Compiling and Verifying DNA-Based Chemical Reaction Network Implementations

Thesis by

Seung Woo Shin

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2011
(Submitted)

Acknowledgements

I am deeply indebted to my advisor Erik Winfree. It is only through his patience and guidance that this thesis could be finished. He always opened new doors for me when I would spend days stuck at a dead end. The insights that he offered me at such times, I believe, is what distinguishes a true scientist from a mere problem solver. Recognizing the difference was an indispensable lesson, for it has led me to strive to become both a good problem solver and scientist myself. Erik also played a large part in my decision to pursue an academic career.

I would also like to thank David Soloveichik not only for his achievements in CRN implementations, which served as a major motivation for my research project, but also for all the invaluable discussions. I thank Brian Wolfe for the development of the reaction enumerator, without which my verifier could not exist. I also express my appreciation to Joseph Schaeffer, Damien Woods, and David Doty for affording me with priceless comments and discussions, and to all Winfree lab members for providing an ideal learning environment.

I wish to extend my thanks to NSF, Caltech's SURF program, and the Caltech Computer Science department, whose generous support made my research possible. As always, I am grateful for my mother, my father, and my sister for being my greatest support system.

Last but not least, I owe my sincere gratitude to my beloved Alma Mater, Caltech. In retrospect, I am most certain that no other school could have allowed for more rewarding five years. I consider myself very fortunate to have been given an opportunity to choose and attend Caltech. Had it not for the valuable experiences at this incredible place, I could not have become the person that I am today, and for this reason it will always be near and dear to my heart.

Abstract

One goal of molecular programming and synthetic biology is to build chemical circuits that can control chemical processes at the molecular level. Remarkably, it has been shown that synthesized DNA molecules can be used to construct complex chemical circuits that operate without any enzyme or cellular component. However, designing DNA molecules at the individual nucleotide base level is often difficult and laborious, and thus chemical reaction networks (CRNs) have been proposed as a higher-level programming language. So far, several general-purpose schemes have been described for designing synthetic DNA molecules that simulate the behavior of arbitrary CRNs, and many more are being actively investigated.

Here, we solve two problems related to this topic. First, we present a general-purpose CRN-to-DNA compiler that can apply user-defined compilation schemes for translating formal CRNs to domain-level specifications for DNA molecules. In doing so, we develop a language in which such schemes can be concisely and precisely described. This compiler can greatly reduce the amount of tedious manual labor faced by researchers working in the field. Second, we present a general method for the formal verification of the correctness of such compilation. We first show that this problem reduces to testing a notion of behavioral equivalence between two CRNs, and then we construct a mathematical formalism in which that notion can be precisely defined. Finally, we provide algorithms for testing that notion. This verification process can be thought of as an equivalent of model checking in molecular computation, and we hope that the generality of our verification techniques will eventually allow us to apply them not only to DNA-based CRN implementations but to a wider class of molecular programs.

Contents

Chapter 1

Introduction

1.1 Molecular Computer

What is a molecular computer? For outsiders, this name alone may sound bizarre enough because the word computer in our time usually carries the image of an electronic board filled with semiconductor chips made of silicon. However, if we rethink the word computer, it should really be used to refer to any kind of device that is capable of computing. At present, semiconductors are the best-studied material for building such a device. But at the front line of human knowledge, scientists and engineers are striving to realize many other computer models that might overpower electronic digital computers in certain ways. In the field of molecular computation, we study how we can induce computational behavior from molecules in solution.

The key idea of molecular computation draws from the fact that we can use chemistry to predict the behavior of molecules in advance. For instance, suppose we know that molecule A and molecule B tend to bind to each other to form a bigger molecule C when they are present in solution together. Then, if we had a way of testing whether C is present in solution, we can regard this system as an AND gate taking two inputs A and B. The species C would be present if and only if both A and B were present in solution. This analogy gets clearer when we imagine A, B, and C as Boolean variables indicating whether the corresponding molecules are present in solution. If this example is not impressive enough, let us add that we can achieve Turing-universality using the same idea [42, 38, 29], which means that anything that can be computed by an electronic computer can also be computed by the molecular computer.

Provided we will build our molecular computer upon this idea, it is obvious that the better we understand the behavior of a molecule, the easier it is to build a system with it. Then what shall be our choice of this molecule? Which types of molecules do we understand best? At present, the most popular choice is DNA molecules, due to mainly two reasons. First, their behavior is relatively well-understood. As commonly known, a DNA strand is a sequence of nucleotide bases each of which can be either adenine (abbreviated A), cytosine (C), guanine (G), or thymine (T). When these bases

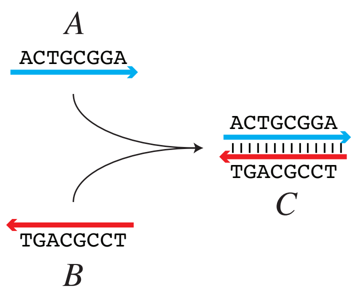


Figure 1.1: A sample implementation of the example given in the previous paragraph. For simplicity, the double helix structure of the species C is neglected in this diagram and only the pairing information is given.

encounter their Watson-Crick complements (A-C and G-T), they may form hydrogen bonds to form the famous double-helix structure. Note that this behavior, namely the Watson-Crick base pairing, is intrinsically algorithmic. In particular, it is very easy to implement the example given in the previous paragraph using DNA strands (Fig. 1.1). Second, there are established ways to synthesize arbitrary DNA sequences. That is, if we design a sequence of bases to use in an experiment, we can easily obtain the actual DNA strand with that sequence.

The first result of DNA computation was reported in 1994 by Leonard Adleman [1] (although the possibility of a molecular computer was suggested by Richard P. Feynman in as early as 1959 [12] and explored in seminal work by Charles H. Bennett in 1982 [2]). Adleman designed DNA strands that will spontaneously self-assemble complexes that encode possible solutions for the Hamiltonian path problem. Then, the actual solutions could be found by applying additional lab techniques to filter out non-solution molecules. The experiment certainly had a strong impact on scientists working in related fields, and it was soon followed by numerous other results exploring the possibilities of a molecular computer. It has been shown that we can build self-assembling DNA tiles [42, 34, 31, 22], digital logic circuitry [33, 25, 28], transcriptional networks [17], entropy-driven DNA catalysts [49], arbitrary shapes [42, 30, 45], arbitrary chemical reaction networks [39, 4, 27], molecular robots [23, 37, 46, 41, 40], oscillators [17, 18], and even Turing-universal stack machines [27]. Many of these results are also closely related to other fields as well, especially bionanotechnology and biorobotics.

Then a natural question arises: it is great to hear that molecular computers can perform all these tasks, but what is their advantage over traditional electronic computers? Why should we study molecular computation rather than ways to build faster electronic computers? Clearly, the goal of molecular computation is not to compete with electronic computers in speed; after all, a molecule can never move faster than electrical current. Rather, the goal of molecular computation lies in learning how to program complicated molecular interactions and in doing so establishing ways to exercise delicate control at the molecular level. We can imagine a great number of immediate applications. For instance, in a near future, we might be drinking – instead of pills – autonomous molecular robots that aid our immune system.

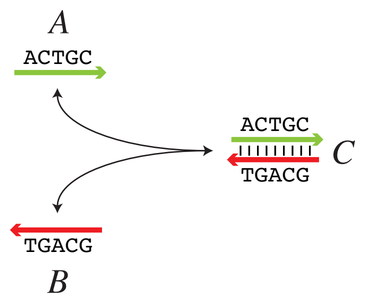
1.2 Importance of Chemical Reaction Networks

This thesis will address a very specific topic in DNA computation, namely the chemical reaction network implementations. Before getting into the main discussion, it will be worthwhile to point out the vital importance that chemical reaction networks possess in the field.

Chemical reaction networks (CRNs) are a universally used formal language for describing chemical systems. A CRN is defined as a set of chemical reactions (which in turn are defined by specifying their reactants and products) paired with corresponding rate constants. The conventional notation for writing chemical reactions is used for describing reactions in a CRN, e.g., $A + B \xrightarrow{k} C + D$. This example says that the reaction at hand completely consumes two reactants A and B and then produces two products C and D, and k is the rate constant that governs how fast this reaction occurs. If A happens to be a catalyst, the reaction can be written $A + B \xrightarrow{k} A + C + D$ to indicate that A is required for the reaction but does not get consumed. In general chemistry, these letter species names are usually replaced by the actual chemical names, e.g., H_2O, $NaCO_3$, etc. However, in DNA computation, each letter name usually denotes a specific DNA complex with a specific structure.

Fig. 1.2 is an example of how a DNA system can be described in the CRN formalism. Note that the two DNA strands A and B are Watson-Crick complementary to each other. Hence, if these two strands come into contact in solution, they will hybridize to form a double-stranded molecule which we call C. In this example, the two strands are not long enough to sustain the bonding and thus will spontaneously detach after a little time. To indicate this behavior, this reaction is modeled as a reversible reaction.

Another point to make in this example is that in reality there exist several intermediate states between the initial state where A and B are completely separate and the final state where they are completely hybridized. Because base pairing does not occur simultaneously, there will be states where only some of the nucleotide bases are hybridized. However, such intermediate configurations



$$A + B \xrightarrow{k_1} C$$

$$C \xrightarrow{k_2} A + B$$

Figure 1.2: A DNA system and its CRN representation

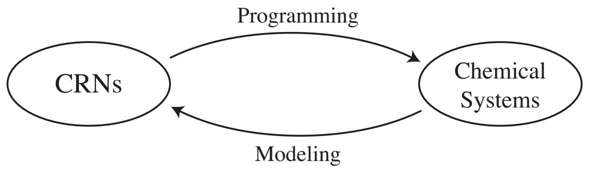


Figure 1.3: The role of CRNs in molecular computing

are often neglected from discussion because they rapidly turn into more stable configurations.

We have seen that CRNs are a good modeling language for DNA systems as they are for any other type of chemical system. In molecular computing, CRNs assume one additional role that is immensely powerful; they become a *programming language* for molecular interactions [10, 21, 35, 36]. Note that our attempt to obtain computational behavior in molecules is the exact reversal of the modeling process; instead of studying the behavior of a given system, we try to create a chemical system that exhibits a given behavior. Thus, if we had a general scheme for building a DNA system that implements the behavior of a given arbitrary CRN, CRNs can effectively function as the "source code" for DNA systems and the CRN formalism becomes a "programming language" [39]. Moreover, it is known that stochastic CRNs can emulate Turing machines with an arbitrarily small probability of error [38], so it is not just any programming language, but a powerful one.

1.3 Implementing Arbitrary Chemical Reaction Networks

For CRNs to serve the roles delineated in the previous section, we need a general scheme for implementing a given CRN using real molecules, i.e., given any CRN, we should be able to design molecules that behave according to the reactions specified in that CRN. Then, since CRNs are a Turing-universal computational model that can encode any kind of computational behavior, it will follow that we can perform arbitrary computation with molecules. Fig 1.4 illustrates the "compilation" steps needed for this procedure.

Note that the first step of this compilation process, from abstract computational ideas to CRNs, need not be general. The proof that this translation is always possible, which was given in [38], used the fact that it is possible to simulate Turing machines using CRNs. However, the existence of a general translation scheme does not necessarily mean that we have to use it. In fact, it is often advantageous to use translation schemes that are more specific to the computational behavior of interest. For instance, if we were to implement the logic gate $Y = ((X_1 \vee X_2) \wedge X_3) \vee X_4$, it is much more efficient to reduce it to the following small CRN than to first translate the logic gate to an equivalent Turing machine and then translate that Turing machine to a CRN using [38].

$$\begin{aligned} X_1 &\to I \\ X_2 &\to I \\ I + X_3 &\to J \\ J &\to Y \\ X_4 &\to Y \end{aligned}$$

Clearly, the output molecule Y will be found if and only if the given logic formula is satisfied (where the truth value of X_i is determined by the presence of the corresponding molecule in solution). This system contains only five reactions whereas translation via Turing machines would require orders of magnitude more reactions. However, generalizing this construction works only for a small class of computation, namely AND-OR logic circuitry, and if we wanted to perform a different type of computation, we would need a different construction.

Since we are likely to use many different constructions for the first step, the construction for the second step, on the other hand, needs to be as general as possible. This is because the fact that we allow many different implementations for the first step makes it impossible to impose any condition on

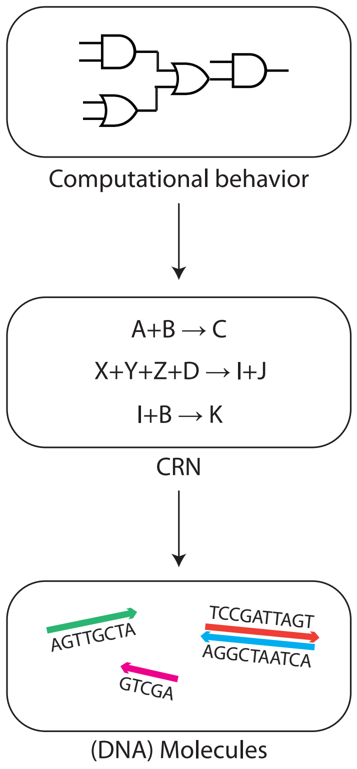


Figure 1.4: CRN-based compilation steps

the CRNs that we encounter in the second step. In other words, the scheme for the second step should work for any CRN. Fortunately, one such general construction was demonstrated by Soloveichik et al. in [39], with a few more that followed later [4, 27, 5] (there also exist implementations that work for a limited class of CRNs [28, 24]). A thorough understanding of such DNA-CRN translation schemes is vital for understanding this thesis, so we will introduce the construction of Soloveichik et al. in this section.

However, before we do so, it is necessary to first introduce a few fundamental concepts of molecular computation that will recur throughout the entire thesis.

First, from this point on, we will annotate DNA strands not by specifying individual nucleotide bases, but by specifying what we call **domains**. Note that the DNA strands in Fig. 1.5 are specified by little numbered segments, or domains, instead of the actual nucleotide base sequences. Here, a starred domain is supposed to have a sequence that is complementary to that of the corresponding un-starred domain. This notation, often called domain specification, is a higher-level abstraction that researchers in the field often employ to relieve the pain of dealing with long DNA sequences. It is clear that for a domain specification to work, it should be enforced that two domains are complementary (either fully or partially) if and only if they are the starred and the un-starred domains of the same number. Otherwise, there will be spurious reactions which are not explained by the domain specification of the system. For this purpose, there are sequence designer programs [47] that translate a given domain specification to actual sequences ensuring that this condition is enforced.

Second, in our domain specifications, we will often distinguish between "long" domains and "short" domains depending on their ability to sustain hybridization. For instance, two complementary domains of length 5 (meaning they have five nucleotide bases) will not provide sufficient binding energy for the two strands to stay together, while two complementary domains of length 20 will. This distinction between long domains and short domains is important, because it provides a foundation for the so-called toehold-mediated branch migration.

In toehold-mediated branch migration, molecules are designed in such a way that they can only react by hybridization of a toehold domain (synonymous with a "short" domain). It might seem that since toehold domains are short, the two molecules will detach after a little time and there will not be any significant behavior in the system. However, the situation is different when there are appropriate domains right next to the toehold domain. For example, in Fig. 1.5, molecules X_1 and L_i share complementary toehold domains 1 and 1* by which they can bind to each other. Note also that the next domains in X_1, namely 2 and 3, are also complementary to the adjacent domains on the bottom strand of L_i. In this case, X_1 will compete with the short top strand of L_i (2-3-4) for the 2* and 3* domains on the bottom strand. If X_1 loses this competition, then it is bound only by one toehold domain and will spontaneously fall off. If X_1 wins, the top strand 2-3-4 is bound only

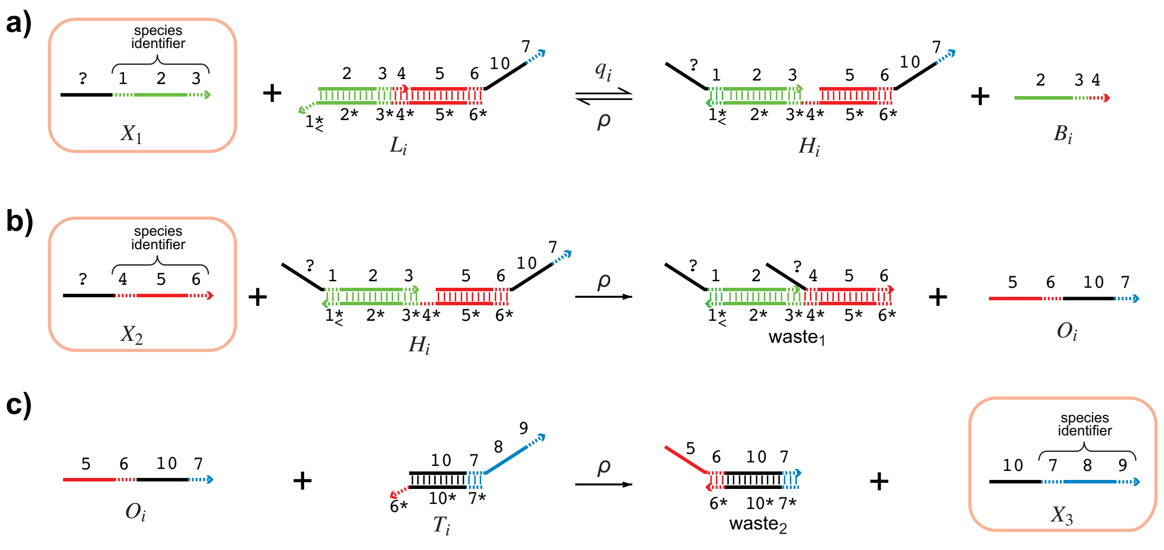


Figure 1.5: Bimolecular module: DNA implementation of the formal bimolecular reaction $X_1+X_2 \rightarrow X_3$ with reaction index i (meaning just that it is the i-th reaction in the CRN). Figure taken from [39].

by one toehold domain and will fall off. In this case, we are left with a different molecule which has the toehold 4 open instead of the toehold 1. This behavior is modeled by the first reaction in Fig. 1.5, which is reversible because the now-free top strand B_i can hybridize with the open toehold 4 on H_i to reverse this process.

This idea, toehold-mediated branch migration, was the basic building block for the construction of Soloveichik et al. Fig. 1.5 shows how a bimolecular reaction (in this thesis, a reaction is bimolecular if it takes exactly two reactants, and similarly, it is unimolecular if it takes exactly one reactant) can be emulated using DNA molecules. Observe that the construction involves some auxiliary species (L_i, B_i, and T_i) as well as the species from the target reaction (X_1, X_2, and X_3). These auxiliary species are usually assumed to be in excess compared to other species, so that a little fluctuation in the concentration of the auxiliary species will not have an effect on the kinetics of the system. This is essential because in most CRN implementations these auxiliary species serve the role of fuel and will constantly get consumed as the system evolves (and the system will deviate from the desired behavior when too much fuel has been consumed).

To understand the logic of this implementation, we shall investigate its execution pathway step by step. Suppose we begin with a huge amount of the auxiliary species L_i, B_i, and T_i and a small amount of the reactant species X_1 and X_2. Since both X_1 and L_i are present in solution, the first step in Fig. 1.5 will start producing the intermediate species H_i. Then, the second step, which consumes X_2 and H_i and produces another intermediate species O_i and some waste, will get activated. Finally, the intermediate O_i reacts with the auxiliary species T_i to produce the target product species X_3. Note that the waste species that get produced along with the desired product have no open toehold and thus are effectively inert. It is important that they cannot participate in any further reaction, because otherwise they can form spurious pathways and cause leak and crosstalk.

Now consider the case where we start with only one of the two reactant species X_1 and X_2. First, if we had only X_2, none of the three reactions can occur and we will not observe any change in the concentration of X_1, X_2, and X_3. This is exactly the behavior expected by the target reaction $X_1 + X_2 \rightarrow X_3$. The case is less simple when we start with X_1 but no X_2. In that case, some of the X_1 molecules will turn into H_i molecules as a result of the first reaction in action. However, the target reaction $X_1 + X_2 \rightarrow X_3$ says that there should not be any fluctuation in the concentration of any of those three species when only X_1 is present. This is why the first reaction in Fig. 1.5 was designed to be reversible. By adjusting the rate constants q_i and ρ appropriately, we can ensure that only a negligible amount of X_1 gets converted to H_i if no further reaction can be triggered. This would not have been possible if the first reaction was irreversible.

This construction works not only for this one reaction but also for CRNs consisting of many different reactions. It is because of the modularity of this construction. Observe that X_1, X_2, and X_3 are of the same form albeit with different domains; in particular, the reactants (X_1 and X_2) and the products (X_3) of this implementation have the same structure. Because of this property, we can use the exact same module for implementing any bimolecular reaction, even when there are cascades of reactions. For instance, we can use the same module to implement the reaction $X_3 + X_4 \rightarrow X_5$. Although X_3 was a product of one reaction $X_1 + X_2 \rightarrow X_3$, it can also be used as a reactant to another reaction $X_3 + X_4 \rightarrow X_5$ because in both cases X_3 has the same structure. Note that this was not the case for the construction shown in Fig. 1.1.

Also, note that the idea used in this construction could be generalized to implement reactions with different number of products. It is pointed out in [39] that by modifying the O_i and T_i molecules to have more domains and strands, we can implement reactions with an arbitrary number of products. For instance, if we lengthen the bottom strand of T_i, put one more signal species X_4 and add the corresponding domains to O_i, we will have implemented $X_1 + X_2 \rightarrow X_3 + X_4$. In contrast, this scheme does not generalize to an arbitrary number of reactants (of course, there are some other schemes that do [27, 4]). However, it can be easily shown that if a scheme can implement unimolecular and bimolecular reactions, then it can implement reactions with an arbitrary number of reactants by composing multiple modules, if we ignore kinetics. For instance, to implement $A + B + C \rightarrow D$, we can rather implement the following three unimolecular and bimolecular reactions: $A + B \rightarrow I$, $I \rightarrow A + B$ and $I + C \rightarrow D$. If we do not count I as a signal species, the net effect would be exactly $A + B + C \rightarrow D$.

Obviously, the construction shown in Fig. 1.5 is merely one of the many ways to implement CRNs using DNA molecules. Indeed, several different implementations [4, 27, 5, 9] were discovered later. It is usually difficult to claim the superiority of one such implementation to another because they have different pros and cons; for example, some implementations admit easier sequence synthesis, some have lower leak rates, etc. Thus it is expected that many more such implementations, or **translation schemes** as we shall call them from now, will be developed and used in the future. Some of them might apply only to a certain subclass of CRNs [28, 24], or might not explicitly translate from a higher-level language [19, 20, 26, 7].

1.4 Outline

This thesis consists of two independent but related projects. The smaller of the two is about an automated compiler for the CRN-to-DNA translation process. The compiler takes two input data which are the translation scheme description and the target CRN and produces the domain specification of the DNA system that implements the given CRN using the given translation scheme. The major features of the compiler are briefly described in Chapter 2 while the details of the software follow in the appendix.

The larger project, which we present over the next four chapters, is about a verifier that tests the correctness of the above-mentioned compiler. However, rather than focusing only on the correctness of our own compiler, we attempt to solve a more general version of the problem: we attempt to test the behavioral equivalence between arbitrary CRNs. Since it is vague what is meant by equivalence here, the first step is to propose a precise definition for it, which turns out to be a nontrivial task and forces us to spend the whole of Chapter 4 on it. In Chapters 5 and 6, we present algorithms for testing that notion of equivalence and report some experimental results. Chapter 3 briefly describes a software tool called reaction enumerator, which is essential for implementing our verification algorithm.

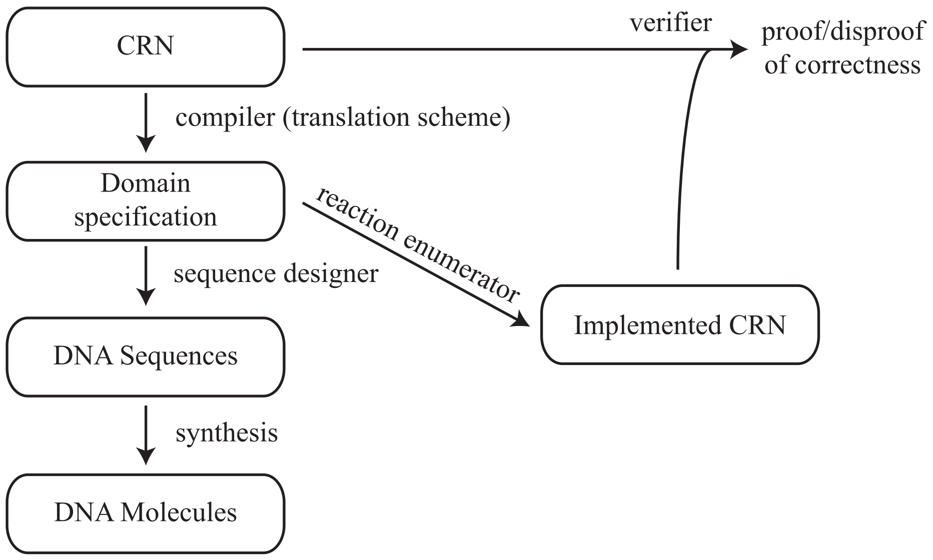


Figure 1.6: CRN-to-DNA compilation schematics

Chapter 2

The BioCRN Compiler

2.1 Introduction

As we delineated in the introduction chapter, there are mainly three steps in which a CRN is compiled down to DNA molecules (Fig. 1.6). First, the given CRN is translated to domain-specified DNA molecules. Second, the domain specification is turned into sequences of nucleotide bases. Finally, real molecules are synthesized according to the sequence information generated in the second step. The procedures for the second and third steps are highly standardized among researchers; the second step is usually performed using software programs called sequence designers such as NUPACK [47] from the California Institute of Technology. The third step is already industrialized and is usually carried out by commercial companies.

Unfortunately, the first step is being performed in many different ways even in a single research group. This is due both to the previously mentioned intrinsic diversity of translation schemes and to the lack of relevant software. Observe that the latter is a consequence of the former. While it is easy to build a compiler that works for a single translation scheme, it is much more difficult to design a general compiler that works for many different translation schemes. This is because translation schemes might build on drastically different ideas; there is no guarantee that all translation schemes will build on the same toehold-mediated branch migration idea presented in the last chapter.

To acquire the generality required for such a compiler, we chose to deviate from the conventional definition of a compiler and adopt the translation scheme description itself as a part of the input data. In other words, our compiler will take two input data instead of one, which are the translation scheme description and the input CRN. Given those data, it will produce domain-specified DNA molecules that implement the given CRN, following the procedures laid out in the given translation scheme. If it is easier to write down a translation scheme in our input language than to write an ad hoc compiler for it, our compiler will contribute greatly to the researchers in this field.

We should mention some previous work done in this regime. First, we note that there are some compilers that work for a single specific translation scheme, usually written by the designers of that translation scheme. For example, David Soloveichik has a compiler written in Mathematica for his translation scheme from [39]. Second, there is Visual DSD [19, 20, 7, 26] which is developed and maintained by Microsoft Research. It is a general tool for analyzing DNA systems which provides a programming language for specifying DNA strand displacement circuits, enumerating their domain-level reactions, and performing simulations. However, their language acts more as a macro language for specifying sets of DNA structures rather than a complete programming language with a specific source language semantics. For instance, it does not have conditionals, loops, or advanced data structures, so the user has to write a separate macro for reaction modules with a different number of reactant or product molecules, even though they all depend on the same logic. In contrast, our compiler will use a representational framework that explicitly begins with a CRN program that supports various high-level programming constructs.

In this chapter, we briefly introduce our 'BioCRN' compiler. We will first describe some of its major features and then provide a few example codes to help understanding. Due to space constraints, the formal syntax and semantics of the languages used in this chapter will only be found in the appendix at the end of the thesis.

2.2 Features

The BioCRN compiler was written in Python 2.6.6 and thus runs on all commonly used platforms. It uses Pyparsing and DNAObjects libraries (the former is a common library and the latter is developed and maintained by the Winfree group at the California Institute of Technology). Its main function, as described in the previous section, is to take two input files, one containing the translation scheme description and the other containing the target CRN, and produce the domain-specified molecule descriptions according to the given input data.

The extensions for the input and output files are .ts (for 'translation scheme'), .crn ('chemical reaction network'), and .dom ('domain specification') respectively. Each file extension has an associated well-defined language in which the data are expressed. Clearly, the core of our compiler is in the TS language. While the other two languages are mere representations of chemical reactions and molecule structure, the TS language is required to be a high-level programming language capable of expressing many different translation schemes. Therefore, we will discuss mainly the TS language in this section and leave the reader to learn the other two languages through the examples presented in the following section (or by reading the appendix).

At first glance it might seem that we could afford to develop a relatively weak programming language for the TS files, because the TS language needs only be able to express valid translation schemes, not arbitrary programs. However, we should not generalize from the little experience that we have in this field that the translation schemes of the future should also be as simple and clean as that of Soloveichik et al. For instance, we can ambitiously imagine a translation scheme that runs simulation on a set of randomly generated molecules and chooses the best one, and the TS language should still be able to express that translation scheme. Thus, we designed it to be a purely functional, dynamically typed programming language that supports conditionals, recursion, lists, pattern matching, and even higher order functions (albeit partially). The only requirement that our language imposes on translation schemes is that they should output a set of domain-specified molecules in the end.

Much of the syntax for our language was inspired by Haskell, which is a general-purpose functional programming language. The following is an example showing some common functions implemented in the TS language.

```
1  # Computes the sum of the given list of integers
2  function sum(x) =
3     if len(x) = 0 then 0 else x[0] + sum(tail(x));
4
5  # Computes the length of the given list
6  function len(x) =
7     if x == [] then 0 else 1 + len(tail(x));
8
9  # Reverses the given list
10 function reverse(x) =
11    if x == [] then [] else reverse(tail[x]) + [x[0]];
12
13 # Applies the function f on each entry of the list x and returns the list of the
      results
14 function map(f, x) =
15    if len(x) == 0 then
16       []
17    else
18       [f(x[0])] + map(f, tail(x))
```

Here, `tail` is a built-in function that returns the given list except the first entry, and `x[i]` indexes the i-th entry of the list `x`. In lines 7 and 11, we can notice that the plus operator is overloaded for multiple data types. Namely, it is being used for adding two integers in line 7 and for concatenating two lists in line 11.

This sample code should be easy to understand for those who have experience in programming. Note that recursion and conditionals are the fundamental idea behind these implementations as it is often the case with functional programming languages. However, since the purpose of this language is to express translation schemes, it should also support some unconventional data types for expressing molecule structure, concentration, etc. Thus, our language supports new data types called `Species`, `Reaction`, `Domain`, `Structure`, and `Solution` as well as traditional data types such as integers, floats, booleans, and lists. The exact specifications of these data types are reserved for the appendix, but in this section we will provide intuitive descriptions of some of the new data types through examples. Also, Fig. 2.1 summarizes the general purposes of these new data types.

The following example shows a use of `Domain` and `Structure` data types. Here, `gate` is a function that takes one argument of the type `Domain` and returns a `Structure` object. Since the `Structure` data type is used to describe a DNA complex, `gate` could be understood as a function that takes one domain and returns a molecule whose structure will somehow depend on that domain.

```
function gate(toehold) =
    "b c + c* b* a*" |
    "( ( + )  )  . "
    where {
        a = toehold;
        b = long();         # long() generates a new unique long domain and although not
        c = long()          # used here, short() generates a new unique short domain.
    }
```

The second and third lines are describing the primary and secondary structures of the molecule that will be returned by the function. This notation system using dots and parentheses is known as dot-paren notation [16]. In this notation, a plus sign indicates strand breaks, a dot indicates that the corresponding domain is unpaired, and a parenthesis indicates that the corresponding domain is paired with the domain with the matching parenthesis. The structure specified in the above code is illustrated in Fig. 2.2. While this notation system cannot express every possible structure that can occur in a DNA molecule (in fact, the class of structures that dot-paren notation can express is exactly those that are free of pseudoknots), it is one of the most convenient notations available today and is widely used in the field. Fig. 2.3 provides some examples that will further help understanding of this notation system.

Then, what is the minimal requirement for a TS source code to be valid? Just as C programmers have to implement the 'main' function, translation scheme programmers need to implement the following two functions:

1. `formal` is a function that constructs molecules that represent species from the input CRN. The name of the function comes from the fact that we call species from the input CRN the **formal species**.

2. `main` is a function that takes a list of reactions and returns all the auxiliary species required to implement those reactions (the return value should be a `Solution` object).

As usual, we should study some examples to understand the purpose of each function more clearly.

The following is an example implementing the `formal` function for the translation scheme from Fig. 1.5.

```
function formal(s) =
   "? a b c" | "? . . ."
   where {
      a = short() ;
      b = long() ;
      c = short()
   };
```

Soloveichik et al. requires that the molecules representing formal species have one "history" domain, which is variable according to the reaction that produced those molecules. Thus, each formal species will not be mapped to a single implemented species, but it will be mapped to a class of species. To express this, here we used a question mark as a wildcard that can match any single unpaired domain. Recall that `short()` and `long()` are functions that generate new unique domains of respective lengths, so `formal` will return a new single-stranded molecule with three new distinct domains every time it is called. This way, no two formal species will share the same domain, as

Data type	Usage
`Species`	Used to represent chemical species. As its only data field is `name` which simply stores the name of the species, its main purpose is to distinguish between different species.
`Reaction`	Stores information about chemical reactions. It has three data fields; `reactants` which holds the list of reactant molecules, `products` which holds the list of product molecules, and `reversible` which tells whether the reaction is reversible.
`Domain`	Used to represent domains. It has two data fields: `id` and `length`. Similarly to the `Species` data type, it is mainly used to distinguish between different domains using the `id` field.
`Structure`	Stores the primary and secondary structure of a DNA complex along with the domains used in it.
`Solution`	Represents a chemical solution. It has one data field called `molecules`, which stores a list of `Structure` objects. Conceptually, it stores the list of molecules that are present in the solution.

Figure 2.1: New data types.

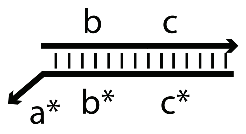
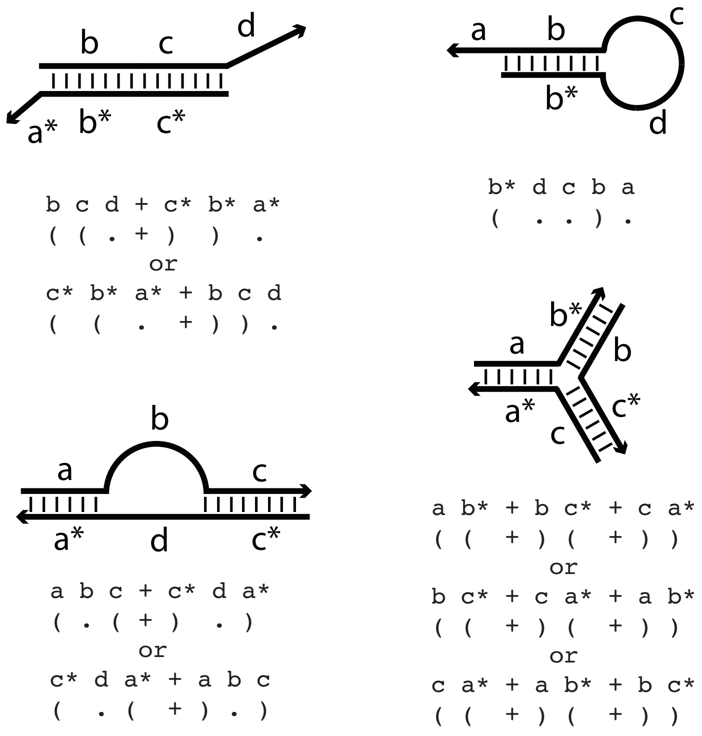


Figure 2.2: DNA secondary structure corresponding to the dot-paren notation "((+)) ." (from the above `gate` function).



```
b c d + c* b* a*
( ( . + )  )  .
      or
c* b* a* + b c d
(  (  .  + ) ) .
```

```
b* d c b a
(  . . ) .
```

```
a b c + c* d a*
( . ( + )  . )
      or
c* d a* + a b c
(  . (  + ) . )
```

```
a b* + b c* + c a*
( (  + ) (  + ) )
        or
b c* + c a* + a b*
( (  + ) (  + ) )
        or
c a* + a b* + b c*
( (  + ) (  + ) )
```

Figure 2.3: Some more examples of dot-paren notation

desired by the scheme. Fig. 2.4 shows the molecule that will be constructed from this code.

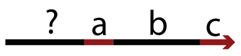


Figure 2.4: The structure from `formal`

Now, note that the following function constructs the B_i molecule from Fig. 1.5.

```
function Bi(x1, x2) =
   "d2 d3 d4" |
   ". . ."
   where {
      d2 = x1.b ;
      d3 = x1.c ;
      d4 = x2.a
   };
```

Here, `x1` and `x2` are assumed to be `Structure` objects produced by the `formal` function. Thus, when we reference `x1.b` or `x1.c`, it will return the corresponding domains as was specified in the dot-paren description in `formal` (if `x1` or `x2` were not produced by `formal`, it might not have domains named `a`, `b`, or `c`, in which case it would result in a runtime error).

Finally, we can write down the following `main` function.

```
function main(crn) = infty(Bi(crn[0].reactants[0], crn[0].reactants[1]))
# 'infty' is a built-in function that takes a Structure instance and puts that the
    species is supposed to have "infinite" concentration. The resulting object will
    be a Solution instance.
```

The argument given to the `main` function, `crn`, is a list of `Reaction` objects that store the reactions from the input CRN. However, their `reactants` and `products` fields contain `Structure` objects that are obtained by running each species that appears in the given reaction through the `formal` function.

Thus, if we ran this translation scheme on the CRN consisting of one reaction $X_1 + X_2 \rightarrow X_3$, it will correctly translate all the formal species according to the scheme from Fig. 1.5, and produce the appropriate B_i molecule. However, as can obviously be seen, this main function ignores every reaction but the first one, it assumes that the first reaction is bimolecular, and most importantly, it does not produce any other auxiliary species than B_i. This is obviously not an implementation of a correct translation scheme, and it requires much more work to make it one. A more thorough example will be given in the next section, but will require a deeper understanding of the language to grasp.

Lastly, note that our compiler completely ignores the kinetic aspect of the system. Thus, we do not specify rate constants in our CRN language and we only specify the concentration of a species as 'infinite' or 'non-infinite.' This is a major limitation of our compiler and should be improved in the future.

2.3 Examples

```
A -> B + C
C -> D
```

Listing 2.1: sample.crn ↑

```
#
# David Soloveichik's translation scheme from "DNA as a universal substrate
# for chemical kinetics", Proceedings of the National Academy of Sciences,
# 107: 5393-5398, 2010.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

function formal(s) = "? a b c"
                   | "? . . ."
```

```
    where {
        a = short() ;
        b = long() ;
        c = short() };

function ugate(s, l)
     = ["b c d + c* b* a*"          # G_i
       | "( ( ~ + )  )  .",
        "e + f c*"                  # T_i
       | "~ + ~ .")
    where {
        a = s.a ;
        b = s.b ;
        c = s.c ;
        [d, e, g] = flip(map(gmac, l), 3) ;
        f = reverse(g) };

function gmac(s)
     = ["d a"
       | ". .",
        "d a b c +"
       | "( ( . . +",
        "a* d*"
       | ")  )"]
    where {
        d = long() ;
        a = s.a ;
        b = s.b ;
        c = s.c };

function unimolecular(r) = infty(g) + infty(t)
    where
        [g, t] = ugate(r.reactants[0], r.products);

function bgate(s1, s2, l)
    = ["b c d + e f g + f* e* d* c* b* a*"        # L_i
      | "( ( ( + ( ( ~ + )  )  )  )  )  .",
       "h + i f*"                                  # T_i
      | "~ + ~ .",
       "b c d"                                     # B_i
      | ". . ."]
    where {
        a = s1.a ;
        b = s1.b ;
        c = s1.c ;
        d = s2.a ;
        e = s2.b ;
        f = s2.c ;
        [g, h, j] = flip(map(gmac, l), 3) ;
        i = reverse(j) };

function bimolecular(r) = infty(l) + infty(t) + infty(b)
    where
        [l, t, b] = bgate(r.reactants[0], r.reactants[1], r.products);

function main(crn) = sum(map(unimolecular, unirxn(crn))) +
                     sum(map(bimolecular, birxn(crn)))
    where
        crn = irrev_reactions(crn)
```

Listing 2.2: soloveichik.ts ↑

```
sequence d1 : 5
sequence d2 : 15
sequence d3 : 5
sequence d4 : 5
sequence d5 : 15
sequence d6 : 5
sequence d7 : 5
sequence d8 : 15
sequence d9 : 5
sequence d10 : 5
sequence d11 : 15
sequence d12 : 5
sequence d13 : 15
sequence d14 : 15
sequence d15 : 15
# Formal species
A :
? d1 d2 d3
? . . .
C :
? d4 d5 d6
? . . .
B :
? d7 d8 d9
? . . .
D :
? d10 d11 d12
? . . .
# Constant species
```

```
automatic4 :
d2 d3 d13 d7 d14 d4 + d3* d2* d1*
( ( . . . . + ) ) .
automatic5 :
d13 d7 d8 d9 + d14 d4 d5 d6 + d4* d14* d7* d13* d3*
( ( . . + ( ( . . + ) ) ) ) .
automatic6 :
d5 d6 d15 d10 + d6* d5* d4*
( ( . . + ) ) .
automatic7 :
d15 d10 d11 d12 + d10* d15* d6*
( ( . . + ) ) .
```

Listing 2.3: sample.dom ↑

In this section, we will investigate a full set of input and output codes to provide a broad picture of the compilation process. In other words, we will look into how the "sample.dom" file above is produced from "sample.crn" and "soloveichik.ts". However, understanding the details of these examples is not necessary for understanding the rest of this thesis, and might require an exhaustive study of the appendix material.

As the compilation begins, first, all the formal species (A, B, C, and D in this case) are translated using the 'formal' function from the TS file. Note that the formal function is written such that we produce three new unique domains for each formal species. Accordingly, the compiler obtains the following after this translation. We can check that the length of each domain agrees with the specification from the formal function. (Here, 5 and 15 are being used as default lengths for 'short' and 'long' domains.)

```
# Formal species
A :
? d1 d2 d3
? . . .
C :
? d4 d5 d6
? . . .
B :
? d7 d8 d9
? . . .
D :
? d10 d11 d12
? . . .
```

Then, the compiler will run the 'main' function of the TS file, supplying the input CRN as the argument. In this case, the argument `crn` will have the following value: `[Reaction([A], [B, C],` False), Reaction([C], [D], False)]. (Reaction([A], [B, C], False) denotes a Reaction object with reactant, products, and reversible data fields having values [A], [B, C], and False respectively. A, B, C, and D here represent the Structure objects that were generated during the first step of compilation using the formal function.)

```
function main(crn) = sum(map(unimolecular, unirxn(crn))) +
                     sum(map(bimolecular, birxn(crn)))
    where
        crn = irrev_reactions(crn)
```

First of all, since we have a where clause, we create a local binding crn which has the value irrev_reactions(crn). irrev_reactions is a function that divides every reversible reaction into a pair of irreversible reactions. Since our input CRN does not contain any reversible reaction, the new crn will be the same as the old crn. Also, note that crn only contains unimolecular reactions, so when the above code computes unirxn(crn) and birxn(crn), the former will be the same as crn itself and the latter will be an empty list. Therefore, the return value of our main function will be sum(map(unimolecular, unirxn(crn))).

To compute this value, we need to apply the function unimolecular to each item in unirxn(crn) and take the sum of the resulting values. Since unirxn(crn) is [Reaction([A], [B, C], False), Reaction([C], [D], False)], we need to first compute unimolecular(Reaction([A], [B, C], False)) and unimolecular(Reaction([C], [D], False)), and compute their sum. Now, we need to study the unimolecular function.

```
function unimolecular(r) = infty(g) + infty(t)
    where
        [g, t] = ugate(r.reactants[0], r.products);
```

Suppose the argument r was Reaction([A], [B, C], False). To compute the return value of unimolecular(r), we first need to evaluate the value of g and t from the where clause. That is, we need to evaluate ugate(r.reactants[0], r.products) first. Clearly, r.reactants[0] is A and r.products is [B, C] where A, B, and C are Structure objects. Thus, the code will call ugate(A, [B, C]) and try to pattern match the result into [g, t].

```
function ugate(s, l)
     = ["b c d + c* b* a*"          # G_i
      | "( ( ~ + ) ) .",
        "e + f c*"                  # T_i
      | "~ + ~ ."]
    where {
        a = s.a ;
        b = s.b ;
        c = s.c ;
        [d, e, g] = flip(map(gmac, l), 3) ;
        f = reverse(g) };
```

Note that the value of the first argument `s` is `A`, which is a `Structure` object created with the `formal` function. If we look at the definition of `formal`, there are three domains that are used to specify the structure: `a`, `b`, and `c`. Thus, when we write `s.a`, `s.b`, and `s.c`, we reference the corresponding `Domain` objects, i.e., `d1`, `d2`, and `d3`, respectively. (If `s` were `B`, we would have referenced `d7`, `d8`, and `d9` instead.) Thus, `a`, `b`, and `c` in the where clause are `d1`, `d2`, and `d3` respectively, but in order to evaluate `d`, `e`, `f`, and `g`, we need to call yet another function, `gmac`. Recall that the value of the second argument `l` was `[B, C]`, which is a list. Thus, when we call `map(gmac, l)`, we will apply `gmac` on `B` and `C` individually, and store the results in a list.

```
function gmac(s)
    = ["d a"
     | ". .",
       "d a b c +"
     | "( ( . . +",
       "a* d*"
     | ") )"]
   where {
       d = long() ;
       a = s.a ;
       b = s.b ;
       c = s.c };
```

Let us first evaluate `gmac(B)`. We first need to evaluate the assignments in the where clause. Note that we need to generate a new unique domain for `d`. Since we already assigned `d1` through `d12` to formal species, we will assign `d` a new `Domain` object named `d13`. Also, since `s` is `B` here, as we discussed above, `s.a`, `s.b`, and `s.c` are `d7`, `d8`, and `d9` respectively. Thus, `a`, `b`, and `c` become `d7`, `d8`, and `d9`. That is, `gmac(B)` has the value `[Structure("d a", ".  .", (d=d13, a=d7)), Structure("d a b c +", "( (  .  .  +", (a=d7, b=d8, c=d9, d=d13)), Structure("a* d*", ") )", (a=d7, d=d13))]`. Similarly, `gmac(C)` will have the value `[Structure("d a", ".  .", (d=d14, a=d4)), Structure("d a b c +", "( ( . . +", (a=d4, b=d5, c=d6, d=d14)), Structure("a* d*", ") )", (a=d4, d=d14))]`.

Now, this means that back in the `ugate` function, the value of `map(gmac, l)` has the form `[[S.., S.., S..], [S.., S.., S..]]`. `flip` is a function that works like a matrix transpose function for lists of lists. Thus, if we compute `flip(map(gmac, l), 3)`, the above list will turn into the form of `[[S.., S..], [S.., S..], [S.., S..]]`. Now we can evaluate all bindings that appear in the `ugate` function, which is shown in Fig. 2.5.

Now, note that there are tilde (∼) signs in the definition of the two `Structure` objects that

a	d1
b	d2
c	d3
d	[Structure("d a", ". .", (d=d13, a=d7)), Structure("d a", ". .", (d=d14, a=d4))]
e	[Structure("d a b c +", "((. . +", (a=d7, b=d8, c=d9, d=d13)), Structure("d a b c +", "((. . +", (a=d4, b=d5, c=d6, d=d14))]
g	[Structure("a* d*", "))", (a=d7, d=d13)), Structure("a* d*", "))", (a=d4, d=d14))]
f	[Structure("a* d*", "))", (a=d4, d=d14)), Structure("a* d*", "))", (a=d7, d=d13))]

Figure 2.5: The values of bindings in the where clause of `ugate`

appear in `ugate`. When a domain in a `Structure` definition has a tilde, that domain should have a list of `Structure` objects as its value like `d`, `e`, and `f` in this case. When this happens, BioCRN concatenates the structural specification given in that list and puts the resulting structure in the place of that domain. For example, the two `Structure` objects being specified in the `ugate` function now have the following structure (Fig. 2.6).

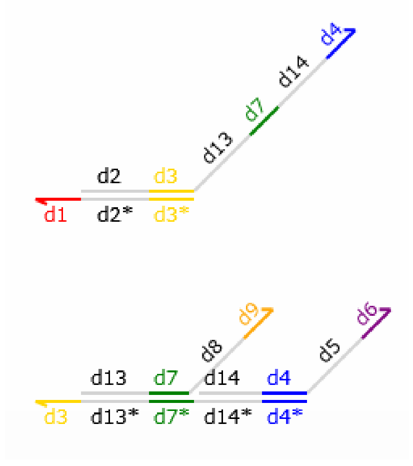


Figure 2.6: G_i (above) and T_i (below) molecules from the `ugate` function. Figure generated using Visual DSD [19].

Now, back in the `unimolecular` function, these two `Structure` objects computed in `ugate` are put into the variables `g` and `t` respectively. Then, the return value of `unimolecular` is simply `infty(g) + infty(t)`, which is a `Solution` object that contains the two species described by `g` and `t` (`infty` is a function that turns a `Structure` object into a `Solution` object that contains only that molecule).

We are finally back in the `main` function, but in order to compute the final return value, we need to repeat the same procedure as above for the second reaction in `crn`. We will omit that second round here because it would be too tedious and laborious to follow it step by step. However, it is clear that the second round will also return a `Solution` object and thus the value of `map(unimolecular, unirxn(crn))` would be a list of `Solution` objects. Finally applying `sum` on that list, we will obtain a `Solution` object that contains every molecule that appears at least once in any `Solution` object in that list. The return value of `main` will be, as promised, a `Solution` object that contains all auxiliary molecules needed to implement the reactions in the given CRN.

Note that this example does not reveal how the above translation scheme will deal with a bimolecular reaction. However, that should be fairly easy to figure out for those that have followed the discussion above, and we leave it as an exercise for the reader.

Chapter 3

Reaction Enumerator

As we saw in the last chapter, the BioCRN compiler produces a structural design for formal and auxiliary molecules, but does not explain how those molecules are supposed to interact with each other. In order to study the chemical pathways involving the produced molecules, we use a software tool called reaction enumerator. Although building a reaction enumerator was not a part of our research, we need to introduce it briefly here because it is necessary for understanding our next project, the CRN implementation verifier.

3.1 Motivation

Despite the diversity of procedures that researchers go through to build a molecular program, they all arrive at the same form in the end, which is the specification for a set of molecules. Then, those molecules are synthesized according to the specification and put into a test tube for experiments. Unfortunately, many molecular programs are error-prone and they exhibit unexpected behaviors that are difficult to understand, because it is hard to figure out in an experimental setting which chemical pathways are active in the test tube. To this end, the use of simulation techniques can be very helpful.

As pointed out in the first chapter, DNA molecules have a simple behavior (relative to other organic molecules) that is easily predicted. In fact, most behaviors seen in today's DNA programs can be predicted solely by the Watson-Crick base pairing. Thus, we can imagine a simulator that, given a set of molecule specifications, enumerates reactions that can occur between those molecules according to the Watson-Crick base pairing. With this "reaction enumerator," one can discover potential spurious pathways even before one performs experiments.

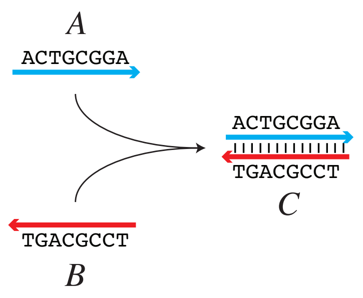


Figure 3.1: A simple example

3.2 Enumeration

A reaction enumerator takes as input a set of molecules and enumerates all reactions that can occur between them. We will call these input molecules the first generation. Note that as enumeration proceeds, we will often discover "new" molecules which are not the first generation but get produced as a result of interactions between the first generation molecules. We can call these new molecules as well as the first generation molecules the second generation. For instance, if the reaction enumerator takes as input the A and B molecules from Fig. 3.1, the first generation will be $\{A, B\}$ but the second generation will be $\{A, B, C\}$, because we discover the new molecule C by the interaction of A and B. Similarly, if we still discover new molecules from interactions between the second generation molecules, we can call them the third generation, and so on. If we continue this procedure until we discover no new molecules, we will clearly enumerate all reactions and species that arise from the given first generation molecules.

The above enumeration algorithm is summarized in the following pseudocode.

```
1 current_generation = input molecules
2 while
3    next_generation = current_generation
4    enumerate all reactions between the molecules in current_generation
5    next_generation += newly discovered molecules
6    if current_generation == next_generation then break
7    current_generation = next_generation
8 end while
```

However, this algorithm is not clearly defined until we explain how to perform the enumeration on the fourth line of the code, i.e., until we define the semantics for enumeration.

3.3 Semantics for Enumeration

3.3.1 Elementary Domain Step Semantics

The following semantics for enumerating DNA reactions is due to Brian Wolfe [43]. Wolfe's reaction enumerator is incapable of dealing with pseudoknotted structures, so if any of the following rules produces a pseudoknotted molecule, that product will be ignored.

3.3.1.1 1-1 Binding

A 1-1 binding reaction is a reaction between two complementary unpaired domains within a single DNA molecule.

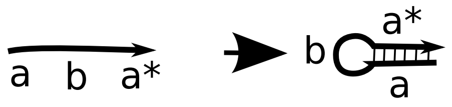


Figure 3.2: 1-1 binding. Figure from [43].

3.3.1.2 2-1 Binding

A 2-1 binding reaction is a reaction between two complementary unpaired domains from two different DNA complexes. The resulting product is one connected molecule.

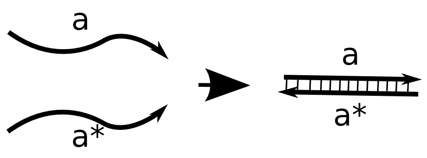


Figure 3.3: 2-1 binding. Figure from [43].

3.3.1.3 1-1 Open

A 1-1 open reaction is a reaction where a series of adjacent, paired domains with a total length less than a threshold (usually 5-8 nts) dissociates and the resulting complex is still connected.

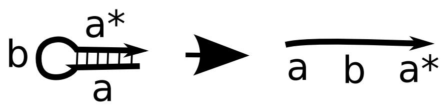


Figure 3.4: 1-1 open, assuming that the length of the opening helix is below the threshold constant. Figure from [43].

3.3.1.4 1-2 Open

A 1-2 open reaction is a reaction where a series of adjacent, paired domains with a total length less than a threshold (usually 5-8 nts) dissociates, leaving two distinct complexes.

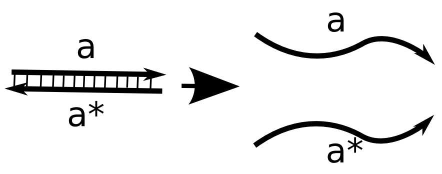


Figure 3.5: 1-2 open, assuming that the length of the opening helix is below the threshold constant. Figure from [43].

3.3.1.5 3-way Branch Migration

A three-way branch migration is a displacement reaction where an unpaired string of domains replaces an adjacent double-stranded region which is fully complementary. This was the building idea for the translation scheme in [39].

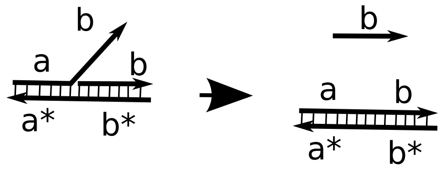


Figure 3.6: 3-way branch migration. Figure from [43]. Shown is the 1-2 variant where one strand dissociates after the branch migration step, but there is also a 1-1 variant in which no strand can dissociate.

3.3.1.6 4-way Branch Migration

A four-way branch migration is a reaction that rearranges double-stranded four-arm junctions. All strands involved start and end completely hybridized.

3.3.1.7 Transient and Resting States

Finally, we have to mention one important caveat that Wolfe's enumerator adopts the kinetic assumption that unimolecular reactions (reactions with a single reactant) are significantly faster than bimolecular reactions (reactions with two reactants). We say that a DNA complex is in a transient state if it has an outgoing unimolecular reaction, because the complex will quickly turn into some other configuration. On the other hand, a DNA complex without any outgoing unimolecular reaction

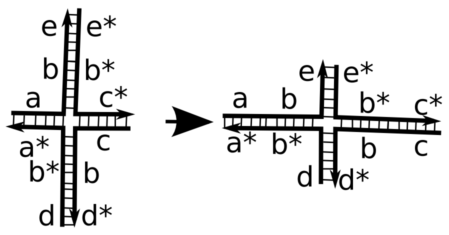


Figure 3.7: 4-way branch migration. Figure from [43]. Shown is the 1-1 variant where no strand dissociates after the branch migration step, but there is also a 1-2 variant.

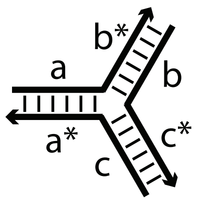


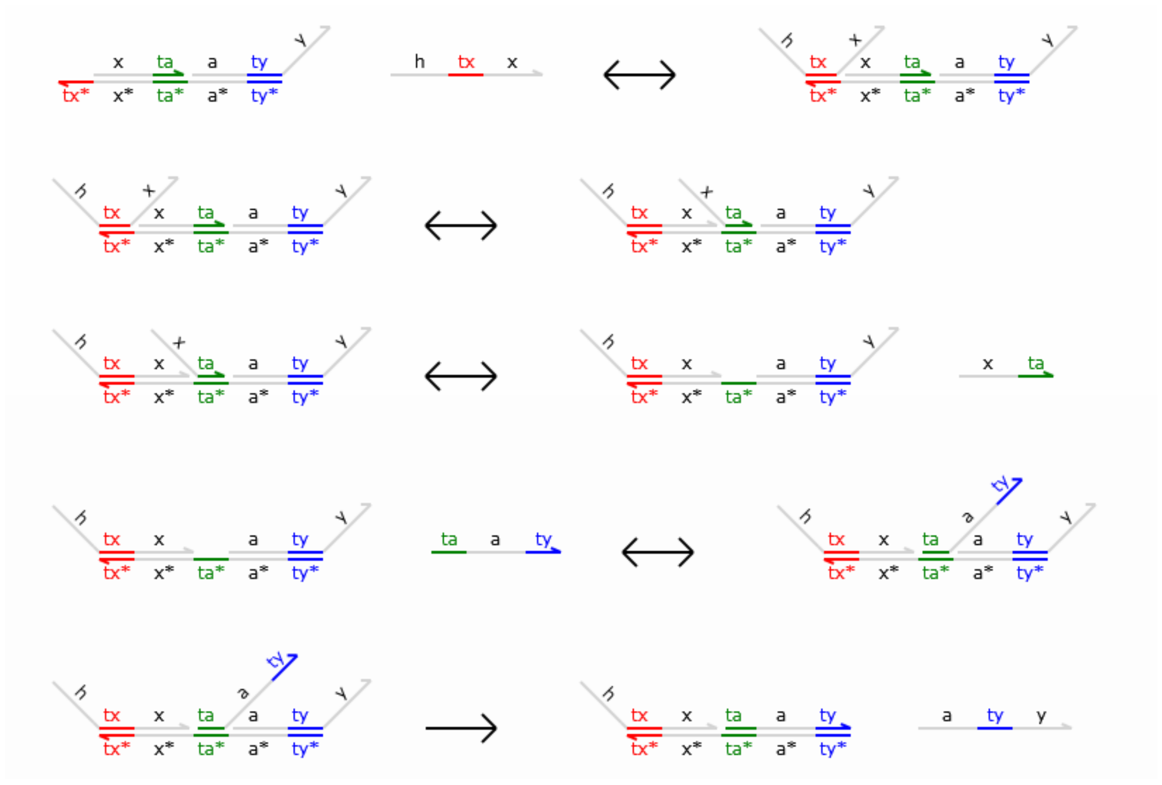
Figure 3.8: A three-way juction

is relatively stable and is said to be in a resting state. In fact, this notion must be generalized to a strongly connected set of states with no outgoing unimolecular reactions. (Here, we say that two states are strongly connected if there are paths of 1-1 unimolecular reactions connecting them in both directions.) In Wolfe's semantics, molecules in a transient state are not allowed to participate in bimolecular reactions, because it is highly likely that the outgoing unimolecular step will always occur first.

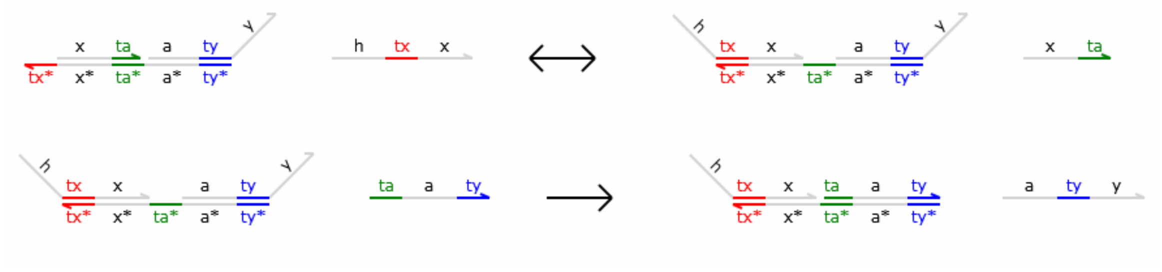
Note that the introduction of this kinetic assumption greatly reduces the number of enumerated reactions. Sometimes, a system that yields an infinite number of DNA complexes and reactions without this kinetic assumption is nicely reduced down to a finite system with the assumption. For instance, a system that contains many copies of three single-stranded molecules "a b*", "b c*", and "c a*", which can form only one type of terminal species under the above kinetic restriction (Fig. 3.8), can polymerize forever if the restriction were removed. Interestingly, there are some reaction enumerators (e.g. Microsoft Research's Visual DSD) that attempt to avoid this issue by syntactically prohibiting systems that have a potential of having infinite-length polymers, but they do so at the expense of the ability to express some interesting systems.

3.3.2 Resting States Semantics

Because the above semantics can still generate a huge number of reactions and species (and sometimes potentially infinite), we often truncate the result using the same kinetic assumption about unimolecular and bimolecular reactions. This is done by enumerating all pathways that visit resting states exactly twice, namely at the start and at the end, and compressing them into single-step reactions by ignoring what happens in the middle. The end result is a reaction graph where all molecules that appear are in resting states (Fig. 3.9). Clearly, the number of enumerated reactions is further decreased and it becomes much easier to investigate the result.



(a) Elementary domain step semantics ("detailed" in DSD)



(b) Resting states semantics ("infinite" in DSD)

Figure 3.9: The difference between the elementary domain step semantics and the resting states semantics. Figures generated using Visual DSD [19] whose semantics agrees with Wolfe semantics in this example.

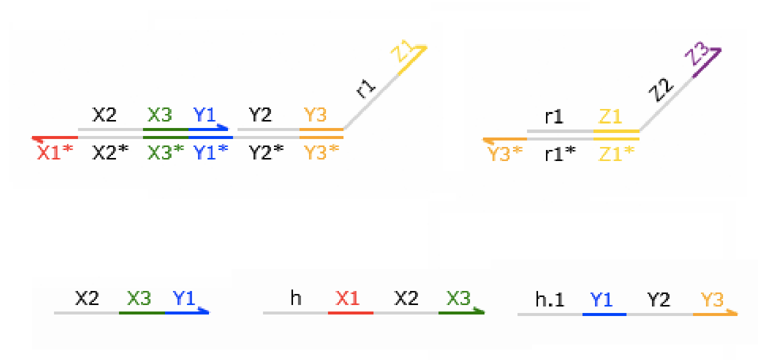
Our work on the CRN implementation verification uses the Wolfe enumerator with this "resting states semantics."

3.3.3 Others

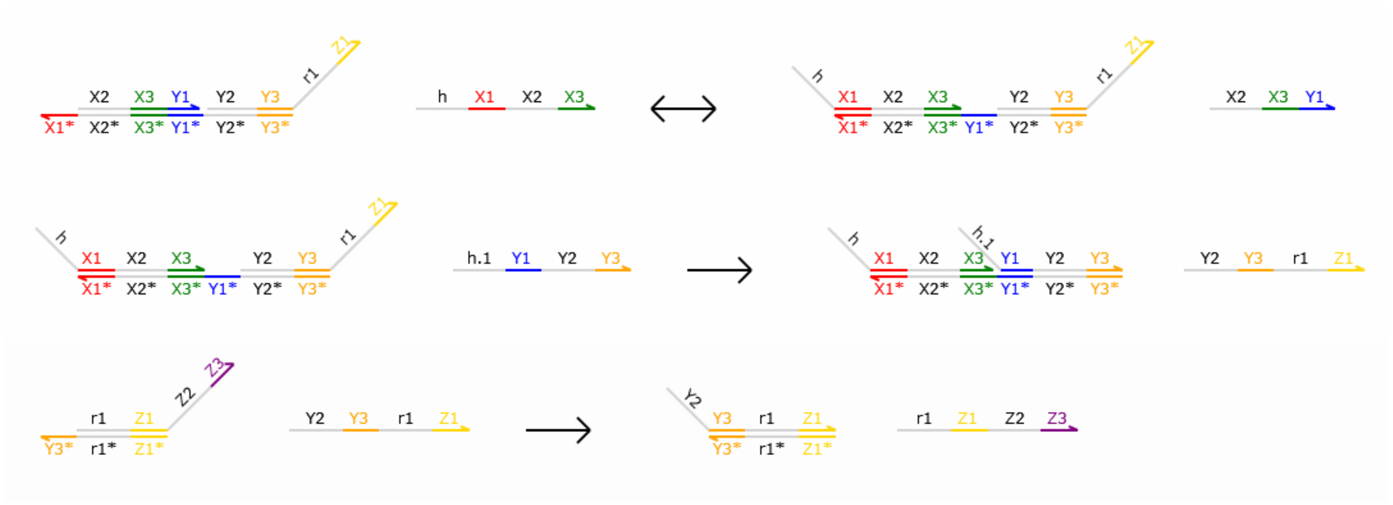
Other than the Wolfe enumerator, there are some other reaction enumerators that may or may not use different semantics from the one described above. For instance, Karthik Sarma at the Winfree lab at the California Institute of Technology is rewriting and extending Wolfe' enumerator to more easily accommodate additional domain-level elementary steps, such as remote toehold mediated branch migration [13]. Also, Visual DSD [19], which is a tool developed and maintained by Microsoft Research, includes reaction enumeration as one of its features. It provides a very convenient graphical user interface, but it has a weakness in that it imposes a limitation on the types of structures and reactions that are permitted. For example, hairpins (most notably the motif from [44]), highly branched structures, and four-way branch migration cannot be represented in their model.

3.4 Examples

To help understanding, some more examples of reaction enumeration are given in Fig. 3.10 and Fig. 3.11.

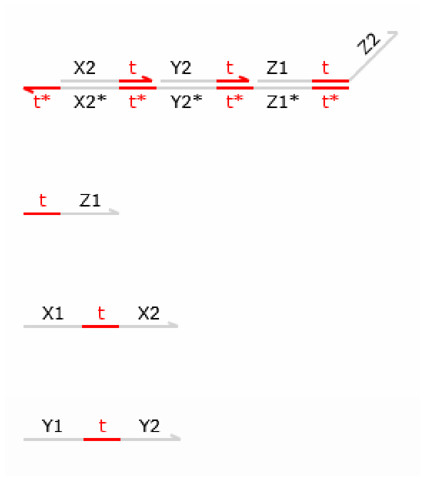


(a) Input molecules

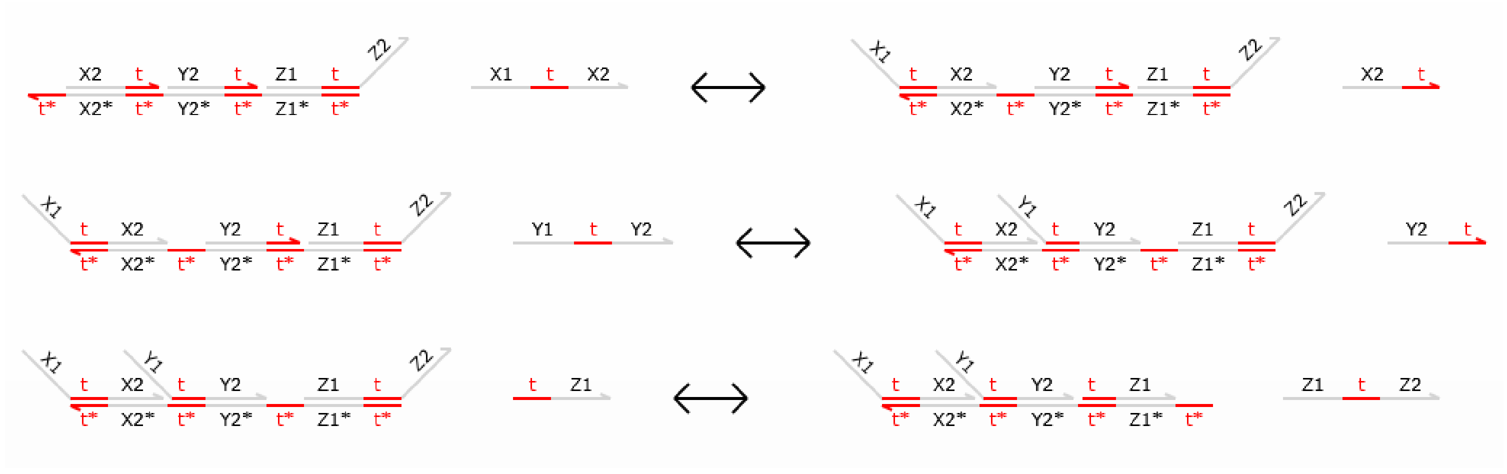


(b) Enumerated reactions

Figure 3.10: Enumeration example. The input is the implementation of $X + Y \rightarrow Z$ using the translation scheme from [39]. Figures were generated using Visual DSD [19] using the "infinite" semantics (Visual DSD equivalent of the resting states semantics).



(a) Input molecules



(b) Enumerated reactions

Figure 3.11: Enumeration example. The input is the implementation of $X + Y \leftrightharpoons Z$ using the translation scheme from [27]. Figures were generated using Visual DSD [19] using the "infinite" semantics (Visual DSD equivalent of the resting states semantics).

Chapter 4

CRN Equivalence

In Chapters 4, 5, and 6, we will discuss methods for verifying CRN implementations. Although partly motivated by the compiler from Chapter 2, our verification techniques are independent of it and self-contained.

4.1 Motivation

Before we go into main discussion, we shall briefly describe our motivations behind this work. For this, we shall recall the translation scheme that we introduced in the first chapter (Fig. 1.5, [39]), where the reaction $X_1 + X_2 \to X_3$ was implemented using the following three reactions. Note that the translation scheme will only output how to design formal and auxiliary species, but we can obtain the following reactions using the reaction enumerator described in the previous chapter.

$$X_1 + L_i \rightleftharpoons H_i + B_i$$

$$X_2 + H_i \longrightarrow \text{waste}_1 + O_i$$

$$O_i + T_i \longrightarrow \text{waste}_2 + X_3$$

To recapitulate the points needed to claim the correctness of this implementation, observe how the first step is reversible as opposed to the other two steps. This ensures that none of the X_1 molecules is permanently consumed when X_2 is not also present in the system. If the first step were irreversible like the other two reactions, then in the absence of X_2, the system will behave as if there were a decaying reaction for X_1 (because the X_1 molecules will turn into H_i, which is not one of the species that we are interested in). However, the researcher might mistakenly think that the implementation is fine with an irreversible first step, since the system would still seem to work when X_1 and X_2 are both present. In general, proving the correctness of a CRN implementation involves a careful study of many different initial conditions, which is not always easy to do. Moreover, the number of distinct initial conditions that need to be checked is usually infinite (the initial state is allowed to have any number of molecules), and thus no technique based on explicit enumeration of finite state spaces can do the job.

The situation is worse when we implement more complex CRNs with a large number of reactions. The above example implemented only one reaction. How difficult would it be to verify an implementation of dozens of reactions? In that case, we would need to prove both that the resulting system implements each reaction correctly and that the auxiliary molecules and the intermediate chemical species used for the construction do not interfere with one another in an unintended manner. Note that the translation scheme from [39] is "modular" in the sense that it has a module with which it implements each desired reaction independently. For these modular constructions, it is important to verify that modules implementing different reactions do not interfere with each other. But at the same time, one might want to optimize these constructions in a way that different modules share some fuel and intermediate species (e.g. reactions $A + B \rightarrow C$ and $A + B \rightarrow X + Y$ are likely to be more efficiently implemented using shared fuels and intermediates). One might even want to use an intrinsically non-modular translation scheme (although there is no such scheme that has been reported thus far). Thus, we have numerous questions to ask before we come up with the final verdict to the correctness of an implementation, and as the size of the target CRN becomes larger, it becomes harder to imagine doing this verification manually.

A skeptical reader might still argue that the correctness can be easily proved at the translation scheme level, i.e., it is easier to prove the correctness of the general method of translation than that of each individual compilation instance. In fact, the translation scheme from [39] can be proved to be generally correct because it was deliberately designed such that there would be no spurious interaction between the auxiliary molecules. However, we should not generalize from that one scheme that all translation schemes will allow for nice and clean correctness proofs. For instance, there are translation schemes that only work for a certain class of CRNs (e.g. [28, 24]) but are much more efficient when they do. Or else, we could go so far as to imagine that there could be randomized translation schemes that only succeed with some probability, and then must be checked.

In any case, the problem of verifying CRN implementations seems to be in need of some automated techniques because sometimes those implementations contain very subtle bugs. Fig. 4.1 shows a translation scheme reported to have arisen during the preparation of [27], implementing the reaction $X + Y \rightarrow A + B$. It seems from this figure that the translation scheme is correctly implementing the target reaction. In fact, for this particular target reaction, it can be shown that this scheme works correctly. Since this scheme correctly implements the most general form of a bimolecular reaction, we might want to conclude here that this scheme will be correct for any bimolecular reaction. Could that be true?

Fig. 4.2 shows the same translation scheme implementing a different reaction $X + Y \rightarrow A + A$.

While it might seem from this figure alone that this implementation works correctly for this new reaction as well, there is a very serious problem in this implementation, as illustrated by Fig. 4.3. In that spurious pathway, only one A molecule is produced as opposed to the desired number two, which indicates that there is a serious logical error in this translation scheme. The astute reader will also notice that this translation scheme will also fail on the CRN $\{X+Y \to A+B,\ Z+W \to B+C\}$, although each reaction is individually compiled correctly. Thus, it is not sufficient to verify each reaction independently. While the corrected version of this translation scheme can be found in [27], this example shows that it can be sometimes very difficult to detect compilation errors by eye.

4.2 The Meaning of Correctness

However, the notion of correctness that we want to test for CRN implementations is presently not very well-defined, since the problem has not been extensively studied before. Clearly, every reaction in the target CRN should somehow be implemented and the reactions that were not in the target CRN should not suddenly appear, but it is not obvious how to define such concepts in a logically rigorous way. Throughout the rest of this chapter, we will formally define the notion of correctness of a CRN implementation in the language of mathematics. In the course, we shall see that this

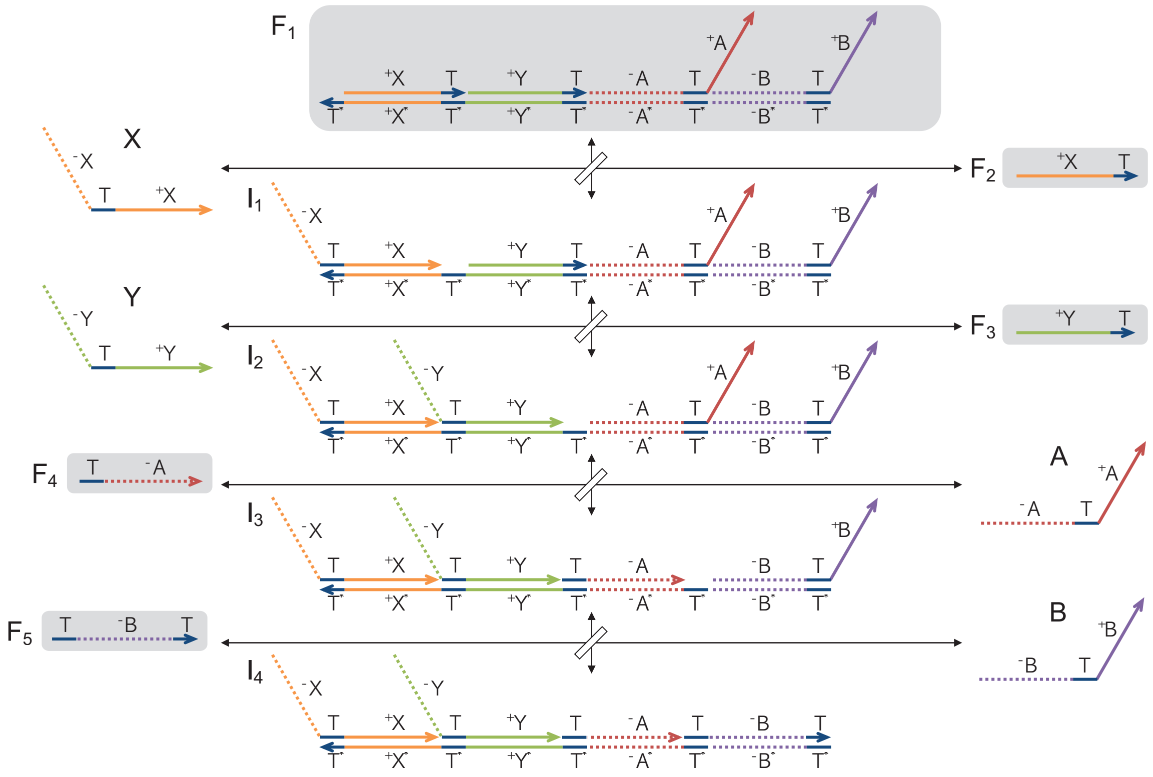


Figure 4.1: An incorrect translation scheme applied to $X+Y \to A+B$ (no exhibited bug). Figure taken and modified from [27].

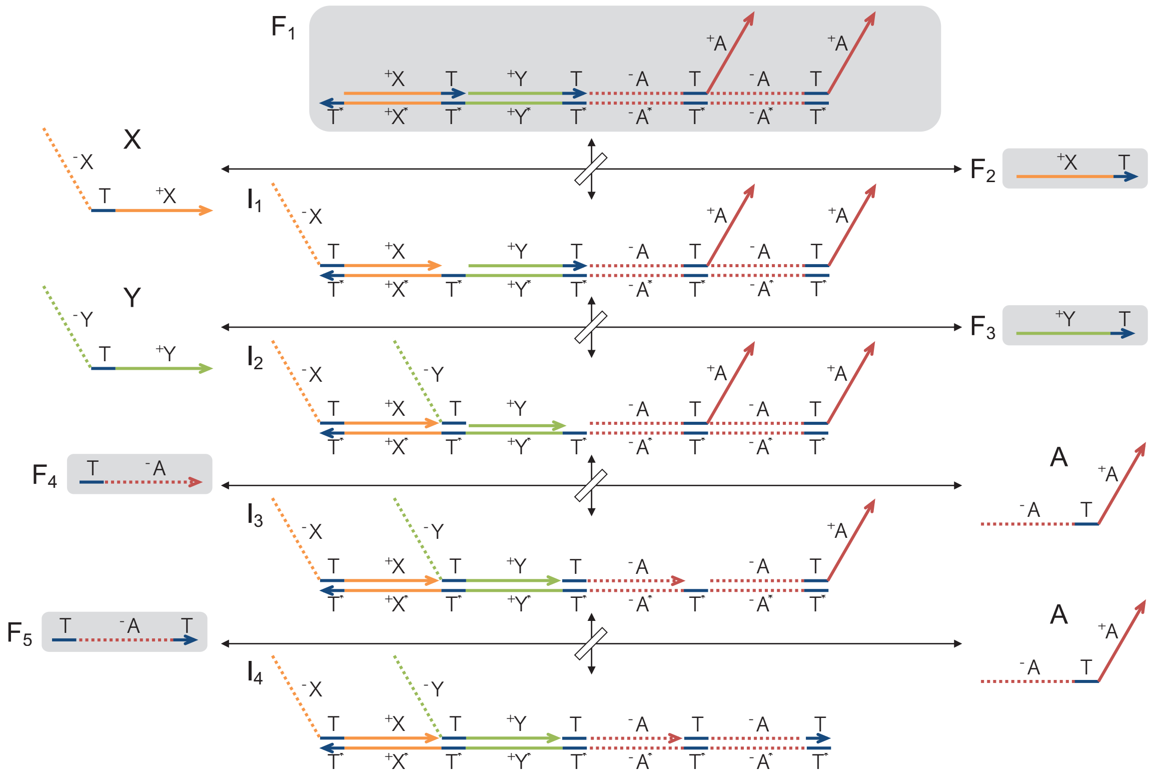


Figure 4.2: An incorrect translation scheme applied to $X + Y \to A + A$ (intended pathway). Figure taken and modified from [27].

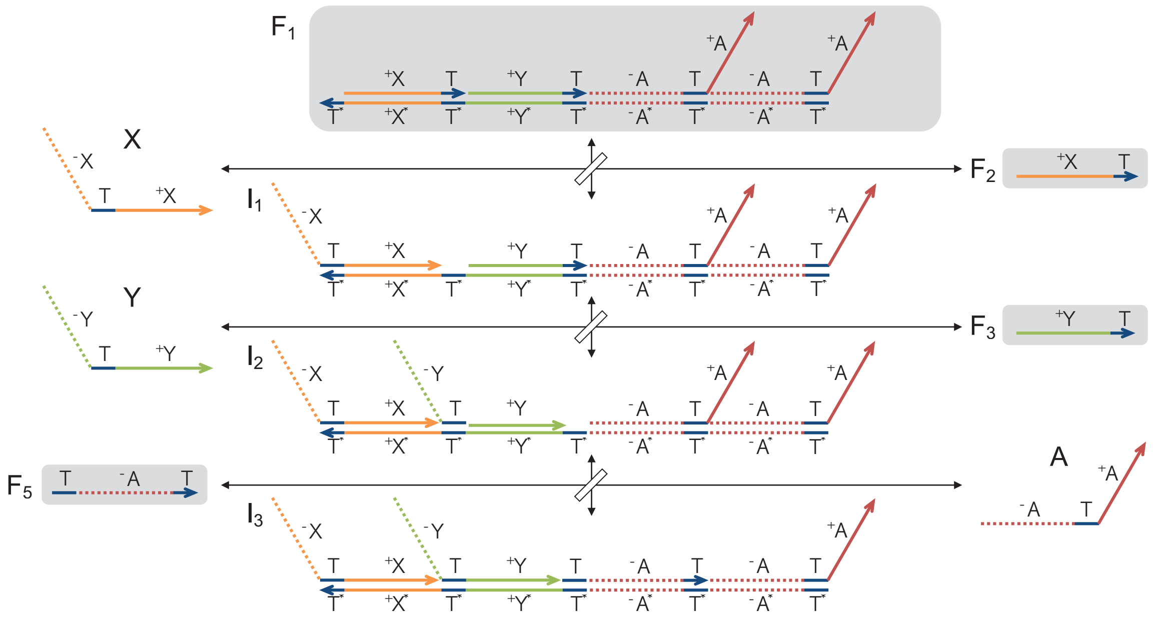


Figure 4.3: An incorrect translation scheme applied to $X + Y \to A + A$ (spurious pathway). Figure taken and modified from [27].

seemingly easy task is in fact very subtle and difficult.

We shall begin by noticing that our problem essentially concerns the comparison of two CRNs. There are two CRNs involved in our verification process, which are the original target CRN and the CRN that models the implemented DNA system, or the *implemented CRN*. For instance, in the example of Fig. 1.5, our task is to compare the target CRN $\{X_1 + X_2 \to X_3\}$ with the implemented CRN $\{X_1 + L_i \rightleftharpoons H_i + B_i,\ X_2 + H_i \to \text{waste}_1 + O_i,\ O_i + T_i \to \text{waste}_2 + X_3\}$. The core of our problem is to develop an algorithm that can decide which reactions are really "implemented" in the implemented CRN. Then we could test the desired correctness by checking whether the reactions "implemented" in the implemented CRN are exactly those reactions from the target CRN.

We will attempt to answer this problem by studying a more general problem, which we call *the CRN equivalence problem*. Instead of trying to exploit the intrinsic properties of our initial problem, such as that the species set of one CRN is a subset of the species set of the other CRN (in the above example, $\{X_1, X_2, X_3\}$ is a subset of $\{X_1, X_2, X_3, L_i, B_i, H_i, O_i, T_i, \text{waste}_1, \text{waste}_2\}$), we choose to neglect all such conditions and face the problem from the most general standpoint. In words, our new problem will be formulated as follows; test whether two given CRNs behave equivalently with respect to a given set of special species (which in the above example would be $\{X_1, X_2, X_3\}$), where the meaning of equivalent behavior is yet to be defined. Note that the original problem of verifying CRN implementations is a special case of this new problem. By solving a more general problem, we hope not only to allow for a wider range of applications but also to be able to safely apply the method to any translation scheme that might employ singular techniques.

Since the BioCRN compiler outputs only the structural information of DNA molecules rather than the implemented CRN, we need to first construct the implemented CRN using the reaction enumerator before running our verifier. As discussed in the previous chapter, software tools such as Brian Wolfe's reaction enumerator or Microsoft Research's Visual DSD can be used. These tools simulate and generate the list of all possible reactions that can occur in the given system, which in our case will be provided by BioCRN.

We have to mention one important caveat about our work. In this thesis, our main objective is to test the logical equivalence of the behaviors of two CRNs, and it is beyond the scope of this thesis to analyze the kinetics of the CRNs, i.e., we will not try to prove that the reactions are implemented with the correct reaction rates. Our goal is to check whether the two CRNs contain the same pathways, not whether those pathways occur at the same speed.

4.3 Basic Concepts

In order to prove properties about chemical systems, it is necessary to first confine them in some mathematical abstraction in which propositions and proofs can be presented. Thus we shall begin by proposing purely mathematical definitions for some of the most fundamental concepts in chemistry such as species, reactions, and CRNs. We will use the letter Σ to denote the namespace from which we will assign names to species. By default, Σ will contain all upper case and lower case letters unless stated otherwise. However, this is a mere convention that we adopt for the sake of convenience. Note that Σ can be infinite in principle. Lastly, we know that our implemented CRNs will have specially designated species called the **formal species**. These are the species that are inherited[1] from the original CRN. To make this distinction, we will also have $\mathcal{F} \subseteq \Sigma$ which is the set of formal species. In the rest of the thesis, we will adopt a species naming strategy that assigns upper case letters to formal species and lower case letters to non-formal species.

Definition. The elements of Σ are called **species**. The elements of $\mathcal{F}$ are called **formal species**.

Definition. A **state** is a multiset of species. If every species in a state S is a formal species, then S is a **formal state**.

Definition. If S is a state, Formal(S) denotes the multiset that consists of exactly all the formal species in S.

Definition. A **reaction** is a pair of multisets of species (R, P) and it is **trivial** if $R = P$. Here, R is called the set of **reactants** and P is called the set of **products**. We say that the reaction (R, P) **can occur** in the state S if $R \subseteq S$. If both R and P are formal states, then (R, P) is a **formal reaction**.

Definition. If (R, P) is a reaction that can occur in the state S, we write $S \oplus (R, P)$ to denote the resulting state $(S \setminus R) \cup P$. $\oplus$ is left-associative.

Definition. A **CRN** is a (nonempty) set of nontrivial reactions. A CRN that contains only formal reactions is called a **formal CRN**.

Definition. A **pathway** p of a CRN $\mathcal{C}$ is a (finite) sequence of reactions $(r_1, \ldots, r_k)$ with $r_i \in \mathcal{C}$ for all i. It is said to be able to occur in the state S if $S \oplus r_1 \oplus r_2 \oplus \cdots \oplus r_k = S'$ for some state S'. Note that given any pathway, we can find a unique **minimal initial state** associated with it. Conveniently, the **initial state** of a pathway $p = (r_1, \ldots, r_k)$ will simply mean its minimal initial state S, and the **final state** of a pathway will mean $S \oplus r_1 \oplus r_2 \oplus \cdots \oplus r_k$. If both the initial and final states of a pathway are formal, then the pathway is said to be a **formal pathway**. Also, a pathway whose initial and final states are identical is called **futile**.

To help understanding, we shall study some examples. Consider the chemical reaction $2A + B \rightarrow C$. In this formalism, this will be written $(\{A, A, B\}, \{C\})$ since reactions are defined to be pairs of

[1]Since translation schemes might allow multiple implementations of a single formal species (for instance, the scheme in Fig. 1.5 does so by allowing the question-marked "history" domain to be arbitrary), this formulation will not be able to directly handle such translation schemes. This problem will be addressed in detail in section 4.7.

multisets. Here, $\{A, A, B\}$ is called the reactants and $\{C\}$ is called the products, complying with the common perception of the terms. Also, this reaction can occur in the state $\{A, A, A, B, B\}$ but cannot occur in the state $\{A, B, C, C, C, C\}$ because the latter state does not have all the required reactants. If the reaction takes place in the former state, then the resulting state will be $\{A, B, C\}$ and thus we can write $\{A, A, A, B, B\} \oplus (\{A, A, B\}, \{C\}) = \{A, B, C\}$. Note that these notations appear different from the conventional notations of chemistry but are equivalent. For the rest of the discussion, we should interchangeably use the conventional notations with our notations where doing so enhances readability. For instance, we will often write $2A + B \rightarrow C + D$ instead of $(\{A, A, B\}, \{C, D\})$.

The definition of a pathway is subtle and deserves careful attention. Since the definition of $S \oplus r$ asserts that the reaction r can occur in the state S, the pathway $(r_0, \ldots, r_k)$ can occur if and only if all of $r_0, r_1, \ldots, r_k$ can occur in that order. More precisely, this means that r_0 can occur in S, r_1 can occur in $S \oplus r_0$, r_2 can occur in $S \oplus r_0 \oplus r_1$, and so on. For example, consider the pathway that consists of $2A + B \rightarrow C$ and $B + C \rightarrow A$. This pathway cannot occur in the state $\{A, A, B\}$ because even though the first reaction can occur in that state, the resulting state from the first reaction $\{C\}$ will not have all the reactants required for the second reaction. In contrast, the pathway can clearly occur in the state $\{A, A, B, B\}$.

Lastly, note that we cannot directly express a reversible reaction in this formalism. Thus, a reversible reaction will be expressed using two different reactions. For example, the reversible reaction $A \rightleftharpoons B$ will be expressed by having two irreversible reactions $A \rightarrow B$ and $B \rightarrow A$.

4.4 Preparing the Implemented CRN

The output from the BioCRN compiler and the reaction enumerator consists of largely three parts. First, the compiler provides us with the correspondence between the formal species of the original CRN and the formal species of the implemented CRN. Second, it tells us which species are the fuel that should be maintained at high concentration. Third, the reaction enumerator gives us the list of reactions that can occur between the species that exist in the system.

In preparing the implemented CRN to fit our formalism, it is obvious what needs to be done with the first and the third part of the input. The formal species given in the input become our $\mathcal{F}$, and the enumerated reactions become our CRN. However, how can we express that the "fuel" species are maintained at high concentration? The answer is simple. We simply add spontaneous reactions that produce those fuel species. Fig. 4.4 shows how the implemented CRN should be preprocessed before it is tested under our notions of equivalence that are soon to be defined.

$$\begin{array}{cc}
X_1 + L_i \rightleftharpoons H_i + B_i & X_1 + L_i \rightleftharpoons H_i + B_i \\
X_2 + H_i \longrightarrow w_1 + O_i & X_2 + H_i \longrightarrow w_1 + O_i \\
O_i + T_i \longrightarrow w_2 + X_3 & O_i + T_i \longrightarrow w_2 + X_3 \\
\text{formal}: X_1, X_2, X_3 & \emptyset \longrightarrow L_i + B_i + T_i \\
\text{fuel}: L_i, B_i, T_i & \mathcal{F} = \{X_1, X_2, X_3\} \\
\textbf{before} & \textbf{after}
\end{array}$$

Figure 4.4: The implemented CRN from Fig. 1.5 before and after the preprocessing

4.5 Equivalence with Respect to a CRN Observer

Now that we have mathematical formalisms for expressing chemical concepts, we will attempt to formulate a notion of equivalence between CRNs. Clearly, the most natural strategy would be to define it with respect to some physical observer.

The main idea of the notion that we are about to introduce, called equivalence with respect to a CRN observer, is that if two CRNs are different, they should be distinguishable by some external formal CRN which presumably has an expanded set of formal species. This "observer" CRN will observe each of the target CRNs by interacting with it in the same test tube, but it is not given any additional information other than the interactions that it experiences. Thus, it can only distinguish between CRNs by checking which pathways can occur in each environment. (Again, note that we are restricting our analysis to notions that are independent of kinetics.) This notion can be clearly articulated as follows. Note that this definition explains precisely (up to kinetics) the way CRNs are studied in the real world, because as human beings we can only observe CRNs by letting them interact with other well-studied chemical reaction networks.

Definition. Let $\mathcal{C}$ be an arbitrary CRN, $\mathcal{O}$ a formal CRN, and S a formal state. Then, we define the sets $\text{Obs\_NT}_{\mathcal{C},\mathcal{O},S}$ and $\text{Obs\_T}_{\mathcal{C},\mathcal{O},S}$ as follows.

$$\begin{aligned}
\text{Obs\_NT}_{\mathcal{C},\mathcal{O},S} = \{p' \quad &: \quad p' \text{ is a pathway of } \mathcal{O}, p \text{ is a pathway of } \mathcal{C}, \text{ and some non-terminal pathway} \\
&\qquad \text{generated by interleaving } p \text{ and } p' \text{ can occur in } S.\} \\
\text{Obs\_T}_{\mathcal{C},\mathcal{O},S} = \{p' \quad &: \quad p' \text{ is a pathway of } \mathcal{O}, p \text{ is a pathway of } \mathcal{C}, \text{ and some terminal pathway} \\
&\qquad \text{generated by interleaving } p \text{ and } p' \text{ can occur in } S.\}
\end{aligned}$$

Here, a pathway q generated by interleaving a pathway of $\mathcal{O}$ and a pathway of $\mathcal{C}$, with respect to S, is said to be **terminal** if for any pathway q' of $\mathcal{C} \cup \mathcal{O}$ that contains at least one reaction from $\mathcal{O}$, $q + q'$ cannot occur in S. For example, if $\mathcal{O} = \{A \to B\}$ and $\mathcal{C} = \{A \to C\}$, the pathway $(A \to B)$

with respect to $\{A\}$ is clearly terminal, but it will not be terminal if $\mathcal{C} = \{B \rightarrow i,\ i \rightarrow A\}$.

Intuitively, the condition in the definitions of these two sets means that p' can occur in S with some help from reactions from $\mathcal{C}$. For example, if $\mathcal{C} = \{A \rightarrow i,\ i \rightarrow B,\ C \rightarrow D\}$ and $\mathcal{O} = \{B \rightarrow C,\ D \rightarrow E\}$, then $p' = (B \rightarrow C,\ D \rightarrow E)$ is in $\text{Obs\_T}_{\mathcal{C},\mathcal{O},\{A\}}$ because $p = (A \rightarrow i,\ i \rightarrow B,\ C \rightarrow D)$ which is a pathway of $\mathcal{C}$ can be interleaved with p' to form a terminal pathway $(A \rightarrow i,\ i \rightarrow B,\ B \rightarrow C,\ C \rightarrow D,\ D \rightarrow E)$ which can occur in $\{A\}$.

Definition. Two CRNs $\mathcal{C}_1$ and $\mathcal{C}_2$ are **equivalent with respect to a CRN observer** if for any formal CRN $\mathcal{O}$ and any formal state S, $\text{Obs\_NT}_{\mathcal{C}_1,\mathcal{O},S} = \text{Obs\_NT}_{\mathcal{C}_2,\mathcal{O},S}$ and $\text{Obs\_T}_{\mathcal{C}_1,\mathcal{O},S} = \text{Obs\_T}_{\mathcal{C}_2,\mathcal{O},S}$.

For example, it is easy to see that the CRNs $\{A \rightarrow B\}$ and $\{A \rightarrow i,\ i \rightarrow B\}$ or the CRNs $\{A + B \rightarrow C + D\}$ and $\{A \rightarrow i,\ i \rightarrow A,\ i + B \rightarrow j,\ j \rightarrow C + k,\ k \rightarrow D\}$ are equivalent in this sense. In contrast, the CRNs $\mathcal{C}_1 = \{A \rightarrow i,\ i \rightarrow C,\ C \rightarrow j,\ j \rightarrow B\}$ and $\mathcal{C}_2 = \{A \rightarrow C\}$ are not equivalent because $\text{Obs\_NT}_{\mathcal{C}_1,\{B \rightarrow A\},\{A\}}$ contains the pathway $(B \rightarrow A)$ but $\text{Obs\_NT}_{\mathcal{C}_2,\{B \rightarrow A\},\{A\}}$ does not. However, they will again become equivalent if we add $C \rightarrow B$ to $\mathcal{C}_2$.

But why do we distinguish between the set of terminal pathways and the set of non-terminal pathways? This is because we cannot recognize the existence of decaying reactions otherwise (or in fact, any reaction that only decreases the molecular count of each species). For example, consider two CRNs $\mathcal{C}_1 = \{A \rightarrow \emptyset\}$ and an empty CRN $\mathcal{C}_2$. Clearly, these should not be deemed equivalent in any case, but if we choose not to distinguish between terminal and non-terminal pathways, they will indeed be considered equivalent. For example, consider the following set.

$$\text{Obs}_{\mathcal{C},\mathcal{O},S} = \{p' \ :\ p' \text{ is a pathway of } \mathcal{O},\ p \text{ is a pathway of } \mathcal{C}, \text{ and some pathway generated by interleaving } p \text{ and } p' \text{ can occur in } S.\}$$

Note that for any choice of $\mathcal{O}$ and S, $\text{Obs}_{\mathcal{C}_1,\mathcal{O},S}$ will be identical to $\text{Obs}_{\mathcal{C}_2,\mathcal{O},S}$. However, since we chose to make a distinction between terminal and non-terminal pathways, we can now distinguish between these two CRNs because $\text{Obs\_T}_{\mathcal{C}_1,\{A \rightarrow A+B\},\{A\}}$ contains $(A \rightarrow A+B)$ whereas $\text{Obs\_T}_{\mathcal{C}_2,\{A \rightarrow A+B\},\{A\}}$ is empty.

Unfortunately, it turns out that this notion of equivalence is weaker than it appears and cannot distinguish between some obviously different formal CRNs. Some example would include the CRNs $\{A \rightarrow B\}$ and $\{A \rightarrow B,\ A + A \rightarrow B + B\}$ or the CRNs $\{A \rightarrow B,\ B \rightarrow C,\ C \rightarrow A\}$ and $\{A \rightarrow C,\ C \rightarrow B,\ B \rightarrow A\}$. Although we are currently working in the logical regime where kinetics is ignored, this does not seem to be good because translation schemes are usually designed with some kinetic considerations in mind and those considerations will be in vain if we consider the above CRNs to be equivalent. Thus, if there exists a notion that solves this problem without appealing to a detailed kinetic analysis, then it will be better to switch to that notion.

In the following sections, we will attempt to establish a stronger notion of equivalence that, when reduced to formal CRNs, becomes the identity function. That is, formal CRNs $\mathcal{C}_1$ and $\mathcal{C}_2$ will be considered equivalent if and only if $\mathcal{C}_1 = \mathcal{C}_2$.

4.6 Equivalence Based on Pathway Decomposition

Intuitively thinking, the weakness of a CRN observer comes from the fact that a CRN observer cannot distinguish between a one-step reaction and a series of reactions. For instance, if a CRN observer observes the reaction $A \to C$ from the CRN $\{A \to B,\ B \to C,\ A \to C\}$, it cannot tell whether it has observed the $A \to C$ reaction or the $(A \to B,\ B \to C)$ pathway. This is because although the CRN observer did not see the intermediate B molecule, it cannot be sure whether it was because there was no B molecule or it was that there was a B molecule but it simply did not choose to interact with the CRN observer. For this reason, the CRN observer cannot distinguish between the CRNs $\{A \to B,\ B \to C,\ A \to C\}$ and $\{A \to B,\ B \to C\}$.

However, it is not so easy to think about this "one-step reachability" when the given CRN is not a formal CRN. For example, we might be tempted to declare that pathways that go through an intermediate formal state (when occurring from its minimal initial state) are not encoding a one-step behavior. This approach draws inspiration from pathways like $(A \to i,\ i \to B,\ B \to j,\ j \to C)$ where the existence of the intermediate formal state $\{B\}$ shows that it is a composition of smaller pathways $(A \to i,\ i \to B)$ and $(B \to j,\ j \to C)$. Similarly, this approach will also be able to catch that the pathway $(A \to i,\ i \to B,\ A \to i,\ i \to B)$ is not one-step because of the intermediate formal state $\{A, B\}$. However, it does not take long before we notice that this approach is not a complete solution, because it cannot rule out $(A \to i,\ A \to i,\ i \to B,\ i \to B)$ which is behaviorally nearly the same as $(A \to i,\ i \to B,\ A \to i,\ i \to B)$.

A new approach that we are about to introduce, called equivalence based on pathway decomposition, provides an elegant solution to this problem. Unlike the CRN observer approach, it exploits the fact that most CRN implementations (in fact, all known implementations at present) share a nice property that we call modularity. It means that any pathway that is implementing a formal reaction consumes only formal species and fuel species and produces only formal species and some inert "waste" species. Modular CRNs are nice in that they allow us to concentrate on these modules that implement the target reactions and not worry about the byproducts of those modules. That is, once a module takes place as a whole, there is no room for spurious interaction between this module and other modules because all it has produced are the desired formal species and some chemically inert species.

4.6.1 Types of Species

Now imagine that we can somehow filter out the fuel and waste. Then, under the assumption that the given implementation is modular, every module will only consume and produce formal species. This will prove to be a very nice property when we discuss the ways to analyze these modules later, so we will first establish a way to filter out the fuel and waste.

Generally, the species of an implemented CRN can be classified in four categories. First, **formal species** are the species inherited from the target CRN (X_1, X_2, and X_3 in Fig. 1.5). Second, **fuel species** are the auxiliary species that are assumed to be maintained at high concentration (L_i, B_i, and T_i in Fig. 1.5). Third, **waste species** are the inert species that get produced along with the desired products of the target chemical reaction (waste$_1$ and waste$_2$ in Fig. 1.5). A refinement of this idea will be defined more rigorously below. Lastly, we call all other species simply **intermediate species** (H_i and O_i in Figure 1.5). Except formal species, intermediate species are the most important species because they carry the information about the computation that is being performed. For instance, an O_i molecule encodes the information that one X_1 and one X_2 molecules have been consumed, because there is no other way that it can be produced.

If we view molecules as information carriers, we need to consider only two of the above four categories. Clearly, a fuel species molecule cannot carry any information because those species are assumed to be maintained at effectively constant concentrations. Thus, producing or consuming one fuel species molecule will not have any effect on the rest of the system. The situation is slightly different for a waste species molecule. It encodes some information in that we can learn which reactions occurred from the existence of waste molecules. However, the fact that they are completely inert prevents that information from being propagated further. For these reasons, we will completely ignore fuel and waste species from our future discussion.

To reflect this point, we perform one more preprocessing step that removes all the fuel and waste species from the implemented CRN, i.e., we will treat the CRN as if those species never existed. To understand this accurately, imagine a world where it is impossible to detect fuel or waste species molecules. In that world, if we ignore kinetics, the CRN on the right in Fig. 4.5 will look like a correct model of the system, even though the one on the left is a more detailed model. In fact, even if we consider kinetics, since fuel species are maintained at constant concentration and waste species are inert, the CRN on the right will still be a valid model, although the effective rate constants may change.

Note that the verifier would need to autonomously distinguish between waste species and intermediate species, whereas formal species and fuel species come in already tagged by the user (or the BioCRN compiler). Moreover, there could be many different criteria for this task. For example, we can define waste species to be the species that do not interact with any other species at all. However, this definition is usually too strict to be used in practice. For instance, waste species interacting with

$$\begin{aligned} X_1 + L_i &\rightleftharpoons H_i + B_i \\ X_2 + H_i &\longrightarrow \text{waste}_1 + O_i \\ O_i + T_i &\longrightarrow \text{waste}_2 + X_3 \\ \emptyset &\longrightarrow L_i + B_i + T_i \end{aligned}$$

before

$$\begin{aligned} X_1 &\rightleftharpoons H_i \\ X_2 + H_i &\longrightarrow O_i \\ O_i &\longrightarrow X_3 \end{aligned}$$

after

Figure 4.5: The implemented CRN from Fig. 1.5 before and after the preprocessing

one another need not be prohibited as long as such interactions do not produce a species that can interact with non-waste species. Thus we propose the following criterion for distinguishing waste species and intermediate species. Note that the definition of an intermediate species is recursive.

Definition. A species is an **intermediate species** if all of the following conditions hold.

1. It is not a formal species or a fuel species.
2. It is a reactant of a reaction that involves at least one formal or intermediate species either as a reactant or a product.

A species that is neither formal, fuel, nor intermediate is called a **waste species**.

Finally, note that fuel species might interact with one another to form non-fuel species. However, because we are assuming infinite supply of fuel species, it means that we will have infinite supply of those non-fuel species as well, which is a trait of a fuel species. Thus, we revise our definition to count such species as fuel species (of course, this does not apply to formal species).

Definition. A non-formal species is called a **fuel species** if at least one of the following conditions hold.

1. It is tagged as a fuel species in the input.
2. It appears as a product of a reaction whose reactants are all fuel species.

This implies that after preprocessing of the input, reactions that do not require any reactant can only produce formal species. Note that the adoption of this broader definition of fuel species may not be conceptually necessary. However, it will turn out later that it greatly helps with the performance of our algorithms.

Thus, we have defined how the four types of species should be distinguished and clearly explained how we should preprocess the implemented CRN such that it only involves formal and intermediate species. From this point, we assume that all CRNs have only formal and intermediate species.

4.6.2 Implementing Formal Reactions as Pathways

Now, because our CRN only has formal and intermediate species, we can think of our modules as pathways that only consume and produce formal species in the net effect (although various intermediate species can be generated in the middle, they should eventually be all removed). Thus, we can introduce the following definition for describing modules.

Definition. We say that a pathway $p = (r_1, \ldots, r_k)$ **implements** a formal reaction (R, P) if it satisfies the following conditions.

1. If S and T are the initial and final states of p, $R = S$ and $P = T$.

2. Let $S_i = S \oplus r_1 \oplus \cdots \oplus r_i$ (so that $S_0, S_1, \ldots, S_k$ are all the states that p goes through). Then, there exists $0 < j \leq k$ such that $\text{Formal}(S_i) \subseteq S$ for all $i < j$ and $\text{Formal}(S_i) \subseteq T$ for all $i \geq j$. When a formal pathway satisfies this condition, we call it **regular**.

While the first condition needs no particular explanation, some time needs to be spent on the second condition because it is somewhat subtle. It articulates that there should be a point in the pathway prior to which we only see the formal species from the initial state and after which we only see the formal species from the final state. The existence of such a turning point allows us to interpret this pathway as a module that implements the reaction (R, P) where the real transition is occurring at that turning point. Before this turning point, because we only see the formal species from the initial state, we can say that we are still in the state R only happening to not detect some of the molecules in it. For a similar reason, we can claim that we are already in state P after the turning point.

Importantly, this condition rules out some counterintuitive implementations such as $(A \to i,\ i \to C + j,\ C + j \to k,\ k \to B)$ or $(A \to i + B,\ i + B \to j + A,\ j + A \to B)$ as implementations of $A \to B$.

4.6.3 Pathway Decomposition

However, the above definition will still interpret pathways like $(A \to i,\ A \to i,\ i \to B,\ i \to B)$ as correctly implementing $A + A \to B + B$. To prevent this, we again take note of the fact that this pathway can be thought of as a composition of two smaller pathways each implementing $(A \to i,\ i \to B)$. Clearly, if a module is merely a composition of smaller modules, it should not be considered a module in the first place. Therefore, we introduce the following notion of decomposability on pathways. For better understanding, Fig. 4.6 provides some examples of decomposable formal pathways.

Definition. A formal pathway p is **decomposable** if p can be partitioned into two nonempty subsequences (which need not be contiguous) that are each formal pathways. If a pathway is not decomposable, it is said to be **prime**.

Now, we have an important theorem that almost immediately follows, which says that any formal pathway in a system can be retrieved from "composing" prime formal pathways. However, one should also notice that this does not imply that every formal pathway has a unique decomposition into prime pathways. For example, the pathway $(A \to i,\ B \to i,\ i \to C,\ i \to D)$ can be decomposed in two different ways: $(A \to i,\ i \to C)$ and $(B \to i,\ i \to D)$, and $(A \to i,\ i \to D)$ and $(B \to i,\ i \to C)$.

Theorem 4.1. Any formal pathway can be generated by interleaving one or more prime formal pathways.

Proof. Trivial. □

Finally, we are ready to articulate what we mean by implementing a CRN using pathways.

Definition. A CRN is **regular** if every prime formal pathway implements some formal reaction. Equivalently, a CRN is regular if every prime formal pathway is regular.

Theorem 4.2. If a CRN $\mathcal{C}$ is regular, for any formal CRN $\mathcal{C}'$, $\mathcal{C} \cup \mathcal{C}'$ is regular.

Proof. This immediately follows from the fact that any pathway of length greater than one that contains a formal reaction is decomposable. □

This means that an irregular CRN contains some formal pathway that cannot be interpreted as implementing any formal reaction or series of reactions. In such cases, we will say that the given CRN does not implement any formal CRN by pathway decomposition (of course, it can still be a non-modular implementation that is correct in some other sense). On the other hand, if the

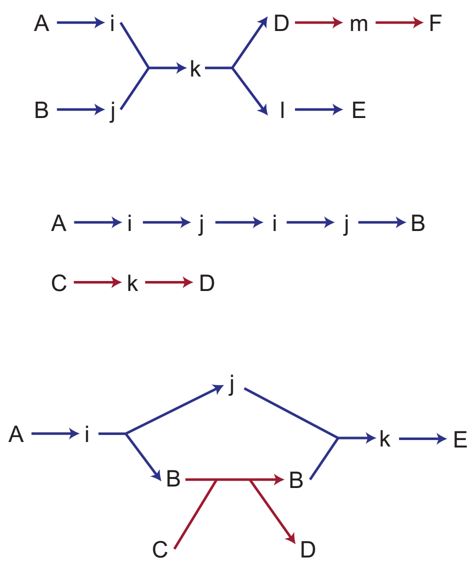


Figure 4.6: Some examples of decomposable formal pathways. The partition of reactions is marked by different colors. Note that only in the last case, a pathway is not regular after decomposition.

given CRN $\mathcal{C}$ is regular, it is now obvious what formal CRN $\mathcal{C}'$ that $\mathcal{C}$ should be interpreted as implementing. We call $\mathcal{C}'$ the **formal basis** of $\mathcal{C}$.

Definition. The set of prime formal pathways in a given CRN is called the **elementary basis** of the CRN. The **formal basis** is the set of (initial state, final state) pairs of the nonfutile pathways in the elementary basis.

One final problem is that since the formal basis and regularity both only concern formal pathways, we might fail to catch a problem that arises with non-formal pathways. To avoid this problem, we introduce the following property.

Definition. Let p be a pathway with a formal initial state and T its final state. Then, a pathway $p' = (r_1, \ldots, r_k)$ is said to be the **closing pathway** of p if p' can occur in T and $T \oplus r_1 \oplus \cdots \oplus r_k$ is a formal state. A CRN is **confluent** if every pathway with a formal initial state has a closing pathway.

This means that the given CRN is capable of cleaning up all the intermediate species that it produced. For example, the CRN $\{A \to i,\ i + B \to C\}$ will not be confluent because if the system starts from the state $\{A\}$, it will immediately turn into $\{i\}$ and this i molecule will fail to be removed. Similarly to before, we will say that non-confluent CRNs do not implement any formal CRN, but it can be a non-modular implementation that is correct by some other definition.

For a more subtle example, let us consider the CRN $\{A \to i + B,\ i + B \to B\}$, which is clearly confluent according to the definition above. In fact, there will be no problem with this implemented CRN when it is operating by itself. However, when intermediate species require some formal species in order to get removed, the implemented CRN might not work correctly when there are other reactions in the test tube as well. For instance, if the above implementation runs in an environment that also contains the reaction $B \to C$, there is no longer a guarantee that i will always be removed, i.e., the CRN consisting of these three reactions will not be confluent. Thus, we often want to enforce that CRNs should be able to remove intermediate species without any help from formal species.

Definition. A closing pathway is **strong** if its reactions do not consume any formal species. A CRN is **strongly confluent** if every pathway with a formal initial state has a strong closing pathway.

Theorem 4.3. If a CRN $\mathcal{C}$ is strongly confluent, for any formal CRN $\mathcal{C}'$, $\mathcal{C} \cup \mathcal{C}'$ is strongly confluent.

Proof. Suppose p is a pathway of $\mathcal{C} \cup \mathcal{C}'$ that has a formal initial state. Clearly, if we remove from p all reactions that belong to $\mathcal{C}'$ and call the resulting pathway p', then p' has exactly the same intermediate species in its final state as p does. This is because the removed reactions are all formal reactions. Since $\mathcal{C}$ is strongly confluent, p' has a strong closing pathway, and so too does p. $\square$

In a sense, strong confluence means that we can always get an immediate (formal) interpretation of a chemical state (or of the intermediate species in a state). From this point, whenever we say confluence, we will implicitly mean strong confluence. Similarly, we will use the word 'closing pathway' to mean 'strong closing pathway.'

$A + B \rightarrow i$ $i + C \rightarrow D$ $i \rightarrow E$	$A \rightarrow i$ $i + B \rightarrow C$	$A \rightarrow i$ $i \rightarrow A$ $i + B \rightarrow C$
confluent	**not confluent**	**confluent**

Figure 4.7: Some examples of confluent and non-confluent CRNs

4.6.4 Conclusion

We have introduced all necessary concepts for defining our problem. The purpose of this section is to define with utmost clarity the notion of equivalence between CRNs, which is exactly the following.

Definition. Two confluent and regular CRNs are said to be **equivalent with respect to pathway decomposition** if their formal bases are identical. Note that since formal bases by definition do not contain any trivial reaction, two formal CRNs will be deemed equivalent if they differ only by trivial reactions.

The following theorems provide some justification for this definition.

Theorem 4.4. If two confluent and regular CRNs $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent with respect to pathway decomposition, they are equivalent with respect to a CRN observer.

Proof. Fix a formal CRN $\mathcal{O}$ and a formal state S. Let p' be a pathway of $\mathcal{O}$ and p a pathway of $\mathcal{C}_1$. Let p'' be a pathway that can be formed by interleaving p and p', and we also assume that p'' can occur in S. Since $\mathcal{C}_1$ is strongly confluent, we will assume that p is a formal pathway because we can always make it one by finding a strong closing pathway and appending it. Thus, p'' will also be a formal pathway.

Then, we completely decompose p'' into prime pathways. By regularity, we know that each prime pathway has a unique turning point where the intermediate state becomes no longer contained in the formal state. Now, we can reorder the prime pathways in p'' according to the position of these turning points so that reactions that belong to the same prime pathway are adjacent. For example, if p'' was $(A \rightarrow i, B \rightarrow j, j \rightarrow C, i \rightarrow D, C + D \rightarrow E)$, which consists of three prime pathways $(A \rightarrow i, i \rightarrow D)$, $(B \rightarrow j, j \rightarrow C)$, and $(C + D \rightarrow E)$, it will become $(B \rightarrow j, j \rightarrow C, A \rightarrow i, i \rightarrow D, C + D \rightarrow E)$ after this reordering. Now, since $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent with respect to pathway decomposition, for each prime pathway in p'', except for the reactions from $\mathcal{O}$, there should be a prime pathway of $\mathcal{C}_2$ that consumes and produces the same species. Thus, after some substitutions, we can obtain a pathway p''' of $\mathcal{C}_2$ and $\mathcal{O}$ that can occur in S (note that the definition of regularity was chosen carefully to allow this). Most importantly, p''' still has p' as a subsequence.

Now, assume towards a contradiction that p'' was terminal with respect to S and p''' is non-terminal. Then, we can find a pathway $q = p''' + q'$ of $\mathcal{C}_2 \cup \mathcal{O}$ such that q' contains at least one reaction from $\mathcal{O}$. Again, by strong confluence, we can assume that q is formal. Then, since p''' was formal, q' should also be formal. Thus we can apply the same procedure as above to obtain a pathway of $\mathcal{C}_1 \cup \mathcal{O}$ that has the same initial state as q' and has at least one reaction from $\mathcal{O}$. This pathway can then be appended to p'' to draw a contradiction to the assumption that p'' was terminal. Similarly, if p'' was non-terminal and p''' is terminal, we can also draw a contradiction. Thus, we have shown that $\text{Obs\_NT}_{\mathcal{C}_1,\mathcal{O},S} \subseteq \text{Obs\_NT}_{\mathcal{C}_2,\mathcal{O},S}$ and $\text{Obs\_T}_{\mathcal{C}_1,\mathcal{O},S} \subseteq \text{Obs\_T}_{\mathcal{C}_2,\mathcal{O},S}$.

Noting that a symmetric argument should also hold, we can easily see that $\text{Obs\_NT}_{\mathcal{C}_1,\mathcal{O},S} = \text{Obs\_NT}_{\mathcal{C}_2,\mathcal{O},S}$ and $\text{Obs\_T}_{\mathcal{C}_1,\mathcal{O},S} = \text{Obs\_T}_{\mathcal{C}_2,\mathcal{O},S}$, as desired. □

Theorem 4.5. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are formal CRNs, they are equivalent with respect to pathway decomposition if and only if $\mathcal{C}_1 = \mathcal{C}_2$.

Proof. Trivial. □

To test this new notion of equivalence, we need an algorithm for enumerating the formal basis of a given CRN, which we will address in the next chapter. In addition, we will also explain how to identify irregular and non-confluent CRNs.

4.7 History Domains

Before we move on to the algorithms, we have one final issue to discuss. In previous sections, we assumed that our implemented CRN allows only one implementation for each formal species, which is often not the case. Some implementations [39, 4] use the idea of a history domain, in which a formal species gets implemented as a class of species that have some variable domains that depend on previous reactions but have the same domains everywhere else. In the BioCRN compiler, this concept was implemented using the question-mark wildcard, which can match any single unpaired domain.

In such cases, we will want to identify all strands that match the same wildcard specification as in Fig. 4.8, because they should all have the same behavior (meaning that they all can participate in the same reactions and produce the same product species) and in doing so, we can obtain a CRN in which each formal species has only one representation. For this transformation to be consistent, we should also identify all species that contain such species as a part, as in Fig. 4.9. Note that the representations of a formal species might not necessarily be single-stranded molecules. However, even when they are multi-strand complexes, we can locate the strands with question-marked domains inside them and apply the same grouping procedure as above.

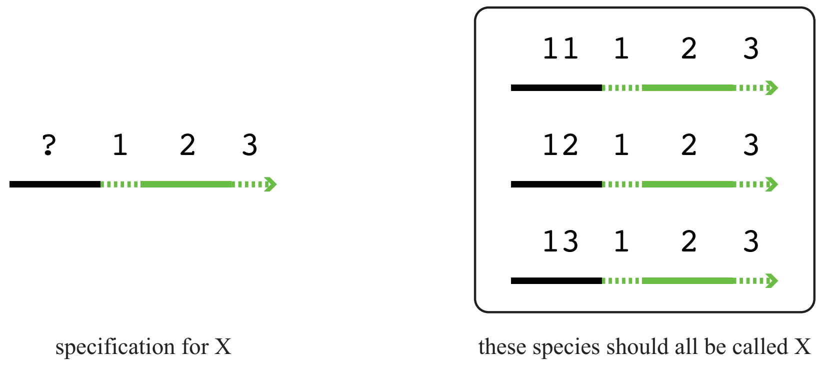


Figure 4.8: Grouping species using wildcards. Figure taken and modified from [39].

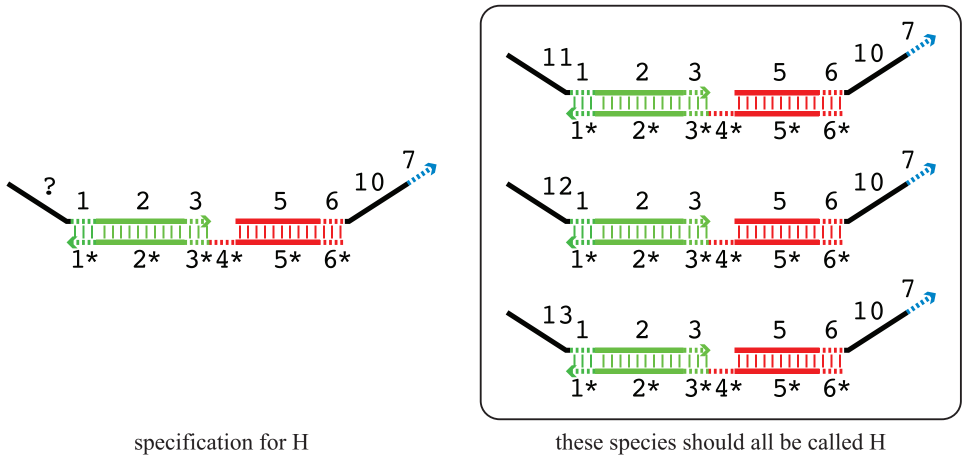


Figure 4.9: Grouping species using wildcards. Figure taken and modified from [39].

In order to ensure that this grouping is safe, we also carry out a safety check where it is tested whether species grouped under the same name actually have the same behavior. Let $\mathcal{C}$ be our original implemented CRN, and $f : \Sigma \to \Sigma$ be the new naming function for the species in $\mathcal{C}$, i.e., species that are being grouped together will have the same value of f. Then, we check the following property.

Definition. The naming function f is said to be **consistent** if the following is true. If $(R, P) \in \mathcal{C}$, then for every state S such that $f(S) = f(R)$, there exists $(S, T) \in \mathcal{C}$ such that $f(T) = f(P)$. Here, $f(S)$ denotes the multiset obtained by replacing every element x of S by $f(x)$.

This property ensures that same reactions are implemented for every species under the same name. For instance, if $f(X_1) = f(X_2) = X$, $f(Y) = Y$ and $(\{X_1\}, \{Y\})$ is in $\mathcal{C}$, then we should also have $(\{X_2\}, \{Y\})$ in $\mathcal{C}$. Otherwise, we cannot really say X_1 and X_2 are different representations of the same species, because one can produce Y and one cannot. What if Y also had two different representations, i.e., $f(Y_1) = f(Y_2) = Y$? In this case, if $(\{X_1\}, \{Y_1\}) \in \mathcal{C}$, it would suffice to have either $(\{X_2\}, \{Y_1\})$ or $(\{X_2\}, \{Y_2\})$ in $\mathcal{C}$. Because Y_1 and Y_2 are being regarded as the same species, producing a Y_1 molecule and producing a Y_2 molecule have no recognizable difference.

If the naming function obtained from the wildcard-based grouping is consistent under the above definition, we can obtain a simplified version of the implemented CRN in which each formal species has a single representation. Otherwise, we can say that the use of a wildcard domain was erroneous.

Chapter 5

Algorithm for Enumerating the Formal Basis

In the previous chapter, we reduced the CRN implementation verification to the CRN equivalence testing. We developed mathematical formalisms that can express a strong notion of equivalence between CRNs, called equivalence based on pathway decomposition, and concluded that we can test this notion by simply comparing the two CRNs' formal bases. In this chapter, we present an algorithm for finding the formal basis of a given CRN and testing whether the given CRN is confluent and regular. Then, we can simply apply that algorithm to the two CRNs that we want to compare and decide whether they are equivalent.

5.1 Objective

Because the formal basis is defined in terms of the elementary basis, the most obvious way to enumerate the formal basis is to first enumerate the elementary basis and obtain the formal basis from it. However, even for CRNs with a finite formal basis, the elementary basis is usually infinite. For example, consider the CRN $\{A \rightarrow i,\ i \rightarrow j,\ j \rightarrow i,\ j \rightarrow B\}$. This is obviously a correct implementation of $\{A \rightarrow B\}$, and indeed its formal basis is exactly $\{A \rightarrow B\}$. However, its elementary basis contains infinitely many pathways because we can have infinitely many rounds of $i \rightarrow j$ and $j \rightarrow i$ between $A \rightarrow i$ and $j \rightarrow B$ reactions (note that these pathways are all undecomposable). Moreover, this kind of reversible steps is pervasive in existing CRN-to-DNA translation schemes, so this issue of potentially infinite pathways in the elementary basis is unavoidable.

Ideally, we will want an algorithm that always terminates when the given CRN has a finite formal basis, but unfortunately, the algorithm that we are about to present does not match the ideal. In fact, it is not clear to us at this point whether the problem of deciding the finiteness of the formal basis of a given CRN is even decidable. Alternatively, we will present an algorithm that terminates under a very moderate assumption which is almost never violated in a practical CRN implementation.

To explain this condition, we first need to extend the notion of decomposability that we previously defined on formal pathways to all pathways that have formal initial states (but do not necessarily terminate in one). This is because our algorithm will constructively generate all undecomposable pathways in the CRN and those pathways will be incomplete while being worked on.

Definition. A pathway that has a formal initial state is called **semiformal**.

Definition. Let p be a semiformal pathway. We say that p is **decomposable** if p can be partitioned into two nonempty subsequences (which do not need to be contiguous) each of which are semiformal pathways. We also say that these two subsequences are a **decomposition** of p.

It is easy to check that this definition is identical to our previous definition on formal pathways. That is, if p is a formal pathway, it is decomposable with our previous definition if and only if it is decomposable with this new definition.

Then, we define the width of a pathway as follows.

Definition. Let $p = (r_1, \ldots, r_k)$ be a pathway and let $S_i = S \oplus r_1 \oplus \cdots \oplus r_i$ where S is the initial state of p. The **width** of p is defined to be $\max_i |S_i|$.

In other words, the width of a pathway is the maximum number of molecules that need to exist simultaneously in order for the pathway to occur.

Starting in the next section, we will present an algorithm that terminates if and only if the given CRN has an upper bound to the width of an undecomposable semiformal pathway. This is a moderate condition because if a CRN does not have such an upper bound, it would mean that the CRN contains "modules" that produce an infinite number of intermediate molecules. It is a structure that is highly unlikely to occur in a correct practical implementation. If necessary, we can also put an artificial limit on the width so that when this limit is exceeded, we will force-stop the algorithm and produce a warning.

5.2 Main Theorems

Before we present the pseudocode of our algorithm, we will first prove some important theorems that will argue the correctness and identify the termination condition of our algorithm.

Definition. The **branching factor** of a CRN $\mathcal{C}$ is defined to be the following value.

$$\max_{(R,P)\in\mathcal{C}} \max\{|R|, |P|\}$$

In the case of a DNA-based implementation, this constant is usually 2 (refer to the enumeration rules from Chapter 3), but it is possible to have DNA-based implementations with arbitrary branching factors [4, 5, 39].

Proposition 5.1. Suppose a pathway p is obtained by interleaving pathways $p_1, \ldots, p_l$. Let S be the initial state of p and $S_1, \ldots, S_l$ the initial states of $p_1, \ldots, p_l$. Then, $S \subseteq S_1 + S_2 + \cdots + S_l$.

Proof. Trivial. □

Theorem 5.2. If p is an undecomposable semiformal pathway of width $w > 0$, there exists an undecomposable semiformal pathway of width smaller than w but at least $(w - b)/b$, where b is the branching factor of the CRN. (Note that if $w = 1$, the lower bound $(w - b)/b$ might be negative. In this case, it would simply mean that there exists an undecomposable semiformal pathway of width 0, which would be the empty pathway.)

Proof. Since $w > 0$, p is nonempty. Let p_{-1} denote the pathway obtained by removing the last reaction (R, P) from p. Also, let $S_0, \ldots, S_k$ be the states that p goes through, and $S'_0, \ldots, S'_{k-1}$ the states that p_{-1} goes through. S_i is potentially unequal to S'_i because if the last reaction in p consumes some new formal species, then the minimal initial state of p_{-1} might be smaller than that of p.

It is obvious that the minimal initial state of p_{-1} is smaller than the minimal initial state of p by at most $|R|$, i.e., $|S_0| - |S'_0| \leq |R|$. This means that for all $0 \leq i \leq k-1$, we have that $|S_i| - |S'_i| \leq |R|$. Clearly, if there exists some $0 \leq i \leq k-1$ such that $|S_i| = w$, then $|S'_i| \geq |S_i| - |R| = w - |R| \geq w - b$, so p_{-1} has width at least $w - b$. If there exists no such i, then we have that $|S_k| = w$. Clearly, $|S_{k-1}| = |S_k| - |P| + |R|$ and it follows that

$$|S_k| - |P| + |R| - |S'_{k-1}| = |S_{k-1}| - |S'_{k-1}| \leq |R|.$$

This is equivalent to $|S_k| - |S'_{k-1}| \leq |P|$. Since $|S_k| = w$, we have that $|S'_{k-1}| \geq w - |P| \geq w - b$. Thus, p_{-1} achieves width at least $w - b$.

Then, we decompose p_{-1} until it is no longer decomposable. As a result, we will end up with $l \geq 1$ undecomposable pathways $p_1, p_2, \ldots, p_l$ which by interleaving can generate p_{-1}. Also, they are all semiformal. First, we show that l is at most b. Assume towards a contradiction that $l > b$. Then, by the pigeonhole principle, there exists i such that (R, P) can occur in the sum of the final states of $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$ (since $|R| \leq b$ and (R, P) can occur in the sum of the final states of $p_1, \ldots, p_l$, the b reactants of (R, P) are distributed among $l > b$ pathways and there exists at least one p_i that does not provide a reactant and can be omitted). Then, consider the decomposition (p_i, p'_i) of p_{-1} where p'_i denotes the pathway we obtain by interleaving $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$ in the same order that those reactions occur in p_{-1}. By Proposition 5.1, p'_i is semiformal. Since p_i's are all semiformal, the proposition also tells us that the intermediate species in the final state of p'_i will be exactly the same as those in the sum of the final states of $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$. That is, the final state of p'_i contains all the intermediate species that (R, P) needs to occur, i.e., p'_i with (R, P) appended at the end should have a formal initial state. However, this means that p is decomposable which is a contradiction. Hence, $l \leq b$.

Now, note that if we have l pathways each with widths $w_1, \ldots, w_l$, any pathway obtained by interleaving them can have width at most $\sum_{i=1}^{l} w_i$ (for the same reason as in the previous paragraph). Since p_{-1} had width at least $w - b$, we have that $w - b \leq \sum_{i=1}^{l} w_i$. Then, if $w_i < (w - b)/b$ for all i, then $\sum_{i=1}^{l} w_i < w - b$, which is contradiction. Thus, we conclude that at least one of $p_1, \ldots, p_l$ has width greater than or equal to $(w - b)/b$. It is also clear that its width cannot exceed w. Thus, we have found a pathway p' which

1. has a smaller length than p, and
2. has width at least $(w - b)/b$ and at most w.

If p' has width exactly w, then we have failed to meet the requirements of the claim. However, since we have decreased the length of the pathway by at least one and the width of a zero-length pathway is always 0, we can eventually get a smaller width than w by repeating this process. □

Corollary 5.3. Suppose that $w, w_{\max}$ are integers and that $(w + 1)b \leq w_{\max}$. Then, if there are no undecomposable semiformal pathway of width greater than w and less than or equal to $w_{\max}$, then there exists no undecomposable semiformal pathway of width greater than w.

Proof. Assume towards a contradiction that there exists an undecomposable semiformal pathway p of width $w' > w$. If $w' \leq w_{\max}$, then it is an immediate contradiction. Thus, assume that $w' > w_{\max}$. By Theorem 5.2, we can find an undecomposable semiformal pathway q of width v where $(w' - b)/b \leq v < w'$. Since $w' > w_{\max}$, we have that $v \geq (w' - b)/b > (w_{\max} - b)/b \geq ((w + 1)b - b)/b = w$. If $v \leq w_{\max}$, we have a contradiction. If $v > w_{\max}$, then take q as our new p and repeat the above process. Since v is smaller than w' by at least one, we will eventually reach a contradiction.

Thus, there exists no undecomposable semiformal pathway of width greater than w. □

This theorem gives us a hint on how to exploit the bounded width condition that we imposed on the input CRN, but it does not illuminate a way to enumerate all undecomposable semiformal pathways of bounded width in finite time (because there will still be an infinite number of them). To solve this problem, we need to define a few more concepts.

Definition. Let p be a semiformal pathway. The **decomposed final states (DFS)** of p is the set of unordered pairs (T_1, T_2) where T_1 and T_2 are the final states of some decomposition of p (which are not necessarily formal).

Note that for an undecomposable pathway, the DFS is an empty set.

Definition. Let $p = (r_1, \ldots, r_k)$ be a semiformal pathway. Also, let $S_i = S \oplus r_1 \oplus \cdots \oplus r_i$ where S is the initial state of p. Let $j \leq k$ be the greatest integer such that $\text{Formal}(S_i) \subseteq S$ for all $i \leq j$. Then, there exists a unique minimal state S' which contains $\text{Formal}(S_i)$ for all $i > j$ (clearly, it will be an empty state if $j = k$). We call such S' the **regular final state** of p.

Proposition 5.4. If p is a formal pathway, then it is regular if and only if its final state is equal to the regular final state or the regular final state is empty.

Proof. For the forward direction, suppose $p = (r_1, \ldots, r_k)$ is a regular formal pathway. It would suffice to show that if its regular final state is not empty, it must be equal to the final state. Note that for p, j from the definition above will have to be strictly less than k, because otherwise the regular final state of p will be empty. Also, since p is regular, we have that $\text{Formal}(S_i) \subseteq S_k$ for all $i > j$ where $S_i = S \oplus r_1 \oplus \cdots \oplus r_i$ as before. Thus, it is clear that the regular final state of p must be exactly S_k.

For the reverse direction, suppose p is a formal pathway. If its final state is equal to its regular final state, it is trivial that p is regular. If its final state is empty, then it means that $\text{Formal}(S_i) \subseteq S_0$ for all i, so clearly p is regular, too. □

Definition. The quintuple of the initial state, the final state, the width, the DFS, and the regular final state of a pathway is called the **signature** of the pathway.

Theorem 5.5. If m is any number, the set of signatures of all semiformal pathways of width at most m is finite.

Proof. Clearly, there is only a finite number of possible initial states, final states, widths, and regular final states. Also, since there is only a finite number of possible final states, there is only a finite number of possible DFS's. □

Theorem 5.6. Suppose p_1 and p_2 are two pathways with the same signature. Then, for any reaction r, $p_1 + (r)$ and $p_2 + (r)$ also have the same signature.

Proof. Let $p_1' = p_1 + (r)$ and $p_2' = p_2 + (r)$. It is trivial that p_1' and p_2' have the same initial and final states and the same width. It is also clear that they should have the same regular formal state. Thus we need only show that p_1' and p_2' have the same DFS.

Suppose (T_1, T_2) is in the DFS of p_1'. That is, there exists a decomposition (q_1', q_2') of p_1' where q_1' and q_2' have final states T_1 and T_2. The last reaction r is either contained in q_1' or q_2'. Without loss of generality, suppose the latter is the case. Then, if $q_1 = q_1'$ and $q_2 + (r) = q_2$, then (q_1, q_2) should decompose p_1, which is a prefix of p_1'. Since p_1 and p_2 have the same DFS, there should be a

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
A	A	A	A	h	k	l	X	X	o	X
B	i	i	i				m	Y	Y	Y
C	C	h	C	C				n		
						j				

The j from the definition occurs at S_6 and the regular final state is $\{X, Y\}$.
Since this coincides with the final state, the pathway is regular.

S_0	S_1	S_2	S_3	S_4	S_5	S_6
A	i	k	C	h	D	D
B	B		l		m	E
		j				

The j from the definition occurs at S_2 and the regular final state is $\{C, D, E\}$.
Since this is different from the final state, the pathway is irregular.

S_0	S_1	S_2	S_3	S_4
A	i	k	X	X
B	B	X	l	Y
		B		
	j			

The j from the definition occurs at S_1 and the regular final state is $\{B, X, Y\}$.
Since this is different from the final state, the pathway is irregular. Note that the intuitive reason that this pathway is irregular is that the product X is produced before the reactant B is consumed.

Figure 5.1: Finding regular final states

decomposition (s_1, s_2) of p_2 that has the same final states as q_1 and q_2. Clearly, $(s_1, s_2 + (r))$ should be a decomposition of p'_2 and thus (T_1, T_2) is also in the DFS of p'_2.

By symmetry, it follows that p'_1 and p'_2 have the same DFS. $\square$

Note that, if we can enumerate the set of signatures of all undecomposable semiformal pathways, we can immediately find the formal basis or test the regularity of the given CRN (Proposition 5.4). Thus, the final piece of the puzzle needed here is an algorithm that enumerates the signatures of all undecomposable semiformal pathways, given that there is an upper bound to the width of such pathways.

5.3 Algorithm

Our algorithm, which is explained in the following pseudocode, consists of two subroutines that are very short and easy to understand.

```
1  function enumerate(p, w, ret)
2      if p has width greater than w then return ret
3      if p is not semiformal then return ret
4      sig = signature of p
5      if sig is in ret then return ret
6      add sig to ret
7      for every reaction rxn
8          ret = enumerate(p + [rxn], w, ret)
9      end for
10     return ret
11 end function
12
13 function main()
14     w_max = 0
15     while true
16         signatures = enumerate([], w_max, {})
17         w = maximum width of an undecomposable pathway in signatures
18         if (w+1)*b <= w_max then break
19         w_max = (w+1)*b
20     end while
21 end function
```

The subroutine `enumerate` is a function that is designed to enumerate the set of signatures of all semiformal pathways with width at most `w`. It takes three arguments `p`, `w`, and `ret`. Here, `p` is the semiformal pathway that is currently being worked on, `w` is the width bound, and `ret` is the set of signatures that have been discovered so far. It is easy to note from the above pseudocode that `enumerate(p, w, ret)` generates and returns the set of signatures of all pathways of width at most `w` that contain `p` as a prefix. It uses the memoization technique in that it does not further expand a pathway whose signature has been already found and stored in `ret` (line 5). This is justified by Theorem 5.6. Note that this observation alone proves the overall correctness of this subroutine, because this subroutine basically works by enumerating every single semiformal pathway except for the ones discarded by memoization. Also, because of memoization, the termination condition for this subroutine is clearly articulated by Theorem 5.5.

The subroutine `main` repeatedly calls `enumerate`, increasing the width bound appropriately. The correctness of this bound is given by Corollary 5.3. It is trivial that `main` will terminate if and only if there exists a bound to the width of an undecomposable semiformal pathway. It is also trivial how to extract the formal basis from the found signature set.

5.4 Testing Confluence and Regularity

Finally, we discuss how to test confluence and regularity.

Theorem 5.7. A CRN is confluent if and only if every undecomposable semiformal pathway has a closing pathway.

Proof. The forward direction is trivial. For the reverse direction, we show that if a CRN is not confluent, there exists an undecomposable semiformal pathway that does not have a closing trajectory.

By definition, we have a semiformal pathway p that does not have a closing pathway. If this pathway is undecomposable, then we are done. Otherwise, we can decompose it into two pathways p_1 and p_2. Suppose both of these pathways have closing pathways. Then, since the final state of p has the same intermediate species as the sum of the final states of p_1 and p_2 (by Proposition 5.1 and the fact that p_1 and p_2 are semiformal), the two closing pathways concatenated will be a closing pathway of p (because a closing pathway does not ever consume a formal species). This is a contradiction. Thus, we conclude that at least one of p_1 or p_2 does not have a closing pathway.

Thus, either p is undecomposable or we can find a shorter pathway that does not have a closing pathway. Since a pathway of length 0 has a closing pathway, we will eventually find an undecomposable pathway with no closing pathway by repeating this process. $\square$

By the above theorem, we can test confluence by considering only undecomposable pathways. Since we already have in our signature set all the states that can be the final state of some undecomposable semiformal pathway, it would suffice to check that each of those states has a closing trajectory. This can easily be done using a simple BFS-based algorithm.

Also, as was pointed out earlier, we can easily test regularity from our signature set using Proposition 5.4.

5.5 Output of the Verifier

Note that the pseudocode from the previous section can be easily modified so that it stores one "representative" pathway for each signature found. This will be very useful in practice, because

1. in case the formal basis of the implemented CRN contains a reaction that is not in the target CRN, the verifier can provide the spurious pathway corresponding to that reaction to the user,
2. if the CRN is not confluent, the verifier can output a semiformal pathway which does not have a closing pathway,
3. and if the CRN is not regular, the verifier can output an irregular prime pathway.

These outputs, which can be interpreted as a counterexample to the given implementation, can greatly help the user locate the error in the implementation.

5.6 Improving Performance

One problem with our algorithm is its exponential worst-case time complexity (exponential in w, the upper bound on the width of undecomposable semiformal pathways, because the DFS of a pathway with width w can be as large as $\sim 2^w$). However, we can use some ad hoc techniques to greatly improve the empirical performance of the algorithm. For example, the following theorem allows us to dramatically reduce the number of pathways to enumerate.

Definition. If S is a state, Intermediate(S) denotes the multiset that consists of exactly all the intermediate species in S.

Theorem 5.8. If p is an undecomposable semiformal pathway with an initial state of size $m > 0$, there exists an undecomposable semiformal pathway with an initial state of size smaller than m but at least $\min_{(R,P)\in\mathcal{C}}\{(m - |\text{Formal}(R)|)/|\text{Intermediate}(R)|\}$ (where $\mathcal{C}$ denotes the CRN at hand).

Proof. Since $m > 0$, p is nonempty. Let p_{-1} denote the pathway obtained by removing the last reaction (R, P) from p. Let x be the number of formal species in R. Also, let S and S_{-1} denote the initial states of p and p_{-1} respectively.

It is obvious that the initial state of p_{-1} is smaller than the initial state of p by at most x, i.e., $|S_{-1}| \geq |S| - x$. Then, we decompose p_{-1} until it is no longer decomposable. As a result, we will end up with l undecomposable pathways $p_1, p_2, \ldots, p_l$ which by interleaving can generate p_{-1}. Also, they are all semiformal. First, we show that l is at most y, where y is the number of intermediate species in R (clearly, $x + y = |R|$). Assume towards a contradiction that $l > y$. Then, by the pigeonhole principle, there exists i such that (Intermediate(R), P) can occur in the sum of the final states of $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$ (as in the proof of Theorem 5.2). Then, consider the decomposition (p_i, p_i') of p_{-1} where p_i' denotes the pathway we obtain by interleaving $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$ in the same order that those reactions occur in p_{-1}. By Proposition 5.1, p_i' should have a formal initial state and the intermediate species in the final state of p_i' are exactly the same as those in the sum of the final state of $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l$. Thus, the final state of p_i' contains all the intermediate species that (Intermediate(R), P) needs to occur, i.e., p_i' with (R, P) appended at the end should have a formal initial state. However, this means that p is decomposable which is a contradiction. Hence, $l \leq y$.

Now, note that if we have l semiformal pathways whose initial states have size $m_1, \ldots, m_l$, any pathway obtained by interleaving them can have an initial state of size at most $\sum_{i=1}^{l} m_i$. Since p_{-1} had an initial state of size at least $m - x$ and $l \leq y$, we can conclude that at least one of $p_1, \ldots, p_l$ has an initial state of size at least $(m - x)/y$. It is also clear that the size of its initial state cannot exceed m. Thus, we have found a pathway p' which

1. has a smaller length than p, and

2. has an initial state of size at least $\min_{(R,P)\in\mathcal{C}}\{(m - |\text{Formal}(R)|)/|\text{Intermediate}(R)|\}$ and at most m.

If p' has an initial state of size exactly m, then we have failed to meet the requirements of the claim. However, since we have decreased the length of the pathway by at least one and a zero-length pathway has an empty initial state, we can eventually get an initial state of size smaller than m by repeating this process. □

The following theorem helps us further eliminate a huge number of pathways from consideration.

Definition. Let p be a semiformal pathway. We say that p is **strongly decomposable** if p is decomposable and at least one of the decomposed subsequences is a formal pathway.

Theorem 5.9. Let p be a semiformal pathway. If it is strongly decomposable, any pathway that contains p as a prefix is decomposable.

Proof. Trivial. □

Finally, note that we can optimize the constants in Theorem 5.2 as follows.

Theorem 5.10. If p is an undecomposable semiformal pathway of width $w > 0$, there exists an undecomposable semiformal pathway of width smaller than w but at least $(w - b)/b_r$, where $b_r = \max_{(R,P)\in\mathcal{C}}\{|\text{Intermediate}(R)|\}$.

Proof. Similar to the proofs of Theorems 5.2 and 5.8. □

These three theorems allow us to modify our algorithm as follows, which shows a considerably faster performance in practice. The subroutine `enumerate` now has one additional argument `i` which is the bound to the size of the initial state (as in line 3). By Theorem 5.8, the value of this bound, which is set in lines 22-23, is justified. Also, Theorem 5.10 allows us to use a tighter bound in lines 24-25. Finally, line 5 was added in the light of Theorem 5.9. Note that if the user wants even faster termination, one can integrate algorithms that test confluence and regularity into `enumerate` and terminate the whole algorithm as soon as a counterexample is found.

```
1 function enumerate(p, w, i, ret)
2     if p has width greater than w then return ret
3     if p has an initial state of size greater than i then return ret
4     if p is not semiformal then return ret
5     if p is strongly decomposable then return ret
6     sig = signature of p
7     if sig is in ret then return ret
8     add sig to ret
9     for every reaction rxn
```

```
        ret = enumerate(p + [rxn], w, ret)
    end for
    return ret
end function

function main()
    w_max = 0
    i_max = 0
    while true
        signatures = enumerate([], w_max, i_max, {})
        w = maximum width of an undecomposable pathway in signatures
        i = maximum initial state size of an undecomposable pathway in signatures
        i_bound = max(i*y+x) taken over all reactions (R,P) where
                  x means |Formal(R)| and y means |Intermediate(R)|
        if w*b_r+b <= w_max and i_bound <= i_max then break
        w_max = w*b_r+b
        i_max = i_bound
    end while
end function
```

Chapter 6

Results

6.1 Test Runs on Existing CRN Implementations

In this section, we will present the results of the test runs of our algorithm on existing translation schemes. The six translation schemes and four sample CRNs used in the test runs are described in Table 6.1 and Table 6.2, and the actual input files that were used are presented in the appendix. In each test run, the sample CRN was first converted to a DNA implementation by BioCRN using the specified translation scheme and then provided to Brian Wolfe's reaction enumerator to find the reaction network of the implementation under the resting states semantics. With translation schemes that use history domains, this implemented CRN was also simplified according to the process explained in Section 4.7. The consistency check for this simplification process, which is also explained in Section 4.7, was also performed and none of the translation schemes exhibited any inconsistency problem. Finally, the fuel and waste species were removed from this reaction network and the resulting CRN was finally fed to our algorithm for testing. All test runs were performed on a personal computer with 2.4GHz Core 2 Duo processor and 4GB RAM. The codes used for these test runs can be found at "http://dna.caltech.edu/biocrn_verifier/biocrn_verifier.tgz".

Translation Scheme	Description
Soloveichik	David Soloveichik's translation scheme from [39].
Qian	Lulu Qian's translation scheme from [27].
Qian_Bug	A slightly different version of the above translation scheme that was illustrated in Fig. 4.3.
Cardelli_FJ	A translation scheme using Luca Cardelli's fork and join gates from [4], which was pointed out to have bugs in the same paper.
Cardelli_C	The translation scheme proposed by Luca Cardelli, as shown in Fig. 9 of [4].
Cardelli_2D	Luca Cardelli's translation scheme from [5].

Table 6.1: List of the tested translation schemes

Table 6.3 is the summary of the 24 different test runs that were performed. For each test run, the size of the input CRN, the size of the implemented CRN, the final verdict, and the elapsed

CRN	Description
crn1	$\{A \to B,\ B+B \to A,\ B \to \emptyset\}$
crn2	$\{A+B \to C,\ C \to D,\ D+E \to A\}$
crn3	$\{A+B \to C,\ C \to D,\ D+E \to A,\ E+F \to G,\ D+C \to H,$ $A+G \to H+I+E+B,\ D \to E+E\}$
crn4	$\{A+B \to B+B,\ B+C \to C+C,\ C+A \to A+A\}$

Table 6.2: List of the tested CRNs

running time were recorded. Surprisingly, many of the known translation schemes were shown to be incorrect in the four CRNs that we tried them on.

For instance, the translation scheme from [27] (Qian in the table) was shown to produce an irregular implemented CRN for all four inputs. We investigated the output of the verifier, and the problem in this scheme is that a module is finalized only after all its products are produced. For instance, a reaction $A \to B$ will be translated to the following three reactions in this scheme.

$$\begin{aligned} A &\leftrightharpoons i \\ i &\leftrightharpoons j+B \\ j &\to \emptyset \end{aligned}$$

In this case, it is clear that the prime pathway ($A \to i,\ i \to j+B,\ j+B \to i,\ i \to A$) will be irregular. Is it really problematic? Yes[1], because if there were also a $B \to B+C$ reaction present, a system starting from the state $\{A\}$ can reach the state $\{A, C\}$, which should be impossible if the above module correctly implemented the $A \to B$ reaction. The authors of [27] independently discovered this problem, and it will be corrected in the journal version of that paper.

Interestingly, the scheme that was modified from [27] (Qian_Bug in the table) which was believed to have a fatal bug (Fig. 4.3) was shown to be rather correct in the first two input cases. This is because this scheme avoids the above problem when the number of products of the reaction at hand is one. In fact, $A \to B$ is implemented as below, which does not show any problem.

$$\begin{aligned} A &\leftrightharpoons i \\ i &\to B \end{aligned}$$

However, when the number of products of any reaction in the target CRN is larger than one, this scheme exhibits the same problem as above.

Cardelli_FJ [4] also had the same problem. Cardelli_FJ was already pointed out to have a problem by the authors of [4], but it was a different type of a problem that arises due to crosstalk between

[1]However, if we can guarantee that there exists no other formal reaction in the system, there can be some notions of correctness which consider this example to be correct (although it still violates the temporal logical property that a system that started from $\{A\}$ cannot observe an A molecule after it observes a B molecule at least once).

Translation Scheme	Input CRN	Size of the Input CRN	Size of the Implemented CRN	Verdict	Running Time
Soloveichik	crn1	3	7	Correct	0.04s
Soloveichik	crn2	3	10	Correct	0.08s
Soloveichik	crn3	7	24	Correct	0.66s
Soloveichik	crn4	3	12	Correct	0.09s
Qian	crn1	3	15	Irregular	0.00s
Qian	crn2	3	19	Irregular	0.01s
Qian	crn3	7	53	Irregular	0.01s
Qian	crn4	3	27	Irregular	0.04s
Qian_Bug	crn1	3	11	Correct	0.18s
Qian_Bug	crn2	3	13	Correct	0.15s
Qian_Bug	crn3	7	51	Irregular	0.01s
Qian_Bug	crn4	3	36	Irregular	0.04s
Cardelli_FJ	crn1	3	9	Irregular	0.00s
Cardelli_FJ	crn2	3	13	Irregular	0.01s
Cardelli_FJ	crn3	7	60	Irregular	0.02s
Cardelli_FJ	crn4	3	24	Correct	1.03s
Cardelli_C	crn1	3	20	?	∞[a]
Cardelli_C	crn2	3	28	?	∞
Cardelli_C	crn3	7	68	?	∞
Cardelli_C	crn4	3	36	?	∞
Cardelli_2D	crn1	3	32	Irregular	1874.78s
Cardelli_2D	crn2	3	38	Not confluent	0.08s
Cardelli_2D	crn3	7	98	Not confluent	0.29s
Cardelli_2D	crn4	3	48	Not confluent	0.04s

Table 6.3: Test run results

[a]Did not terminate in three hours.

Translation Scheme	Input CRN	Size of the Input CRN	Size of the Implemented CRN	Verdict	Running Time
Cardelli_C	crn1	3	20	Correct*	64.39s
Cardelli_C	crn2	3	28	Correct*	4.96s
Cardelli_C	crn3	7	68	Correct*	725.79s
Cardelli_C	crn4	3	36	Correct*	4.66s
Cardelli_C_noGC	crn1	3	11	Correct	0.18s
Cardelli_C_noGC	crn2	3	13	Correct	0.15s
Cardelli_C_noGC	crn3	7	31	Correct	1.32s
Cardelli_C_noGC	crn4	3	15	Correct	0.32s
Cardelli_2D_noGC	crn1	3	30	Irregular	4459.89s
Cardelli_2D_noGC	crn2	3	34	Irregular	0.15s
Cardelli_2D_noGC	crn3	7	88	Irregular	0.46s
Cardelli_2D_noGC	crn4	3	42	Irregular	0.27s
Qian_Fixed	crn1	3	17	Correct	21.96s
Qian_Fixed	crn2	3	22	Correct	9.58s
Qian_Fixed	crn3	7	60	?	∞
Qian_Fixed	crn3	7	60	Correct*	2357.60s
Qian_Fixed	crn4	3	30	Correct	114.65s

Correct* means that the verifier was run under some artificial bounds on the width and the initial state size of pathways.

Table 6.4: Test run results

different modules. Here, the scheme was shown to generate the same irregular structure as the above two schemes (Figs. 6 and 7A in [4]). However, this problem does not occur with their join gate (Fig. 7 in [4]), so crn4 which only has bimolecular reactions was translated correctly.

The algorithm failed to terminate on Cardelli_C [4] in the three hours that we allowed it to run. This translation scheme uses a complicated "garbage collection" mechanism to remove "garbage" intermediate species that it produces, and apparently this mechanism blows up the number of distinct pathway signatures in the system. Thus, we tried the algorithm with a restriction that it only enumerates pathways with an initial state of size at most two (because all the reactions in the target CRNs had less than two reactants), and obtained the results in Table 6.4. These results do not guarantee that this translation scheme is correct on the four input CRNs, but they at least tell us that if there is a problem, then it must be due to crosstalk between different modules because the problematic pathway will have an initial state of size greater than two. We also tested a simpler version of Cardelli_C in which the garbage collection mechanism was removed (Cardelli_C_noGC in Table 6.4). As shown in Table 6.4, it was shown to be correct for all four CRNs.

Cardelli_2D was shown to be irregular in the first input case, which is because of essentially the same problem as that of Qian, Qian_Bug, and Cardelli_FJ. It was also shown to be not confluent in the other three input cases, but when we analyzed the output of the verifier, it turned out that it was because the scheme uses a reaction mechanism that our reaction enumerator does not support (Fig. 8 of [5] uses a trimolecular reaction which was originally proposed in [48]). This problem can be fixed by using a more thorough reaction enumerator in the future[2]. Since this trimolecular mechanism also occurs in the garbage collection module, we implemented a simplified version in which the garbage collection was removed (Cardelli_2D_noGC in Table 6.4). Now, the scheme exhibited the same irregularity problem in all four cases.

Finally, the authors of [27] provided us with the corrected version of their translation scheme (Qian_Fixed in Table 6.4), which was shown to be correct for crn1, crn2, and crn4. However, the algorithm failed to terminate in reasonable time on crn3, and accordingly we performed another run in which we limited the width of enumerated pathways to 6 and the size of the initial state to 2 (this time, limiting of the initial state size alone did not help much). Under those restrictions, the scheme was also shown to be correct* for crn3.

Thus, our verifier was able to locate many subtle errors from various known translation schemes. Although it failed to terminate quickly on some input cases, it finished in less than one second in most cases, which is surprising considering the exponential worst-case time complexity of the algorithm. Especially, when there is an error, the algorithm tended to find a counterexample very quickly, which will be useful in practice. If the verifier does not terminate for a long time, the user can also opt to impose certain limits on the algorithm (in the form of artificial constant bounds) to achieve a verification certificate with various different levels of confidence. In conclusion, we claim that our algorithm is not only theoretically correct, but will also be practically useful to the researchers studying DNA-based CRN implementations.

[2]Note that if more enumerators with various different semantics are built in the future, our verifier will be able to analyze systems under different physical assumptions.

Chapter 7

Discussion

In this thesis, we have presented two useful tools for molecular programming: the BioCRN compiler and the CRN implementation verifier. These tools can be readily used in the research of DNA-based CRN implementations. They will not only contribute to the automation of molecular programming, but also help many researchers have more theoretical confidence in their implementations and concentrate on reducing experimental flaws. In particular, our verifier is capable of producing the exact pathway in which the implementation fails, which will be very helpful in debugging errors.

It seems that our algorithm for enumerating undecomposable semiformal pathways may be useful for many more applications. In any chemical system in which we are only interested in certain species, our techniques can be used to enumerate independent pathways in the system and analyze them, even if the system has nothing to do with using DNA to implement chemical reaction networks. In these cases, we will need to drop CRN-specific conditions such as confluence or regularity if necessary.

For this reason, our research might be a promising first step toward automated verification of any DNA system. As we mentioned in the introduction chapter, CRNs are a very powerful modeling language that can express a wide range of chemical behavior. In this sense, CRNs can be used as a specification language for molecular programs, where we specify the desired behavior between key species. In fact, the system does not even need to be based on DNA strand displacement as long as we have a way to enumerate possible reactions (i.e. we have an equivalent of the reaction enumerator).

Also, there is an indication that similar techniques can be used to verify DNA systems not only at the domain level but also at the individual base pair level. This is easily seen by observing that a sequence-level design can be viewed as a domain-level design that uses four different ‘short’ domains named A, G, C, and T. At present, however, this kind of direct translation will produce too many reactions and species for the reaction enumerator and the verifier to terminate in a reasonable time.

There are some more key areas in which improvements can be made about CRN implementation verification:

1. Our verification algorithm is intrinsically exponential, and can fail to produce the answer in a reasonably short time for some input. There might be ways to reduce the time complexity of the algorithm, either by a mathematical breakthrough or by ad hoc observations.

2. Our method can verify a single compilation instance (i.e. a single translation scheme applied on a single CRN), but cannot prove the correctness of a given translation scheme in general (for all possible input CRNs). It would be interesting to provide an automated method for proving the correctness of a translation scheme.

3. There have been some published results that use polymer molecules [27]. Our formulation of CRNs in this thesis regards every polymer molecule as a different species, whereas ideally there should be a way to group them in a smart way such that we can express a potentially infinite number of polymer species efficiently. This will be an interesting area of study because the polymer molecules allow us to build a theoretically infinite memory with a finite number of component species.

4. The reaction enumerators introduced in this thesis are not capable of dealing with pseudoknotted structures. For a broader range of applications, it would be advantageous to have a verifier that can work with pseudoknot-capable enumerators.

5. It is possible that an implementation of a CRN that is incorrect according to our definitions might still work very well in practice. This can occur when the erroneous pathways are kinetically extraordinarily rare, because the discussion of chemical kinetics was entirely left out in our research. There is an interesting idea to consider in this light; if we can somehow enumerate a full list of the elementary basis (although it is often infinite), we might be able to analyze each prime formal pathway kinetically, giving rise to provable kinetic properties for the full system.

Historically, the ways to model and analyze chemical systems have been studied in various perspectives, ranging from Petri nets and bisimulation to temporal logic [8, 11, 32, 3]. None of these approaches seems to be directly applicable to the logical verification of DNA implementations in an obvious way. This was the main reason that we came to invent an entirely new set of definitions rather than develop on existing ones. When we limit our scope to DNA-based CRN implementations, there have been some papers that attempted to define their own notions of correctness and prove it [4, 5, 39]. However, these notions were usually either too specific to be widely adopted or not strong enough to be interesting in general. Finally, there is a software tool called PRISM, which is designed for automatic verification of probabilistic systems [15]. Some researchers have successfully used it to test temporal logical properties on chemical systems [14, 6], but it is not obvious what temporal logical properties will ensure the implementation's correctness.

In this thesis, we proposed a mathematical formalism for chemical systems that is fundamentally different from any method mentioned above. Although our results are certainly not perfect at this point, we hope that they will not only contribute to the field's current understanding about the CRN implementations, but also inspire scientists to think about the verification problem in its own right.

Bibliography

[1] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] Charles H. Bennett. The thermodynamic of computation - a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[3] Muffy Calder, Vladislav Vyshemirsky, David Gilbert, and Richard Orton. Analysis of signalling pathways using continuous time markov chains. *Transactions on Computational Systems Biology VI*, 4220:44–67, 2006.

[4] Luca Cardelli. Strand algebras for DNA computing. *Natural Computing: an international journal*, 10(1):407–428, 2011.

[5] Luca Cardelli. Two-domain DNA strand displacement. *Mathematical Structures in Computer Science*, 2011. In press.

[6] Luca Cardelli, Marta Kwiatkowska, Matthew R. Lakin, David Parker, and Andrew Phillips. Design and analysis of DNA circuits using probabilistic model checking. Preprint, available at `http://lucacardelli.name`, 2010.

[7] Luca Cardelli, Andrew Phillips, and Simon Youssef. Exploring DNA strand-displacement computational elements. In *The 16th International Conference on DNA Computing and Molecular Programming*, 2010. Two-page abstract at conference proceedings.

[8] Nathalie Chabrier-Rivier, Marc Chiaverini, Vincent Danos, Franois Fages, and Vincent Schchter. Modeling and querying biomolecular interaction networks. *Theoretical Computer Science - Special issue: Computational systems biology*, 325:25–44, 2003.

[9] Ehsan Chiniforooshan, David Doty, Lila Kari, and Shinnosuke Seki. Scalable, time-responsive, digital, energy-efficient molecular circuits using DNA strand displacement. *Lecture Notes in Computer Science*, 6518:25–36, 2010.

[10] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. Technical Report ETR090, California Institute of Technology, 2008.

[11] Mark Ettinger. The complexity of comparing reaction systems. *Bioinformatics*, pages 465–469, 2002.

[12] Richard P. Feynman. There's plenty of room at the bottom. *Caltech Engineering and Science*, 23:22–36, 1960.

[13] Anthony J. Genot, David Yu Zhang, Jonathan Bath, and Andrew J. Turberfield. Remote toehold: A mechanism for flexible control of DNA hybridization kinetics. *Journal of the American Chemical Society*, 133:2177–2182, 2011.

[14] John Heath, Marta Kwiatkowska, Gethin Norman, David Parker, and Oskana Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science - Special issue on Converging Sciences: Informatics and Biology*, 391:239–257, 2008.

[15] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. *Lecture Notes in Computer Science*, 3920:441–444, 2006.

[16] Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, Sebastian L. Bonhoeffer, Manfred Tacker, and Peter Schuster. Fast folding and comparison of RNA secondary structures. *Monatsh. Chem.*, 125:167–188, 1994.

[17] Jongmin Kim. *In vitro synthetic transcriptional networks*. PhD thesis, California Institute of Technology, 2006.

[18] Jongmin Kim and Erik Winfree. Synthetic in vitro transcriptional oscillators. *Molecular Systems Biology*, 7:465, 2011.

[19] Matthew R. Lakin and Andrew Phillips. *Visual DSD*. Microsoft Research.

[20] Matthew R. Lakin, Simon Youssef, Luca Cardelli, and Andrew Phillips. Abstractions for DNA circuit design. *Journal of the Royal Society Interface*, 2011. Published online, doi: 10.1098/?rsif.2011.0343.

[21] Anthony M. L. Liekens and Chrisantha T. Fernando. Turing complete catalytic particle computers. *Lecture Notes in Computer Science*, 4648:1202–1211, 2007.

[22] Chenxiang Lin, Yan Liu, Sherri Rinker, and Hao Yan. DNA tile-based self-assembly: building complex nanoarchitectures. *ChemPhysChem*, 7:1641–1647, 2006.

[23] Kyle Lund, Anthony T. Manzo, Nadine Dabby, Nicole Michelotti, Alexander Johnson-Buck, Jeanette Nangreave, Steven Taylor, Renjun Pei, Milan N. Stojanovic, Nils G. Walter, Erik Winfree, and Hao Yan. Molecular robots guided by prescriptive landscapes. *Nature*, 465:206–210, 2010.

[24] Kevin Oishi and Eric Klavins. A biomolecular implementation of linear I/O systems. *IET Systems Biology*, 5(4):252–260, 2011.

[25] Akimitsu Okamoto, Kazuo Tanaka, and Isao Saito. DNA logic gates. *Journal of the American Chemical Society*, 126:9458–9463, 2004.

[26] Andrew Phillips and Luca Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 6:S419–S436, 2009. doi: 10.1098/?rsif.2009.0072.focus.

[27] Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. *Lecture Notes in Computer Science*, 6518:123–140, 2011.

[28] Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuites. *Journal of the Royal Society Interface*, 8(62):1281–1297, 2011.

[29] Paul W. K. Rothemund. A DNA and restriction enzyme implementation of Turing machines. In *DNA Based Computers*, pages 75–120, 1996.

[30] Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440:297–302, 2006.

[31] Paul W. K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *Public Library of Science Biology*, 2:424, 2004.

[32] Ph. Schnoebelen and N. Sidorova. Bisimulation and the reduction of Petri nets. *Lecture Notes in Computer Science*, 1825:409–423, 2000.

[33] Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314:1585–1588, 2006.

[34] Nadrian C. Seeman. DNA in a material world. *Nature*, 421:427–431, 2003.

[35] Phillip Senum and Marc Riedel. Rate-independent constructs for chemical computation. *PLoS ONE*, 6:e21414, 2011.

[36] Adam Shea, Brian Fett, Marc D. Riedel, and Keshab Parhi. Writing and compiling code into biochemistry. In *Pacific Symposium on Biocomputing*, volume 15, pages 456–464, 2010.

[37] Jong-Shik Shin and Niles A. Pierce. A synthetic DNA walker for molecular transport. *Journal of the American Chemical Society*, 126:10834–10835, 2004.

[38] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7:615–633, 2008.

[39] David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107:5393–5398, 2010.

[40] Ye Tian, Yu He, Yi Chen, Peng Yin, and Chengde Mao. A DNAzyme that walks processively and autonomously along a one-dimensional track. *Angewandte Chemie*, 44:4355–4358, 2005.

[41] Suvir Venkataraman, Robert M. Dirks, Paul W. K. Rothemund, Erik Winfree, and Niles A. Pierce. An autonomous polymerization motor powered by DNA hybridization. *Nature Nanotechnology*, 2:490–494, 2007.

[42] Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

[43] Brian R. Wolfe. Wolfe enumerator specification: Version 4. A project as a graduate student on rotation, 2009.

[44] Peng Yin, Harry M. T. Choi, Colby R. Calvert, and Niles A Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451:318–322, 2008.

[45] Peng Yin, Rizal F. Hariadi, Sudheer Sahu, Harry M. T. Choi, Sung Ha Park, Thomas H. LaBean, and John H. Reif. Programming DNA tube circumferences. *Science*, 321:824–826, 2008.

[46] Peng Yin, Hao Yan, Xiaoju G. Daniell, Andrew J. Turberfield, and John H. Reif. A unidirectional DNA walker that moves autonomously along a track. *Angewandte Chemie*, 43:4906–4911, 2004.

[47] Joseph N. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. NUPACK: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32:170–173, 2011.

[48] David Yu Zhang. Cooperative hybridization of oligonucleotides. *Journal of the American Chemical Society*, 133:1077–1086, 2011.

[49] David Yu Zhang, Andrew J. Turberfield, Bernard Yurke, and Erik Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318:1121–1125, 2007.

Appendix A

The BioCRN Compiler

A.1 Introduction

Recently, it has been shown that chemical reaction networks (CRNs) can be algorithmically translated down to synthetic DNA molecules that emulate their behavior [39, 4, 27]. The existence of such a general translation scheme assigns CRNs a very important role in molecular programming, because now they can serve as a programming language for DNA systems. It means that any computational idea that can be encoded in a CRN can also be implemented using DNA molecules.

Conceptually, there are mainly three steps in which a CRN is compiled down to DNA molecules. First, the given CRN is translated to domain-specified DNA molecules. Second, the domain specification is turned into sequences of nucleotide bases. Third, real DNA molecules are synthesized according to the sequence information generated in the second step. The procedures for the second and third steps are highly standardized among researchers; the second step is usually performed using software programs called sequence designers, e.g., NUPACK [47]. The third step is highly industrialized and is usually carried out by commercial companies.

Unfortunately, the first step is being performed in many different ways even in a single research group. The reasons are twofold. First, there are many different "translation schemes" with different pros and cons. Second, there is no standardized software that can carry out this task. Observe that the second reason is a consequence of the first. While it is easy to build a compiler that works for a single translation scheme, it is much more difficult to design a general compiler that works for all translation schemes. This is because translation schemes might build on drastically different ideas.

BioCRN is a general compiler for DNA-to-CRN translation that is designed to work for a wide range of translation schemes. The purpose of this compiler is to allow researchers to easily switch between different translation schemes, modify existing translation schemes, and even verify the correctness of the compilation[1].

[1]For this, one needs to interface BioCRN with a few other software tools such as the reaction enumerator and the CRN equivalence tester.

A.2 Overview

The BioCRN compiler is written in Python 2.6.6 and thus runs on all commonly used platforms. It uses Pyparsing and DNAObjects libraries. (The former is a common library and the latter is developed and maintained by the Winfree group at the California Institute of Technology. For information about the latter, contact Joseph Schaffer, Karthik Sarma or Seung Woo Shin.)

BioCRN takes two input files and produces one output file. The two input files are the target CRN and the translation scheme description, and the output file is the domain specification of the translated molecules. The extensions for the input and output files are .crn ('chemical reaction network'), .ts (for 'translation scheme'), and .dom ('domain specification') respectively.

A.3 Syntax and Semantics

A.3.1 CRN Language

Since the CRN language is very simple, we replace the formal syntax definition with the following example.

```
X1 -> X2 + X3
X2 + g ->
-> E
E + x <=> Ex
# This is a comment.
```

The CRN language adopts the Python style comment, ignoring everything that appears after a sharp (`#`) character. The first four lines of the above code illustrates how to write a normal irreversible reaction, a decaying reaction, a spontaneous reaction, and a reversible reaction in the CRN language.

A.3.2 DOM Language

The DOM language has two types of statements. First, the user can specify the length of a domain as follows.

```
sequence <identifier> : <integer>
```

Here, `identifier` is an alphanumeric string (which has to start with an alphabet) which is the name of the domain whose length is to be specified. The following `integer` is the length of the domain. For instance, we can write the following to mean that the length of a domain `d1` is 5 nucleotide bases.

```
sequence d1 : 5
```

The second type of a statement is the structure specification for a DNA complex. This is always done in three lines, as follows.

```
<identifier>:
<primary structure>
<secondary structure>
```

Here, `identifier` is an alphanumeric string that specifies the name of the DNA complex to be described. The following two lines, which store the primary and secondary structure information about the complex, use the dot-paren notation (as in NUPACK [47]). For instance, we can write the following to describe the molecule shown in Fig. A.1.

```
A:
b c + c* b* a*
( ( + )  )  .
```

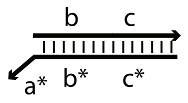


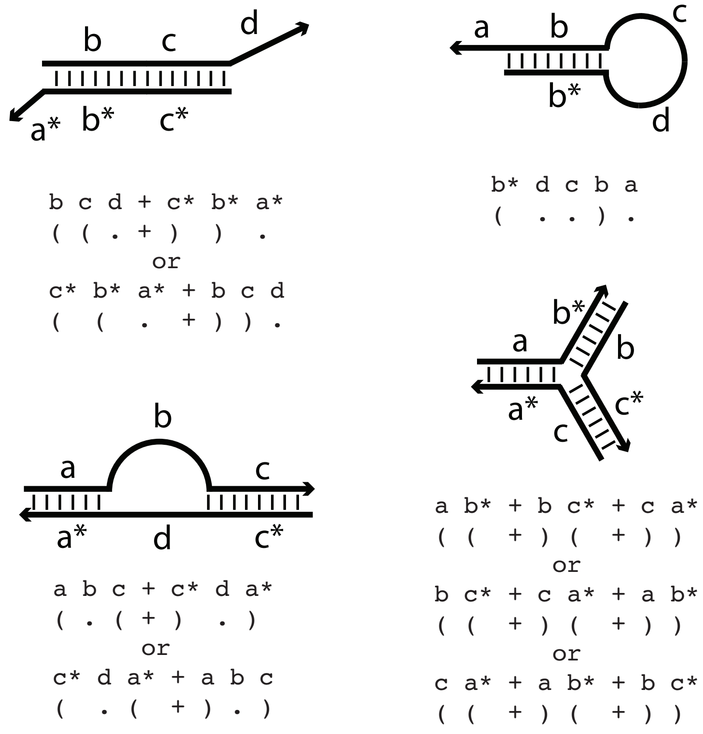
Figure A.1: A dot-paren notation example from the above code

In this notation system, a plus sign indicates strand breaks, a dot indicates that the corresponding domain is unpaired, and a parenthesis indicates that the corresponding domain is paired with the domain with the matching parenthesis. While this notation system cannot express every possible structure that can occur in a DNA molecule (in fact, the class of structures that dot-paren notation can express is exactly those that are free of pseudoknots), it is one of the most convenient notations available today and is widely used in the field. Fig. A.2 provides some examples that will further help understanding of this notation system.

The following is an example of a typical DOM file.

```
sequence a : 5
sequence b: 15
sequence c: 15

# The following is a formal species.
A:
b c + c* b* a*
( ( + )  )  .
```



```
b c d + c* b* a*
( ( . + )  )  .
       or
c* b* a* + b c d
(  (  .  + ) ) .
```

```
b* d c b a
(  . . ) .
```

```
a b c + c* d a*
( . ( + )  . )
      or
c* d a* + a b c
(  . (  + ) . )
```

```
a b* + b c* + c a*
( (  + ) (  + ) )
        or
b c* + c a* + a b*
( (  + ) (  + ) )
        or
c a* + a b* + b c*
( (  + ) (  + ) )
```

Figure A.2: Some more examples of dot-paren notation

Like the CRN language, the DOM language also supports the Python style comment.

A.3.3 TS Language

A.3.3.1 Function definition

As the TS language is a functional programming language, the top-level statements are mostly function definitions. It is usually done in the following way.

```
function <identifier>(<arguments>) = <body>;
```

For instance, we can define some common functions as follows.

```
# The TS language also supports the Python style comments.
# Computes the sum of the given list of integers
function sum(x) =
    if len(x) = 0 then 0 else x[0] + sum(tail(x));
# Note that list indices start from zero.


# Computes the length of the given list
function len(x) =
    if x == [] then 0 else 1 + len(tail(x));


# Reverses the given list
function reverse(x) =
    if x == [] then [] else reverse(tail[x]) + [x[0]];


# Applies the function f on each entry of the list x and returns the list of the results
function map(f, x) =
    if len(x) == 0 then
        []
    else
        [f(x[0])] + map(f, tail(x));
```

Here, `tail` is a built-in function that returns the given list except the first entry, and `x[i]`, as usual, indexes the i-th entry of the list `x`. In the definitions for `len` and `reverse`, we can notice that the plus operator is overloaded for multiple data types. Namely, it is being used for adding two integers in `len` and for concatenating two lists in `reverse`.

The TS language provides many different names for functions. The keywords `class`, `module`, `macro` all have the same meaning as `function`, but the programmer can use them appropriately to enhance the readability of the code.

A.3.3.2 Global declaration

At the top-level, the user can also declare global constants. For instance, the following code will generate a toehold domain that can be used universally in the implementation.

```
global toehold = short();
```

A.3.3.3 Conditionals

The TS language supports conventional if-then-else statements. It can be used the following way.

```
function power2(x) =
    if x == 1 then 2
    elseif x == 2 then 4
    elseif x == 3 then 8
    else 16;
# this function will be incorrect because it returns 16 whenever x is greater
#    than 3.
```

The number of `elseif` clauses is unlimited.

A.3.3.4 Local bindings

The TS language also supports the Haskell style where clauses. It allows the user to create local bindings. The following example illustrates its usage.

```
function area(r) =
    r * r * pi where pi = 3.14159265358979323846264338327950288419;
```

If one wishes to create multiple bindings at once, one can use the following syntax as well.

```
global x = e + f where {
        a = 2;
        b = a + 1;
        c = b + 1;
        d = c + 1;
        e = d + 1;
         f = 4
    }
```

The bindings will be created in the order that they are listed, so the value of `x` in this case will be 10.

A.3.3.5 Pattern matching

The `global` and `where` keywords support basic pattern matching. For instance, we can write the following.

```
global [x, y] = [2, 3];
```

The values of `x` and `y` will be 2 and 3, respectively.

A.3.3.6 Function calls, attributes, and list indexing

The function calls in the TS language are made the same way as most other programming languages. For instance, if we want to call a function called `sum` on a variable `l`, we write `sum(l)`. To reference an attribute `x` of an object `A`, we simply write `A.x`. List indexing is done in the same way as in Python, e.g., `l[0]`, `l[1]`, `l[2]`, etc. The list index always starts from 0.

A.4 Types

A.4.1 Conventional Data Types

The TS language supports the following common data types: integer, float, boolean, and list. These data types are handled the exact same way as in Python except that the boolean data type does not explicitly exist in the TS language. There is no ‘true’ or ‘false’ keyword in the TS language, so a boolean value only arises as a result of logical operators (such as `==` or `!=`). The following are some examples of how to use these data types.

```
global x = [1,2,3,4,5]; # defines a list of five elements
global y = 2.35 + x[1]; # the result should be 4.35
global z = x == y; # the result will be false
```

A.4.2 New Data Types

The TS language supports new (object) data types called `Species`, `Reaction`, `Domain`, `Structure`, and `Solution` in order to facilitate handling molecule information. Fig. A.3 summarizes the general purposes of these new data types.

Data type	Usage
`Species`	Used to represent chemical species. As its only data field is `name` which simply stores the name of the species, its main purpose is to distinguish between different species. In the current version of the compiler, this data type has no particular use.
`Reaction`	Stores information about chemical reactions. It has three data fields; `reactants` which holds the list of reactant molecules, `products` which holds the list of product molecules, and `reversible` which tells whether the reaction is reversible.
`Domain`	Used to represent domains. It has two data fields: `id` and `length`. Similarly to the `Species` data type, it is mainly used to distinguish between different domains using the `id` field.
`Structure`	Stores the primary and secondary structure of a DNA complex along with the domains used in it.
`Solution`	Represents a chemical solution. It has one data field called `molecules`, which stores a list of `Structure` objects. Conceptually, it stores the list of molecules that are present in the solution.

Figure A.3: New data types.

A.4.2.1 `Species` data type

The `Species` data type was designed in anticipation of new features to be implemented in the future, and it currently has no particular use.

A.4.2.2 `Reaction` data type

The `Reaction` data type is used when the input CRN is given to the TS code’s `main` function (which is an analog to C/C++’s main function and will be introduced shortly). A CRN in the TS language is expressed as a list of `Reaction` objects, which contain reactant and product species as `Structure` objects.

A.4.2.3 `Domain` data type

The `Domain` data type is usually used to define `Structure` objects, which require the primary and secondary structure information. Unlike the above two data types, a `Domain` object can be created using built-in functions like `short` or `long`.

A.4.2.4 `Structure` data type

The `Structure` data type is used to store structural information of a DNA molecule. It can be created in runtime using the following syntax:

```
global X = "a b c + c* b*" | ". ( ( + ) )"
    where {
        a = short();
        b = long();
        c = long();};
```

`X` will have the structure shown in Fig. A.1. Note that the quotation marks here do not indicate that what comes between them is a string, but they are simply syntactic tokens for creating a `Structure` object. Similarly, the vertical bar is not an operator, but merely a syntactic token.

The domains that were used to create a `Structure` object can be referenced later using the names that were used at the time of creation. For instance, the user can reference the three domains that were used in `X` by writing `X.a`, `X.b`, and `X.c`.

A.4.2.5 `Solution` data type

A `Solution` object is basically a set of molecules. In later versions, it will eventually have to specify the concentration of each molecule as well, but currently it is assumed that every molecule has an infinitely large concentration. A `Solution` object can be created from a `Structure` object using the built-in function called `infty`.

A.5 Built-In Operators

A.5.1 Arithmetic Operators

The TS language implements the following binary arithmetic operators: `+`, `-`, `*`, and `/`. While these operators basically support numerical data types like integers and floats, the plus operator is also overloaded for other data types such as lists and `Solution` objects. It works as a concatenation operator for lists and as a set union operator for `Solution` objects. The language also supports a unary minus operator, used to negate a numerical value.

A.5.2 Logical Operators

Like in C/C++, one can use the following numerical comparison operators on integer and float values: `==`, `!=`, `<`, `>`, `>=`, and `<=`. Also, one can use the keywords `and` and `or` to perform logical and and or operations respectively.

A.6 Built-In Functions

A.6.1 `tail` function

It takes a list argument and returns the same list starting from the second element. For example, `tail([1,2,3])` returns `[2,3]`. If the given list is empty, `tail` reports a runtime error.

A.6.2 `complement` function

The `complement` function takes one argument. If given a `Domain` object, it computes the Watson-Crick complement of the given domain. If given a `Structure` object that describes a single strand molecule, it returns a `Structure` object that is the mirror image of the given `Structure` object. The mirror image refers to the structure obtained by reversing the list of the domains and replacing each of them with its complement.

A.6.3 `infty` function

This function simply converts a given `Structure` object into a `Solution` object containing only that molecule.

A.6.4 `short` function and `long` function

As discussed earlier, these functions create a new unique domain of respective length. Currently, 5 and 15 are being used as the default lengths for these functions. If the user wants to create a domain of specific length, `unique` function can be used.

A.6.5 `unique` function

This function takes an integer as its only argument and returns a new unique domain of that length.

A.6.6 `flip` function

This function takes two arguments: a list `l` and an integer `n`. Here, `l` is assumed to be a `len(l)`×`n` matrix expressed as a list of lists, and otherwise, it will result in a runtime error. Then, `flip(l, n)` will return the transpose of the given matrix, which will be a `n`×`len(l)` matrix. For instance, if `M` is `[[1,2,3],[4,5,6]]`, then `flip(M, 3)` will return `[[1,4],[2,5],[3,6]]`. The usage of this function will be further explained in the examples section.

A.6.7 `rev_reactions` function and `irrev_reactions` function

These functions take one argument, which should be a list of `Reaction` objects. First, `irrev_reactions` turns every reversible reaction into a pair of irreversible reactions that it comprises. This is because some translation schemes can implement reversible reactions only as two separate irreversible reactions. Conversely, `rev_reactions` finds all pairs of matching irreversible reactions and merge them into one reversible reaction.

A.6.8 `sum` function

This function simply computes the sum of a list using the built-in plus operator. This means that it can be used on either a list of integers, a list of lists, or a list of `Solution` objects because the plus operator is overloaded for those types. If given an empty list, the function will return an empty `Solution` object.

```
# sum([1,2,3,4,5]) will evaluate to 15
# sum([[1,2], [3,4,5]]) will evaluate to [1,2,3,4,5]
# sum([infty(x), infty(y)]) will evaluate to a Solution object containing both x and
#   y molecules.
```

A.6.9 `len` function

This function, as usual, computes the length of a given list, e.g., `len([1,2,3])` will be 3.

A.6.10 `reverse` function

This function reverses a given list.

A.6.11 `unirxn` function, `birxn` function, and `rxn_degree` function

The `unirxn` function takes one argument, which is a list of `Reaction` objects, filters out reactions that do not take exactly one reactant, and returns the resulting list. The `birxn` function performs the same task but leaves only the reactions that take exactly two reactants. The most general version of these functions is the `rxn_degree` function, which takes an additional argument that specifies the number of reactants that we want. These functions are conveniently provided to the user because some translation schemes apply different constructions for reactions with different number of reactants (e.g. Soloveichik et al., 2010 [39]).

A.6.12 `map` function

This function works the same way as the map function in any other functional language. In other words, `map(f, [a,b,c,...,z])` where `f` is a function evaluates to `[f(a), f(b), f(c), ..., f(z)]`.

A.7 Runtime

In order for a TS code to be functional, it has to implement two functions.

1. `formal` is a function that produces `Structure` objects that describe how the species from the input CRN should be constructed. The name of the function comes from the fact that we call species from the input CRN the **formal species**.

2. `main` is a function that takes a list of reactions and returns all the auxiliary species required to implement those reactions (the return value should be a `Solution` object).

We should study some examples to understand the purpose of each function more clearly. The following is an example implementing the `formal` function for the translation scheme from Soloveichik et al., Proceedings of the National Academy of Sciences, 2010 [39].

```
function formal(s) =
    "? a b c" | "? . . ."
    where {
        a = short() ;
        b = long() ;
        c = short()
    };
```

Soloveichik et al. requires that the molecules representing formal species have one "history" domain, which is variable according to the reaction that produced those molecules. Thus, each formal species will not be mapped to a single implemented species, but it will be mapped to a class of species. To express this, here we used a question mark as a wildcard that can match any single unpaired domain. Recall that `short()` and `long()` are functions that generate new unique domains of respective lengths, so `formal` will return a new single-stranded molecule with three new distinct domains every time it is called. This way, no two formal species will share the same domain, as desired by the scheme. Fig. A.4 shows the molecule that will be constructed from this code.

Now, note that the following function constructs the B_i molecule from Fig. 3 of the same paper.

```
function Bi(x1, x2) =
    "d2 d3 d4" |
    ". . ."
```

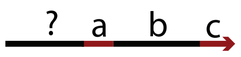


Figure A.4: The structure from `formal`

```
  where {
     d2 = x1.b ;
     d3 = x1.c ;
     d4 = x2.a
  };
```

Here, `x1` and `x2` are assumed to be `Structure` objects produced by the `formal` function. Thus, when we reference `x1.b` or `x1.c`, it will return the corresponding domains as was specified in the dot-paren description in `formal` (if `x1` or `x2` were not produced by `formal`, it might not have domains named `a`, `b`, or `c`, in which case it would result in a runtime error).

Finally, we can write down the following `main` function.

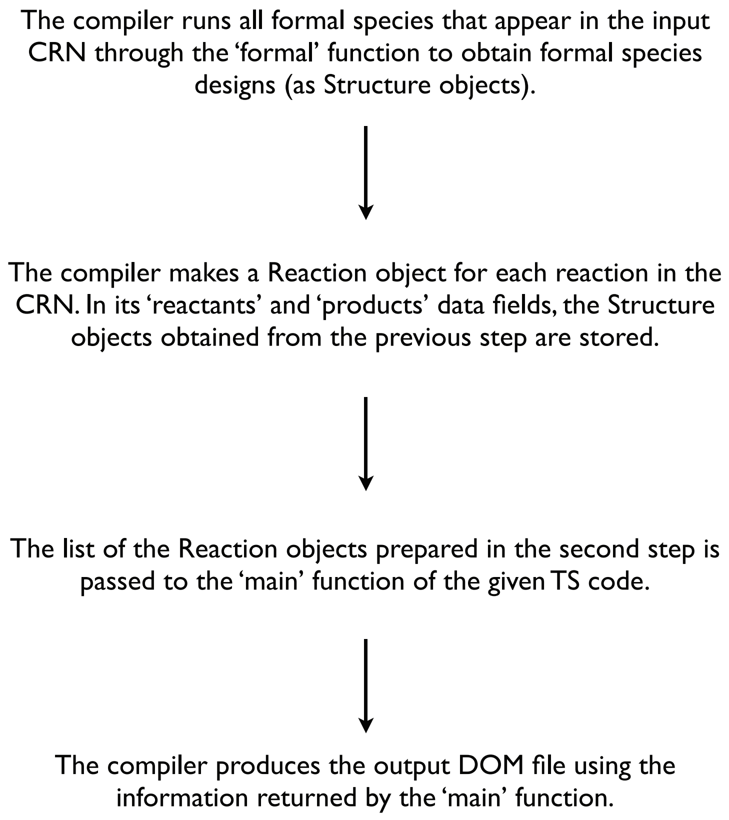
```
function main(crn) = infty(Bi(crn[0].reactants[0], crn[0].reactants[1]))
# 'infty' is a built-in function that takes a Structure instance and puts that the
    species is supposed to have "infinite" concentration. The resulting object will
    be a Solution instance.
```

The argument given to the `main` function, `crn`, is a list of `Reaction` objects that store the reactions from the input CRN. However, their `reactants` and `products` fields contain `Structure` objects that are obtained by running each species that appears in the given reaction through the `formal` function.

Thus, if we ran this translation scheme on the CRN consisting of one reaction $X_1 + X_2 \to X_3$, it will correctly translate all the formal species according to the scheme from Soloveichik et al., and produce the appropriate B_i molecule. However, as can obviously be seen, this main function ignores every reaction but the first one, it assumes that the first reaction is bimolecular, and most importantly, it does not produce any other auxiliary species than B_i. This is obviously not an implementation of a correct translation scheme, and it requires much more work to make it one. A complete implementation of the translation scheme from Soloveichik et al. will be presented in the next section.

To enhance the reader's understanding of the compiler, Fig. A.5 pictures how the compiler uses the TS code that the user provides.

Lastly, note that our compiler completely ignores the kinetic aspect of the system. Thus, we do not specify rate constants in our CRN language and we only specify the concentration of a species as 'infinite' or 'non-infinite.' This is a major limitation of our compiler and should be improved in



The compiler runs all formal species that appear in the input CRN through the 'formal' function to obtain formal species designs (as Structure objects).

The compiler makes a Reaction object for each reaction in the CRN. In its 'reactants' and 'products' data fields, the Structure objects obtained from the previous step are stored.

The list of the Reaction objects prepared in the second step is passed to the 'main' function of the given TS code.

The compiler produces the output DOM file using the information returned by the 'main' function.

Figure A.5: The compiler flow chart

the future.

A.8 Examples

```
A -> B + C
C -> D
```

Listing A.1: sample.crn ↑

```
#
# David Soloveichik's translation scheme from "DNA as a universal substrate
# for chemical kinetics", Proceedings of the National Academy of Sciences,
# 107: 5393-5398, 2010.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

function formal(s) = "? a b c"
                   | "? . . ."
    where {
        a = short() ;
        b = long() ;
        c = short() };

function ugate(s, l)
     = ["b c d + c* b* a*"         # G_i
       | "( ( ~ + ) ) .",
        "e + f c*"                 # T_i
       | "~ + ~ ."]
    where {
        a = s.a ;
        b = s.b ;
        c = s.c ;
        [d, e, g] = flip(map(gmac, l), 3) ;
        f = reverse(g) };

function gmac(s)
     = ["d a"
       | ". .",
        "d a b c +"
       | "( ( . . +",
        "a* d*"
       | ") )"]
    where {
        d = long() ;
        a = s.a ;
        b = s.b ;
        c = s.c };

function unimolecular(r) = infty(g) + infty(t)
    where
        [g, t] = ugate(r.reactants[0], r.products);

function bgate(s1, s2, l)
    = ["b c d + e f g + f* e* d* c* b* a*"          # L_i
      | "( ( ( + ( ( ~ + ) ) ) ) ) .",
        "h + i f*"                                  # T_i
      | "~ + ~ .",
        "b c d"                                     # B_i
      | ". . ."]
    where {
        a = s1.a ;
        b = s1.b ;
        c = s1.c ;
        d = s2.a ;
        e = s2.b ;
        f = s2.c ;
        [g, h, j] = flip(map(gmac, l), 3) ;
        i = reverse(j) };

function bimolecular(r) = infty(l) + infty(t) + infty(b)
    where
        [l, t, b] = bgate(r.reactants[0], r.reactants[1], r.products);

function main(crn) = sum(map(unimolecular, unirxn(crn))) +
                   sum(map(bimolecular, birxn(crn)))
    where
        crn = irrev_reactions(crn)
```

Listing A.2: soloveichik.ts ↑

```
sequence d1 : 5
sequence d2 : 15
sequence d3 : 5
sequence d4 : 5
sequence d5 : 15
sequence d6 : 5
sequence d7 : 5
sequence d8 : 15
sequence d9 : 5
sequence d10 : 5
sequence d11 : 15
sequence d12 : 5
sequence d13 : 15
sequence d14 : 15
sequence d15 : 15
# Formal species
A :
? d1 d2 d3
? . . .
C :
? d4 d5 d6
? . . .
B :
? d7 d8 d9
? . . .
D :
? d10 d11 d12
? . . .
# Constant species
automatic4 :
d2 d3 d13 d7 d14 d4 + d3* d2* d1*
( ( . . . . + ) ) .
automatic5 :
d13 d7 d8 d9 + d14 d4 d5 d6 + d4* d14* d7* d13* d3*
( ( . . + ( ( . . + ) ) ) ) .
automatic6 :
d5 d6 d15 d10 + d6* d5* d4*
( ( . . + ) ) .
automatic7 :
d15 d10 d11 d12 + d10* d15* d6*
( ( . . + ) ) .
```

Listing A.3: sample.dom ↑

In this section, we will investigate a full set of input and output codes to provide a broad picture of the compilation process. In other words, we will look into how the “sample.dom” file above is produced from “sample.crn” and “soloveichik.ts”.

First of all, all the formal species (A, B, C, and D in this case) are translated using the ‘formal’ function from the TS file. Note that the formal function is written such that we produce three new unique domains for each formal species. Accordingly, the compiler obtains the following after this translation. We can check that the length of each domain agrees with the specification from the formal function. (Here, 5 and 15 are being used as default lengths for ‘short’ and ‘long’ domains.)

```
# Formal species
```

```
A :
? d1 d2 d3
? . . .
C :
? d4 d5 d6
? . . .
B :
? d7 d8 d9
? . . .
D :
? d10 d11 d12
? . . .
```

Then, the compiler will run the 'main' function of the TS file, supplying the input CRN as the argument. In this case, the argument `crn` will have the following value: `[Reaction([A], [B, C], False), Reaction([C], [D], False)]`. (`Reaction([A], [B, C], False)` denotes a `Reaction` object with `reactant`, `products`, and `reversible` data fields having values `[A]`, `[B, C]`, and `False` respectively. `A`, `B`, `C`, and `D` here represent the `Structure` objects that were generated during the first step of compilation using the formal function.)

```
function main(crn) = sum(map(unimolecular, unirxn(crn))) +
                     sum(map(bimolecular, birxn(crn)))
    where
        crn = irrev_reactions(crn)
```

First of all, since we have a where clause, we create a local binding `crn` which has the value `irrev_reactions(crn)`. Since our input CRN does not contain any reversible reaction, the new `crn` will be the same as the old `crn`. Also, note that `crn` only contains unimolecular reactions, so when the above code computes `unirxn(crn)` and `birxn(crn)`, the former will be the same as `crn` itself and the latter will be an empty list. Therefore, the return value of our main function will be `sum(map(unimolecular, unirxn(crn)))`.

To compute this value, we need to apply the function `unimolecular` to each item in `unirxn(crn)` and take the sum of the resulting values. Since `unirxn(crn)` is `[Reaction([A], [B, C], False), Reaction([C], [D], False)]`, we need to first compute `unimolecular(Reaction([A], [B, C], False))` and `unimolecular(Reaction([C], [D], False))`, and compute their sum. Now, we need to study the `unimolecular` function.

```
function unimolecular(r) = infty(g) + infty(t)
    where
        [g, t] = ugate(r.reactants[0], r.products);
```

Suppose the argument `r` was `Reaction([A], [B, C], False)`. To compute the return value of `unimolecular(r)`, we first need to evaluate the value of `g` and `t` from the where clause. That is, we need to evaluate `ugate(r.reactants[0], r.products)` first. Clearly, `r.reactants[0]` is `A` and `r.products` is `[B, C]` where `A`, `B`, and `C` are `Structure` objects. Thus, the code will call `ugate(A, [B, C])` and try to pattern match the result into `[g, t]`.

```
function ugate(s, l)
     = ["b c d + c* b* a*"           # G_i
       | "( ( ~ + )   )   .",
         "e + f c*"                  # T_i
       | "~ + ~ ."]
    where {
        a = s.a ;
        b = s.b ;
        c = s.c ;
        [d, e, g] = flip(map(gmac, l), 3) ;
        f = reverse(g) };
```

Note that the value of the first argument `s` is `A`, which is a `Structure` object created with the `formal` function. If we look at the definition of `formal`, there are three domains that are used to specify the structure: `a`, `b`, and `c`. Thus, when we write `s.a`, `s.b`, and `s.c`, we reference the corresponding `Domain` objects, i.e., `d1`, `d2`, and `d3`, respectively. (If `s` were `B`, we would have referenced `d7`, `d8`, and `d9` instead.) Thus, `a`, `b`, and `c` in the where clause are `d1`, `d2`, and `d3` respectively, but in order to evaluate `d`, `e`, `f`, and `g`, we need to call yet another function, `gmac`. Recall that the value of the second argument `l` was `[B, C]`, which is a list. Thus, when we call `map(gmac, l)`, we will apply `gmac` on `B` and `C` individually, and store the results in a list.

```
function gmac(s)
     = ["d a"
       | ". .",
         "d a b c +"
       | "( ( . . +",
         "a* d*"
       | ") )"]
    where {
        d = long() ;
        a = s.a ;
        b = s.b ;
        c = s.c };
```

Let us first evaluate `gmac(B)`. We first need to evaluate the assignments in the where clause. Note that we need to generate a new unique domain for `d`. Since we already assigned `d1` through `d12` to formal species, we will assign `d` a new `Domain` object named `d13`. Also, since `s` is `B` here, as we discussed above, `s.a`, `s.b`, and `s.c` are `d7`, `d8`, and `d9` respectively. Thus, `a`, `b`, and `c` become `d7`, `d8`, and `d9`. That is, `gmac(B)` has the value `[Structure("d a", ".  .", (d=d13, a=d7)), Structure("d a` b c +", "((. . +", (a=d7, b=d8, c=d9, d=d13)), Structure("a* d*", "))", (a=d7, d=d13))]. Similarly, gmac(C) will have the value [Structure("d a", ". .", (d=d14, a=d4)), Structure("d a b c +", "((. . +", (a=d4, b=d5, c=d6, d=d14)), Structure("a* d*", "))", (a=d4, d=d14))].

Now, this means that back in the ugate function, the value of map(gmac, l) has the form [[S.., S.., S..], [S.., S.., S..]]. Thus, if we compute flip(map(gmac, l), 3), it will turn into the form of [[S.., S..], [S.., S..], [S.., S..]]. Now we can evaluate all bindings that appear in the ugate function, which is shown in Fig. A.6.

a	d1
b	d2
c	d3
d	[Structure("d a", ". .", (d=d13, a=d7)), Structure("d a", ". .", (d=d14, a=d4))]
e	[Structure("d a b c +", "((. . +", (a=d7, b=d8, c=d9, d=d13)), Structure("d a b c +", "((. . +", (a=d4, b=d5, c=d6, d=d14))]
g	[Structure("a* d*", "))", (a=d7, d=d13)), Structure("a* d*", "))", (a=d4, d=d14))]
f	[Structure("a* d*", "))", (a=d4, d=d14)), Structure("a* d*", "))", (a=d7, d=d13))]

Figure A.6: The values of bindings in the where clause of ugate

Now, note that there are tilde (∼) signs in the definition of the two Structure objects that appear in ugate. When a domain in a Structure definition has a tilde, that domain should have a list of Structure objects as its value like d, e, and f in this case. When this happens, BioCRN concatenates the structural specification given in that list and puts the resulting structure in the place of that domain. For example, the two Structure objects being specified in the ugate function now have the following structure (Fig. A.7).

Now, back in the unimolecular function, these two Structure objects computed in ugate are put into the variables g and t respectively. Then, the return value of unimolecular is simply infty(g) + infty(t), which is a Solution object that contains the two species described by g and t.

We are finally back in the main function, but in order to compute the final return value, we need to repeat the same procedure as above for the second reaction in crn. We will omit that second round here because it would be too tedious and laborious to follow it step by step. However, it is clear that the second round will also return a Solution object and thus the value of map(unimolecular, unirxn(crn)) would be a list of Solution objects. Finally applying sum on that list, we will obtain a Solution object that contains every molecule that appears at least once in any Solution object in that list. The return value of main will be, as promised, a Solution object that contains all

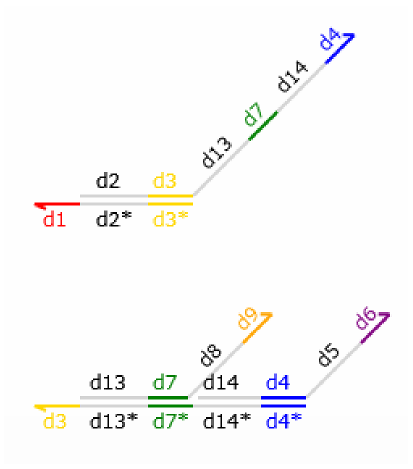


Figure A.7: G_i (above) and T_i (below) molecules from the `ugate` function. Figure generated using Visual DSD.

auxiliary molecules needed to implement the reactions in the given CRN. The molecules contained in this `Solution` object, along with the formal species structures that were produced using the `formal` function, will constitute the final output of the compiler.

Note that this example does not reveal how the above translation scheme will deal with a bi-molecular reaction. However, that should be fairly easy to figure out for those that have followed the discussion above, and we leave it as an exercise for the reader.

Appendix B

Input Data Used for Test Runs

B.1 Translation Schemes

```
#
# David Soloveichik's translation scheme from "DNA as a universal substrate
# for chemical kinetics", Proceedings of the National Academy of Sciences,
# 107: 5393-5398, 2010.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

class formal(s) = "? a b c"
                | "? . . ."
    where {
        a = short() ;
        b = long() ;
        c = short() };

class ugate(s, l)
     = ["b c d + c* b* a*"          # G_i
       | "( ( ~ + ) ) .",
        "e + f c*"                  # T_i
       | "~ + ~ ."]
    where {
        a = s.a ;
        b = s.b ;
        c = s.c ;
        [d, e, g] = flip(map(gmac, l), 3) ;
        f = reverse(g) };

macro gmac(s)
     = ["d a"
       | ". .",
        "d a b c +"
      | "( ( . . +",
        "a* d*"
      | ") )"]
    where {
        d = long() ;
        a = s.a ;
        b = s.b ;
        c = s.c };

module unimolecular(r) = infty(g) + infty(t)
    where
        [g, t] = ugate(r.reactants[0], r.products);

class bgate(s1, s2, l)
    = ["b c d + e f g + f* e* d* c* b* a*"        # L_i
     | "( ( ( + ( ( ~ + ) ) ) ) ) .",
       "h + i f*"                                 # T_i
     | "~ + ~ .",
       "b c d"                                    # B_i
     | ". . ."]
    where {
        a = s1.a ;
        b = s1.b ;
        c = s1.c ;
        d = s2.a ;
        e = s2.b ;
        f = s2.c ;
        [g, h, j] = flip(map(gmac, l), 3) ;
        i = reverse(j) };

module bimolecular(r) = infty(l) + infty(t) + infty(b)
    where
        [l, t, b] = bgate(r.reactants[0], r.reactants[1], r.products);

module main(crn) = sum(map(unimolecular, unirxn(crn))) +
                   sum(map(bimolecular, birxn(crn)))
    where
        crn = irrev_reactions(crn)
```

Listing B.1: soloveichik.ts ↑

```
#
# Lulu Qian's translation scheme from "Efficient Turing-universal
# computation with DNA polymers", Lecture Notes in Computer Science,
# 6518: 123-140, 2011.
```

```
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

global toehold = short();

class formal(s) = "hist th recog"
                | ". . ."
    where {
        hist = long();
        th = toehold;
        recog = long()};

macro reactant(s)
    = ["a b +" | "( ( +",
       "b* a*" | ") )",
       "a b" | ". ."]
    where {
        a = s.recog;
        b = s.th };

macro product(s)
    = ["a b c +" | "( ( . +",
       "b* a*" | ") )"]
    where {
        a = s.hist;
        b = s.th;
        c = s.recog };

class maingate(r, p)
    = [["a b f + f* c d e*"
     | "~ ~ ( + )  ~ ~ .",
       "e f"
     | ". ."], extra]
    where {
        [a, temp, extra] = flip(map(reactant, r), 3);
        d = reverse(temp);
        [b, temp2] = flip(map(product, p), 2);
        c = reverse(temp2);
        e = toehold;
        f = long() };

class suppgate(s)
    = "a b" | ". ."
    where {
        a = s.th;
        b = s.hist };

module rxn(r) = infty(g) + infty(h) + sum(map(infty, map(suppgate, r.products))) +
    sum(map(infty, gates))
    where
        [[g, h], gates] = maingate(r.reactants, r.products);

module main(crn) = sum(map(rxn, irrev_reactions(crn)))
```

Listing B.2: qian.ts ↑

```
#
# Lulu Qian's translation scheme from "Efficient Turing-universal
# computation with DNA polymers", Lecture Notes in Computer Science,
# 6518: 123-140, 2011.
#
#   Note : this was a scheme that the authors of the above paper discovered
#          while they were working on that paper, but it was later shown to
#          have a bug and was not included in the published version of the
#          paper. It was reproduced here for demonstrative purposes.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

global toehold = short();

class formal(s) = "hist th recog"
                | ". . ."
    where {
        hist = long();
        th = toehold;
        recog = long()};

macro reactant(s)
    = ["a b +" | "( ( +",
       "b* a*" | ") )",
       "a b" | ". ."]
    where {
        a = s.recog;
        b = s.th };

macro product(s)
    = ["a b c +" | "( ( . +",
       "b* a*" | ") )"]
    where {
        a = s.hist;
        b = s.th;
        c = s.recog };

class maingate(r, p)
    = [["a b + c d e*"
      | "~ ~ +  ~ ~ ."], extra]
    where {
        [a, temp, extra] = flip(map(reactant, r), 3);
        d = reverse(temp);
        [b, temp2] = flip(map(product, p), 2);
        c = reverse(temp2);
        e = toehold;
        f = long() };

class suppgate(s)
    = "a b" | ". ."
    where {
        a = s.th;
        b = s.hist };

module rxn(r) = infty(g) + sum(map(infty, map(suppgate, reverse(tail(reverse(p))))))
    + infty("t b t" | ". . .") + sum(map(infty, gates))
    where {
        p = if len(r.products) == 0 then [formal(0)] else r.products;
        [[g], gates] = maingate(r.reactants, p);
        t = toehold;
        b = (p[len(p) - 1]).hist };

module main(crn) = sum(map(rxn, irrev_reactions(crn)))
```

Listing B.3: qian_bug.ts ↑

```
#
# Luca Cardelli's translation scheme from "Strand Algebras for DNA
# Computing", Natural Computing, 10: 407-428, 2011. Can be found at
# http://lucacardelli.name/
#
#   Note : this implements the 'fork' and 'join' gates from the paper,
#          which are pointed out to be problematic in the paper. This
#          implements the schemes shown in Figs 4-7.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

class formal(s) = "? t b"
```

```
                   | "? . ."
    where {
        t = short();
        b = long() };

class signal() = "? t b"
                 | "? . ."
    where {
        t = short();
        b = long() };

class forkgate(x, y, z)
    = [ "xb yt yb + a zt zb + zt* a* yt* xb* xt*"
      | "(  (  .  + (  (  .  + )   )  )   )   .",
        "yt a zt"
      | ".  . ."]
    where {
        xt = x.t;
        xb = x.b;
        yt = y.t;
        yb = y.b;
        zt = z.t;
        zb = z.b;
        a = long() };

module fork2(x, y, z) = infty(gb) + infty(gt)
    where
        [gb, gt] = forkgate(x, y, z);

class joingate(x, y, z)
    = [ "xb yt + yb a + b zt zb + zt* b* a* yb* yt* xb* xt*"
      | "(  (  + (  ( + ( (  .  + )   )  )  )   )   )   .",
        "a b zt"
      | ". . .",
        "xb yt"
      | ".  .",
        "yb a"
      | ".  ."]
    where {
        xt = x.t;
        xb = x.b;
        yt = y.t;
        yb = y.b;
        zt = z.t;
        zb = z.b;
        a = short();
        b = long() };

module join2(x, y, z) = infty(gb) + infty(gt) + infty(r1) + infty(r2)
    where
        [gb, gt, r1, r2] = joingate(x, y, z);

class transgate(x, y)
    = [ "xb yt yb + a + a* yt* xb* xt*"
      | " (  (  . + ( + )  )   )   .",
        "yt a"
      | " . ."]
    where {
        xt = x.t;
        xb = x.b;
        yt = y.t;
        yb = y.b;
        a = long() };

module transducer(x, y) = infty(gb) + infty(gt)
    where
        [gb, gt] = transgate(x, y);

class anhlgate(x)
    = "xb + xb* xt*"
    | " ( +  )   ."
    where {
        xt = x.t;
        xb = x.b };

module annihilator(x) = infty(g)
    where
        g = anhlgate(x);

module fork(r, p) =
    if len(p) == 2 then
        fork2(r, p[0], p[1])
    else
        fork2(i, p[0], p[1]) + join2(p[0], p[1], i) +
        fork(r, tail(tail(p)) + [i])
        where
            i = signal();

module join(r, p) =
    if len(r) == 2 then
```

```
        join2(r[0], r[1], p)
    else
        fork2(i, r[0], r[1]) + join2(r[0], r[1], i) +
        join(tail(tail(r)) + [i], p)
        where
            i = signal();

module joinfork(r, p) =
    if len(p) == 0 then
        if len(r) == 1 then
            annihilator(r[0])
        else
            fork(r, i) + annihilator(i)
            where
                i = signal()
    elseif len(r) == 1 and len(p) == 1 then
        transducer(r[0], p[0])
    elseif len(r) == 1 then
        fork(r[0], p)
    elseif len(p) == 1 then
        join(r, p[0])
    else
        join(r, i) + fork(i, p)
        where
            i = signal();

module reaction(r) = joinfork(r.reactants, r.products);

module main(crn) = sum(map(reaction, crn))
```

Listing B.4: cardelli_fj.ts ↑

```
#
# Luca Cardelli's translation scheme from "Strand Algebras for DNA
# Computing", Natural Computing, 10: 407-428, 2011. Can be found at
# http://lucacardelli.name/
#
#   Note : this implements the scheme shown in Fig 9. Note that the gate
#          species at the bottom seems like one molecule, but it is in
#          fact two molecules because of an arrowhead separating them.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

class formal(s) = "? t b"
                | "? . ."
```

```
    where {
        t = short();
        b = long() };

class signal() = "? t b"
                | "? . ."
    where {
        t = short();
        b = long() };

macro input(s)
    = ["xt + xb"
     | "(  + (",
       "xb* xt*"
     | ")   )"]
    where {
        xt = s.t;
        xb = s.b };

macro output(s)
    = ["yh yt yb +"
     | "(  (  . +",
       "yt* yh*"
     | ")   )",
       "yh yt"
     | ".  . "]
    where {
        yh = long();
        yt = s.t;
        yb = s.b };

macro gc(s)
    = ["xt cb + ct xb +"
     | " (  ( + (  (  +",
       "xb* ct* cb* xt*"
     | " )   )   )   )",
       "cb ct"
     | " .  .",
       "cb + cb* xt*"
     | "(  +  )   .",
       "xb + xb* ct*"
     | "(  +  )   . "]
    where {
        xt = s.t;
        xb = s.b;
        cb = long();
        ct = short() };

class nmgate(r, p)
    = [ "x1b in1 t + out1 out2 t* in2 x1b* x1t*"
      | " (   ~   ( + ~     ~   )   ~    )     .",
        "t out3"
      | ".  ~",
        "x1b + gc1 t* gc2 x1b*"
      | " (  +  ~   .   ~   )",
      small,
      big1,
      big2]
    where {
        x1b = (r[0]).b;
        x1t = (r[0]).t;
        t = short();
        [in1, in2r] = flip(map(input, tail(r)), 2);
        [out1, out2r, out3] = flip(map(output, p), 3);
        in2 = reverse(in2r);
        out2 = reverse(out2r);
        [gc1, gc2r, small, big1, big2] = flip(map(gc, tail(r)), 5);
        gc2 = reverse(gc2r) };

module nmmodule(r, p) = infty(gb) + infty(gt) + infty(gc1) + sum(map(infty, gc2)) +
    sum(map(infty, gc3)) + sum(map(infty, gc4))
    where
        [gb, gt, gc1, gc2, gc3, gc4] = nmgate(r, p);

module reaction(r) =
    if len(r.products) == 0 then
        nmmodule(r.reactants, [i])
        where
            i = signal()
    else
        nmmodule(r.reactants, r.products);

module main(crn) = sum(map(reaction, irrev_reactions(crn)))
```

Listing B.5: cardelli_c.ts ↑

```
#
# Luca Cardelli's translation scheme from "Two-Domain DNA Strand
# Displacement", Mathematical Structures in Computer Science.
# Can be found at http://lucacardelli.name/
#
```

```
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

global toehold = short();

class formal(s) = "t x"
                | ". ."
    where {
        t = toehold;
        x = long() };

class signal() = "t x"
               | ". ."
    where {
        t = toehold;
        x = long() };

macro output(s)
    = ["t x +"
      | "( ( +",
        "x* t*"
      | ")  )",
        "x t"
      | ". ."]
    where {
        t = toehold;
        x = s.x };

class outgate(r, p, a)
    = [ "x + out1 t a + t* a* t* out2 x*"
      | "( +  ~   ( ( + .  )  )   ~   )"] + l
    where {
        x = (r[0]).x;
        [out1, out2r, l] = flip(map(output, p), 3);
        out2 = reverse(out2r);
        t = toehold };

class oneinput(r, p)
    = [ "x t + a t + a + a* t* a* t* x* t*"
      | "( ( + ( ( + ( + )  )  )  )  )  . ",
        "t a"
      | ". ."] + outgate(r, p, a)
    where {
        a = long() ;
        x = (r[0]).x ;
        t = toehold };

class twoinput(r, p)
    = [ "x t + y t + a t + a + a* t* a* t* y* t* x* t*"
      | "( ( + ( ( + ( ( + ( + )  )  )  )  )  )  )  . ",
        "t a"
      | ". .",
        "b y + t* y* b* t*"
      | "( ( + .  )  )  . "] + outgate(r, [bs] + p, a)
    where {
        x = (r[0]).x;
        y = (r[1]).x;
        t = toehold;
        a = long();
        bs = signal();
        b = bs.x };

module reaction(r) =
    if len(r.reactants) == 1 then
        sum(map(infty, oneinput(r.reactants, r.products)))
    else
        sum(map(infty, twoinput(r.reactants, r.products)));

module main(crn) = sum(map(reaction, crn))
```

Listing B.6: cardelli_2d.ts ↑

```
#
# Luca Cardelli's translation scheme from "Strand Algebras for DNA
# Computing", Natural Computing, 10: 407-428, 2011. Can be found at
# http://lucacardelli.name/
#
#   Note : this implements the scheme shown in Fig 9, without the garbage
#          collection module. Note that the gate species at the bottom
#          seems like one molecule, but it is in fact two molecules
#          because of an arrowhead separating them.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

class formal(s) = "? t b"
                | "? . ."
    where {
        t = short();
        b = long() };

```

```
class signal() = "? t b"
               | "? . ."
    where {
        t = short();
        b = long() };

macro input(s)
    = ["xt + xb"
     | "(  + (",
       "xb* xt*"
     | ")   )"]
    where {
        xt = s.t;
        xb = s.b };

macro output(s)
    = ["yh yt yb +"
     | "(   (  . +",
       "yt* yh*"
     | ")   )",
       "yh yt"
     | ".  . "]
    where {
        yh = long();
        yt = s.t;
        yb = s.b };

class helper(r) =
    if len(r) < 2 then
        []
    else
        ["b t" | ". ."] + helper(tail(r))
        where {
            b = (r[0]).b;
            t = (r[1]).t };

class nmgate(r, p)
    = [ "x1b in1 t + out1 out2 t* in2 x1b* x1t*"
      | " (   ~  ( + ~    ~    )  ~   )    .",
        "t out3"
      | ".  ~",
        "l t"
      | ". ."] + inf
    where {
        x1b = (r[0]).b;
        x1t = (r[0]).t;
        t = short();
        [in1, in2r] = flip(map(input, tail(r)), 2);
        inf = helper(r);
        l = ((reverse(r))[0]).b;
        [out1, out2r, out3] = flip(map(output, p), 3);
        in2 = reverse(in2r);
        out2 = reverse(out2r) };

module nmmodule(r, p) = sum(map(infty, sp))
    where
        sp = nmgate(r, p);

module reaction(r) =
    if len(r.products) == 0 then
        nmmodule(r.reactants, [i])
        where
            i = signal()
    else
        nmmodule(r.reactants, r.products);

module main(crn) = sum(map(reaction, crn))
```

Listing B.7: cardelli_c_nogc.ts ↑

```
#
# Luca Cardelli's translation scheme from "Two-Domain DNA Strand
# Displacement", Mathematical Structures in Computer Science.
# Can be found at http://lucacardelli.name/
#
#   NOTE : The garbage collection module was omitted in this implementation.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

global toehold = short();

class formal(s) = "t x"
                | ". ."
    where {
        t = toehold;
        x = long() };

class signal() = "t x"
               | ". ."
    where {
        t = toehold;
        x = long() };

macro output(s)
    = ["t x +"
      | "( ( +",
        "x* t*"
      | ") )",
        "x t"
      | ". ."]
    where {
        t = toehold;
        x = s.x };

class outgate(r, p, a)
    = [ "x + out1 t a + t* a* t* out2 x*"
       | "( +  ~   ( ( + .  )  )   ~   )"] + l
    where {
        x = (r[0]).x;
        [out1, out2r, l] = flip(map(output, p), 3);
        out2 = reverse(out2r);
        t = toehold };

class oneinput(r, p)
    = [ "x t + a t + a + a* t* a* t* x* t*"
       | "( ( + ( ( + ( + )  )  )  )  )  . ",
         "x t"
       | ". .",
         "t a"
       | ". ."] + outgate(r, p, a)
    where {
        a = long();
        x = (r[0]).x;
        t = toehold };

class twoinput(r, p)
    = [ "x t + y t + a t + a + a* t* a* t* y* t* x* t*"
       | "( ( + ( ( + ( ( + ( + )  )  )  )  )  )  )  . ",
         "t a"
       | ". .",
         "x t"
       | ". .",
         "y t"
       | ". ."] + outgate(r, p, a)
    where {
        x = (r[0]).x;
        y = (r[1]).x;
        t = toehold;
        a = long() };

module reaction(r) =
    if len(r.reactants) == 1 then
        sum(map(infty,oneinput(r.reactants, r.products)))
    else
        sum(map(infty,twoinput(r.reactants, r.products)));

module main(crn) = sum(map(reaction, crn))
```

Listing B.8: cardelli_2d_nogc.ts ↑

```
#
# Lulu Qian's translation scheme from "Efficient Turing−universal
# computation with DNA polymers", Lecture Notes in Computer Science,
# 6518: 123−140, 2011.
#
#   NOTE : This is the corrected version of the scheme in the above paper
#          which was provided to us by the authors of that paper.
#
# Coded by Seung Woo Shin (lagnared@gmail.com).
#

global toehold = short();

class formal(s) = "hist th recog"
                | ". . ."
    where {
        hist = long();
        th = toehold;
        recog = long() };

macro reactant(s)
    = ["a b +" | "( ( +",
       "b* a*" | ") )",
       "a b" | ". ."]
    where {
        a = s.recog;
        b = s.th };

macro product(s)
    = ["a b c +" | "( ( . +",
       "b* a*" | ") )"]
    where {
        a = s.hist;
        b = s.th;
        c = s.recog };

class maingate(r, p)
    = [[”a f + g e + b + f + f* c e* g* f* d e*”
      | ”˜ ( + ( ( + ˜ + ( + ) ˜ ) ) ) ˜ .”,
        ”e f g”
      | ”. . .”], extra]
    where {
        [a, temp, extra] = flip(map(reactant, r), 3);
        d = reverse(temp);
        [b, temp2] = flip(map(product, p), 2);
        c = reverse(temp2);
        e = toehold;
        f = long();
        g = long() };

class suppgate(s)
    = ”a b” | ”. .”
    where {
        a = s.th;
        b = s.hist };

module rxn(r) = infty(g) + infty(h) + sum(map(infty, map(suppgate, r.products))) + sum(map(infty, gates))
    where
        [[g, h], gates] = maingate(r.reactants, r.products);

module main(crn) = sum(map(rxn, crn))
```

Listing B.9: qian_fixed.ts ↑

B.2 CRNs

```
A -> B
B + B -> A
B ->
```

Listing B.10: crn1.crn ↑

```
X3 + X4 -> X5
X5 -> X1
```

```
X1 + X2 -> X3
```

Listing B.11: crn2.crn ↑

```
X3 + X4 -> X5
X5 -> X1
X1 + X2 -> X3
X2 + X6 -> X7
X1 + X5 -> X9
X3 + X7 -> X9 + X10 + X2 + X4
X1 -> X2 + X2
```

Listing B.12: crn3.crn ↑

```
A + B -> B + B
B + C -> C + C
C + A -> A + A
```

Listing B.13: crn4.crn ↑