticCircuit.genes
60   get() = entities.filterIsInstance<Gene>()
61     .filter { it !is RegulatedGene }
62
63 private val GeneticCircuit.agctEntities
64   get() = entities.filter { it !is BoundEntity<*, *> }
65
66 private val DegradingEntity.degradation
```

```
67    get() = context!!.reactionsOf(this)
68      .filterIsInstance<Degradation>()
69      .single()
70
71  private val Gene.transcriptions
72    get() = context!!.reactionsOf(this)
73      .filterIsInstance<Transcription<*>>()
74
75  private val Transcription<*>.regulations
76    get() = context!!.reactions
77      .filterIsInstance<Regulation>()
78      .filter { it.reaction == this }
79
80  private fun Level.chemicalReactions(
81    block: Level.(ChemicalReaction) -> Unit
82  ) = context!!.reactions
83    .filterIsInstance<ChemicalReaction>()
84    .let { reactions ->
85      if (reactions.isNotEmpty()) {
86        "chemical reactions" {
87          reactions.forEach {
88            block(it)
89          }
90        }
91      }
92    }
93
94  private val ChemicalReaction.line
95    get() = reactions.single()
96      .toString()
97      .substringAfter(":")
98      .substringBefore(",")
99      .replace("[", "\"")
100     .replace("]", "\"")
101     .replace("-->", "to")
102     .trim() + " with rate ${rate.string}"
103
104 private val Entity.string
105   get() = when (this) {
106     is Gene -> "gene"
107     is Protein -> "protein"
108     is RegulatingEntity -> "molecule"
109     else -> throw UnsupportedClassException(this)
110   } + " ${id.string}"
111
112 private val Variable<*>.string
113   get() = if (values.size == 1) "of ${values.single()}"
```

```
114       else "into values${values.joinToString(", ", "(", ")")}"
```

La classe *Level*, di fatto, funziona come uno *StringBuilder* in cui è possibile aumentare il livello di indentazione delle linee e creare dei blocchi a partire da delle *Collections* di oggetti.

Generatore per Alchemist

Vista la complessità dell'operazione, anche nel generatore per il simulatore *Alchemist* si è fatto ricorso alla classe *Level*.

```
1  object Alchemist : AlchemistGenerator(
2    { "$defaultDirectory/alchemist.yml" }
3  )
4
5  open class AlchemistGenerator(
6    filename: GeneticCircuit.() -> String
7  ) : AbstractGenerator({ file ->
8    val variables = mutableListOf<Variable<*>>(
9      *dslConcentrations.toTypedArray(),
10     *dslRates.toTypedArray()
11   )
12
13   file[filename()] =
14     start(
15       prefix = null,
16       postfix = null,
17       indentation = "  ",
18       stringSeparator = ": "
19     ) { // Level
20       "incarnation"("biochemistry")
21       line()
22       "variables: " { // Level
23         variables({"${it.id}: &${it.id}"}) { // Level
24           "type"("ArbitraryVariable")
25           "parameters"(
26             "[${it.values.first()}, ${it.values.joinToString(separator = ", ")}]"
27           )
28         }
29       }
30       line()
31       "seeds:" { // Level
32         "scenario"(0)
33         "simulation"(1)
34       }
35       line()
```

```
36      "displacements:" { // Level
37        "- in:" { // Level
38          "  type"("Point")
39          "  parameters"("[0, 0]")
40        }
41        "" { // Level
42          "contents:" { // Level
43            dslConcentrations(line = { "- molecule: ${it.info.molecule}" }) { // Level
44              "concentration"("*${it.id}")
45            }
46          }
47          line()
48          "programs:" { // Level
49            "- " { // Level
50              dslRates(line = { "- time-distribution: *${it.id}" }) { // Level
51                "program"(it.info.program)
52              }
53            }
54          }
55        }
56      }
57    }.toString()
58  })
59
60  private data class Variable<I : Any>(
61    val id: String,
62    val values: Collection<*>,
63    val info: I
64  )
65
66  private val GeneticCircuit.dslConcentrations
67    get() = entities.filter { it !is BoundEntity<*, *> }
68      .mapIndexed { index, entity ->
69        Variable(
70          "conc$index",
71          entity.initialConcentration.values,
72          object {
73            val molecule = entity.id
74          }
75        )
76      }
77
78  private val GeneticCircuit.dslRates
79    get() = reactions.flatMap { it.reactions }
80      .mapIndexed { index, reaction ->
81        Variable(
82          "rate$index",
```

```
83          reaction.rate.values,
84          object {
85            val program = buildString {
86              append("\"")
87              append(reaction.reagents.string)
88              append(" --> ")
89              append(reaction.products.string)
90              append("\"")
91            }
92          }
93        )
94      }
95
96  private val Map<Entity, Int>.string
97    get() = entries.joinToString(" + ", "[", "]") {
98      if (it.value != 1)
99        "${it.value}${it.key.id}"
100     else
101       it.key.id
102   }
```

Inoltre, al fine di automatizzare il plotting dei dati successivamente alla simulazione in *Alchemist*, in un modulo secondario di AGCT si è creato anche un ulteriore generatore, chiamato *ExportableAlchemist*, il quale permette di appendere dell'ulteriore testo al file *.yaml* esportato.

```
1   class ExportableAlchemist(
2     directoryPath: GeneticCircuit.() -> String =
3       { "$defaultDirectory/alchemist.yml" },
4     exportBlock: StringBuilder.(GeneticCircuit) -> Unit =
5       { }
6   ) : AlchemistGenerator(
7     directoryPath
8   ) {
9     private val block: GeneticCircuit.() -> String =
10      { StringBuilder().also { it.exportBlock(this) }.toString() }
11
12    override val files: GeneticCircuit.() -> Map<String, String> = {
13      super.files(this).mapValues { (_, data) ->
14        buildString {
15          append(data)
16          append("\n\n")
17          append("export:")
18          append("\n  ")
19          append(block().replace("\n", "\n  "))
20        }
21      }
```

```
22   }
23 }
24
25 fun StringBuilder.line(line: String): StringBuilder =
26   append("$line\n")
```

3.3 Esempi e Risultati

Come ultimo atto prima di chiudere il capitolo, verranno ora esposte le descrizioni, fatte tramite AGCT, degli esempi presentati nella sezione 1.1.5, e i relativi file *.yaml* di configurazione per il simulatore *Alchemist*.

3.3.1 Repressilator

```
1  import agct.*
2
3  fun main() {
4    Create circuit "Repressilator" containing {
5      gene("gTetR") that codes {
6        protein("TetR")
7        regulated by { protein("LacI") }
8      }
9
10     gene("gLacI") that codes {
11       protein("LacI")
12       regulated by { protein("AcI") }
13     }
14
15     gene("gAcI") that codes {
16       protein("AcI")
17       regulated by { protein("TetR") }
18     }
19   } with {
20     a default initial.concentration of 1
21     a default degradation.rate of 0.1
22     a default basal.rate of 0
23     a default regulating.rate of 10
```

```
    a default binding.rate of 0.01
    a default unbinding.rate of 0.01
  } then export to Alchemist
}
```

Listing 3.1: Repressilator modellato in AGCT

```
incarnation: biochemistry

variables:
  conc0: &conc0
  type: ArbitraryVariable
  parameters: [1, 1]
  conc1: &conc1
  type: ArbitraryVariable
  parameters: [1, 1]
  conc2: &conc2
  type: ArbitraryVariable
  parameters: [1, 1]
  conc3: &conc3
  type: ArbitraryVariable
  parameters: [1, 1]
  conc4: &conc4
  type: ArbitraryVariable
  parameters: [1, 1]
  conc5: &conc5
  type: ArbitraryVariable
  parameters: [1, 1]
  rate0: &rate0
  type: ArbitraryVariable
  parameters: [0.0, 0.0]
  rate1: &rate1
  type: ArbitraryVariable
  parameters: [0.01, 0.01]
  rate2: &rate2
  type: ArbitraryVariable
  parameters: [0.01, 0.01]
  rate3: &rate3
  type: ArbitraryVariable
```

```
33    parameters: [10.0, 10.0]
34    rate4: &rate4
35    type: ArbitraryVariable
36    parameters: [0.1, 0.1]
37    rate5: &rate5
38    type: ArbitraryVariable
39    parameters: [0.01, 0.01]
40    rate6: &rate6
41    type: ArbitraryVariable
42    parameters: [0.01, 0.01]
43    rate7: &rate7
44    type: ArbitraryVariable
45    parameters: [10.0, 10.0]
46    rate8: &rate8
47    type: ArbitraryVariable
48    parameters: [0.1, 0.1]
49    rate9: &rate9
50    type: ArbitraryVariable
51    parameters: [0.0, 0.0]
52    rate10: &rate10
53    type: ArbitraryVariable
54    parameters: [0.01, 0.01]
55    rate11: &rate11
56    type: ArbitraryVariable
57    parameters: [0.01, 0.01]
58    rate12: &rate12
59    type: ArbitraryVariable
60    parameters: [10.0, 10.0]
61    rate13: &rate13
62    type: ArbitraryVariable
63    parameters: [0.1, 0.1]
64    rate14: &rate14
65    type: ArbitraryVariable
66    parameters: [0.0, 0.0]
67
68  seeds:
69    scenario: 0
70    simulation: 1
```

```
71
72 displacements:
73   - in:
74     type: Point
75     parameters: [0, 0]
76
77   contents:
78     - molecule: gTetR
79     concentration: *conc0
80     - molecule: TetR
81     concentration: *conc1
82     - molecule: LacI
83     concentration: *conc2
84     - molecule: gLacI
85     concentration: *conc3
86     - molecule: AcI
87     concentration: *conc4
88     - molecule: gAcI
89     concentration: *conc5
90
91   programs:
92     -
93     - time-distribution: *rate0
94       program: "[gTetR] --> [gTetR + TetR]"
95     - time-distribution: *rate1
96       program: "[gTetR + LacI] --> [gTetR_LacI]"
97     - time-distribution: *rate2
98       program: "[gTetR_LacI] --> [gTetR + LacI]"
99     - time-distribution: *rate3
100      program: "[gTetR_LacI] --> [gTetR_LacI + TetR]"
101    - time-distribution: *rate4
102      program: "[TetR] --> []"
103    - time-distribution: *rate5
104      program: "[gAcI + TetR] --> [gAcI_TetR]"
105    - time-distribution: *rate6
106      program: "[gAcI_TetR] --> [gAcI + TetR]"
107    - time-distribution: *rate7
108      program: "[gAcI_TetR] --> [gAcI_TetR + AcI]"
```

```
109     - time-distribution: *rate8
110       program: "[LacI] --> []"
111     - time-distribution: *rate9
112       program: "[gLacI] --> [gLacI + LacI]"
113     - time-distribution: *rate10
114       program: "[gLacI + AcI] --> [gLacI_AcI]"
115     - time-distribution: *rate11
116       program: "[gLacI_AcI] --> [gLacI + AcI]"
117     - time-distribution: *rate12
118       program: "[gLacI_AcI] --> [gLacI_AcI + LacI]"
119     - time-distribution: *rate13
120       program: "[AcI] --> []"
121     - time-distribution: *rate14
122       program: "[gAcI] --> [gAcI + AcI]"
```

Listing 3.2: File di configurazione del Repressilator descritto nella listing 3.1

3.3.2 Brusselator

```
1  import agct.*
2
3  fun main() {
4    Create circuit "Brusselator" containing {
5      gene("A") that codes { protein("X") }
6
7      chemical reactions {
8        "2X + Y" to "3X"
9        "B + X" to "B + Y"
10     }
11
12     "A" {
13       has an initial.concentration of 10
14     }
15
16     "B" {
17       has an initial.concentration of 200
18     }
19   } with {
20     a default initial.concentration of 0
```

```
    a default degradation.rate of 1
    a default basal.rate of 1
    a default reaction.rate of 1
  } then export to Alchemist
}
```

Listing 3.3: Brusselator modellato in AGCT

```
incarnation: biochemistry

variables:
  conc0: &conc0
  type: ArbitraryVariable
  parameters: [10, 10]
  conc1: &conc1
  type: ArbitraryVariable
  parameters: [0, 0]
  conc2: &conc2
  type: ArbitraryVariable
  parameters: [0, 0]
  conc3: &conc3
  type: ArbitraryVariable
  parameters: [200, 200]
  rate0: &rate0
  type: ArbitraryVariable
  parameters: [1.0, 1.0]
  rate1: &rate1
  type: ArbitraryVariable
  parameters: [1.0, 1.0]
  rate2: &rate2
  type: ArbitraryVariable
  parameters: [1.0, 1.0]
  rate3: &rate3
  type: ArbitraryVariable
  parameters: [1.0, 1.0]

seeds:
  scenario: 0
  simulation: 1
```

```
32
33 displacements:
34   - in:
35     type: Point
36     parameters: [0, 0]
37
38   contents:
39     - molecule: A
40     concentration: *conc0
41     - molecule: X
42     concentration: *conc1
43     - molecule: Y
44     concentration: *conc2
45     - molecule: B
46     concentration: *conc3
47
48   programs:
49     -
50     - time-distribution: *rate0
51       program: "[A] --> [A + X]"
52     - time-distribution: *rate1
53       program: "[X] --> []"
54     - time-distribution: *rate2
55       program: "[2X + Y] --> [3X]"
56     - time-distribution: *rate3
57       program: "[B + X] --> [B + Y]"
```

Listing 3.4: File di configurazione del Brusselator descritto nella listing 3.3

3.3.3 Genetic Toggle Switch

```
1 import agct.*
2
3 fun main() {
4   Create circuit "Genetic Toggle Switch" containing {
5     gene("promoter1") that codes {
6       protein("repressor2")
7       regulated by {
8         protein("repressor1")
```

```
9         with a regulating.rate of 0.1
10      }
11    }
12
13    gene("promoter2") that codes {
14      protein("repressor1")
15      regulated by {
16        protein("repressor2")
17        with a regulating.rate of 0.1
18      }
19    }
20
21    chemical reactions {
22      "repressor1 + inducer1" to "repressor1_inducer1"
23      "repressor1_inducer1" to "repressor1 + inducer1"
24      "repressor2 + inducer2" to "repressor2_inducer2"
25      "repressor2_inducer2" to "repressor2 + inducer2"
26    }
27
28    "promoter1" {
29      has an initial.concentration of 1
30    }
31
32    "promoter2" {
33      has an initial.concentration of 1
34    }
35  } with {
36    a default initial.concentration of 0
37    a default degradation.rate of 1
38    a default basal.rate of 1
39    a default binding.rate of 1
40    a default unbinding.rate of 1
41    a default reaction.rate of 1
42  } then export to reactions
43 }
```

Listing 3.5: Genetic Toggle Switch modellato in AGCT

```
1 incarnation: biochemistry
```

```
2
3  variables:
4    conc0: &conc0
5    type: ArbitraryVariable
6    parameters: [1, 1]
7    conc1: &conc1
8    type: ArbitraryVariable
9    parameters: [0, 0]
10   conc2: &conc2
11   type: ArbitraryVariable
12   parameters: [0, 0]
13   conc3: &conc3
14   type: ArbitraryVariable
15   parameters: [1, 1]
16   conc4: &conc4
17   type: ArbitraryVariable
18   parameters: [0, 0]
19   conc5: &conc5
20   type: ArbitraryVariable
21   parameters: [0, 0]
22   conc6: &conc6
23   type: ArbitraryVariable
24   parameters: [0, 0]
25   conc7: &conc7
26   type: ArbitraryVariable
27   parameters: [0, 0]
28   rate0: &rate0
29   type: ArbitraryVariable
30   parameters: [1.0, 1.0]
31   rate1: &rate1
32   type: ArbitraryVariable
33   parameters: [1.0, 1.0]
34   rate2: &rate2
35   type: ArbitraryVariable
36   parameters: [1.0, 1.0]
37   rate3: &rate3
38   type: ArbitraryVariable
39   parameters: [0.1, 0.1]
```

```
40    rate4: &rate4
41    type: ArbitraryVariable
42    parameters: [1.0, 1.0]
43    rate5: &rate5
44    type: ArbitraryVariable
45    parameters: [1.0, 1.0]
46    rate6: &rate6
47    type: ArbitraryVariable
48    parameters: [1.0, 1.0]
49    rate7: &rate7
50    type: ArbitraryVariable
51    parameters: [0.1, 0.1]
52    rate8: &rate8
53    type: ArbitraryVariable
54    parameters: [1.0, 1.0]
55    rate9: &rate9
56    type: ArbitraryVariable
57    parameters: [1.0, 1.0]
58    rate10: &rate10
59    type: ArbitraryVariable
60    parameters: [1.0, 1.0]
61    rate11: &rate11
62    type: ArbitraryVariable
63    parameters: [1.0, 1.0]
64    rate12: &rate12
65    type: ArbitraryVariable
66    parameters: [1.0, 1.0]
67    rate13: &rate13
68    type: ArbitraryVariable
69    parameters: [1.0, 1.0]
70
71  seeds:
72    scenario: 0
73    simulation: 1
74
75  displacements:
76    - in:
77      type: Point
```

```
78      parameters: [0, 0]
79
80    contents:
81      - molecule: promoter1
82        concentration: *conc0
83      - molecule: repressor2
84        concentration: *conc1
85      - molecule: repressor1
86        concentration: *conc2
87      - molecule: promoter2
88        concentration: *conc3
89      - molecule: inducer1
90        concentration: *conc4
91      - molecule: repressor1_inducer1
92        concentration: *conc5
93      - molecule: inducer2
94        concentration: *conc6
95      - molecule: repressor2_inducer2
96        concentration: *conc7
97
98    programs:
99      -
100     - time-distribution: *rate0
101       program: "[promoter1] --> [promoter1 + repressor2]"
102     - time-distribution: *rate1
103       program: "[promoter1 + repressor1] --> [promoter1_repressor1]"
104     - time-distribution: *rate2
105       program: "[promoter1_repressor1] --> [promoter1 + repressor1]"
106     - time-distribution: *rate3
107       program: "[promoter1_repressor1] --> [promoter1_repressor1 +
            repressor2]"
108     - time-distribution: *rate4
109       program: "[repressor2] --> []"
110     - time-distribution: *rate5
111       program: "[promoter2 + repressor2] --> [promoter2_repressor2]"
112     - time-distribution: *rate6
113       program: "[promoter2_repressor2] --> [promoter2 + repressor2]"
114     - time-distribution: *rate7
```

```
115       program: "[promoter2_repressor2] --> [promoter2_repressor2 +
              repressor1]"
116     - time-distribution: *rate8
117       program: "[repressor2 + inducer2] --> [repressor2_inducer2]"
118     - time-distribution: *rate9
119       program: "[repressor2_inducer2] --> [repressor2 + inducer2]"
120     - time-distribution: *rate10
121       program: "[repressor1] --> []"
122     - time-distribution: *rate11
123       program: "[promoter2] --> [promoter2 + repressor1]"
124     - time-distribution: *rate12
125       program: "[repressor1 + inducer1] --> [repressor1_inducer1]"
126     - time-distribution: *rate13
127       program: "[repressor1_inducer1] --> [repressor1 + inducer1]"
```

Listing 3.6: File di configurazione del Genetic Toggle Switch descritto nella listing 3.6

Capitolo 4

Conclusioni

In questa trattazione si è presentato *Another Genetic Circuit Transcriber* (*AGCT*), un linguaggio dominio-specifico per la costruzione di reti geniche sviluppato in Kotlin.

Dopo una breve introduzione al contesto, nel quale sono stati riportati i principali concetti relativi sia all'ingegneria genetica che ai linguaggi dominio-specifici, si è quindi mostrato il percorso svolto durante lo sviluppo del software – dalla fase di analisi fino a quella di ispezione e testing, passando inevitabilmente attraverso quella di design – per fornire infine, contestualmente al risultato ottenuto dalla loro esportazione, alcuni esempi di descrizione di sistemi biologi.

Complessivamente, pertanto, si è avuto modo di valutare le potenzialità di *AGCT*, le quali variano dalla pura formalizzazione delle reti – siano esse modelli semplici come il *Repressilator*, il *Brusselator* o il *Genetic Toggle Switch* oppure sistemi complessi come quelli che descrivono il ritmo circadiano o il ciclo cellulare – alla stima *in-silico* dei rate delle reazioni in seguito all'esecuzione seriale di simulazioni con differenti set di parametri.

4.1 Sviluppi Futuri

Pur garantendo un sufficiente insieme di funzionalità, questa prima versione di *AGCT* presenta ancora delle lacune che ne limitano il potenziale.

Come prima cosa, infatti, si ricorda che la codifica in due step – ovvero tramite trascrizione e traduzione – è presente soltanto all'interno del modello, mentre il DSL

obbliga il passaggio diretto da gene a proteina. Per quanto vincolata a una parziale ristrutturazione della sintassi che, allo stesso tempo, possa auspicabilmente garantire una piena retrocompatibilità con quella odierna – sarebbe ideale, infatti, mantenere la possibilità di trascrizione diretta anche una volta introdotto lo step intermedio, o se non altro mappare internamente su due diverse reazioni quello che, da DSL, per semplicità, viene espresso soltanto con una –, è bene che questa feature venga aggiunta quanto prima, così da garantire una miglior aderenza del linguaggio alla realtà.

Successivamente, sarebbe opportuno supportare anche la descrizione di sistemi multicellulari e meccanismi di comunicazione fra cellule, i quali permetteranno di effettuare studi multiscala – ovvero su singola cellula e popolazione – di circuiti complessi.

Va inoltre rivista e maggiormente controllata la possibilità di utilizzare le keywords del linguaggio in modi arbitrari. Al momento, infatti, è possibile scrivere, senza che questo generi alcun errore, delle istruzioni come *the.the.the gene "g" that*, oppure invocare l'istruzione *codes* dentro al blocco di una proteina prendendo come riferimento il gene che la codifica, tutte sottigliezze che, però, sembrano in qualche modo inevitabili poiché figlie delle politiche di Kotlin relative allo scope delle proprietà e delle funzioni. Prima di gettarsi in ulteriori sviluppi del codice esistente, quindi, bisognerebbe valutare se e quanto l'idea di mantenere *AGCT* come DSL interno sia vantaggiosa o meno.

In conclusione, andrebbe aumentato il supporto per l'esportazione verso software di terze parti – i sopra citati SBOL e GCDL, ad esempio, sono i primi candidati di una lista che, tuttavia, sembra essere in costante aumento –, e allo stesso tempo aggiungere delle classi utili all'import dei dati, così da rendere *AGCT* non un semplice DSL ma un vero e proprio linguaggio "internazionale" che riesca a interpretare e tradurre tutti i vari dialetti utilizzati per la descrizione di reti geniche.

Bibliografia

[1] O. T. Avery. «Studies on the chemical nature of the substance inducing transformation of pneumococcal types. Inductions of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type III». In: *Journal of Experimental Medicine* 149.2 (feb. 1979), pp. 297–326. DOI: 10.1084/jem.149.2.297. URL: https://doi.org/10.1084/jem.149.2.297.

[2] A. D. Hershey. «INDEPENDENT FUNCTIONS OF VIRAL PROTEIN AND NUCLEIC ACID IN GROWTH OF BACTERIOPHAGE». In: *The Journal of General Physiology* 36.1 (set. 1952), pp. 39–56. DOI: 10.1085/jgp.36.1.39. URL: https://doi.org/10.1085/jgp.36.1.39.

[3] J. D. Watson. «MOLECULAR STRUCTURE OF NUCLEIC ACIDS». In: *JAMA* 269.15 (apr. 1993), p. 1966. DOI: 10.1001/jama.1993.03500150078030. URL: https://doi.org/10.1001/jama.1993.03500150078030.

[4] Gregor Mendel e William Bateson. *Experiments in plant-hybridisation / By Gregor Mendel.* Harvard University Press, 1925. DOI: 10.5962/bhl.title.4532. URL: https://doi.org/10.5962/bhl.title.4532.

[5] F. S. Collins. «The Human Genome Project: Lessons from Large-Scale Biology». In: *Science* 300.5617 (apr. 2003), pp. 286–290. DOI: 10.1126/science.1084564. URL: https://doi.org/10.1126/science.1084564.

[6] Gabriele Lillacci e Mustafa Khammash. «Parameter Estimation and Model Selection in Computational Biology». In: *PLoS Computational Biology* 6.3 (mar. 2010). A cura di Anand R. Asthagiri, e1000696. DOI: 10.1371/journal.pcbi.1000696. URL: https://doi.org/10.1371/journal.pcbi.1000696.

[7] D Pianini, S Montagna e M Viroli. «Chemical-oriented simulation of computational systems with ALCHEMIST». In: *Journal of Simulation* 7.3 (ago. 2013), pp. 202–215. DOI: 10.1057/jos.2012.27. URL: https://doi.org/10.1057/jos.2012.27.

[8] Jennifer A N Brophy e Christopher A Voigt. «Principles of genetic circuit design». In: *Nature Methods* 11.5 (apr. 2014), pp. 508–520. DOI: 10.1038/nmeth.2926. URL: https://doi.org/10.1038/nmeth.2926.

[9] Francis Crick. «Central Dogma of Molecular Biology». In: *Nature* 227.5258 (ago. 1970), pp. 561–563. DOI: 10.1038/227561a0. URL: https://doi.org/10.1038/227561a0.

[10] Sumit G. Gandhi. «Synthetic Biology for Production of Commercially Important Natural Product Small Molecules». In: *Current Developments in Biotechnology and Bioengineering*. Elsevier, 2019, pp. 189–205. DOI: 10.1016/b978-0-444-64085-7.00008-3. URL: https://doi.org/10.1016/b978-0-444-64085-7.00008-3.

[11] D. V. Goeddel et al. «Expression in Escherichia coli of chemically synthesized genes for human insulin.» In: *Proceedings of the National Academy of Sciences* 76.1 (gen. 1979), pp. 106–110. DOI: 10.1073/pnas.76.1.106. URL: https://doi.org/10.1073/pnas.76.1.106.

[12] Miller Tran et al. «Production of anti-cancer immunotoxins in algae: Ribosome inactivating proteins as fusion partners». In: *Biotechnology and Bioengineering* 110.11 (giu. 2013), pp. 2826–2835. DOI: 10.1002/bit.24966. URL: https://doi.org/10.1002/bit.24966.

[13] Youyou Tu. «The discovery of artemisinin (qinghaosu) and gifts from Chinese medicine». In: *Nature Medicine* 17.10 (ott. 2011), pp. 1217–1220. DOI: 10.1038/nm.2471. URL: https://doi.org/10.1038/nm.2471.

[14] Dae-Kyun Ro et al. «Production of the antimalarial drug precursor artemisinic acid in engineered yeast». In: *Nature* 440.7086 (apr. 2006), pp. 940–943. DOI: 10.1038/nature04640. URL: https://doi.org/10.1038/nature04640.

[15] Michal Galdzicki et al. «The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology». In: *Nature Biotechnology* 32.6 (giu. 2014), pp. 545–550. DOI: 10.1038/nbt.2891. URL: https://doi.org/10.1038/nbt.2891.

[16] William Waites et al. «A Genetic Circuit Compiler: Generating Combinatorial Genetic Circuits with Web Semantics and Inference». In: *ACS Synthetic Biology* 7.12 (nov. 2018), pp. 2812–2823. DOI: 10.1021/acssynbio.8b00201. URL: https://doi.org/10.1021/acssynbio.8b00201.

[17] Michael B. Elowitz e Stanislas Leibler. «A synthetic oscillatory network of transcriptional regulators». In: *Nature* 403.6767 (gen. 2000), pp. 335–338. DOI: 10.1038/35002125. URL: https://doi.org/10.1038/35002125.

[18] Polly Yingshan Hsu e Stacey L. Harmer. «Wheels within wheels: the plant circadian system». In: *Trends in Plant Science* 19.4 (apr. 2014), pp. 240–249. DOI: 10.1016/j.tplants.2013.11.007. URL: https://doi.org/10.1016/j.tplants.2013.11.007.

[19] I. Prigogine. «Time, Structure, and Fluctuations». In: *Science* 201.4358 (set. 1978), pp. 777–785. DOI: 10.1126/science.201.4358.777. URL: https://doi.org/10.1126/science.201.4358.777.

[20] Timothy S. Gardner, Charles R. Cantor e James J. Collins. «Construction of a genetic toggle switch in Escherichia coli». In: *Nature* 403.6767 (gen. 2000), pp. 339–342. DOI: 10.1038/35002131. URL: https://doi.org/10.1038/35002131.

[21] D. Jemerov e S. Isakova. *Kotlin in Action*. Manning Publications Company, 2017. ISBN: 9781617293290. URL: https://books.google.it/books?id=qtcIkAEACAAJ.

[22] B. J. McCarthy e J. J. Holland. «Denatured DNA as a direct template for in vitro protein synthesis.» In: *Proceedings of the National Academy of Sciences* 54.3 (set. 1965), pp. 880–886. DOI: 10.1073/pnas.54.3.880. URL: https://doi.org/10.1073/pnas.54.3.880.

[23] Joshua Bloch. *Effective Java*. Addison-Wesley, 2018. ISBN: 9780134685991. URL: https://www.amazon.com/Effective-Java-Joshua-Bloch/dp/0134685997.

Ringraziamenti

Italo Calvino scrisse che "*l'arte di scriver storie sta nel saper tirar fuori da quel nulla che si è capito della vita tutto il resto, ma finita la pagina si riprende la vita e ci s'accorge che quel che si sapeva è proprio un nulla*", ed è più o meno così che io mi sento al termine di questo percorso, consapevole che in questi ultimi tre anni ho senza dubbio imparato molte cose, ma di certo non abbastanza per sentirmi completo, sotto qualsiasi punto di vista. Se, però, la prospettiva di un futuro ignoto e di tutte le prove che esso porterà con sé può indurre a uno stato di leggera apprensione, allo stesso modo devo ritenermi fiero di aver raggiunto questo enorme traguardo, ulteriore passo in avanti verso una maturità culturale che, giustamente, diventa sempre più importante nel mondo moderno.

I miei sforzi da soli, tuttavia, non sarebbero mai stati sufficienti, ed è per questo motivo che una sezione di ringraziamenti alle persone che, direttamente o indirettamente, mi hanno accompagnato verso il conseguimento di questo titolo, è d'obbligo.

Ringrazio quindi i miei genitori, Marco e Monica, mia sorella Anna e tutto il resto della mia famiglia, per avermi permesso, da sempre, di esprimere le mie passioni senza alcun intralcio, e per continuare a farlo anche durante quei momenti in cui il mio atteggiamento, ne sono consapevole, possa risultante eccessivamente incline al silenzio.

Ringrazio i miei coinquilini, Diego e Milo, per aver reso splendida la mia permanenza in questa città – che senza dubbio è bella, ma c'è voluto un po' per rendersene conto – dal primo fino all'ultimo giorno di convivenza.

Ringrazio Sofia, la mia – ahilei! – ragazza, che più di una volta ha svolto il ruolo di deterrente verso le mie crisi esistenziali, senza mai mostrarsi turbata o annoiata, ma anzi prendendo la palla al balzo per farsi un po' di sana esperienza lavorativa.

Ringrazio Augusto, Beatrice, Chiara, Pierdomenico, Filippo e tutte le altre persone

conosciute durante il mio soggiorno a Cesena, con cui ho avuto modo di condividere esperienze che mi hanno aiutato a crescere come persona e come amico.

Ringrazio anche gli amici di casa, Federico, Matteo e Michele, che nonostante la lontananza non mi hanno mai rimpiazzato, e rimarranno per sempre parte del mio passato, del mio presente e, credo fermamente, anche del mio futuro.

Ringrazio infine l'Ingegner Pianini, l'Ingegner Cortesi e il Professor Viroli per avermi seguito con competenza e interesse durante questi mesi, rendendo più leggera e godibile la realizzazione di un progetto che, senza il loro aiuto, sarebbe stato fin troppo impegnativo.

Sperando di non aver dimenticato nessuno – non me ne vogliate, in caso; cercherò di farmi perdonare in qualche modo – lascio quindi, in un ultimo scatto di egocentrismo, un ringraziamento a me stesso e, in particolare, alla mia caparbietà ed eccessiva costanza che tanto odio in certi casi ma che in altri, come questi, mi permette di raggiungere risultati di cui potrò sempre andare fiero.