M.Sc Thesis

A Programming Language and Compiler for Molecular Robot Swarm in Biomedical Application

Inbal Wiesel-Kapah

A thesis submitted to the faculty of
Bar-Ilan University
in partial fulfillment of the requirements for the degree of

Master of Science

Advisor: Gal A. Kaminka
Noa Agmon
and Ido Bachelet

Department of Computer Science

Bar-Ilan University

September 2016

ABSTRACT

A Programming Language and
Compiler for Molecular Robot Swarm
in Biomedical Application

Inbal Wiesel-Kapah
Department of Computer Science
Master of Science

The development of new drugs and treatments is one of the fundamental activities in medicine. The main challenge is to translate medical knowledge to a drug that treats a medical condition. This process is long and complex, much because of the different areas of expertise that are required to produce a drug: physiological knowledge about the disease or condition being treated, medical ideas about possible remedies (in the sense of how to fundamentally address the condition), chemical and biological knowledge of how these remedies can be implemented in the form of drug or treatment, and the testing of those to verify their safety. Recently, molecular robots on the nanometer scale (nanobots) have been proposed to alleviate the complexity of this task. Nanobots deliver compounds to specific targets in the body, without worrying about side-effects and long testing periods. However, every nanobot must be designed by an expert, for the specific case. Complex procedures, or those requiring complex targeting, lead to prohibitively complex nanobot designs. Instead, we advocate an approach where a mixture (cocktail) of heterogeneous simple nanobots carries out complex tasks. We present *Tolkien*, a novel framework for programing biomedical nanobots, by synthesizing heterogeneous nanobot mixtures from on high-level biomedical programming code, and generic nanobot designs that can be specialized, e.g., for specific payloads. Specifically, we present the *Athelas* rule-based programming language (for describing medical procedures), and the *Bilbo* compiler that transforms Athelas programs into combinations of specialized nanobots, that together execute the procedures prescribed.

ACKNOWLEDGMENTS

I want to thank my dear husband, Neria, who without his support I could not have done this. I would like to thank my parents, Yossi and Michal, who raised me to believe that I can do anything I want, if I just believe in myself and work hard enough. Of course I want to and must thank Gal, my main advisor, who has taught and shown me how to research and explore in the best way, and that "only those who see the invisible can do the impossible". Thanks for being such a great teacher and for making the process so enjoyable as well. Noa and Ido, thanks for all your insightful advice, helpful tips, and the precious time you gave me. Noa, special thanks for your help with the proofs as well.

Finally and also most importantly, I want to thank God, Hashem-Yitbarch, for all of the endless good He does for me, and for the meaning and purpose he gives to my life. Of course no thesis can be written without His will and desire, so I guess that mine should add some good to our world. I can only hope and believe so.

Contents

List of Algorithms

Chapter 1

Introduction

1.1 Molecular Robots for Medicine

The development of new pharmaceutical treatments is one of the fundamental activities of medicine. The main challenge is to translate medical knowledge and treatment ideas to a pharmaceutical recipe. This process is long and complex, requiring widely varying expertise in multiple areas of science and medicine.

Conceptually, there are two main areas of expertise in the process, each responsible for a different stage in the drug development process (Figure 1.1). The first is medical expertise which focuses on understanding the medical problem, and proposing a treatment approach. This is done, for example, in terms of which specific substances should be delivered to what specific organ in the body in order to solve the problem. The second is the biomedical expertise which focuses on developing a pharmaceutical which implements the approach, checking for feasibility and toxicity to the body, etc. For example, using a chemical compound to carry the substances to the area of interest. The pharmaceutical will be noted as the *chemical implementation*. After the *chemical implementation* is determined, the medicine is produced according to it by the drug producer.

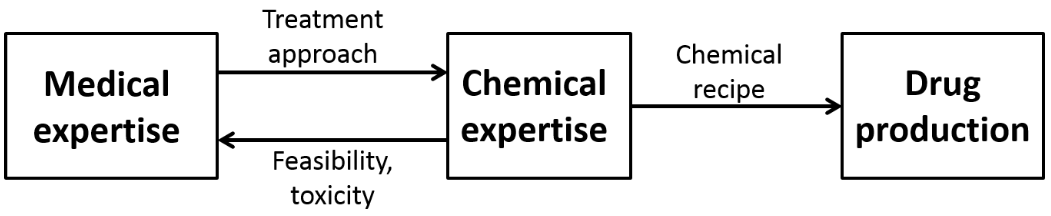


Figure 1.1 Drug development process today

The main challenge is how to find a feasible recipe; the ingredients and their dosage, such that the medicine will do the desired job. In particular, the necessity to determine and address side effects is a particularly prohibitive component in the drug development process costs and duration, and in its usage. That is because that even if the implementation for delivering the drug to the desired location is known, there is still need to make sure that it does not cause side effects that will lead to secondary damages. The issue is so severe, that sometimes a toxic drug is used in combination with secondary drugs to combat its side effects, as is often the case in chemotherapy.

One difficulty in this process lies with the need for very varied expertise and therefor cumbersome knowledge sharing between the medical expert and the chemical expert, as the first raises ideas for treatments and the second tries to translate them into potential drugs, while updating the first about their potential toxicity and feasibility. The complex cooperation leads to long development times. An illustration of the process is shown in Figure 1.1.

In an effort to curb these difficulties, molecular robotics has emerged as a promising approach for *targeted drug delivery*. Molecular robotics is a field that explores nanometer-scale robots (affectionally called "nanobots" in this thesis), that can operate inside a living body [1, 9]. Nanobots can carry out simple molecular actions, such as releasing a molecular payload only under some environmental conditions, and shield the body from toxic payloads [10, 30, 36]. If used as a platform for drug delivery, a nanobot can, in principle, carry out simple pharmaceutical tasks, much alleviating the need for experiments (as the drugs are released only in the presence of their targets). An illustration of the process is shown in Figure 1.2, when this time, the *chemical implementation* is replaced by a *nanobots swarm recipe* which developed by a nanobots expert.

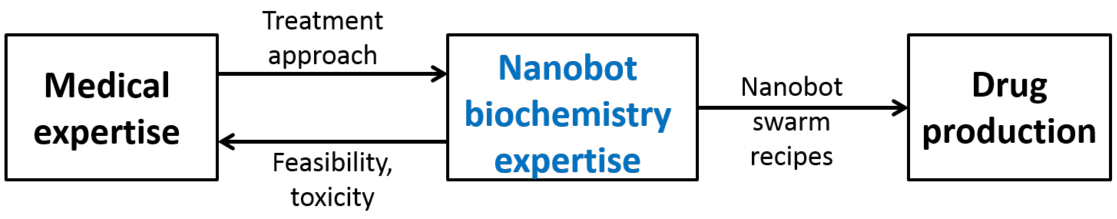


Figure 1.2 Drug development's process as for Nanorobotics.

So the uses of nanobots in that process decreases the feasibility and toxicity problems significantly [3]. However, today, every nanobot is designed by an expert, for a specific task. Furthermore, as each such design is specific to the treatment approach, it must be re-tested for side-effects. Thus a key challenge remains, of the varied expertise that is required for drug development: medical expertise must meet nanobot design expertise.

1.2 The Tolkien project

To address this challenge, the ***Tolkien project*** was founded, to research and develop **programmable** nanobot technology. The project is inspired by modern software development environments. In these, there is a strict separation between different layers of abstraction and areas of technical expertise. Application programmers, for example, do not know—nor care about—the instruction set used by the CPU, or where information is stored in memory. A compiler translates programs in high-level languages used by programmers, to the instruction set of the specific CPU (known as "machine language"). Additional levels of abstraction, e.g., for memory management and even for the instruction set itself, are provided by linking, operating systems, etc.

Analogously, the Tolkien project aims to allow medical professionals to directly program treatments in a human- and machine- readable medicine programming language [19, 34]. A compiler then translates programs in this language to a representation that describes the nanobots implementing the treatment. To do this, the compiler relies on a library of nanobot arch-types, and specializes them to create the specific nanobots required. An illustration of the process is shown in Figure 1.3.

By relying on this library, the Tolkien project avoids the significant challenges of automatically synthesizing, from first principles, nanobot designs that carry out a program. Complex programs (e.g., those requiring complex targeting), would require complex nanobot designs, well beyond current automated design capabilities. This is equivalent to a compiler that creates a new CPU for each program. Moreover, each such new design would need to be extensively tested.

Alternatively, we advocate an approach where a swarm of heterogeneous simple nanobots is synthesized to carry out complex tasks. The key is to rely on *generic* nanobot platforms, provided as a library, that can be specialized (parameterized) for specific payloads or targeting capabilities. Given a library of generic nanobot designs, Tolkien's compiler creates a swarm of differently-specialized nanobots (all based on the same generic design, or mixing designs), to carry out a task that no one specific type of nanobot can carry out. Moreover, the generic designs can be tested once for safety, and their specializations may be allowed to fast-track through testing[1].

Using Tolkien, we hope the design of a new medical treatment will be done one day by programming rather than by experiments. Medical experts will program treatments. Molecular roboticists will develop generic nanobots. And compilers will synthesize swarms of nanobots that carry out the programs, with performance and safety guarantees.

[1]This is not yet determined by the FDA, as for as we know.

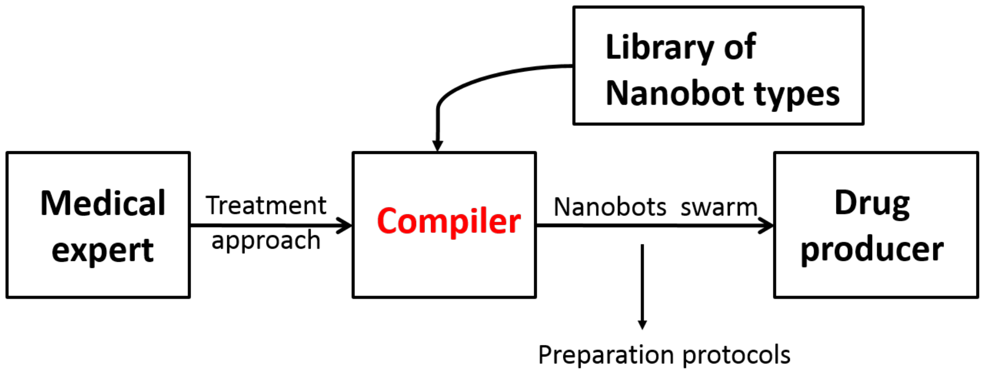


Figure 1.3 Drug development's process as for Tolkien approach

1.3 This thesis

In this thesis, I introduce two important components of the project, with focus on the medicine description language and the compilation algorithms. I developed a rule-based language for describing medicines. The medical expertise will be writing using that language, allowing the separation of the treatment idea from its implementation.

In addition I developed a compiler, which receives as an input a medical treatment written in the medicines description language, and outputs a nanobots swarm recipe that implements it.

Since we are dealing with people's lives, I prove soundness and completeness of the compilation process.

I also tested the compiler's results and showed that it produces valid output. Some of the swarms were also tested in vitro, with actual nanobots, and demonstrated correctness.

Some of the research in this thesis was presented in the following publication:

- **DNA 21** DNA Computing Conference August 2015. I. Wiesel, G. A. Kaminka, N. Agmon, G. Hachmon, and I. Bachelet. First Steps Towards Automated Implementation of Molecular Robot Tasks.

- **FNANO 2016** FOUNDATIONS OF NANOSCIENCE Conference, April 2016. I. Wiesel , G. A. Kaminka, N. Agmon, G. Hachmon, and I. Bachelet, A Compiler for Pro- gramming Molecular Robots.
- **IJCAI 2016** International Joint Conference on Artificial Intelligence, July 2016. I. Wiesel-Kapah, G. A. Kaminka, N. Agmon, G. Hachmon, and I. Bachelet, Rule-Based Programming of Molecular Robot Swarms for Biomedical Applications.

Chapter 2

Background and Motivation

In the past four decades remarkable progress has been made in the area of drug development. Improved spatial control of therapeutics is currently achieved mainly through conjugation of the therapeutic molecule to target-specific carriers such as antibodies or interleukins [14]. Temporal control is mostly achieved by embedding the drug in a matrix that hinders and prolongs its dissolution and diffusion, thereby causing the sustained, more stable distribution of the drug [22]. Various elaborations of this approach have been proposed, including matrices comprised of multiple phases, which release the drug stepwise [29]. Other matrices can be triggered externally by physical means such as infrared laser or ultrasound [28]. However, even these modifications do not always improve drug performance [12]. While definitely representing improvements over conventional, unmodified drug administration, many desired features are not enabled by these techniques. Examples for such features include reversing the availability of a molecule at will; coordinating the activity of two or more molecules by turning one molecule off as the other is turned on and vice versa, such that they do not collide or compete with each other; or activating a molecule by a complex combination of biological conditions, for example a target cell expressing molecules A, B and C but not D and not E.

Lately, nanobots have arisen as an approach to to carrying out such molecular-level treatments.

Nanobots are nanometer-scale devices that can operate inside of a living body [1, 6, 9], and have the potential to revolutionize medicine [13], in a variety of ways. Specifically for drug delivery, heterogeneous nanobot swarms can deliver chemicals directly to molecular targets, with little or no secondary damages due to side effects [27, 32]. However, all nanobot and their interactions are currently manually planned.

Recent advances have begun to explore generic nanobot arch-types, which can be ”programmed” (specialized) to specific conditions and treatments. Recently developed nano-particles [31] serve as an example. These nanobots are built from a nanometer-scale gold bead, to which various DNA strands can be attached, e.g., to bind with specific biomarkers. These particle nanobots cannot shield their payload—it is always exposed. However, as the DNA strands hybridize with the target bio-markers, the exposure will take place at the target.

The DNA-based *clamshell* nanobot is another example [10]. It is an autonomous, logic-guided DNA-based nanobot that [8, 11, 16, 33, 35], that resembles a hexagonal clamshell, open at both ends (Figure 2.1). On one side, a gate consisting of two dsDNA (double-stranded DNA) arms controls the nanobot state. When the arms are in dsDNA configuration, the two halves of the clamshell are held locked. However, when these duplexes unzip, the nanobot can entropically open, exposing its internal side.

One can program the nanobot by specializing its structure in specific ways: choosing the appropriate components such that the clamshell opens when it encounters the pre-defined signature of molecules and biological conditions. The internal side can be programmed by loading it with a variety of payloads, including small molecules, drugs, and proteins. The number, stoichiometry, position and order of payloads can be carefully planned. Similarly, one can choose the components of the gate, and assemble them on the nanobot chassis, in turn setting the nanobot to activate when it encounters the pre-defined signature of molecules and biological conditions recognized as correct input by the gate. Upon activation, the nanobot exposes its payload, which was previously shielded from the environment, enabling it to interface with the point of destination, e.g. tumor cell. The nanobot can also revert back to its inactive state in the absence of the required inputs, making its payload concealed again.

This programming is a manual process, requiring expert knowledge of the nanobot structure and capabilities. Given this, a suitable nanobot and its components can be found when a payload transforming action is needed.

However, when more complex actions are required, a heterogeneous swarm is needed in order to execute them [24]. For example, the clamshell robots essentially respond as a two-input AND gate. Thus as long as the targets are identifiable by one or two markers, a single nanobot type can be built to respond to these markers by opening. However, if the target is identifiable by three or more markers in combination, lets assume A, B and C, no single clamshell nanobot type can correctly open only in the target location. Instead, a heterogeneous combination of clamshells is needed, such that one responding to markers A and B by releasing a compound T, and one responding to T and C by opening. That way, together they make the second open in the target location only (A, B and C).

We consider nanobots to be state machines, whose transitions are governed by interactions with each other and with the environment, somewhat akin to population protocols [2, 4, 7].

To date, all such single and swarm nanobot programming (specializations) were manually planned. But the emergence of nanobot arch-type as described opens the door for automated generation of specialization procedures, based on parameterizable template preparation protocols. Motivated by the (sometimes forgotten) success of rule-based systems at capturing expert knowledge [15, 17, 18, 21] we propose a rule-based approach to nanobot programming. The rule based language, *Athelas*, allows specification of biomedical applications, without considering individual swarm members or the swarm composition. It differs from other swarm programming languages, such as *Buzz* [26], and Proto [5] which focus on spatial swarm behaviors, computation, and syn-

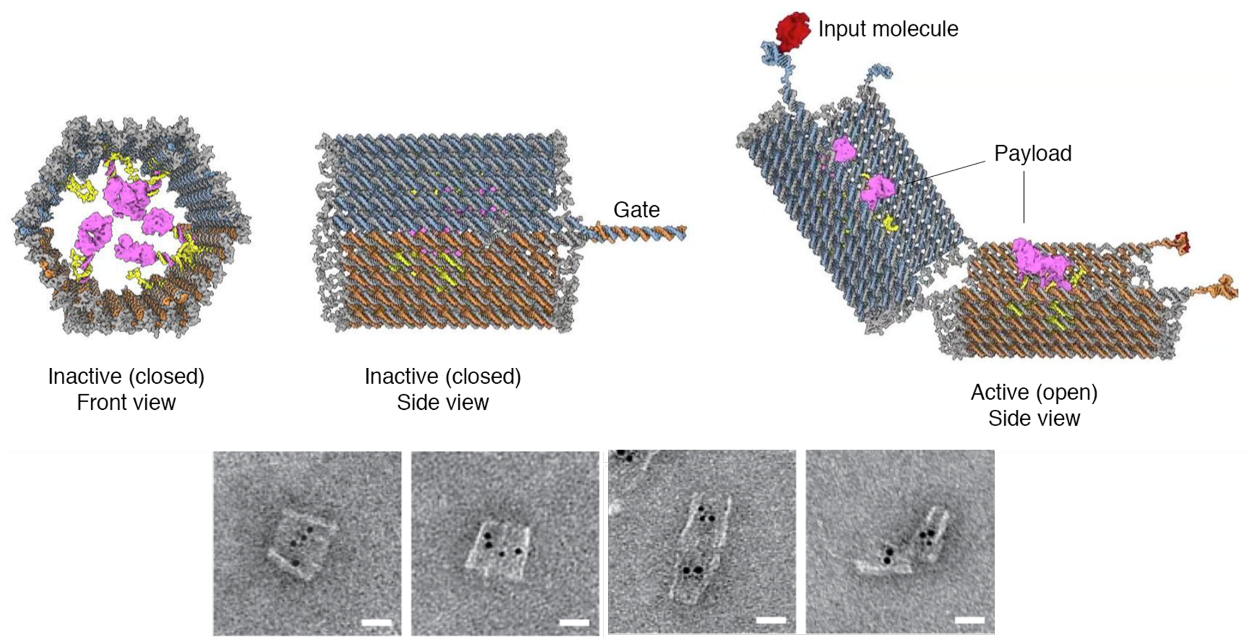


Figure 2.1 A clamshell DNA nanobot [10]. Up: a schematic view of the two states: closed (left) and open (right). Down: TEM images (scale bar, 25nm).

chronized knowledge.

Chapter 3

The Athelas Programming Language

Athelas is a programmable medicine description language, and pert of a project named *Tolkien* that described in Section 3.1. The language itself described in Section 3.2.

3.1 The Tolkien Project: Overview

The Tolkien project aims to develop a development environment for nanobot-based biomedical applications. It is inspired by modern software development environments. When we develop a program, we think in an abstract level rather than the precise instructions set of the CPU. In such environments we describe a program in a high level programming language. The compiler translates the program into machine instructions. To do so, it has to know the architecture of the CPU, and therefore another input that it receives, in addition to the program, is a library of architecture specifications. This separation between implementation levels, allows each of the disciplines to focus on its expertise. In particular, the programmer does not have to know about the instruction set and electronics (Figure 3.1). An illustration of the process is shown in Figure 3.1.

Inspired by this design, we developed a compiler (*Bilbo*) that receives two inputs. The first, is a treatment program, described in a high-level rule-based programming language called *Athelas*.

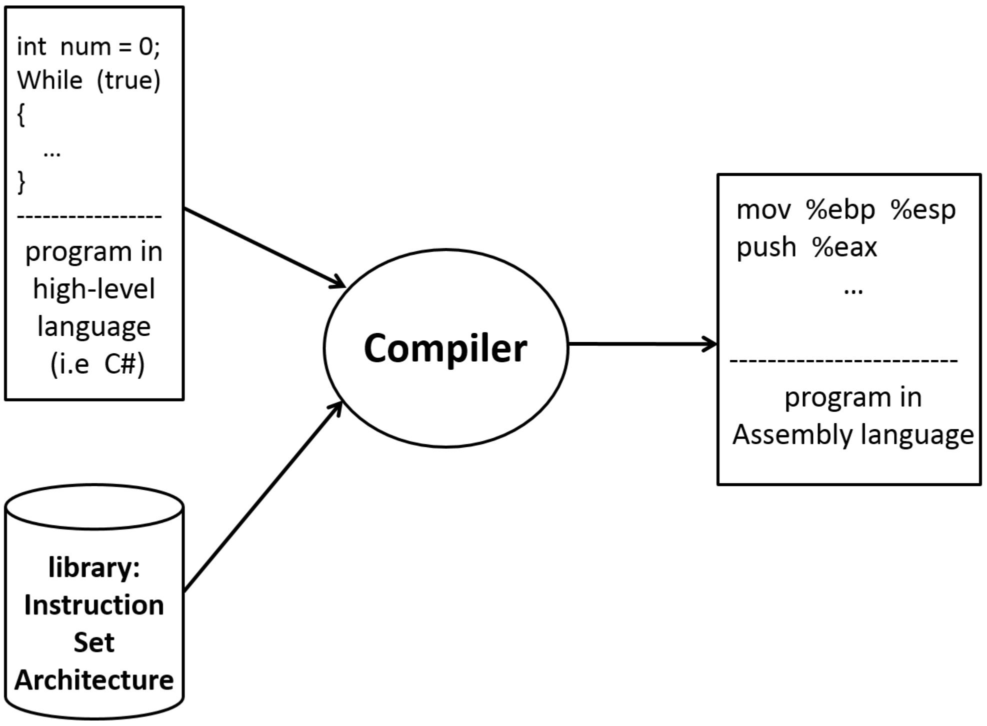

int num = 0;
While (true)
{
...
}
----
program in high-level language (i.e C#)
Compiler
library: Instruction Set Architecture
mov %ebp %esp
push %eax
...
----
program in Assembly language

Figure 3.1 Compilation process illustration.

The second is a library of generic nanobot types, that can be specialized to specific payloads or triggering mechanisms. The library describes all the nanobots that are available for the compiler to use, and all the ways in which they can be specialized. Bilbo can mix and match some or all of these nanobots in its output such that the resulting mix of nanobots—the swarm—carries out the treatment program. An illustration of the process is shown in Figure 3.2.

The benefit of this library is quite significant. Whenever a nanobot developer develops a new type of nanobot (or changes existing capabilities), she only updates the nanobot library. The next compilation can uses the new library, and may produce a better implementation. Such separation can be a key for facilitating cooperation between medical and nanobot experts. It allows each of them to focus on her mission: the medical expert to focus on investing and specifying the treatment approach, and the nanobots developer on developing generic nanobots. The compiler is the bridge between them. Below, we discuss the components of this process, focusing on Athelas in the

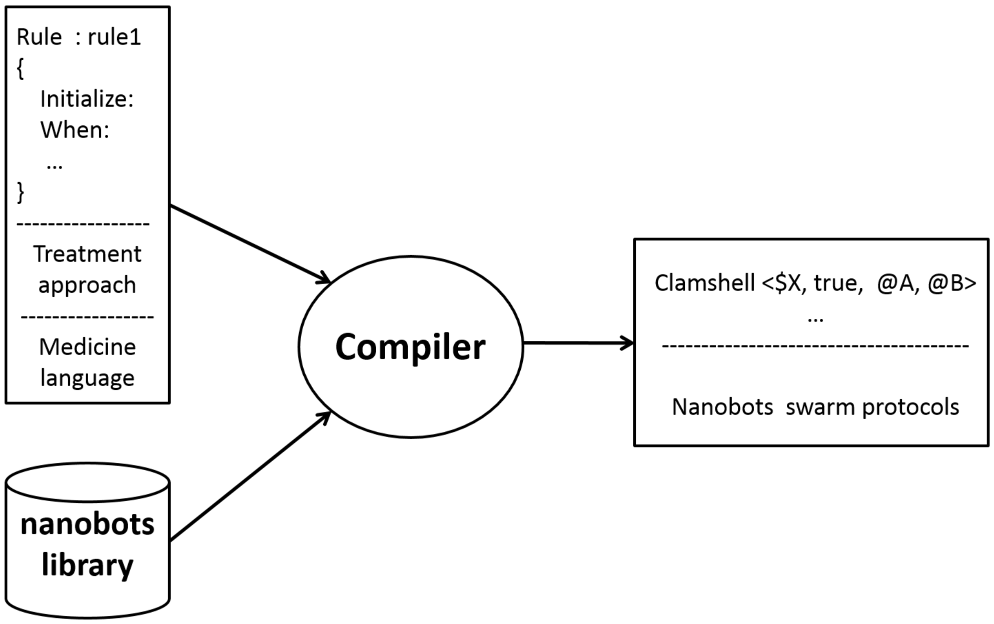


Figure 3.2 Nanobots compilation process illustration.

remainder of this chapter, and on Bilbo in the next chapter.

3.2 The Athelas Language

In order to define Athelas, first we need to understand what are the basic operations that the robots needs to be able to perform for executing drug's tasks. Medications work by moving compounds between locations in the body. So generally, a drug's task can be represented as **carrying** compounds from a given place, and—**under specific conditions**—exposing (and sometimes releasing) them at a desired place. We can refer to those tasks as Pick , which means collecting substance from desirable place, Drop-releasing substance at desirable place, Protect-concealing substance at desirable place, and Expose-exposing substance at desirable place. We consider these tasks to be programmed by a user, and carried out by nanobots. In addition, an ideal drug acts only at the proper time. A proper time can be determined according to location constrains, concentrations, existence of specific chemical and more. So the language should take care of this also.

3.2.1 Athelas Language Definition

We adopt a *rule-based programming* paradigm, in which programs are described by rules that are triggered simultaneously when their conditions are met. At any given time the system is in a specific state, which can be accessed through the contents of the "working memory". There is a set of several rules, which is called the "rule memory". Each rule defines that given some condition are true, a specific action is taken. At any given time, all the rules' conditions are checked against the working memory, and every rule whose conditions are true, is *fired*, i.e., its actions are taken.

This paradigm fits perfectly here, because nanobots act simultaneously and have a direct access to the local environment around them, serving as the "working memory". The rules describe the treatment approach, by testing the body's state, and the actions operate inside it, changing its state accordingly. An example to a rule will be one that describes that when the concentration of some chemical is less then some fixed threshold, an action of releasing this chemical will start to take place. It will continue until the concentration will pass other threshold, then the action will stop.

Indeed, one could think about the clamshell nanobots previously described as simple IF-THEN rules: when the IF (LHS) conditions hold, the robot opens, exposing its payload to interact with the environment or other robots.

```
Medicine: medicine_name
{
        Rule_1
        Rule_2
        ...
        Rule_n
}
```

An Athelas program is described as a set of rules (see full BNF specification in Appendix

A). Each rule has four components (Figure 3.3). The `<initialize-clause>` specifies the set of payloads to be built into the drug when it is injected (i.e., before any action is taken). For example, a robot carrying insulin for a diabetes patient would be assumed to have an initial insulin payload, differently from a robot which begins empty, and is tasked to locate some specified matter and pick it. The `<when-clause>` is the set of conditions that must be true in order for the rule to take action (e.g., insulin concentration is too low). The `<action-clauses>` is the set of actions to be executed when the conditions are met (e.g., expose insulin payload, allowing it to be released), and the `<until-clause>` specifies termination conditions for the drug (e.g., appropriate levels of insulin are detected).

The semantics of rule execution are easily explained by the following pseudo-code (Algorithm 1). After initialization, the rule is assumed to loop in lines 2–6 forever.

```
Rule:<rule-name>
{
        Initialize: <initialize-clause>;
        When: <when-clause> ;
        Actions: <action-clauses> ;
        Until: <until-clause> ;
}
```

Figure 3.3 General structure of an Athelas rule.

Each rule has four clauses, discussed below (Figure 3.4 shows an example). As a matter of notation, Athelas code uses `$` as a prefix to denote payloads, and `@` for a prefix in order to denote location expressions. Such expressions refer to biological markers (e.g., CD Antigens).

- **Initialize**: The Initialize clause specifies the set of payloads to be built into the drug when it is injected (i.e., before any action is taken). For example, a nanobot carrying insulin for

Algorithm 1 Pseudo-code of rule execution semantics.

```
1: Initialize with <initialize-clause>
2: if <when-clause> then
3:     do
4:         execute <action-clauses>
5:     while not <until-clause>
6: goto 2
```

a diabetes patient would be assumed to have an initial insulin payload, differently from a nanobot which begins empty, and is tasked to locate some specified matter and pick it. In this case, the commend will be: `Initialize: $insulin`

- **When** and **Until**: These clauses are each composed of a set of tests, e.g., pH level, existence or concentration of a specific chemical in specific location in the following format:

  `When: <cond>($<chemical> @<location>) <relation> <constant>`

  - `cond` can be pH level, concentration or existence.
  - `chemical` can be any chemical we want to test.
  - `location` can be defined in terms of logical combinations of biomarkers, given by molecular specifications (e.g., CD Antigens).

    The logic combinations are important, since almost each place in the body that can be recognized through molecules combinations needs more than one molecule in the combination. That case requires a logic AND. In addition, there might be a place that could be specified through a couple of combinations, and that case requires a logic OR. Consequently, the syntax enables the mix of Boolean operators such as NOT, AND, OR etc. For instance, `@(cd1 AND cd2) OR NOT cd3` is a valid location expression, using predefined markers `cd1, cd2, cd3`. The expression specifies locations that are marked by both `cd1` and `cd2`, or do not have marker `cd3`. `@*` is used to denote the wildcard location (everywhere).

- `relation` is a numerical comparison operator (e.g., <).

- `constant` specifies the concentration in standard molecular units.

For example, a test for concentration of `C` everywhere uses the following format:

```
When: conc($C @*) > threshold
```

The When and Until clauses give us the ability to control the activation times of the treatment according to the patient's state. Let's assume for example a diabetes' treatment. A good medical care will be one that as soon as the blood-sugar levels gets high, release insulin in the blood. With When and Until clauses we can easily do so by the following rule, when Glc indicates glucose:

```
Rule: Diabetes_Care
{
        Initialize: $insulin;
        When: conc ($Glc @*) > thresh ;
        Actions: drop ($insulin @*) ;
        Until: conc ($Glc @*) < thresh - 5 ;
}
```

The rule creates group of nanobots loaded with insulin and ready to spread it everywhere in the blood. They will not get active until the blood-glucose level gets dangerously high, and will stop right after the level get safer, in this case 5 ppm under a predefined threshold.

- **Actions**: The Actions clause contains the actions to be executed when the drug is active. We allow the following actions:
  - `pick ($payload @location)`: Pick the specified payload from the specific location, allowing it to be carried around. These instructions cause the nanobots to be built with appropriate compounds to bind `$payloads` (if encountered near `@location`) so that is carried by the nanobot.
  - `drop ($payload @location)`: Remove the specified payload at the specific location, allowing it to float freely.
  - `protect($payload @location)`: Protect payload from the environment (at the specific location).
  - `expose ($payload @location)`: Expose payload to the environment at the specific location (this does not, however, release the payload from the robots).

Another action such is Disable, which operates on specific rules, to disable them from being active. This type of programming language reflection is needed to address inter-drug interactions, such as to set priorities of drugs over each other. The action's syntax is `disable (<other-rule-name>)`, when <other-rule-name>refers the rule that should be disabled. The action should be written in rules as before, only that this time their meaning is not operating treatments, but keeping the safety of the patient from the treatment itself. The syntax is:

```
Rule: <rule-name>
{
        When: <start-condition>;
        Actions: disable (<other-rule-name>);
  }
```

As soon as the start condition occurs, the other rule is disabled. I discuss this subject in more details in Chapter 4. A detailed example shown in the experimental results section.

An Example Rule

An example of a rule is shown in Figure 3.4. The meaning of this rule is as follows. Payloads `Z` and `Y` are to be built into the robots. If the concentration of matter `X` in location `T` is bigger than 5 *mol*/*m*^{3}, then the robots will continuously release `Z` at location `@T` and `$Y` at the location `A AND B AND C`, i.e., a location marked by the presence of biomarkers `cd1, cd2, cd3`. This will continue until the concentration of `X` in location `T` drops below 2.

```
Rule:ToxicDrugClean
{
  Initialize: Z, X;
  When: conc ($Y @T) > 5;
  Actions:
        drop   ($Z @T)
        expose ($X @(A AND B AND C));
  Until: conc ($Y @T) < 2;
}
```

Figure 3.4 An example rule with all components.

We are not aware of any nanobot design capable of implementing this rule in a singe nanobot.

For instance, the clamshell nanobot previously discussed is capable of dropping a payload in a location marked by at most two markers (e.g., `cd1 AND cd2`). And the clamshell cannot selectively drop only `Z` or only `Y` in a location. Thus in order to have nanobots execute this rule, a mixture of different nanobots (a heterogeneous nanobot swarm) is needed. The role of the compiler is to synthesize this swarm, choosing between multiple options if possible to optimize cost, yield, and reliability.

We note that we are not aware of any nanobot design that is capable of implementing this rule in a singe nanobot. For instance, the clamshell nanobot previously discussed is capable of dropping a payload in a location marked by at most two markers (e.g., `cd1 AND cd2`). And the clamshell cannot selectively drop only `Z` or only `Y` in a location. Thus in order to have nanobots execute this rule, a mixture of different nanobots (a nanobot cocktail) is needed. The role of the compiler is to synthesize this cocktail, choosing between multiple options if possible to optimize cost, yield, and reliability.

In order to increase the usability of the system, ideally, the location is not specified in terms of molecules combinations, but using a popular medical name, for example Pancreas. This name is replaced with the appropriate molecules combinations using a macro mechanism (essentially, the C pre-processor `#define`) using a given molecules combinations table. For example, the medical expert will specify `#Pancreas` and the preprocessor will replace it with `cd4 AND cd5`.

Chapter 4

The Bilbo Compiler

The Bilbo compiler takes two inputs: an Athelas program, and a library of generic robot types. The library should consists of robots descriptions, each describing the Athelas actions the robot can execute, the dependencies it holds, costs in terms of money and safety, duration of activity and its preparation protocol, which describes in details how to manufacture it. Those protocols are parameterized, which means one can create specific robot from their robot type by selecting different parameters to the protocol. As for today, we do not have all of those parameters, such as half-life and costs, but as a beginning their possible actions and preparation protocol is good enough for us.

Bilbo synthesizes a specification for a heterogeneous swarm of specialized nanobots, which would carry out the program, once deployed. The output specification for each robot includes a specialized preparation protocol, according to the robot type and its role in the swarm.

The compilation process is done in two phases. A front-end phase consists of the lexical, syntax and semantic analyzers, and the intermediate code generator. Its output is a set of Finite State Machines (FSMs), used as intermediate representation for the given Athelas code. The back-end phase then transforms such FSMs into a final nanobots swarm specification, given as templates which merge with their protocols into the final solution. We discuss both in detail below, focusing on their operation when compiling the `initialize` and the `actions` clauses. An illustration of the entire process is shown in Figure 4.1.

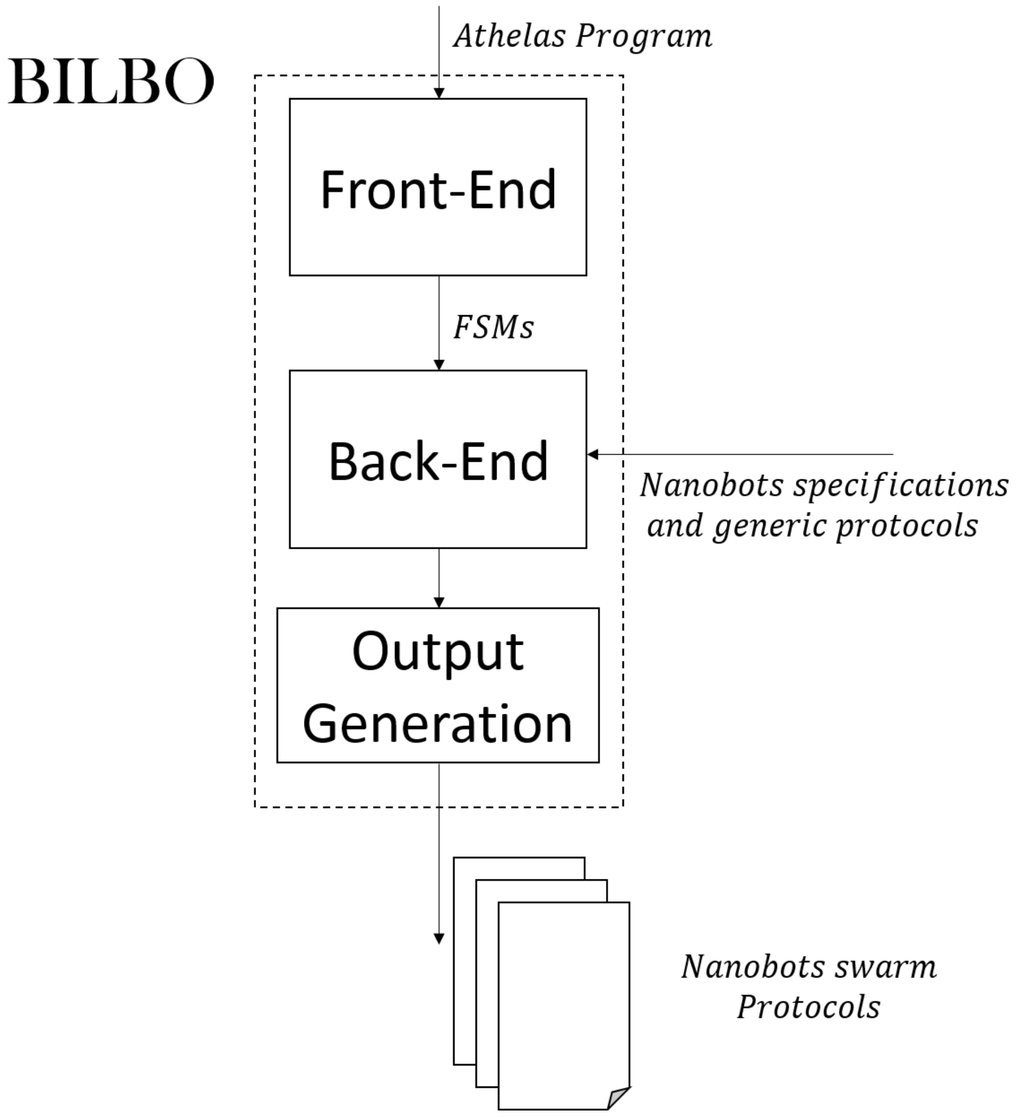


Figure 4.1 Illustration of Bilbo from high

One assumption we must take in order to solve our problem, is the *main-robot assumption*. We assume that for a given thread of actions, if all actions are executed on the same payload, then only one robot is in charge of those actions, i.e., should execute them all by itself. For example, there cannot be a solution in which the main payload is delivered between two robots, only one robot should carry it. We denote the in-charge robot *the main robot*. Although the main robot should be the only robot dealing with the payload itself, it can assist other robots for completing the mission. For example, for recognize the desired location its payload should be released. Those robots are referred to as *the assistant robots*. More details about the assumption is in the discussions section.

The Front-End, is responsible for translating the source code into an intermediate representation. The representation is in a form of finite state machines when each described a part of the medical process. Then, the Back-End, when its responsibility is to translate the intermediate representation into alternative swarm recipes. The alternatives represented by an AND/OR tree, where each path from its root to its leaf represents one alternative recipe. Finally, an optimization step select the optimal solution. In the current version of the compiler that step is disabled, due to a lack of nanobots costs parameters, but in the future we are hoping to use AO* algorithm on the tree for finding the optimal solution, when costs will be denoted by weights on the tree's edges.

4.1 Basic Compilation Process

Firstly I describe the basic process, which take care of the basic actions of the rules, without handling *disable* action and *When* and *Until* clauses, that require a special treatment.

4.1.1 The Front-End

The general idea is to transform every rule into a group of threads of instructions representing the rule. A thread of instructions is an ordered sequences of actions that depend on each other in some way. For example, the Drop $X action should follow either a Pick $X action, or an `Initialize: $X` clause.

The front-end transforms each thread into a Finite State Machines (FSM). Each state consists of three parts: the payload status (*compound* if held, *empty* otherwise); the payload exposure (*exposed* or *protected*); and the location. For example, ($\$X$, *exposed*, $@A$) indicates the process contains payload $\$X$ that is exposed to the environment and in location $@A$. Actions specified in the rule mark transitions between states.

An FSM is a mathematical model of computation that can be in one of a finite number of states. The current state can be changed from one state to another when a triggering event or condition occurs; this is called a transition. In our model, each FSM is a tuple (Σ, S, s_0, δ, F) where:

- $\boldsymbol{\Sigma}$ is the set of the permitted actions (e.g., Drop, Pick) and Initialize.
- $\boldsymbol{S}$ is a finite set of states representing our medical process state. A state consists of three parts: the first indicates what payload the state holds, if at all. The second indicates whether the payload is exposed or protected, and the third indicates the location of the state. For example, ($\$X$, *Exposed*, $@A$) indicates the process contains payload $X that exposed to the environment and in location @A.
- $\boldsymbol{s_0}$ is an initial state, an element of S.
- $\boldsymbol{\delta}$ is the state-transition function: $\delta : S \times \Sigma \to S$.
- $\boldsymbol{F}$ is the set of final states, a subset of S.

The FSMs are built by going through the rules in the given code and adding states and transitions according to their parts. When reaching key word **Rule**, the front-end adds a new rule to the rules list. When processing an `Initialize` clause, the front-end creates a new FSM for each material $M in the clause. The new FSM starts with the states $(null, ?, ?)$, and $(\$M, ?, ?)$ with a transition *Initialize* $\$M$ connecting them. This represents the medical process starting without payload (*null*) and gaining one by an initialization action (which the compiler will add to the nanobot generation protocol) that cause the robot to be loaded with the payload in its creation. The other parts of the state are labeled with ?, since they are unknown at this stage, thus can be anything. An illustration of this step is shown in Figure 4.2.

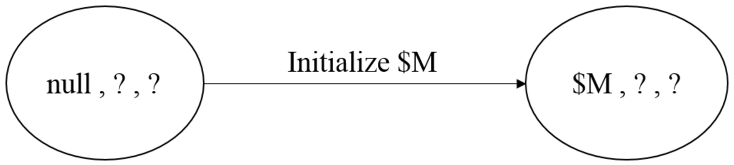


Figure 4.2 Illustration of Initialize case in the front-end

For an action `pick($M @B)` the front-end adds a state $(null, ?, @B)$, and a transition from it to another new state $(\$M, ?, @B)$. This represents a transition from a state in location $@B$ without a payload, to a state in the same location with the payload $\$M$.

See Figure 4.3.

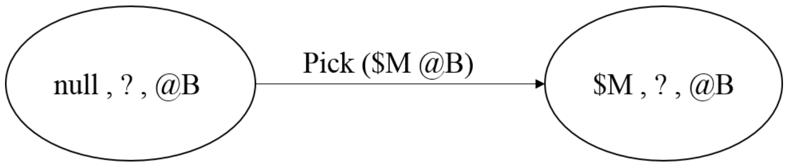


Figure 4.3 Illustration of Pick case in the front-end

For an action `drop($M @B)`, the front-end searches for an FSM that ends with the state $(\$M, *, @B)$, where * stands for any exposure state. If one is found, the front-end connects it to a new state, $(null, ?, @B)$ with the transition $drop(\$M\ @B)$. Otherwise, it searches for a terminal state $(\$M, *, @*)$, i.e., any state with payload $M, but not in location @B. It then connects it via an ε-transition to a new state $(\$M, ?, @B)$ (and then continues as in the first case). The ε-transition represents traveling via the bloodstream until arrival at the drop location. Figure 4.4 shows the two cases.

The compilation process for actions Expose or are handled similarly. However, here the ending state is $(\$M, exposed, @B)$ and $(\$M, protected, @B)$, respectively.

An example of the output from the front end, is shown in Figure 4.5. We'll note that the ε-transition in the second FSM is because of the relocation of the process through the bloodstream

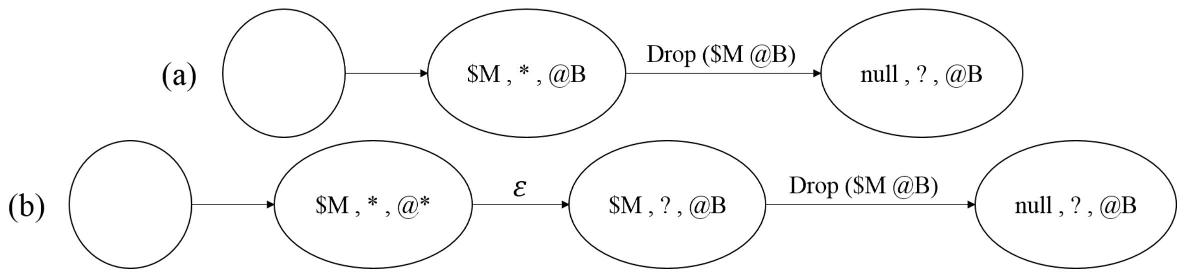


Figure 4.4 An Illustration of the two cases of Drop action compilation in the front-end.

until arriving to the desired location.

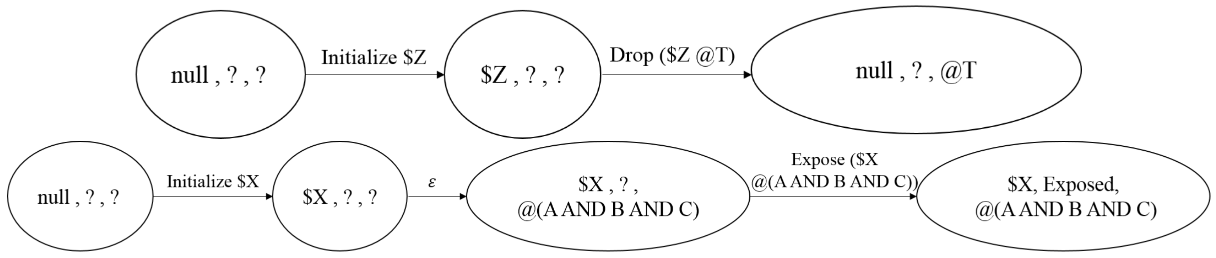


Figure 4.5 The front-end's output, the back-end's input. Two FSMs generated for the rule in Figure 3.4.

4.1.2 The Back-End

The input to the Back-End phase includes a set of FSMs, which is the output of the Front-End phase (see example in Figure 3.4) and a library of generic nanobot arch-types, which the compiler uses in the swarms.

Algorithm BackEnd (Algorithm 2) transforms each FSM to an AND/OR graph (see below) which represents alternative swarm specifications (nanobot mixes). It then runs a specialized dependency-check procedure to filter possible solutions in which the robots conflict with each other, and uses AO* [23] to determine an optimal AND/OR path in the graph. The path is corresponds to a specific heterogeneous swarm, made of specialized nanobot arch-types and their preparation protocols.

An AND/OR tree contains two types of nodes: OR states which represent alternative ways of solving the problem, and AND states which represent subproblems of the problem, that all need be solved. In terms of our problem, the meaning is that each AND state will produce one path including all of its descendants, while each OR node will produce one path for each of its descendants. In our illustrations an arc between couple of regular nodes will represent an AND node between them, otherwise the nodes will be considered as connected with OR nodes. The back-end's data structure is supposed to describe different alternative cocktails (OR), when each of them consist of possibly many nanobots (all of which are needed for the cocktail, i.e., and AND). For that reason, a variant of AND/OR tree is suitable, when every AND/OR path from the start node to end node is one distinct swarm.

Construct-Tree The heart of the back-end phase is Algorithm 3 (ConstructTree). It responsible for the transformation of the FSMs into the AND/OR graphs. It uses a *graph rewriting* approach to carry out the transformation, working with four graph-rewriting operators: SUBSTITUTE, MERGE, REJECT and DECOMPOSE. SUBSTITUTE's mission is to exchange a transition to a group of abstract robots that aims to be able of executing it. Since there are transitions that one single robot cannot carry out by itself, DECOMPOSE breaks down an abstract robot, that cannot be produced in order to operate its tasks alone. It decompose it to a terminal robot – that is manufacturable – and to another action, representing by a transition, that should assist the robot completing its mission. In order that all the FSM's actions will be executed by the same robot, as mentioned above, MERGE merges compatible implementations of different transitions, such that each path from start to end in the tree consists of one main robot. Then REJECT complete MERGE's mission by removing all the paths that cannot be merged.

We present the algorithms and then elaborate each of them.

We illustrate the operation of Algorithm 3 by transforming the EXPOSE FSM in Figure 4.5. For this demonstration, we assume the two nanobot arch-types previously described (Chapter 2):

Algorithm 2 BackEnd (input: $FSMs\_list$, $Nanobot-library$)

```
1: $ruleRecipe \leftarrow \emptyset$
2: for each $FSM$ in $FSMs\_list$ do
3:     $AOGraph \leftarrow$ ConstructTree($FSM$, $Nanobot-library$)
4:     Do
5:         $OPT \leftarrow$ optimization on $AOGraph$
6:     While DependencyCheck($OPT$) is $true$
7:     add the robots recipe of $OPT$ to $ruleRecipe$
8: return $ruleRecipe$
```

Algorithm 3 ConstructTree (input: FSM, Nanobot-library)

```
1:  // We denote a non-$\varepsilon$-transition $t$ by $\alpha t$
2:  $AOGraph \leftarrow FSM$
3:  while $\exists \alpha t$ or non-terminal robot-state do
4:      for all transitions $\alpha t$ connecting states $A$, $B$ do
5:          SUBSTITUTE($t$, $A$, $B$, $AOGraph$)
6:      for all compatible, $\varepsilon$-connected states $A, B$ do
7:          MERGE ($A$, $B$, $AOGraph$)
8:      for all incompatible, $\varepsilon$-connected states $A, B$ do
9:          REJECT($A$, $B$, $AOGraph$)
10:     for all non-terminal robot-state $t$ do
11:         DECOMPOSE($t$, $AOGraph$)
12: return $AOGraph$
```

the clamshell [10], and the nano-particle [31].

The SUBSTITUTE operator replaces all non-ε transitions with subgraphs of abstract robot-states, a robot-state for each generic nanobot in the nanobot library with the ability of executing the transition action (and ignoring location and payload on the transition action). In case the transition cannot be replaced, it is removed.

The transition Expose($X @(A AND B AND C)) in Figure 4.5 is substituted in Figure 4.6 by a clamshell and by a nano-particle, which has the ability to perform an expose action, without the clamshell's inability of recognizing more than two locations. In addition, the transition Initialize $X is substituted also by clamshell and nano-particle since they both can be initialized with $X. The robot-states are connected to the incoming vertex of the original transition with an OR and ε-transitions.

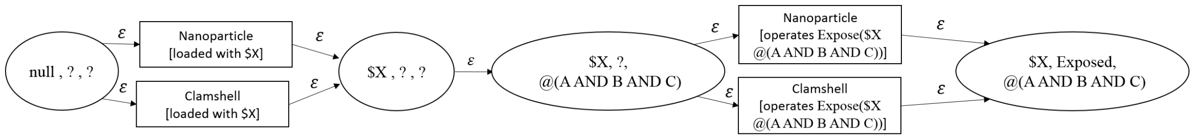

null , ? , ?
ε
Nanoparticle [loaded with $X]
Clamshell [loaded with $X]
$X , ? , ?
$X, ?, @(A AND B AND C)
Nanoparticle [operates Expose($X @(A AND B AND C))]
Clamshell [operates Expose($X @(A AND B AND C))]
$X, Exposed, @(A AND B AND C)

Figure 4.6 Illustration of SUBSTITUTE operation on input shown in Figure 4.5.

Next, Algorithm ConstructTree MERGES each pair of *compatible* robot-states connected by a path of ε-transitions. A pair of robot-states is *compatible* if their parameters are either identical or complementary. For example, let us assume the clamshell parameters are `<$, Init, @L, @R>`, when *$* represents which payload it will carry, *Init* indicates if is it initialized with that payload, and @*L*, @*R* represents the locations its left and right arms will recognize.

Then a robot-state represents a clamshell with the parameters `<$X, true, @A, ?>` (where '?' stands for an undetermined parameter) is compatible with one who represents a clamshell with `<$X, true, ?, @B>`, since there is no conflict between both parameters so they can be replaced by one robot having all their parameters together without conflicts.

The MERGE operation removes the ε-transitions between a given pair of robot-states and merges the states to a one robot-state representing both of them. The new state is connected to the first robot-state's incoming vertex and to the second robot-state's outgoing vertex with ε-transitions. Applying MERGE to Figure 4.6 results in Figure 4.7.

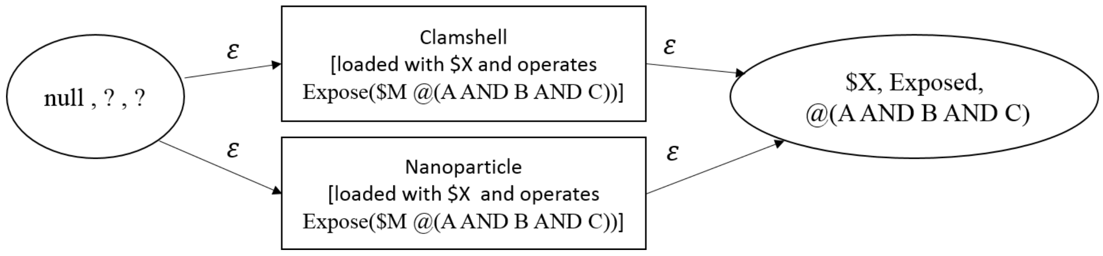

null , ? , ?
ε
Clamshell
[loaded with $X and operates
Expose($M @(A AND B AND C))]
ε
Nanoparticle
[loaded with $X and operates
Expose($M @(A AND B AND C))]
ε
ε
$X, Exposed,
@(A AND B AND C)

Figure 4.7 Illustration of MERGE operation.

The REJECT operator removes all pairs of *incompatible* robot-states, connected by ε-transitions, i.e., states whose parameters are not identical, and conflict (i.e., cannot be merged). For example, this would apply to a robot-state representing a clamshell and a robot-state representing a gold nano-particle. The REJECT operation removes the pair of states from the tree, and thereby rejects their path from the back-end's output.

Algorithm 3 uses DECOMPOSE on all non-terminal robot-states. A robot-state is terminal if and only if all of its parameters can be produced together in exactly one robot of its kind. In other words, if it describes a fully specified, producible robot. An example is a robot-state describing a clamshell with five locations parameter as opposed to one who describes a clamshell with only two locations. Since a clamshell can recognize only two locations, the second state is *terminal* while the first is not. The DECOMPOSE operation replaces a given non-terminal robot-state with a terminal one and an action transition, representing two actions that together complete the original robot-state action. In case the non-terminal robot-state cannot be decomposed, it is removed.

In Figure 4.8, the robot-state representing Expose($X @(A AND B AND C)) by a clamshell is replaced by a transition of Expose($T @(A and B)) and a terminal robot-state representing the original expose action, but this time in locations T and C: Expose($X @(T AND C)). Logically, together they complete the original Expose action. Note that these two states are connected to the incoming (original) state by two ε-transitions marked by an AND, i.e., both have to be taken if this option is selected.

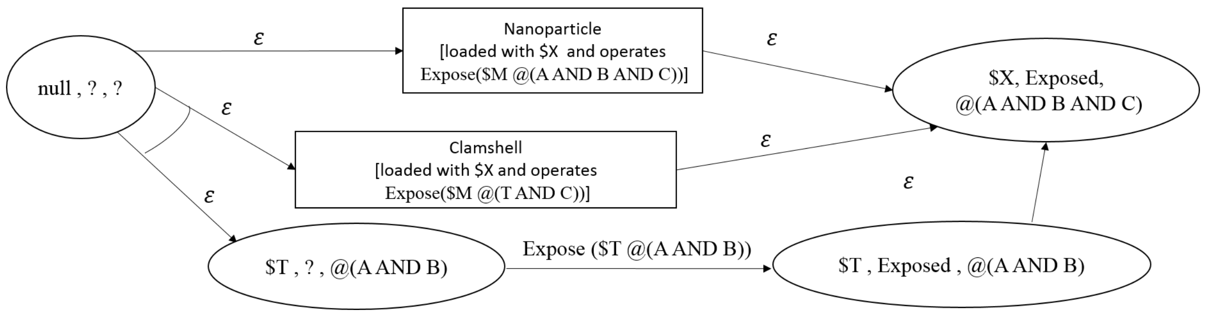


Figure 4.8 DECOMPOSE operation on the results of previous steps. The arc marks AND transitions in the AND/OR graph.

Finally, ConstructTree iterates back to apply the graph rewriting operators on the rewritten graph. The process repeats until no non-ε-transitions and non-terminal robot states are left. Then the final graph is returned.

Then, in the future, when there will be enough cost criteria for the robots, an optimization stage will choose an optimal path from the initial state to the end, where costs will be denoted by weight on the graph's edges (or based on number of steps, otherwise). Since today we do not have such criteria, we return all the options.

DependencyCheck is responsible for making sure that there are not exist any dependencies between the robots in the final recipe. Dependencies can be, for example, signals collision, i.e., when two signals from the same type are used for different purposes and misleading the system. Or dependencies between the robots complexes themselves. Indeed, after selecting a specific recipe, DependencyCheck checks that there are not such dependencies in it. If there are, it activate the selection once again for choosing a new different recipe, and then check again. The process continue until a non-dependencies recipe is found.

The final recipe is consist of all the protocols belong to the robots in the selected path, such that each protocol describes in details the way of producing its robot according to the robot's specification in its state from the graph.

4.2 Extended Compilation Process

We now describe the extended process for compiling the Disable action (Section 4.2.1), and the When/ Until clauses (Section 4.2.2).

4.2.1 Disable Action

In many cases, during the medical treatment a conflict occurs, in which completing the treatment may harm the patient. In such cases, one solution is on the one hand to execute the treatment, and on the other, stop it as soon as it might causes damage.

To do this, Athelas supports the Disable action. Disable operates on specific rule in the treatment by stopping its execution. It can be defined with a start condition, noting the hazardous situation in which the rule must to be stopped. This allows the medical expert decide when exactly the action should happen, in terms of concentration of a substance, existences, etc.

The action's syntax is `disable (<other-rule-name>)`, when <other-rule-name>refers the rule that should be disabled. The action should be written in rules as before, only that this time their meaning is not operating treatments, but keeping the safety of the patient from the treatment itself. The syntax shown in Chapter 3.

To compile the Disable command, changes must be done in the compilation process and in the robots description. We explain the main idea and then elaborate on the changes.

The basic idea is to implement each disable action by *reverse actions*, preventing the disabled rule's functioning. There are two ways of implementation.

The first is disabling the disabled rules' robots, who implement it. This can be done by suitable robots with the ability to block the aforementioned robots' actions, e.g, a robot that can lock a clamshell that in an expose state (Section 6.3 in experimental results section).

The second is canceling effect of the robots implementing the rule. This is done using complementary actions, which means ones that when executed parallel to the rule, converge the system, i.e, the patient, to its state before the rule's execution. An example is a pick action for a rule with a drop action on the same chemical: together, the drop seems like it never happened.

Given disable action on specific rule, the compilation algorithm go through the rule's robotic implementation and creates a disabling implementation for it, consist of robots or actions disabling those robots. In order to so, it needs to know for each robot and action, what their disabling actions/ robots. So another parameter for the robot's description is its reverse actions, for all of its possible actions. For example, for a clamshell with expose action, its reverse robot is another clamshell with the ability of locking the gates of the first, causing its expose's disabling.

In the front-end, for each disable action a new type of FSM is added to the FSMs list. The new FSMs consist of state with two parts: the first is the disabled rule's name and the second is its activation status. The disable action is the transition between a state with the activation status "active" and a state with the activation status "inactive". It represents a transition from a state where the disabled rule is active to a state where the disabled rule is inactive, because of the disable action. An illustration of such an FSM is shown in Figure 4.9.

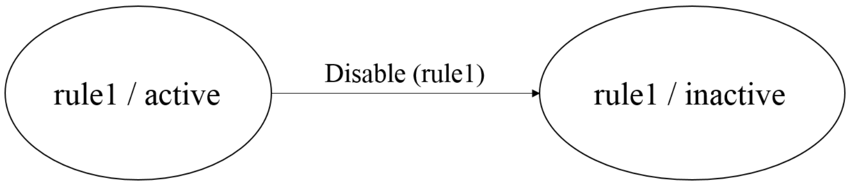


Figure 4.9 Disable's FSM illustration

In the back-end, a new operator is added after the regular operators loop, named DISABLE_CARE. It operates in a loop after the construct of the regular FSMs recipes, and its responsibility is to translate each disable FSM to suitable FSMs which represent the reverse actions by complementary actions or by blocking robots as explained before. For each disable's FSM, DISABLE_CARE goes over the disabled rule's implementation that in ruleRecipe and exchanges the disable FSM in set of FSMs consist of reverse actions or blocking robots that belong to the actions of the implementation's robots. The recognition of those reverse actions is done by a key of robot type and a specific action. Afterwards ConstructTree works on those FSM as before - SUBSTITUTE, MERGE, DECOMPOSE, and so on. An Example of process is shown using the rules shown in Figure 4.10.

```
Rule: rule1
{
        Initialize: X;
        Actions:  drop ($X @A)
                  expose ($X @B);
}

Rule: Disabling_rule1
{
        When: conc ($X @* > 50);
        Actions: disable (rule1);
}
```

Figure 4.10 An example use of Disable

Here, there are two rules displayed, when the second is responsible for disabling the first's actions. As described before, the Front-End creates the FSM showed in Figure 4.9. Next DISABLE_CARE creates the FSMs that are shown in Figure 4.11. We can see that an FSM with a Pick transition was created as a complementary action for the Drop that in the preview rule, as well as an FSM with robot-state used as a blocking robot for stopping the action of the robot that was selected to implement Expose in the rule. In addition, all those actions are being taken only given the desired concentration.

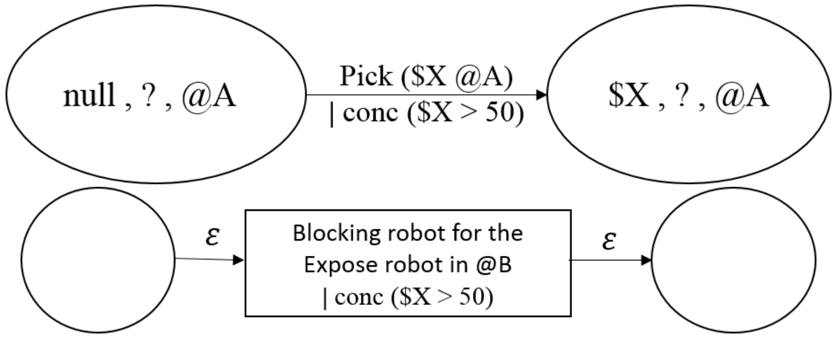


Figure 4.11 Disable's Back-End illustration

In order to implement the Disable action, the back-end needs to find for every robot in the implementation of the disabled rule its reverse action, for the specific action it implements in the rule. So in that case, implementation with the robots only is not enough - the implemented actions have to be recorded too. So another update is in SUBSTITUTE, in which a transition is replaced by an abstract robot and the action itself, as a pair. Accordingly, MERGE saves both the robots' actions, in the merged robot. In that way all the desired details exist for the disable, and for the final recipe only the robots are taken.

One problem that might occur, is that if the rules without the disable (noted as "regular rules") are implemented before the disable rules, the robots that are being selected for the implementation might not have reversed actions. In such case, the disable rules cannot be implemented, while there might be another implementation, in which the robots' actions can be disabled, that wasn't selected in the optimization process.

In order to solve that problem, the back-end works back and forth: first, it collects constraints list from the disable FSMs, which contains the names of all the rules need to be disabled. Second, it works as usual on the regular FSMs, only that for the rules that are in the constraints list, only the implementations that can be disabled are selected to the AND/OR tree. That is done by the improved SUBSTITUTE_WITH_CONSTRAINS Finally, it returns to the disable FSMs and implements them as explained above, with the help of DISABLE_CARE. In case there is a rule that cannot be disabled by any implementation, i.e, its tree is empty, the back-end outputs a matching error message (e.g., "the following rule cannot be disabled"). The revised algorithms are shown in Algorithm 4 and Algorithm 5.

Algorithm 4 BackEnd version 2 (input: $disable\_FSMs\_list$, $FSMs\_list$, $Nanobot-library$)

```
1:  $ruleRecipe \leftarrow \emptyset$
2:  for each $disable\_FSM$ in $disable\_FSMs\_list$ do
3:      $constrainsList$.add($disable\_FSMs\_list$.disabledRule)
4:  for each $FSM$ in $FSMs\_list$ do
5:      $AOGraph \leftarrow$ ConstructTree($constrainsList, FSM, Nanobot-library$)
6:      Do
7:          $OPT \leftarrow$ optimization on $AOGraph$
8:      While DependencyCheck($OPT$) is $true$
9:      add the robots recipe of $OPT$ to $ruleRecipe$
10: for each $disable\_FSM$ in $disable\_FSMs\_list$ do
11:     DISABLE_CARE($disable\_FSM, ruleRecipe$)
12:     $AOGraph \leftarrow$ ConstructTree($FSM, Nanobot-library$)
13:     Do
14:         $OPT \leftarrow$ optimization on $AOGraph$
15:     While DependencyCheck($OPT$) is $true$
16:     add the robots recipe of $OPT$ to $ruleRecipe$
17: return $ruleRecipe$
```

Algorithm 5 ConstructTree version 2 (input: *constrainsList*, *FSM*, Nanobot-library)

```
1:  // We denote a non-$\varepsilon$-transition $t$ by $\alpha t$
2:  $AOGraph \leftarrow FSM$
3:  while $\exists \alpha t$ or non-terminal robot-state do
4:      for all transitions $\alpha t$ connecting states $A$, $B$ do
5:          if $FSM$ in $constrainsList$ then
6:              SUBSTITUTE_WITH_CONSTRAINS($t$, $A$, $B$, $AOGraph$)
7:      for all compatible, $\varepsilon$-connected states $A, B$ do
8:          MERGE ($A$, $B$, $AOGraph$)
9:      for all incompatible, $\varepsilon$-connected states $A, B$ do
10:         REJECT($A$, $B$, $AOGraph$)
11:     for all non-terminal robot-state $t$ do
12:         DECOMPOSE($t$, $AOGraph$)
13: return $AOGraph$
```

The above is also the reason for the choice to make the disable-rules a part of the compilation input, among the rest of the rules (and not in the linking stage). The cause is that for implementing the disable an intervention must happened in the compilation process, where the selection between the different implementations happens, and the robots that are right for the following disable can be chosen. For that reason, *disable* needs to be declared inside the rules. In the future, the compilation process may have an option for automatic generation of disable rules for a given medication, described in Athelas.

4.2.2 When and Until clauses

The Athelas language allows the programmer to specify start and stop conditions for rule activation. This is done by *When* and *Until* clauses, where *When* represents a condition that signals the action clause should be active and *Until* specifies condition signals the actions to stop. We will describe Bilbo's approach implementing those parts.

First, *When* clause. Its possible conditions are existence, concentration and pH level. In general, we can say that for their implementation Bilbo uses robots with an ability of recognizing the conditions and releasing a signal according to it. The signal should be seen by the robots implementing the actions clause and be used as a green light for their activation, so they will start their actions only after receiving the signal. An example is shown in the following rule:

```
Rule: WhenExample
{
        When: conc ($X @A) > 3;
        Actions: pick ($X @A);
}
```

For that rule, Bilbo creates concentration robots that signal when the concentration of $X in loca-

[htbp]

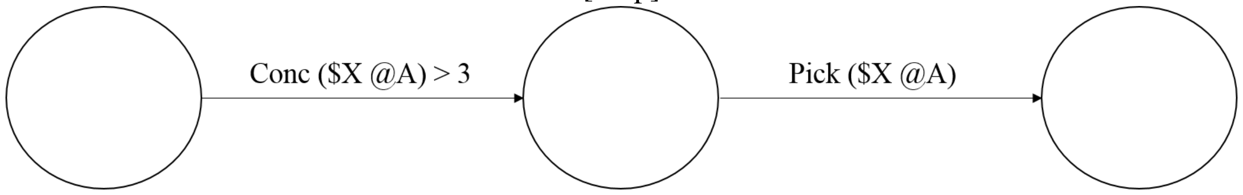


Figure 4.12 *When*'s FSM illustration

tion @A is more than 3. In addition, it outputs actions robots for performing the *pick* action, that starts working only when receiving the signal. That way, the *When* condition is the cause for the rule firing.

More specifically, for each *When* clause the Front-End creates new FSMs that are the heads of all the FSMs it creates. Their transition is

```
<cond> ($<chemical> @<location>) <relation> <constant>
```

according to the *When* condition. An illustration of the FSM is shown in Figure 4.12, where we can see that the condition is added to the *pick* FSM.

After that, in the Back-End stage a new operator is added: CONDITION_CARE. It is located before the regular operators and operates only once. Its mission is creating a robotic implementation for the condition, for example in terms of concentration robots. Then it adds the robots to the building swarm and replaces the transition with one that forcing the FSM's actions to start only when receiving the robots' signal. An illustration of the Back-End process is shown in Figure 4.13. Here, another constraint to the robot implementing *pick*, is starting its action only when receiving $\$signal$.

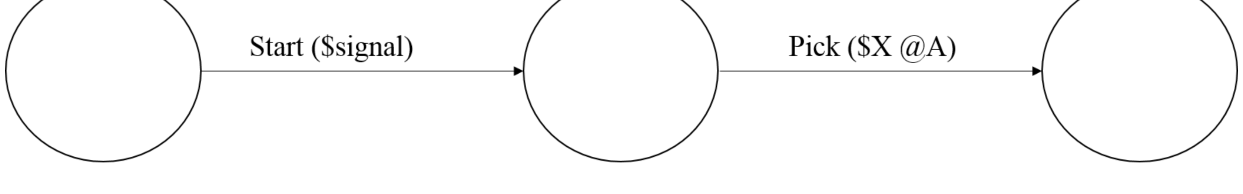


Figure 4.13 CONDITION_CARE illustration

The possible conditions of the *Until* clause are identical to those of *When*. An example for its use is shown below:

```
Rule: UntilExample
{
        Actions: pick ($X @A);
        Until: conc ($X @A) < 5;
}
```

The rule describes a *pick* action at occurs until the concentration of $X in location @A is less that 5. Bilbo implements it by two steps: the first is creating a *signal robot* acting according to *Until*'s condition (in terms of concentration, etc). The second is creating the *reverse actions* which responsible for the actions canceling, as described in *disable* subsection, when the aforementioned signal is the trigger for their operating.

More specifically, the Front-End creates an FSM which identical to the one that created for the *When* clause, and connect a *disable* FSM to its tail. That way, canceling the actions is done only after the *Until* condition is verified. After that, the Back-End works with CONDITION_CARE as in *When* clause and the process continue as usual.

An illustration of the process is shown in Figure 4.14.

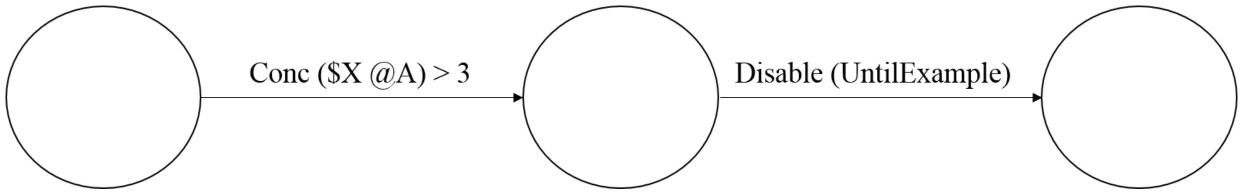


Figure 4.14 Illustration of the Front-End's output on *Until*

4.3 From AND/OR Tree To Swarm Specification

After the back-end generates implementation path of the robots (from the tree), it collect the robots' specifications to the final recipe. The details represented as tuples, where each tuple represents one robot with its parameterized features. For example, the basic syntax of the Clamshell's tuple is: Clamshell <$, Init, @L, @R>, when *$* represents which cargo it will carry, *Init* indicates if is it initialized with that cargo, and @*L*, @*R* represents the locations its left and right arms will recognize. So if the path contains a clamshell that should expose chemical $T at location described by bio-markers @(A AND B), then its tuple wil looks as the following:

```
Clamshell: <$T, true, @A, @B>
```

Finally, each tuple replaced by a preparation protocol, which describes in details how to actually manufacture it. Since those protocols belong to each nanobot type, only the parameterized features in it are set up, selected from the tuples. For example, the protocol from the last exampled tuple is shown in Appendix B.

Chapter 5

Proofs

In this section we prove that the back-end's algorithm is complete, sound and halting. As explained in Chapter 4, we assume that for a given thread of actions, if all actions are executed on the same material, then only one robot is in charge of those actions. For example, there cannot be a solution in which the main material is delivered between two robots. We denote the in-charge robot the main robot. The main robot is assisted by other robots for completing the mission. Those robots are referred to as the assistant robots.

Lemma 1. *The back-end's algorithm is complete, i.e., if a solution to the input exists, the algorithm will return it (and if more than one solution exists, the algorithm will return at least one solution).*

Proof. Assume that there exists a solution to the problem, yet the algorithm failed to return it (the algorithm returns *no solution* when the AND/OR tree is empty or *cannot be disabled* when there is a disabled that cannot be disabled). Following the algorithm's steps, this could happen in one of the following cases:
(i) In the SUBSTITUTE step not all transitions were substituted to abstract robots. However, if a solution exists, there must be a *main* robot of some type X that implements all the FSM's transitions. Therefore SUBSTITUTE must offer it as an option to all the transitions.

(ii) REJECT removed all the possible paths from the tree because no robotic option could have been merged. However, since robot type X was offered to all the transitions, MERGE merges all those robots, and specifically REJECT does not reject them.
(iii) DECOMPOSE could not decompose the solution's abstract robot and canceled its path. However, if in the main robot X should be assisted by other robots, then DECOMPOSE must offer their help, thus does not cancel the path.
(iv) DependencyCheck removed it because of incompatibility of its robots. By the assumption that there exists a solution to the problem, necessarily it does not have dependencies problems, thus DependencyCheck cannot remove it.
(v) In DISABLE_CARE step not all transitions were substituted to blocking robots or to complementary actions. However, if a solution exists, there must be reverse actions in a form of blocking robots or complementary actions. Therefor DISABLE_CARE must offer them as solutions to the transitions.
Thus, given that the input is correct, the algorithm will not remove a valid solution along its way. □

Lemma 2. *An assistant robot does not have an ε-transitions path to any other robot.*

Proof. Let r be an assistant robot. Since r was created in order to implement a transition generated by DECOMPOSE, there is no other transition next to it, because DECOMPOSE creates only one transition each time. Since DECOMPOSE acts after MERGE and SUBSTITUTE, all the transitions in the level before (which belong to the initial FSM) were already implemented and merged to its main robot, so the main robot is connected to the final state and specifically not to another robot, and so does r. □

Lemma 3. *The back-end algorithm is sound, i.e., the output is a correct implementation of the given FSMs.*

Proof. Given a nanobot cocktail recipe R that was returned as an output by the back-end algorithm for a given FSM f, then R can be incorrect (i.e. it is not a valid implementation of f) due to one of the following reasons: a) it does not contain all the needed robots. b) it contains unnecessary robots, and specifically harmful ones. c) the interaction between the different robots is problematic due to dependencies they share.

Case a: Assume that there is a transition in f such that R is lacking a robot for its implementation, and let r be the missing robot and t the unimplemented transition. If r is missing in t's implementation, then it can be either a main or an assistant robot. Assume, towards contradiction, that the recipe is missing the main robot. Since it is a main robot, SUBSTITUTE necessarily offered it for t. Hence, if it is not in the recipe then REJECT or DECOMPOSE removed it. That means that at some point, r did not match any transition t', thus REJECT removed it from the tree. But that contradicts the assumption that r is a main robot, which means it can implement *all* the transitions, and specifically t'. From this point on in the algorithm, r cannot be removed (DECOMPOSE only adds assistant robots, it does not remove existing ones). Following all the above, clearly the missing robot r cannot be a main robot.

In case r is an assistant robot missing from the final recipe, then (by being an assistant robot) it was created by DECOMPOSE for some transition t', and removed by REJECT at some point. Since r is supposed to assist the main robot, there is some transition $\tilde{t}$ which represents the action r needs to execute. Therefore, DECOMPOSE created $\tilde{t}$ and SUBSTITUTE translated it to the matching robots. Since r matches SUBSTITUTE offered it.

Due to lemma 2, r did not have an ε-transitions path to any other robot. Therefore it was not required to be merged with any other robot, and hence REJECT didn't removed it and it is in the recipe. This contradicts the assumption that the recipe doesn't contain all the needed robots.

Last case is when r is a disable-robot, means it should disable some action, by stopping it directly or through a executing complementary action. In this case DISABLE_CARE was must find it or the action in the disabled action description. Then, it necessarily created a matching FSM with suitable transition or robot-state. After that point, the responsible goes to the other operators so the proof is as above.

b) Assume the output of the algorithm (recipe R) contains a redundant robot r that does not implement any transition in f. Since r irrelevant there is no transition in f it is required to implement, neither as a main nor as an assistant robot. A robot can be added to the tree by DISABLE_CARE or by SUBSTITUTE, which operates on original transitions from f or on those created by DECOMPOSE or by DISABLE_CARE. As mentioned, r does not implement any transition of f, so SUBSTITUTE or DISABLE_CARE who adds robots according to transitions only could not have generate it, and also DISABLE_CARE cold not have add a transition for it. In addition, since it does not help to implement any transition, there is no DECOMPOSE that could have generated r. Therefore r is not in the tree, thus cannot be in the recipe.

c) The interaction between the different robots is problematic due to dependencies they share. However, DependencyCheck goes over all possible combinations of robots that may depend on one another (directly or indirectly), thus this case is also impossible. □

Lemma 4. *The number of the robots in the final recipe is finite.*

Proof. Since each FSM implemented by one main robot and number of assistant robots, and due to the fact that the last number is limited by the number of the locations they help to recognize, which is finite, then the number of all the robots in the recipe is finite. □

Theorem 5. *The back-end algorithm halts, and is complete and sound.*

Proof. Completeness and soundness of the algorithm are achieved by Lemmas 1 and 3. In addition, the number of the FSM's in the FSMs list which the back-end works on is finite, and due to the fact that in each recursive call at least one robot created and the number of the robots in the final recipe is finite (Lemma 4), the recursion's depth of construct-tree algorithm for each FSM is limited by the number of the robots in the longest recipe. Therefore, the algorithm will necessarily halt. □

Chapter 6

Experimental Results

Bilbo's basic compilation process has been implemented. To evaluate its use, we conducted several compilation experiments, and followed these with in-vitro experiments, to confirm the compilation results. In these experiments, Bilbo compiled Athelas programs, none of which could be implemented using a single robot type. In all, the compiler generated nanobot swarm recipes, sometimes proposing several options. We implemented these by carefully mixing robots according to the compiler-generated recipes, and show their effectiveness in in-vitro experiments.

For the sake of comfortable we omit the preparations protocols in the outputs demonstrated in this section, and present only templates that integrate with them. Each robot has a template for its own that consist of the configurable parameters in the protocols. For example, the Clamshell protocol needs to be set with the location its gates should recognize, the payload it should carry and so on. The syntax of the Clamshell's template is: Clamshell <$, Init, @L, @R>, when *$* represents which cargo it will carry, *Init* indicates if is it initialized with that cargo, and *@L*, *@R* represents the locations its left and right arms will recognize. In the following experiments we will see this representation.

6.1 AND decomposition

In our first experiment we want a dummy molecular payload denoted by \$., and normally protected from the environment, to be exposed, when in the vicinity of beads marked by three different DNA strands, denoted A, B, and C (as in the `expose()` instruction used in the rule in Figure 3.4). We limited the compiler to using only clamshell nanobots. Each of these has two gates only and as a consequence can recognize only two markers. Thus a single clamshell nanobot cannot be specialized to correctly recognize the target location.

```
Rule:ExposeDrugAtLoc
{
  Actions: expose ($. @(A AND B AND C));
}
```

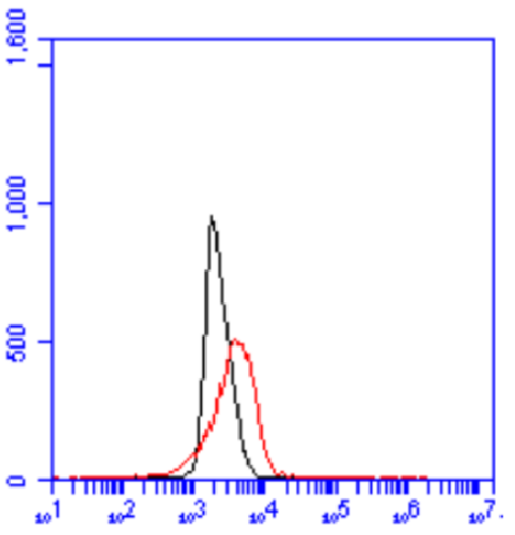
Figure 6.1 A rule requiring decomposing an AND.

The Bilbo compiler's output is given in Figure 6.2 (we omit here the preparation protocols which those templates are combine with). It solved the problem by splitting the strands detection into two steps, each to be executed by one specialized type of clamshell such that together they complete the task. The first responds to A and B by releasing an intermediate compound T. The second responds to T and C by exposing \$.. This cascade causes \$. to be exposed only in the presence of A and B and C, as specified.

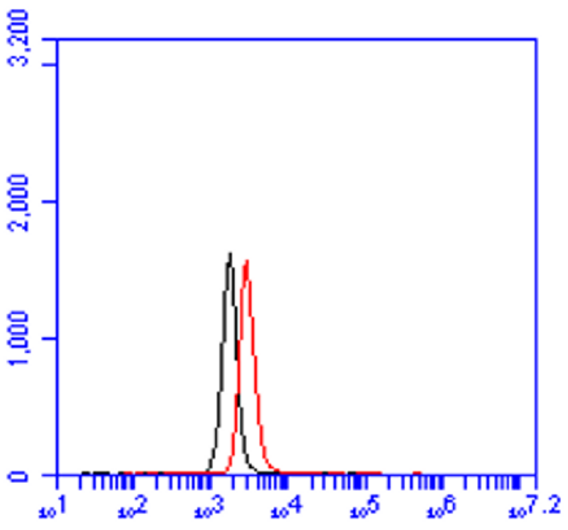
```
1: Clamshell: <$T, true, @A, @B>
2: Clamshell: <$., true, @T, @C>
```

Figure 6.2 Bilbo compiler output for AND decomposition experiment.

We conducted in-vitro experiments to evaluate the compiler output. In a first experiment, we mixed a first type of nanobot, to demonstrate that it can detect strands A and B. We then added a second type of nanobot, detecting C. To measure the results, we use fluorescent materials to mark the activity of the robots. Figure 6.3(a) presents the flow cytometry results from this experiment. The histogram plots the fluoresceine-isothiocyanate intensity (horizontal axis, *log scale*) against the number of events detected (vertical axis). The figure shows lower or no responses when only the first nanobot is interacting with the beads (black line, left peak). But when the second nanobot is added, we see high response (red line, right peak), evidence of the two robots interacting together to cause exposure of the dummy payload.



(a) Exp. 1. Beads and one nanobot type (left peak) vs. beads and the two nanobot types (right peak)



(b) Exp. 2. Beads only (left peak) vs. beads and nanobot (right peak)

Figure 6.3 Flow cytometry histograms of both experiments.

6.2 AND/OR, Multiple Options

In a second experiment we demonstrate the compiler's capability to generate different implementation alternatives for the same program. To do this, we extended the nanobot library to also include the gold nano-particles [31] previously described[1](see Chapter 2). We forced the compiler to generate all alternatives, by disabling the path selection stage. Thus all implementation alternatives are produced.

The benefits with number of recipes are numerous, since they differ in perspective of cost-in terms of the nanobots manufacturing cost; in perspective of chances of success with completing the mission; in perspective of treatment time, etc. So with the varied recipes, the optimal solution can be chosen for each criteria.

We compiled the Athelas program shown in Figure 6.4. The task is to expose the dummy molecular payload $. when in the vicinity of beads marked by DNA strands A, B, and C, or by D and E. The more complex target specification gives rise to different implementation alternatives.

```
Rule:ExposeDrugAtComplicatedLoc
{
  Actions: expose ($. @(   (A AND B AND C)
                        OR (D AND E));
}
```

Figure 6.4 Rule used in the second experiment.

The compiler offers three different implementations. The first uses a nano-particle nanobot (marked `au` in Figure 6.5). The second, uses a combination of clamshell and nano-particles (Figure 6.6), and the third combines three types of clamshell nanobots (Figure 6.7).

[1]Thanks to Rachela Popovtzer and Eran Barnoy for their help in this.

As the first implementation, the compiler used a single type of particle nanobot, with two types of strands of DNA are attached to the gold core, either of which (or both) may bind to targets (thus forming an OR). One DNA strand binds to A, B, and C (concatenated). The other to D and E, concatenated (Figure 6.5). Multiple copies of both these types uniformly cover the surface of the core, so they should have equal probability of binding to locations thus marked. However, here the compiler is also displaying its limits: in practice, this solution will work only as long as the different biomarkers (e.g., D and E) are in the same order on the same strand, but not if they are spatially separated, which is the more general case. This is a limitation of the current nanobot modeling language used in the nanobot library, which we hope to address in future work.

```
1: au: <$.,true,@A AND B AND C,@E AND D>
```

Figure 6.5 nanoparticle implementation for rule in Fig 6.4.

As a second alternative implementation, the compiler proposes a swarm composed of a single clamshell and a particle nanobot (Figure 6.6). The particle nanobot is almost the same as above, so the clamshell may seem redundant. However, it is not. The particle nanobot will bind in the same location. But it carries a payload $T, rather than the dummy payload $.. The clamshell responds to the payload $T, by attaching itself to the particle and releasing $. This has the nuanced difference from the first implementation in that the payload $. is shielded from the environment throughout until activation of the nanobots (useful, e.g., when the target payload is toxic). Had a `protect()` instruction been used, this implementation would have been preferred.

Finally, a clamshell-only solution decomposes the OR condition to its constituent parts. The `(A AND B AND C)` part is identical to above, and the compiler issues the same implementation (lines 2–3, Figure 6.5). The `(D AND E)` implementation uses a single clamshell, reacting to pres-

```
1: au: <$T,true,@A AND B AND C, @E AND D>
2: Clamshell: <$., true, @T, @T>
```

Figure 6.6 Gold particle and clamshell nanobot swarm for rule in Fig. 6.4.

ence of both D and E (line 1).

```
1: Clamshell: <$., true, @D, @E>


2: Clamshell: <$T, true, @A, @B>
3: Clamshell: <$., true, @T, @C>
```

Figure 6.7 Clamshell-only implementation of rule in Figure 6.4.

We unfortunately do not have the facilities to conduct in-vitro experiments involving gold particle nanobots. However, we are able to test the final compilation result in-vitro. The `@(A AND B AND C)` results are identical to those previously presented (Figure 6.3(a)). Figure 6.3(b) measures the success of the second component (`D AND E`). We see a significant boost in fluorescence when the `D AND E` clamshell binds itself to the beads.

6.3 Asimov experiment

Our next experiment tests a special treatment, that uses *disable* action as protection for itself. In the treatment, damage to the cells can happen while being executed. But when this occurs, a special chemical named miR-16 is being released around the harmed place and spreads around. The meaning of that is that we can recognize the problem by detecting the existence of miR-16, and as soon as it occurs - stop the treatment to avoid additional damages. So the protection of the rule described as the following:

```
Rule: AsimovExp
{
        Actions: expose ($X @*);
}


Rule: ProtectRule
{
        When: exist ($miR-16 @*);
        Actions: disable (AsimovExp);
}
```

Figure 6.8 Rule and its protection used in the third experiment.

In order to implement the protection, we compiled manually by the extended compilation algorithm two types of robots. The first is a clamshell that exposes $X in the blood, and the second is a clamshell too, only that this time it has the ability of locking the first, causing its exposing action to be stopped. The compilation parametrizes the second robot with the property of acting once recognizing miR-16 (by its gates). So once the first robot causes damage to the cells, miR-16 starts to appear. Then the second robot, that was closed before, opens and reveals its cargo gate. The first robot's gates are synchronized with the second's internal (cargo) gate and so when it is revealed, the first is attached to the second by closing its gates and ending the *expose* action.

To sum up, recognizing miR-16 as the triggering gate causes the second robot to lock the first robot and by that to stop its action. Figure 6.9A demonstrates how the two robots are combined into a filtering process, with miR-16 being the input, and the effector activity of the robots being the output. Note that in the figures, the first robot denoted as L2 and the second as L1.

Figure 6.9B shows an illustration of the process, where in the left the two robots act without miR-16 intervention, L2 by being exposed and L1 by being closed. Once the trigger arrived, L1 opens and closes L2, preventing it from executing its *expose* action.

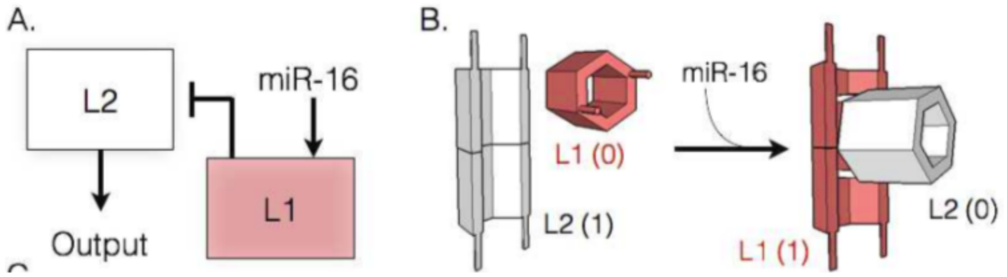


Figure 6.9 From: [20]

Chapter 7

Discussions

In this chapter we go in depth into specific topics that require more thorough discussion. We discuss the assumption from Section 4.1.2 and also explore some safety issues and prodrugs.

7.1 The Role of Medical Knowledge

We start with the assumption that the compiler does not have medical knowledge, while the programmer of the treatment does. According to that, there are several things that the compiler cannot do on its own, without specific instructions. For example using temporary locations as intermediate places to release payloads there by one robot, and after to picking them up by another robot. This cannot be done because the compiler cannot know if the specific payload is harmless and not toxicity for that location in the patient's body, unless have been specified other. That is the reason for the following assumption.

In the compilation algorithms, the front-end creates FSMs. Each represents a thread derived from a rule. The back-end, with the operations SUBSTITUTE, MERGE, REJECT and DECOMPOSE, translates each FSM to an AND/OR tree where each of its paths is a legal solution, or cocktail to the thread.

In that complex process we have made an assumption. The assumption was that for a given thread of actions, that all of them are executed on the same payload, only one robot is in-charge on them and will be dealing with the payload itself - the *main-robot*. The meaning is that once a robot carries the original payload from the rule (not temporaries signals), no other robot will be dealing with that payload. Still, the other robots will be able to assist the main robot with recognizing the desired locations. The only constrain is that the payload should be carried by the main robot itself.

In order to prevent the back-end from creating invalid paths in the AND/OR tree, REJECT operation was added. REJECT implements this assumption by removing the paths that lead to the wrong solutions, by disconnecting the problematic robot states from the tree. That causes the removal of their path from the start node to the end and by that to their removal. REJECT is required when a sequence of actions cannot be MERGEd, because each of the actions was implemented by a different robot type. An example is a thread with the actions `initialize $P` and `drop ($P @*)`, shown in Figure 7.1. Assume SUBSTITUTE translate it to a type-Y-robot that cannot be initialized with \$P and a type-Z-robot that cannot operate drop, and also to a type-X-robots that can do both actions. The illustration in Figure 7.2 demonstrates the transition from the thread's FSM to SUBSTITUTE's result, with those robots.

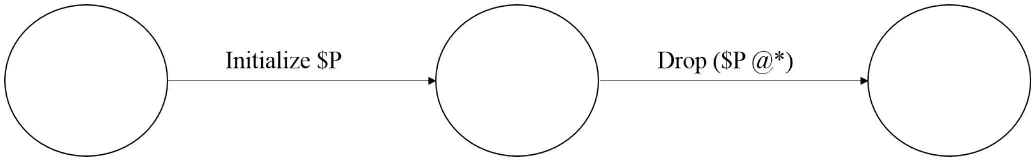


Figure 7.1 Discussions' FSM illustration

In the next step, MERGE tries to merge every couple of robots. An illustration is shown in Figure 7.3. Here, since type-X-robots are *compatible*, they were merged. However, type-Y and Z-robots differ in their types and are *incompatible* and therefor were not merged. Obviously a recipe that contains them both is not applicable, since the result will be one robot initialized with \$P and another robot with the ability to drop it, but without the payload itself (\$P) for doing so.

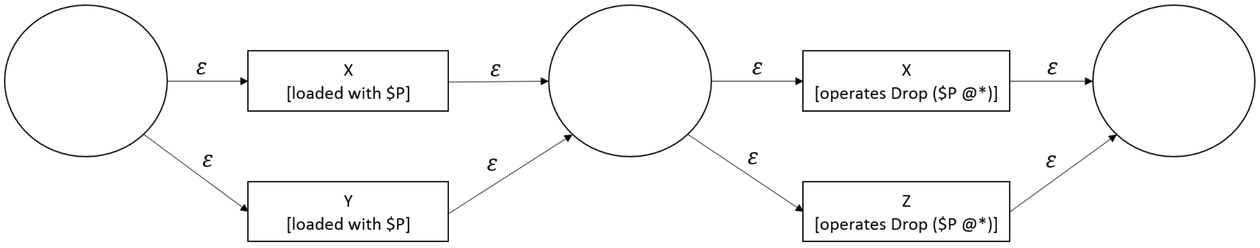


Figure 7.2 Illustration of the tree after SUBSTITUTE

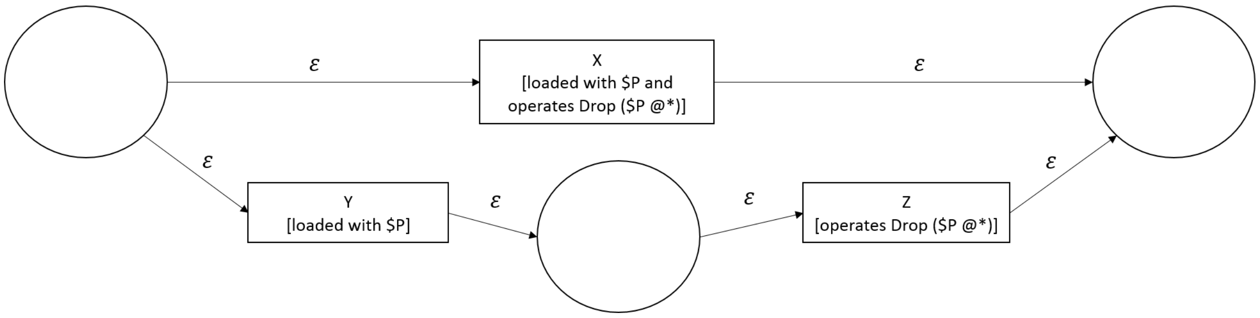


Figure 7.3 Illustration of the tree after MERGE

Since the compiler cannot add intermediate location in which the first robot will drop the payload and the second will pick it up, that solution cannot be outputted.

Without REJECT, which removes these cases and cancels all the solutions in which the robots that operate on the main payload of the thread cannot be merged to one and only robot, at the end of the back-end all of the paths—from the start to the end node—would have taken from the tree, and particularly Y and Z path, which is erroneous. Thanks to REJECT, those paths are removed. An Illustration of the REJECT's effect is shown in Figure 7.4.

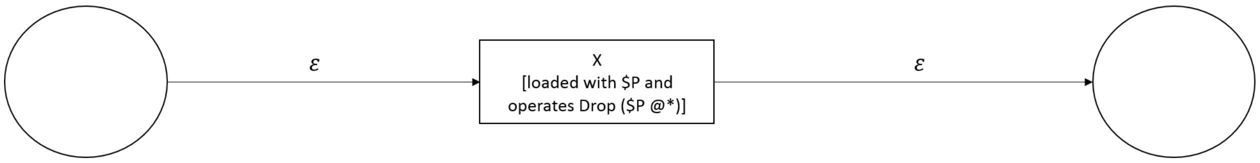


Figure 7.4 Illustration of the tree after REJECT

In conclusion, this assumption solves the problem of creating paths that cannot be asserted as valid.

7.2 Safety

In Section 4.2 we saw the *disable* action, which used in order to protect the patient from damages caused by from the treatment itself. We explained that when the medicine programmer knows about possible problems that can occur from the treatment, she adds *disable* actions for stopping the damaging actions when needed.

We propose to organize such safety consideration by allowing every medicine file to have an associated safety file. The file will be with *.asmv* suffix, for *asimov rules*, as a tribute to Asimov, the famous science fiction writer and the inventor of "the three laws of robotics". Now, every treatment is described by two files. The first is from *.ath* type, describes the treatment approach, as we already know. And the second is from *.asmv* type, in which asimov rules are describes, which protect the patient from the treatment damages by disable the relevant rules and the harmful actions. An example to a use of an asimov file is shown below.

```
example.ath:            // the treatment's logic

Rule: example_athelas
{
        When: conc ($Y @*) > 256;
        Actions: pick ($Y @*)
                protect ($Y @*);
        Until: conc ($Y @*) < 100;
}

example.asmv:           // the treatment's protection

Rule: example_asimov
```

```
{
        When: conc ($X @* > 3);
        Actions: disable (example_athelas);
}
```

Another example is shown bellow, for the experiment tested in vitro (Section 6.3).

```
asimovExperiment.ath:

Rule: AsimovExp
{
        Actions: expose ($x @*);
}

asimovExperiment.asmv:

Rule: ProtectRule
{
        When: exist ($miR-16 @*);
        Actions: disable (AsimovExp);
}
```

The difference between the two sets of rules: In the first the conditions related to the actions themselves without thinking about side effects, in contrast to the second, where they refer to optional side effects of the rules' actions. The meaning is that *.ath* file should not deal with the side effects but in the treatment itself.

This way there can be another separation of expertise: between the medical expert who write the treatment (in *.ath*) to the one who write its protection (in *.asmv*). The advantage is that it allows the first to focus on the treatment she want to make without need to worry for its implications, and leave the "cleaning" to the second, who needs to know nothing about the treated problem, and to only be cared about the implications of the actions described in *.ath* rules.

A disadvantage is in case there are number of optional treatments and accordingly a couple of *.ath* files, when one of them is preferable by its matching asimov file. This can be happen in terms of price in case of less disabling robots, or more robustness of the treatment since it is stopped much more rarely, and more. In that case, thanks to the expertise separation, the medical expert doesn't know about the implications of its decision (since it doesn't have the asimov files) and might choose the expensive treatment. That could have been avoided if she would have write the protection by herself.

I believe the two files separation is preferred, because that way there is an order and clear location to each of the treatments and its protection, and also safety rules are separated and do not burden the treatment's code.

Another reason for is the future option to automate the asimov file creation, by defining an *Safety Library*, which contains all the possible indications for problems, e.g., the appearance of chemical *X* that points on a problem with the pancreas. The library will be another input of the compiler, in addition to the nanobots library and the athelas code, and when given a new treatment (written in *.ath* file) the compiler will automatically create a *.asmv* file for it, with the matching protections.

7.3 Prodrugs and Equivalence of Payloads

Let us assume we have a toxic drug that should be delivered to some location in the patient's body and be exposed there, while being protected anywhere else. If we use a Clamshell for the mission, it should be OK, since it can carry out *protect* action. But on the other hand, if we want to use nanoparticle for the same task, we encounter a problem since it does not have this ability. It seams like using nanoparticles here is impossible, unless it's a prodrug.

A prodrug is a medication or compound that is metabolized into a pharmacologically active drug within the body. An example is a toxic medication denoted by Z, that consists of materials X and Y in a way that as long as X and Y are separated, they are not damaging the body. But once they combine, when near, they turn to toxic Z.

In a case of a prodrug there is a solution for using non-protect nanobots, such as those based on nanoparticles. We explain the main idea and then elaborate.

The idea is to carry the components separately and combine them only at the targeted location. Since when they are apart they are not harmful, so each of them can be carried without need to be protected, and thus to cause the whole drug to be protected wherever it should be and to be exposed only at the target location. That way the mission can be complete by using non-protect nanobots.

For implementing the idea, changes must to be done in the compilation process. First, a new clause is added, called the *Equivalence Map*. The map contains all the equivalences defined by the medical expert. The keys are the prodrugs' names and the values are the equivalent materials. For example: $\{\$Z : \$(X, Y)\}$.

In order to implement the creation of the map, we allow a new syntax in *Athelas*:

`Eq: <key> = <values>;` For example, an addition of the last equivalence to the map is done by the following: `Eq: Z = X AND Y;` When the Front-End parses the above, it directly adds the new definition to *Equivalence Map*.

Next, the Back-End is updated too. I introduce EQUIVALENCE, a new operator with the responsibility of translating the equivalences. EQUIVALENCE places right before SUBSTITUTE (and after CONDITION_CARE) and it is simply adds, for every transition that contains a key from the map, a new transition with the key's values. The new transition is connected with an OR node, in order to denote an alternative path. Let us assume our treatment is described in the following:

```
equivalenceExample.ath:


Eq: Z = X AND Y;


Rule: ProdrugExm
{
        Initialize: Z;
        Actions: protect ($Z @(NOT A))
                expose ($Z @A));
}
```

Here an equivalence of $\$Z$ is defined, and in the rule a protection of the drug is declared anywhere except then in $@A$. The Back-End's input is the FSM shown in Figure 7.5.

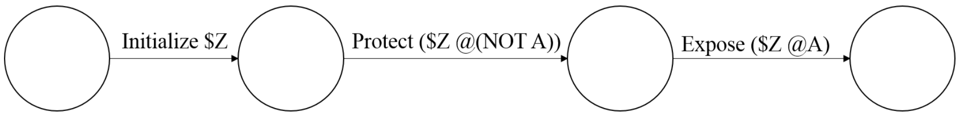


Figure 7.5 Illustration of the Back-End's input

After EQUIVALENCE's operation, the result is the graph in Figure 7.6.

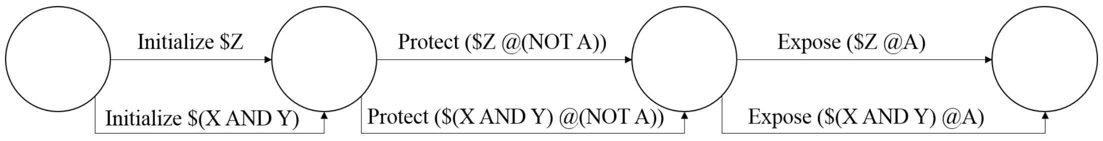


Figure 7.6 Illustration of EQUIVALENCE's output

We can see that for every transition with $\$Z$, a new one with $\$(XANDY)$ was added. Next, SUBSTITUTE. SUBSTITUTE translates each transition that contains a number of materials (e.g.,

$\$(XANDY))$ to transitions with all the possible combinations of robot types implementations. Between every combination there is an OR node, and inside every combination, i,e. between its robots, there is an AND node.

An illustration of the process is shown in Figure 7.7. Notice that the operation is demonstrates only for the transitions with more then one material. The others are substitutes as usual, so we are not focus on them.

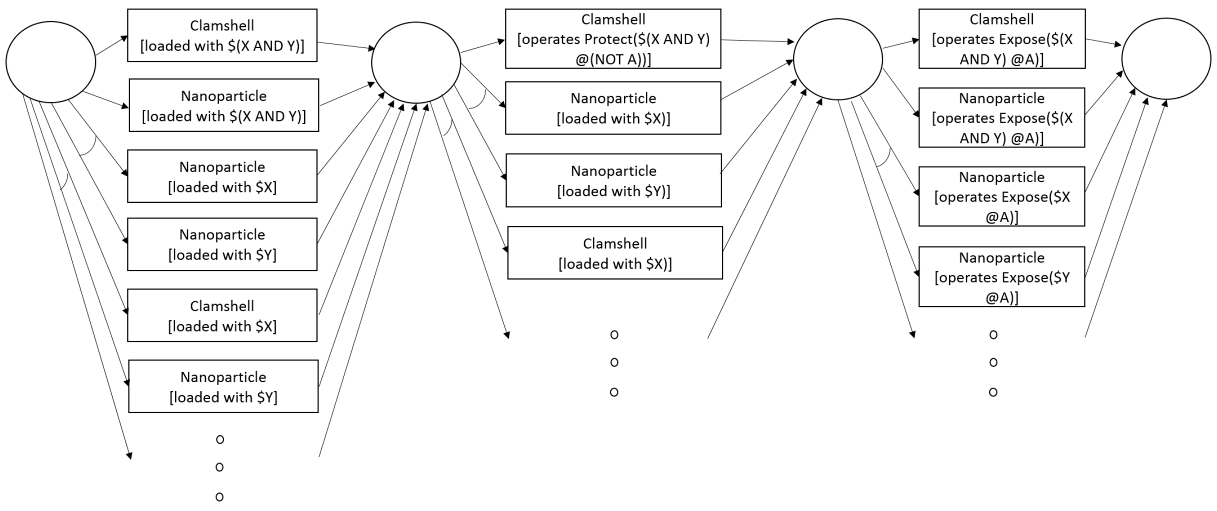

Clamshell [loaded with $(X AND Y)]
Nanoparticle [loaded with $(X AND Y)]
Nanoparticle [loaded with $X]
Nanoparticle [loaded with $Y]
Clamshell [loaded with $X]
Nanoparticle [loaded with $Y]
Clamshell [operates Protect($(X AND Y) @(NOT A))]
Nanoparticle [loaded with $X)]
Nanoparticle [loaded with $Y)]
Clamshell [loaded with $X)]
Clamshell [operates Expose($(X AND Y) @A)]
Nanoparticle [operates Expose($(X AND Y) @A)]
Nanoparticle [operates Expose($X @A)]
Nanoparticle [operates Expose($Y @A)]

Figure 7.7 Illustration of SUBSTITUTE's output

We can see that every transition is substituted to all the robotic combinations that can implement it. Notice that in contrast to the regular SUBSTITUTE, this time the operator not defines a *protect* action for the nanoparticles nodes, as it does for the clamshells, and satisfied with *"loaded with"*. The reason is that since this is a prodrug, the entire drug is exposed by definition by the fact that its components are not mixed together, so it performs the *protect* action as a result. In addition, since the Clamshell can protect all the components when carry them together, the solution in which it carry them all is part of the implementation. But since we cannot promise the Nanoparticle will not make them to become active while they are exposed, there is not such Nanoparticle implementation and thus there will not be any path or solution such as *"Nanoparticle [loaded with $(X AND Y)]"*.

Generally, carrying all the components in one robot is allowed only if it can carry them without mistakenly mixing them, or if it can *protect* the mix.

The last update is in MERGE. Since now there can be a group of robots connected with an AND node that together acting as the *main robot*, it demands that there will be a matching group at the other side of the ε-path in order to merge them together. In case there is not such as group, REJECT removes them.

An illustration of MERGE's action is shown in Figure 7.8. Notice that the second option that SUBSTITUTE offered in the left side of the graph that in Figure 7.7 was removed, since it does not have any *compatible* robot on the right side to merge with. However, the two robot connected with an AND node below it were merged with *compatible* ones on their right.

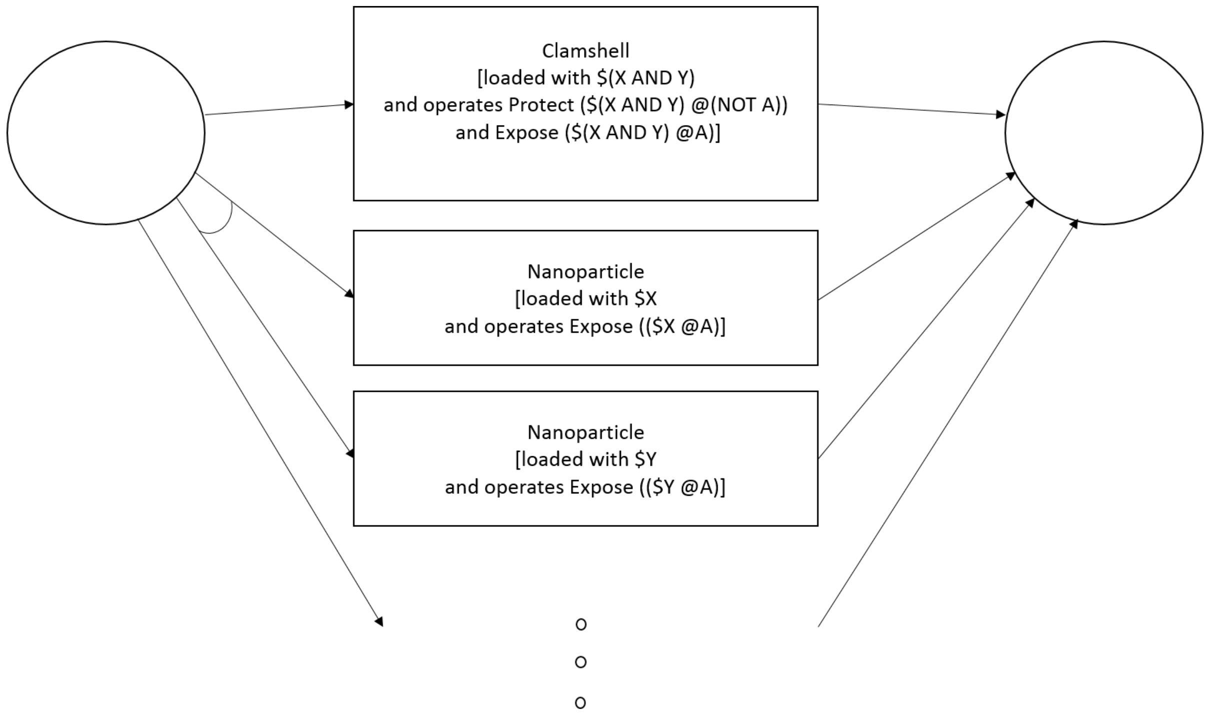

Clamshell
[loaded with $(X AND Y)
and operates Protect ($(X AND Y) @(NOT A))
and Expose ($(X AND Y) @A)]
Nanoparticle
[loaded with $X
and operates Expose (($X @A)]
Nanoparticle
[loaded with $Y
and operates Expose (($Y @A)]
o
o
o

Figure 7.8 Illustration of MERGE's output

Chapter 8

Future Work

This thesis presents the world's first medical treatment programming language for nanobots. It thus takes a step toward a full treatment development environment based on computer science. It raises many issues for future investigation. Some are discussed below.

8.1 Number of Payloads

Bilbo's current algorithms and implementation compiles a subset of the Athelas language, and depends much on the knowledge of the programmer. For example, the discussions in Chapter 7. In the current compilation framework, only one payload can be carried in a process and only one payload can be carried by each nanobot. For example, we cannot execute the thread of the following actions:

```
pick ($(X and Y and Z) @A)
drop ($X @B)
drop ($(Y AND Z) @C)
```

In order to do so the algorithms must be updated, since right now SUBSTITUTE, MERGE, REJECT and DECOMPOSE do not handle nanobots with multi payloads. In addition, the number of the parameters in the FSM's states should grow in perspective to the number of the payloads, what makes it complicated. By obtaining this kind of power, we could utilize the best of the nanobots ability and implements much more sophisticated treatments.

8.2 Representing Nanobots

In Chapter 4 we introduced the nanobots library. We explained it consists of robots descriptions, which hold the Athelas actions each robot can execute. In the future, we would like to add the dependencies it holds, costs in terms of half-life, money and safety, and its preparation protocol. In order to describe all those parameters to the library, a robot description language is needed.

The robot description language named *Hobbit*, is an open topic of investigating that is being researched in our lab. The current version of Bilbo uses hand-coded representation for now. Every nanobot has some parameters that can be configured in order to specialize it for a specific task. In the current representation, each robot in the library is represented by an FSM that describes its capabilities and its configurable parameters. The configurable parameters is depicted by a <param> notation.

For example, the clamshell robot type is described by the FSM in Figure 8.1:

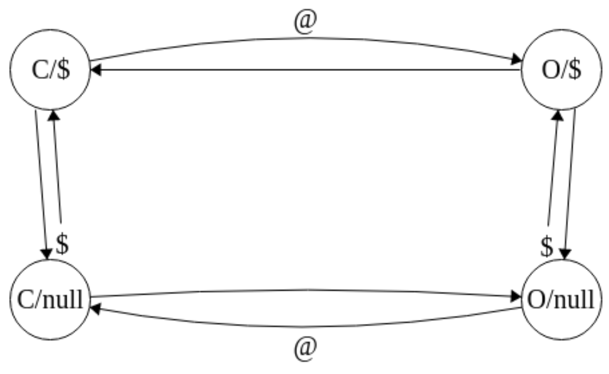


Figure 8.1 Clamshell's FSM

When C stands for state Close ,O stand for state Open, $ stand for state that contains a cargo and null stand for containing nothing. The transitions are marked by @, $, <null >, as an indication that it is affected by the parameter that the symbol represents.

The problem in this way of representation is that as much as the number of the parameters increases, so also the number of the FSM's states become exponential, and very quickly the model turns to be too heavy to work with, so another model is needed. A future work may implement the language using Petri Nets [25], which overcome that issue.

8.3 Optimization Criteria

Right now our compiler offers a set of nanobots solution to a treatment. In the future we would like to be able to choose a preferable result according to criteria of cost and half life. In addition, we want the compiler to automatically add probabilities of success to the different implementations and also to investigate another optimization criteria. That way the solution will not only useful, but optimal too.

Chapter 9

Conclusions

My thesis presents a novel approach to programming nanobots for biomedical applications. Inspired by modern compilation paradigms, I advocate separation of expertise: medical experts to use *Athelas*, a high-level rule-based language to program medications, and nanobot builders will develop nanobots which can be used by the *Bilbo* compiler to compile Athelas programs into heterogeneous nanobot swarm specifications. Concerned with safety, I prove the soundness and completeness of the Bilbo back-end, which is at the heart of the compilation process. I demonstrated that the compiler was able to generate novel swarm specifications, utilizing its knowledge of generic nanobots types. These swarms were shown in-vitro to carry out tasks not possible with a single nanobot type of the same underlying design. I discussed interesting topics as *Asimov Protection* and *Prodrugs* and also talked about an assumption I have made in the compilation process. Finally, I noted what I think that should be the future work.

We believe this thesis opens the door for exciting new opportunities for AI research, reusing and innovating technologies (e.g., rule-based languages, robot swarm programming) in service of a revolutionary approach to development of medical treatments.

Appendix A

Appendix A

I present Athelas' BNF:

⟨*MEDICATION_exp*⟩ ::= 'medication' ':' VAR '{' ⟨*MULTI_RULES_exp*⟩ '}'

⟨*MULTI_RULES_exp*⟩ ::= ⟨*MULTI_RULES_exp*⟩ ':' ⟨*SINGLE_RULE_exp*⟩

| ⟨*SINGLE_RULE_exp*⟩

⟨*SINGLE_RULE_exp*⟩ ::= 'rule' ':' VAR '{' ⟨*INITIALIZE_exp*⟩ ';' ⟨*ACTIONS_exp*⟩ ';' '}'

⟨*INITIALIZE_exp*⟩ ::= 'initialize' ':' ⟨*MULTI_INITIALIZE_exp*⟩

⟨*MULTI_INITIALIZE_exp*⟩ ::= ⟨*MULTI_INITIALIZE_exp*⟩ ':' VAR

| VAR

⟨*ACTIONS_exp*⟩ ::= 'actions' ':' ⟨*MULTI_ACTIONS_exp*⟩

⟨*MULTI_ACTIONS_exp*⟩ ::= ⟨*MULTI_ACTIONS_exp*⟩ ⟨*SINGLE_ACTION_exp*⟩

| ⟨*SINGLE_ACTION_exp*⟩

⟨*SINGLE_ACTION_exp*⟩ ::= 'pick' '(' ⟨*PAYLOAD_exp*⟩ ⟨*LOCATION_exp*⟩ ')'

| 'drop' '(' ⟨*PAYLOAD_exp*⟩ ⟨*LOCATION_exp*⟩ ')'

| 'expose' '(' ⟨*PAYLOAD_exp*⟩ ⟨*LOCATION_exp*⟩ ')'

| 'protect' '(' ⟨*PAYLOAD_exp*⟩ ⟨*LOCATION_exp*⟩ ')'

$\langle$*LOCATION_exp*$\rangle$ ::= '@' $\langle$*MULTI_LOCATION_exp*$\rangle$

$\langle$*PAYLOAD_exp*$\rangle$ ::= '$' VAR

$\langle$*MULTI_LOCATION_exp*$\rangle$ ::= $\langle$*MULTI_LOCATION_exp*$\rangle$ 'and' $\langle$*MULTI_LOCATION_exp*$\rangle$
| VAR

When VAR stands for the following regular expression : `[a-zA-Z_][a-zA-Z0-9_]*`

Appendix B

Appendix B

I present the protocol for making the clamshell described in Section 4.3. The tuple is:

```
Clamshell: <$T, true, @A, @B>
```

The following table describes the ingredients of the clamshell, such as:

- The parameters that colored in purple represent the two gates, described by *V* and *P* and composed, each one, from couple chained segments. In our case they are @A and @B, and the compiler is responsible for putting their matching DNA strands in those segments.
- The parameters that colored in orange and green represent the payloads, in our case $T. As before, the compiler's responsibility is to change them to the payload's strands.
- The other segments determined in advance for general use.

ID	Description		Sequence
1	core		AAAAACCAAACCCTCGTTGTGAATATGGTTTGGTC
2	core		GGAAGAAGTGTAGCGGTCACGTTATAATCAGCAGACTGATAG
3	core		TACGATATAGATAATCGAACAACA
4	core		CTTTTGCTTAAGCAATAAAGCGAGTAGA
5	core		GTCTGAAATAACATCGGTACGGCCGCGCACGG
6	core		GGAAGAGCCAAACAGCTTGCAGGGAACCTAA
7	core		AAAATCACCGGAAGCAAACTCTGTAGCT
8	core		CCTACATGAAGAACTAAAGGGCAGGGCGGAGCCCCGGGC
9	core		CATGTAAAAAGGTAAAGTAATAAGAACG
10	core		ATTAAATCAGGTCATTGCCTGTCTAGCTGATAAATTGTAATA
11	core		ATAGTCGTCTTTTGCGGTAATGCC
12	core		AGTCATGGTCATAGCTGAACTCACTGCCAGT
13	core		AACTATTGACGGAAATTTGAGGGAATATAAA
14	core		ATCGCGTCTGGAAGTTTCATTCCATATAGAAAGACCATC
15	core		AAATATTGAACGGTAATCGTAGCCGGAGACAGTCATAAAAAT
16	core		GTCTTTACAGGATTAGTATTCTAACGAGCATAGAACGC
17	core		GCACCGCGACGACGCTAATGAACAGCTG
18	core		AACTTCATTTTAGAATCGCAAATC
19	core		CGTAGAGTCTTTGTTAAGGCCTTCGTTTTCCTACCGAG
20	core		CCAATCAAAGGCTTATCCGGTTGCTATT
21	core		AGAGGCGATATAATCCTGATTCATCATA
22	core		CCGTAATCCCTGAATAATAACGGAATACTACG
23	core		AAATGGTATACAGGGCAAGGAAATC
24	core		TCCTCATCGTAACCAAGACCGACA
25	core		CATTATCTGGCTTTAGGGAATTATGTTTGGATTAC
26	core		ACCCGCCCAATCATTCCTCTGTCC
27	core		CGACCAGTCACGCAGCCACCGCTGGCAAAGCGAAAGAAC
28	core		CTAAAGGCGTACTATGGTTGCAACAGGAGAGA
29	core		TTGGCAGGCAATACAGTGTTTCTGCGCGGGCG
30	core		TATACAGGAAATAAAGAAATTTTGCCCGAACGTTAAGACTTT
31	core		AAGTATAGTATAAACAGTTAACTGAATTTACCGTTGAGCCAC
32	core		ACATTCAGATAGCGTCCAATATTCAGAA
33	core		AAACATCTTTACCCTCACCAGTAAAGTGCCCGCCC
34	core		GAGATGACCCTAATGCCAGGCTATTTTT
35	core		TCCTGAATTTTTTGTTTAACGATCAGAGCGGA
36	core		GCCGAAAAATCTAAAGCCAATCAAGGAAATA
37	core		AGCGTAGCGCGTTTTCACAAAATCTATGTTAGCAAACGAACGCAACAAA
38	core		ACCAATCGATTAAATTGCGCCATTATTA
39	core		ATCTTACTTATTTTCAGCGCCGACAGGATTCA
40	core		CCCTAAAAGAACCCAGTCACA
41	core		GGAAGGGCGAAAATCGGGTTTTTCGCGTTGCTCGT
42	core		CAGACCGGAAGCCGCCATTTTGATGGGGTCAGTAC
43	core		TAATATTGGAGCAAACAAGAGATCAATATGATATTGCCTTTA
44	core		TTCCTTATAGCAAGCAAATCAAATTTTA
45	core		ACTACGAGGAGATTTTTTCACGTTGAAACTTGCTTT

46	core		AAACAGGCATGTCAATCATATAGATTCAAAAGGGTTATATTT
47	core		AACAGGCACCAGTTAAAGGCCGCTTTGTGAATTTCTTA
48	core		TTCCTGAGTTATCTAAAATATTCAGTTGTTCAAATAGCAG
49	core		AAAGAAACAAGAGAAGATCCGGCT
50	core		TTGAGGGTTCTGGTCAGGCTGTATAAGC
51	core		TTTAACCGTCAATAGTGAATTCAAAAGAAGATGATATCGCGC
52	core		ACGAGCGCCCAATCCAAATAAAATTGAGCACC
53	core		AATAAGTCGAAGCCCAATAATTATTTATTCTT
54	core		ACGAAATATCATAGATTAAGAAACAATGGAACTGA
55	core		TTTCATAGTTGTACCGTAACACTGGGGTTTT
56	core		AGGAGCGAGCACTAACAACTAAAACCCTATCACCTAACAGTG
57	core		CAAAGTATTAATTAGCGAGTTTCGCCACAGAACGA
58	core		TGGGGAGCTATTTGACGACTAAATACCATCAGTTT
59	core		ATAACGCAATAGTAAAATGTTTAAATCA
60	core		ACGAATCAACCTTCATCTTATACCGAGG
61	core		TAATGGTTTGAAATACGCCAA
62	core		CGGAACAAGAGCCGTCAATAGGCACAGACAATATCCTCAATC
63	core		ATTAAAGGTGAATTATCAAAGGGCACCACGG
64	core		GGCAACCCATAGCGTAAGCAGCGACCATTAA
65	core		AGAAACGTAAGCAGCCACAAGGAAACGATCTT
66	core		AGAGGTCTTTAGGGGGTCAAAAGGCAGT
67	core		GGGGACTTTTTCATGAGGACCTGCGAGAATAGAAAGGAGGAT
68	core		TTTTAGAACATCCAATAAATCCAATAAC
69	core		AAATGTGGTAGATGGCCCGCTTGGGCGC
70	core		ACGGATCGTCACCCTCACGATCTAGAATTTT
71	core		CGCCATAAGACGACGACAATAGCTGTCT
73	core		AGAGAACGTGAATCAAATGCGTATTTCCAGTCCCC
74	core		AACGAAAAAGCGCGAAAAAAAGGCTCCAAAAGG
75	core		TAATTTAGAACGCGAGGCGTTAAGCCTT
76	core		ACCAGGCGTGCATCATTAATTTTTTCAC
77	core		CAGCCTGACGACAGATGTCGCCTGAAAT
78	core		ATTAGTCAGATTGCAAAGTAAGAGTTAAGAAGAGT
79	core		CTCGAATGCTCACTGGCGCAT
80	core		GGGCAGTCACGACGTTGAATAATTAACAACC
81	core		TAAAAACAGGGGTTTTGTTAGCGAATAATATAATAGAT
82	core		TCAACCCTCAGCGCCGAATATATTAAGAATA
83	core		ATTATACGTGATAATACACATTATCATATCAGAGA
84	core		GCAAATCTGCAACAGGAAAAATTGC
85	core		ATAATTACTAGAAATTCTTAC
86	core		TATCACCGTGCCTTGAGTAACGCGTCATACATGGCCCCTCAG
87	core		AAGTAGGGTTAACGCGCTGCCAGCTGCA
88	core		CCAGTAGTTAAGCCCTTTTTAAGAAAAGCAAA
89	core		TGGCGAAGTTGGGACTTTCCG
90	core		CAGTGAGTGATGGTGGTTCCGAAAACCGTCTATCACGATTTA
91	core		AAATCAAAGAGAATAACATAACTGAACACAGT
92	core		CTGTATGACAACTAGTGTCGA
93	core		ATCATAAATAGCGAGAGGCTTAGCAAAGCGGATTGTTCAAAT
94	core		TTGAGTAATTTGAGGATTTAGCTGAAAGGCGCGAAAGATAAA
95	core		ATAAGAATAAACACCGCTCAA
96	core		CGTTGTAATTCACCTTCTGACAAGTATTTTAA
97	core		AACCGCCTCATAATTCGGCATAGCAGCA
98	core		AAATAGGTCACGTTGGTAGCGAGTCGCGTCTAATTCGC

99	core		CAGTATAGCCTGTTTATCAACCCCATCC
100	core		TTGCACCTGAAAATAGCAGCCAGAGGGTCATCGATTTTCGGT
101	core		CGTCGGAAATGGGACCTGTCGGGGGAGA
102	core		AAGAAACTAGAAGATTGCGCAACTAGGG
103	core		CCAGAACCTGGCTCATTATACAATTACG
104	core		ACGGGTAATAAATTAAGGAATTGCGAATAGTA
105	core		CCACGCTGGCCGATTCAAACTATCGGCCCGCT
106	core		GCCTTCACCGAAAGCCTCCGCTCACGCCAGC
107	core		CAGCATTAAAGACAACCGTCAAAAATCA
108	core		ACATCGGAAATTATTTGCACGTAAAAGT
109	core		CAACGGTCGCTGAGGCTTGATACCTATCGGTTTATCAGATCT
110	core		AAATCGTACAGTACATAAATCAGATGAA
111	core		TTAACACACAGGAACACTTGCCTGAGTATTTG
112	core		AGGCATAAGAAGTTTTGCCAGACCCTGA
113	core		GACGACATTCACCAGAGATTAAAGCCTATTAACCA
114	core		AGCTGCTCGTTAATAAAACGAGAATACC
115	core		CTTAGAGTACCTTTTAAACAGCTGCGGAGATTTAGACTA
116	core		CACCCTCTAATTAGCGTTTGCTACATAC
117	core		GAACCGAAAATTGGGCTTGAGTACCTTATGCGATTCAACACT
118	core		GCAAGGCAGATAACATAGCCGAACAAAGTGGCAACGGGA
119	core		ATGAAACAATTGAGAAGGAAACCGAGGATAGA
120	core		GGATGTGAAATTGTTATGGGGTGCACAGTAT
121	core		GGCTTGCGACGTTGGGAAGAACAGATAC
122	core		TAAATGCCTACTAATAGTAGTTTTCATT
123	core		TGCCGTCTGCCTATTTCGGAACCAGAATGGAAAGCCCACCAGAAC
124	core		TGACCATAGCAAAAGGGAGAACAAC
125	core		CGAGCCAGACGTTAATAATTTGTATCA
126	core		GCTCAGTTTCTGAAACATGAAACAAATAAATCCTCCCGCCGC
127	core		AGACGCTACATCAAGAAAACACTTTGAA
128	core		AGTACTGACCAATCCGCGAAGTTTAAGACAG
129	core		GATTCCTGTTACGGGCAGTGAGCTTTTCCTGTGTGCTG
130	core		GGTATTAAGGAATCATTACCGAACGCTA
131	core		GTTCATCAAATAAAACGCGACTCTAGAGGATCGGG
132	core		AGCCTTTAATTGGATAGTTGAACCGCCACCCTCATAGGTG
133	core		ACAGAGGCCTGAGATTCTTTGATTAGTAATGG
134	core		AACGAGATCAGGATTAGAGAGCTTAATT
135	core		TACCAAGTTATACTTCTGAATCACCAGA
136	core		CAGTAGGTGTTCAGCTAATGCGTAGAAA
137	core		AGGATGACCATAGACTGACTAATGAAATCTACATTCAGCAGGCGCGTAC
138	core		TTTCAACCAAGGCAAAGAATTTAGATAC
139	core		TTGAAATTAAGATAGCTTAACTAT
140	core		CTATTATCGAGCTTCAAAGCGTATGCAA
141	core		CAGGGTGCAAAATCCCTTATAGACTCCAACGTCAAAAGCCGG
142	core		GAGCTTGTTAATGCGCCGCTAATTTTAGCGCCTGCTGCTGAA
143	core		CGAACGTTAACCACCACACCCCCAGAATTGAG
144	core		GTGTGATAAATAAGTGAGAAT
145	core		GCTATATAGCATTAACCCTCAGAGA
146	core		AGGAGAGCCGGCAGTCTTGCCCCCGAGAGGGAGGG
147	core		CGGCCTCCAGCCAGAGGGCGAGCCCCAA
148	core		CCAAAACAAAATAGGCTGGCTGACGTAACAA
149	core		GGCGGTTAGAATAGCCCGAGAAGTCCACTATTAAAAAGGAAG
150	core		ATAAAGGTTACCAGCGCTAATTCAAAAACAGC

152	core		TTTTAAAACATAACAGTAATGGAACGCTATTAGAACGC
153	core		AATTGGGTAACGCCAGGCTGTAGCCAGCTAGTAAACGT
154	edge		TTACCCAGAACAACATTATTACAGGTTTTTTTTTTTTTTT
155	edge		TTTTTTTTTTTTTTTTAATAAGAGAATA
156	edge		TTTTTTTTTTTTTTTCCAGTTTGGGAGCGGGCTTTTTTTTTTTTTTT
157	edge		GGTTGAGGCAGGTCAGTTTTTTTTTTTTTTT
158	edge		TTTTTTTTTTTTTTTGATTAAGACTCCTTATCCAAAAGGAAT
159	edge		TTTTTTTTTTTTTTTCTTCGCTATTACAATT
160	edge		TTTTTTTTTTTTTTTCTTGCGGGAGAAGCGCATTTTTTTTTTTTTTTT
161	edge		TTTTTTTTTTTTTTGGGAATTAGAGAAACAATGAATTTTTTTTTTTTT TT
162	edge		TCAGACTGACAGAATCAAGTTTGTTTTTTTTTTTTTTT
163	edge		TTTTTTTTTTTTTTTGGTCGAGGTGCCGTAAAGCAGCACGT
164	edge		TTTTTTTTTTTTTTTTTTAATCATTTACCAGACTTTTTTTTTTTTTTT
165	edge		TTTTTTTTTTTTTCATTCTGGCCAAATTCGACAACTCTTTTTTTTTTTT T
166	edge		TTTTTTTTTTTTTTTACCGGATATTCA
167	edge		TTTTTTTTTTTTTTTTAGACGGGAAACTGGCATTTTTTTTTTTTTTTT
168	edge		TTTTTTTTTTTTTTTCAGCAAGCGGTCCACGCTGCCCAAAT
169	edge		CTGAGAGAGTTGTTTTTTTTTTTTTTT
170	edge		CAATGACAACAACCATTTTTTTTTTTTTTTT
171	edge		TTTTTTTTTTTTTTTTGAGAGATCTACAAGGAGAGG
172	edge		TCACCAGTACAAACTATTTTTTTTTTTTTTT
173	edge		TTTTTTTTTTTTTGGCAATTCATCAAATTATTCATTTTTTTTTTTTTTT T
174	edge		TAAAGTTACCGCACTCATCGAGAACTTTTTTTTTTTTTTT
175	edge		TTTTTTTTTTTTTTTCACCCTCAGAACCGCC
176	edge		TTTTTTTTTTTTTAGGTTTAACGTCAATATATGTGAGTTTTTTTTTTTT
177	edge		CCACACAACATACGTTTTTTTTTTTTT
178	edge		TTTTTTTTTTTTTTTGCTAGGGCGAGTAAAAGATTTTTTTTTTTTTTT
179	edge		TTTTTTTTTTTTTTTAGTTGATTCCCAATTCTGCGAACCTCA
180	edge		TTATTTAGAGCCTAATTTGCCAGTTTTTTTTTTTTTTTT
181	edge		TTTTTTTTTTTTTTTACGGCGGAT
182	edge		TTTTTTTTTTTTTTTATATGCGTTAAGTCCTGATTTTTTTTTTTTTTT
183	edge		TTTTTTTTTTTTTTTACGATTGGCCTTGATA
184	edge		TTTTTTTTTTTTTTTCAACGCCTGTAGCATT
185	edge		TTTTTTTTTTTTTTTGGCTTTGAGCCGGAACGATTTTTTTTTTTTTTT
186	edge		TTTTTTTTTTTTTTTAAGCAAGCCGTTT
187	edge		TTTTTTTTTTTTTATGTGTAGGTAAGTACCCCGGTTGTTTTTTTTTTTT T
188	edge		ATCGTCATAAATATTCATTTTTTTTTTTTTTTT
189	edge		TTTTTTTTTTTTTTTGTTAATTTCATCT
190	edge		TTTTTTTTTTTTTGTATTAAATCCTGCGTAGATTTCTTTTTTTTTTTT T
191	edge		GCCATATAAGAGCAAGCCAGCCCGACTTGAGCCATGGTT
192	edge		GTAGCTAGTACCAAAAACATTCATAAAGCTAAATCGGTTTTTTTTTT TTT
193	edge		ATAACGTGCTTTTTTTTTTTTTTTT
194	edge		TTTTTTTTTTTTTTTAAAATACCGAACGAACCACCAGTGAGAATTAA C
195	edge		TTTTTTTTTTTTTTTACAAAATAAACA
196	edge		TTTTTTTTTTTTTTTACAAGAAAAACCTCCCGATTTTTTTTTTTTTTT
197	edge		TTTTTTTTTTTTTTTGACGATAAAAAGATTAAGTTTTTTTTTTTTTTT
198	edge		TTTTTTTTTTTTTCAATTACCTGAGTATCAAAATCATTTTTTTTTTTTT T
199	edge		GGTACGGCCAGTGCCAAGCTTTTTTTTTTTTTTT
200	edge		TTTTTTTTTTTTTTGAATAACCTTGAAATATATTTATTTTTTTTTTTT T

201	edge		CACTAAAACACTTTTTTTTTTTTTTT
202	edge		TTTTTTTTTTTTTTTTAACCAATATGGGAACAATTTTTTTTTTTTTTT
203	edge		TACGTCACAATCAATAGAATTTTTTTTTTTTTTT
204	edge		TTTTTTTTTTTTTTTAGAAAGATTCATCAGTTGA
205	edge		TTTTTTTTTTTTTGTGGCATCAATTAATGCCTGAGTATTTTTTTTTTTTTT
206	edge		TTTTTTTTTTTTTTTTTGCATGCCTGCATTAATTTTTTTTTTTTTTTT
207	edge		CCAGCGAAAGAGTAATCTTGACAAGATTTTTTTTTTTTTTT
208	edge		TTTTTTTTTTTTTTTGAATCCCCCTCAAATGCTT
209	edge		AGAGGCTGAGACTCCTTTTTTTTTTTTTTT
210	edge		ACAAACACAGAGATACATCGCCATTATTTTTTTTTTTTTTT
211	edge		TTTTTTTTTTTTTTTCAAGAGAAGGATTAGG
212	edge		TTTTTTTTTTTTTGAATTGAGGAAGTTATCAGATGATTTTTTTTTTTTTT
213	edge		CAGAACAATATTTTTTTTTTTTTTT
214	edge		TTTTTTTTTTTTTAGCCGGAAGCATAAAGTGTCCTGGCC
215	edge		TGACCGTTTCTCCGGGAACGCAAATCAGCTCATTTTTTTTTTTTTTTTT
216	edge		TTTTTTTTTTTTTTTGGTAATAAGTTTTAAC
217	edge		TTTTTTTTTTTTTTGTCTGTCCATAATAAAAGGGATTTTTTTTTTTTTT
218	edge		TTTTTTTTTTTTTTTCCTCGTTAGAATCAGAGCGTAATATC
219	edge		AATTGCTCCTTTTGATAAGTTTTTTTTTTTTTTT
220	edge		CATCGGACAGCCCTGCTAAACAACTTTCAACAGTTTTTTTTTTTTTTT
221	edge		TTTTTTTTTTTTTTTAACCGCCTCCCTCAGACCAGAGC
222	edge		TCTGACAGAGGCATTTTCGAGCCAGTTTTTTTTTTTTTTT
223	edge		TTTTTTTTTTTTTTTTTTCAGCGGAGTTCCATGTCATAAGG
224	edge		TTTTTTTTTTTTTTTCGCCCACGCATAACCG
225	edge		AATTACTTAGGACTAAATAGCAACGGCTACAGATTTTTTTTTTTTTTT
226	edge		CAAGTTTTTTGGTTTTTTTTTTTTTTT
227	edge		TTTTTTTTTTTTTTTCCTTTAGCGCACCACCGGTTTTTTTTTTTTTTT
228	edge		TTTTTTTTTTTTTTTGAATCGGCCGAGTGTTGTTTTTTTTTTTTTTTT
229	edge		TTTTTTTTTTTTTCATCTTTGACCC
230	edge		TTTTTTTTTTTTTATAATCAGAAAATCGGTGCGGGCCTTTTTTTTTTTTT
231	edge		GATACAGGAGTGTACTTTTTTTTTTTTTTT
232	edge		TTTTTTTTTTTTTTTGGCGCAGACAATTTCAACTTTTTTTTTTTTTTT
233	edge		GGAGGTTTAGTACCGCTTTTTTTTTTTTTTT
234	edge		TTTTTTTTTTTTTACCGCCAGCCATAACAGTTGAAAGTTTTTTTTTTTTT
235	edge		TTTTTTTTTTTTTTTATAGCAATAGCT
236	Key handle		AATAAGTTTTGCAAGCCCAATAGGGGATAAGTATCGGATGACTATACT
237	handles		ACATAGCTTACATTTAACAATAATAACGTTGTGCTACTCCAGTTC
238	handles		CCTTTTTGAATGGCGTCAGTATTGTGCTACTCCAGTTC
239	handles		CGTAACCAATTCATCAACATTTTGTGCTACTCCAGTTC
240	handles		CACCAACCGATATTCATTACCATTATTGTGCTACTCCAGTTC
241	handles		CCACCCTCATTTTCTTGATATTTGTGCTACTCCAGTTC
242	handles		AACTTTGAAAGAGGAGAAACATTGTGCTACTCCAGTTC
243	handles		CAAGGCGCGCCATTGCCGGAATTGTGCTACTCCAGTTC
244	handles		CATAGCCCCCTTAAGTCACCATTGTGCTACTCCAGTTC
245	handles		TTTCCCTGAATTACCTTTTTTACCTTTTTTGTGCTACTCCAGTTC
246	handles		AACGGTGTACAGACTGAATAATTGTGCTACTCCAGTTC
247	handles		GATTCGCGGGTTAGAACCTACCATTTTGTTGTGCTACTCCAGTTC
248	guides		AGAGTAGGATTTCGCCAACATGTTTTAAAAACC
249	guides		ACGGTGACCTGTTTAGCTGAATATAATGCCAAC
250	guides		CGTAGCAATTTAGTTCTAAAGTACGGTGTTTTA

Bibliography

[1] Y. Amir, E. Ben-Ishay, D. Levner, S. Ittah, A. Abu-Horowitz, and I. Bachelet. Universal computing by DNA origami robots in a living animal. *Nature Nanotechnology*, 9(5):353–357, May 2014.

[2] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, Nov. 2007.

[3] W. Arap, R. Pasqualini, and E. Ruoslahti. Cancer treatment by targeted drug delivery to tumor vasculature in a mouse model. *Science*, 279(5349):377–380, 1998.

[4] J. Aspnes and E. Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, Oct. 2007.

[5] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19:825–847, 2010.

[6] A. Cavalcanti, B. Shirinzadeh, T. Fukuda, and S. Ikeda. Nanorobot for brain aneurysm. *International Journal of Robotics Research*, 28(4):558–570, 2009.

[7] I. Chatzigiannakis and P. G. Spirakis. The dynamics of probabilistic population protocols. *CoRR*, abs/0807.0140, 2008.

[8] H. Dietz, S. M. Douglas, and W. M. Shih. Folding DNA into twisted and curved nanoscale shapes. *Science*, 325(5941):725–730, 2009.

[9] L. Dong and B. Nelson. Tutorial - robotics in the small part ii: Nanorobotics. *Robotics Automation Magazine, IEEE*, 14(3):111–121, Sept 2007.

[10] S. M. Douglas, I. Bachelet, and G. M. Church. A logic-gated nanorobot for targeted transport of molecular payloads. *Science*, 335(6070):831–834, Feb 2012.

[11] S. M. Douglas, H. Dietz, T. Liedl, B. Högberg, F. Graf, and W. M. Shih. Self-assembly of DNA into nanoscale three-dimensional shapes. *Nature*, 459(7245):414–418, 2009.

[12] E. Estey. Treatment of AML: resurrection for gemtuzumab ozogamicin? *The Lancet*, 379(9825):1468–1469, 2012.

[13] R. A. Freitas. Current status of nanomedicine and medical nanorobotics. *Journal of Computational and Theoretical Nanoscience*, 2(1):1–25, 2005.

[14] S. V. Govindan and D. M. Goldenberg. Designing immunoconjugates for cancer therapy. *Expert opinion on biological therapy*, 12(7):873–890, 2012.

[15] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. *SIGARCH Computer Architecture News*, 14(2):28–37, May 1986.

[16] D. Han, S. Pal, J. Nangreave, Z. Deng, Y. Liu, and H. Yan. DNA origami with complex curvatures in three-dimensional space. *Science*, 332(6027):342–346, 2011.

[17] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, Sep 1985.

[18] A. A. Hopgood. *Intelligent Systems for Engineers and Scientists*. CRC Press, 2001.

[19] G. A. Kaminka, N. Agmon, and I. Bachelet. On the tight coupling between molecular robots and their programming languages: Initial thoughts. In *IROS 2014 workshop on Micro-Nano Robotic Swarms for Biomedical Applications*, 2014.

[20] G. A. Kaminka, R. Spokoini-Stern, Y. Amir, N. Agmon, and I. Bachelet. Molecular robots obeying Asimov's three laws of robotics. *Artificial Life*, 2017. In press.

[21] A. Ligêza. *Logical Foundations for Rule-Based Systems*, volume 11 of *Studies in Computational Intelligence*. Springer, 2006.

[22] M. A. Moses, H. Brem, and R. Langer. Advancing the field of drug delivery: taking aim at cancer. *Cancer cell*, 4(5):337–341, 2003.

[23] N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing, Palo Alto, CA, 1980.

[24] J. H. Park, G. von Maltzahn, L. L. Ong, A. Centrone, T. A. Hatton, E. Ruoslahti, S. N. Bhatia, and M. J. Sailor. Cooperative nanoparticles for tumor detection and photothermally triggered drug delivery. *Advanced Materials*, 22:880–885, 2010.

[25] J. L. Peterson. Petri net theory and the modeling of systems. 1981.

[26] C. Pinciroli, A. Lee-Brown, and G. Beltrame. Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. Available online at http://arxiv.org/abs/1507.05946, 2015.

[27] E. Ruoslahti, S. N. Bhatia, and M. J. Sailor. Targeting of drugs and nanoparticles to tumors. *Journal of Cell Biology*, 188(6):759–768, 2010.

[28] A. Schroeder, R. Honen, K. Turjeman, A. Gabizon, J. Kost, and Y. Barenholz. Ultrasound triggered release of cisplatin from liposomes in murine tumors. *Journal of Controlled Release*, 137(1):63–68, 2009.

[29] S. Sengupta, D. Eavarone, I. Capila, G. Zhao, N. Watson, T. Kiziltepe, and R. Sasisekharan. Temporal targeting of tumour cells and neovasculature with a nanoscale delivery system. *Nature*, 436(7050):568–572, 2005.

[30] J.-S. Shin and N. A. Pierce. A synthetic dna walker for molecular transport. *Journal of the American Chemical Society*, 126(35):10834–10835, 2004.

[31] G. Y. Tonga, Y. Jeong, B. Duncan, T. Mizuhara, R. Mout, R. Das, S. T. Kim, Y.-C. Yeh, B. Yan, S. Hou, et al. Supramolecular regulation of bioorthogonal catalysis in cells using nanoparticle-embedded transition metal catalysts. *Nature chemistry*, 7(7):597–603, 2015.

[32] G. von Maltzahn, J. H. Park, K. Y. Lin, N. Singh, C. Schwöppe, R. Mesters, W. E. Berdel, E. Ruoslahti, M. J. Sailor, and S. N. Bhatia. Nanoparticles that communicate in vivo to amplify tumour targeting. *Nature Materials*, 10:545–552, 2011.

[33] B. Wei, M. Dai, and P. Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.

[34] I. Wiesel, G. A. Kaminka, G. Hachmon, N. Agmon, and I. Bachelet. Late-breaking: First steps towards automated implementation of molecular robot tasks. In *DNA Computing (DNA-21)*, 2015.

[35] P. Yin, R. F. Hariadi, S. Sahu, H. M. Choi, S. H. Park, T. H. LaBean, and J. H. Reif. Programming DNA tube circumferences. *Science*, 321(5890):824–826, 2008.

[36] R. M. Zadegan, M. D. Jepsen, K. E. Thomsen, A. H. Okholm, D. H. Schaffert, E. S. Andersen, V. Birkedal, and J. Kjems. Construction of a 4 zeptoliters switchable 3d dna box origami. *ACS nano*, 6(11):10050–10053, 2012.