Rule-Based Programming of Molecular Robot Swarms for Biomedical Applications

Inbal Wiesel-Kapah[1], **Gal A. Kaminka**[1], **Guy Hachmon**[2], **Noa Agmon**[1], **Ido Bachelet**[3]
[1]Computer Science Department, Bar Ilan University, Israel
[2]XLX Technologies, Israel
[3]Augmanity, Ltd., Israel
wieseli@biu.ac.il, {galk,agmon}@cs.biu.ac.il, guy.hachmon@gmail.com, dogbach@gmail.com

Abstract

Molecular robots (nanobots) are being developed for biomedical applications, e.g., to deliver medications without worrying about side-effects. Future treatments will require swarms of heterogeneous nanobots We present a novel approach to generating such swarms from a treatment program. A compiler translates medications, written in a rule-based language, into specifications of a swarm built by specializing generic nanobot platforms to specific payloads and action-triggering behavior. The mixture of nanobots, when deployed, carries out the treatment program. We describe the medication programming language, and the associated compiler. We prove the validity of the compiler output, and report on in-vitro experiments using generated nanobot swarms.

1 Introduction

Nanometer-scale molecular robotics has emerged as a promising approach for *targeted drug delivery*. Molecular robots (nanobots) can operate inside a living body [Dong and Nelson, 2007; Amir *et al.*, 2014], carrying out simple molecular actions, such as releasing a molecular payload only under some environmental conditions or shielding the body from toxic payloads [Douglas *et al.*, 2012]. If used as a platform for drug delivery, a nanobot can, in principle, overcome many of the safety issues, as drugs are released only in the presence of their targets.

Currently, every nanobot must be designed by an expert, for the specific task: medical expertise must meet nanobot design expertise. As procedures grow in complexity, the challenge is further exacerbated: nanobot developers mix different types of nanobots—each type specifically tailored to its role—in heterogeneous swarms, such that the medical outcome emerges out of the interactions of the nanobots in the swarm [Ruoslahti *et al.*, 2010; Park *et al.*, 2010a; 2010b].

We present a novel approach to developing nanobot swarms. Inspired by modern software development environments, which separates high-level programming languages from specific CPU details, we aim to allow medical professionals to directly program treatments in a *Athelas*, a rule-based medication programming language, modeled after rule-based languages for knowledge-based systems [Hayes-Roth, 1985; Hopgood, 2001; Ligêza, 2006]. A compiler (*Bilbo*) translates Athelas programs to nanobot specifications, guaranteed to implement the written program. The compiler relies on a library of generic nanobot arch-types, and specializes them to create the specific roles needed for the swarm.

We believe this separation between medical expertise and nanobot design expertise can significantly accelerate the development of new medical treatments relying on nanobot technology: 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.

In this paper, we will introduce the Athelas language and the compilation algorithms used in Bilbo. We prove the soundness and completeness of the compilation process, and present results of the compiler in-vitro experiments, with actual nanobots.

2 Background and Motivation

Nanobots are nanometer-scale devices that can operate inside of a living body [Cavalcanti *et al.*, 2009; Dong and Nelson, 2007; Banerjee *et al.*, 2013; Amir *et al.*, 2014], and have the potential to revolutionize medicine [Freitas, 2005], 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 [Ruoslahti *et al.*, 2010; von Maltzahn *et al.*, 2011]. 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) in specific ways. [Banerjee *et al.*, 2013] reports DNA "cages" which can hold small payloads. Both the openning triggers and the payloads can be varied. Recently developed nano-particles [Tonga *et al.*, 2015] serve as an additional 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.

We use the DNA-based *clamshell* nanobot [Douglas *et al.*, 2012] in the experiments. It resembles a hexagonal clamshell, open at both ends (Figure 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. We can program the nanobot by specializing its parameters: choosing the appropriate components such that the clamshell opens when it encounters the predefined 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. When more complex actions are required, a heterogeneous swarm is needed. The clamshell robots essentially respond as a two-input AND gate. If the target is identifiable by three or more markers in combination, no single clamshell nanobot type can correctly open only in the target location. Instead, a heterogeneous combination of clamshells (one responding to markers A and B by releasing an compound T, and one responding to T and C) is needed.

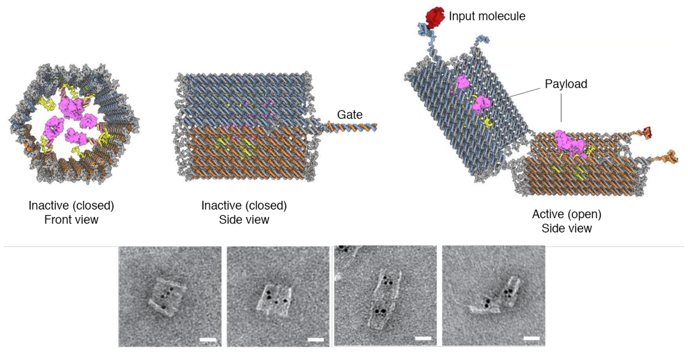


Figure 1: A clamshell DNA nanobot. Up: a schematic view of the two states: closed (left) and open (right). Down: TEM images (scale bar, 25nm).

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 [Hayes-Roth, 1985; Gupta *et al.*, 1986; Hopgood, 2001; Ligêza, 2006] 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* [Pinciroli *et al.*, 2015], and *Proto* [Bachrach *et al.*, 2010] which focus on spatial swarm behaviors, computation, and synchronized knowledge.

3 Compiling Medications into Swarms

The *Tolkien* development environment for nanobot-based biomedical applications includes a high level language for programming medications (Section 3.1) which are compiled into swarm specifications (Section 3.2).

3.1 The Athelas Language

Medications work by moving compounds between locations in the body: picking compounds (by molecular binding) or exposing (and sometimes releasing) them in diseased areas. We consider these to be nanobot swarm tasks, to be programmed by a user.

As activities will be carried out by a swarm of millions of nanobots, we adopt a *rule-based programming* paradigm, in which programs are specified by sets of rules that are continuously considered in parallel, against changing conditions [Gupta *et al.*, 1986]).

Each rule has four clauses, discussed below (Figure 2 shows an example). As a matter of notation, Athelas code uses `$` to denote payloads, and `@` for location expressions denoting biological markers (e.g., CD Antigens), which can be aliased for convenience using `#define` macros. Locations can be specified using logical operators (AND, OR, NOT).

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 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 Figure 2, the rule states that the drug contains compounds `Z` and `X` when it begins.

The `When` and `Until` clauses are each composed of a set of tests, e.g., pH level or concentration of a specific chemical in specific location. The `When` tests must hold in order for the drug to become activated (here, when the concentration of `Y` in the vicinity of `T` is above $5mol/m^3$). The `Until` terminates the activity of the drug (here, when the concentration drops below $2mol/m^3$). Note that the `When` conditions do not need to hold through the activation of the drug; they only trigger it.

The `Actions` clause contains the actions to be executed when the drug is active. `pick($payload @location)` instructions cause the nanobots to be built with appropriate compounds to bind `$payloads` (if encountered near `@location`) so that is carried by the nanobot. Similar actions are defined for releasing the payload, allowing it to float freely (`drop`), and for protecting it from, or exposing it to, the environment (`protect, expose`, e.g., via a mechanism such as a protective outer shell, as in the clamshell nanobot). Other actions include `disable, enable` which operate on rules (and are given rule names). This type of reflection to address inter-drug interactions, e.g., to set drug action priorities.

In Figure 2, there are two `drop` actions. One to drop the compound `$Z` which the drug initially contains, at location `@T`. The other drops `$Y` at the location `@(A AND B AND C)`, i.e., a location marked by the presence of all biomarkers `A, B, C` (`#define#`'d elsewhere).

```
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 2: 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., `A AND B`), and it cannot selectively drop only `Z` or expose only `X` 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.

3.2 The Bilbo Compiler

The Bilbo compiler takes two inputs: an Athelas program, and a library of generic robot types (with defined ways of parameterizing them, including parameterizable preparation protocols). It then synthesizes a specification for a heterogeneous swarm of specialized nanobots, which would carry out the program, once deployed. The output specification for each specialized robot includes a specialized preparation protocols.

The compilation process is done in two phases. A front-end phase consists of the lexical and syntax analyzers, generates finite state machines (FSMs) representing the rules. The back-end phase then transforms such FSMs into a final nanobot swarm specification (recipe). We discuss both in detail below.

The Front-End.

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 2) and a library of generic nanobot arch-types, which the compiler uses in the recipes recipes. The back-end transforms each FSM to an AND/OR graph (see below) which represents alternative swarm specifications (nanobot mixes). It then uses AO* [Nilsson, 1980] to determine an optimal AND/OR path in the graph, which corresponds to a specific heterogeneous swarm, made of specialized nanobot arch-types and their preparation protocols.

Algorithm 1 (ConstructTree), which transforms FSMs into and/or graphs, is the heart of the back-end phase. It uses a *graph rewriting* approach to carry out the transformation, working with four graph-rewriting operators: SUBSTITUTE, MERGE, REJECT and DECOMPOSE.

We illustrate the operation of Algorithm 1 by transforming the EXPOSE FSM in Figure 3 (right). For this demonstration, we assume the two nanobot arch-types previously described (Section 2): the clamshell [Douglas *et al.*, 2012], and the nano-particle [Tonga *et al.*, 2015].

Algorithm 1 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 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 3 is substituted in Figure 4 by a clamshell and nano-particle. 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.

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. 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 (left) results in Figure 4 (right).

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 1 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. 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 5, the robot-

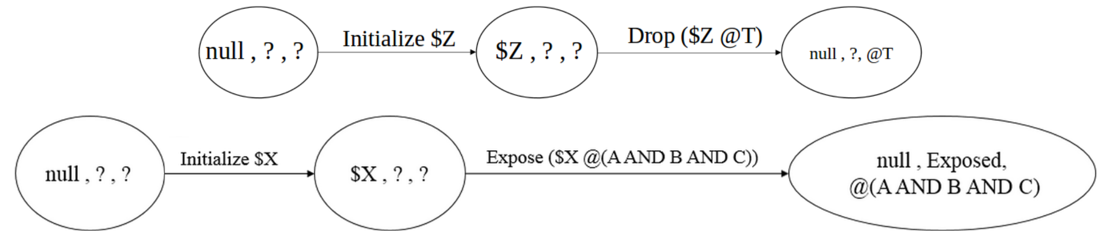


Figure 3: The front-end's output, the back-end's input. Two FSMs generated for the rule in Figure 2.

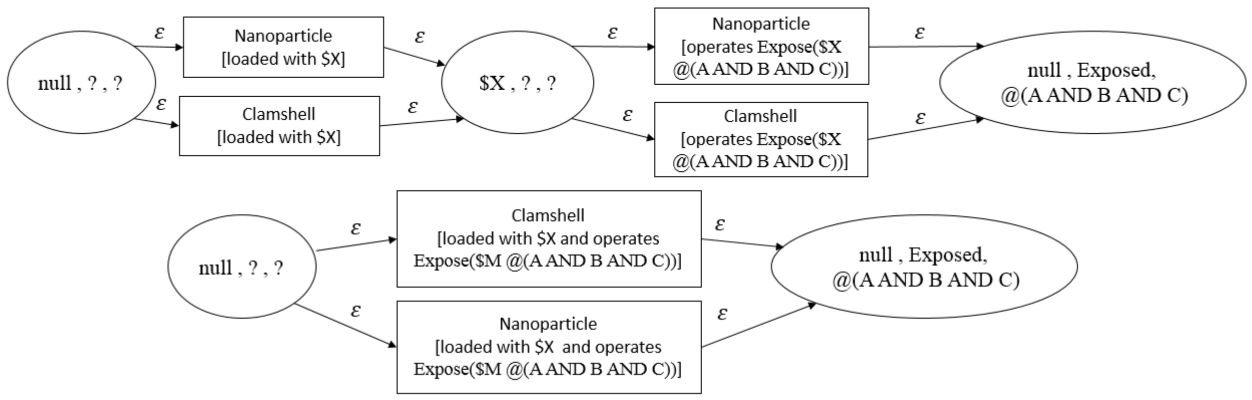

null , ? , ?
ε
Nanoparticle [loaded with $X]
Clamshell [loaded with $X]
$X , ? , ?
Nanoparticle [operates Expose($X @(A AND B AND C))]
Clamshell [operates Expose($X @(A AND B AND C))]
null , Exposed, @(A AND B AND C)
Clamshell [loaded with $X and operates Expose($M @(A AND B AND C))]
Nanoparticle [loaded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

Figure 4: Illustration of SUBSTITUTE (up) and MERGE (down) operations on input shown in Figure 3.

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.

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

The AO* algorithm [Nilsson, 1980] then selects an optimal path from the initial state to the end, where transitions have weights denoting costs (or based on number of steps, otherwise). 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 Proofs

The generated nanobot swarms are intended to one day serve in biomedical applications, thus safety and performance guarantees are crucial. In this section we prove that Algorithm ConstructTree is complete, is sound, and halts. For the sake of the analysis we distinguish between the robot who implements the transitions given as an input and those who implement transitions created by DECOMPOSE. We refer the first as the *main* robot and the others as the *assistant* robots.

Lemma 1. *Algorithm* ConstructTree *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 Algorithm ConstructTree failed to return it (the AND/OR graph is empty). 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 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 the main robot X should be assisted by other robots, then DECOMPOSE must offer their help, thus does not cancel the path. (iv) Dependency-check 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 the dependency-check cannot remove it. Thus, given that the input is correct, the algorithm will not remove a valid solution along its way. □

Lemma 2. *Algorithm* ConstructTree *is sound, i.e., the output is a correct implementation of the given FSMs.*

Proof. (Sketch) 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

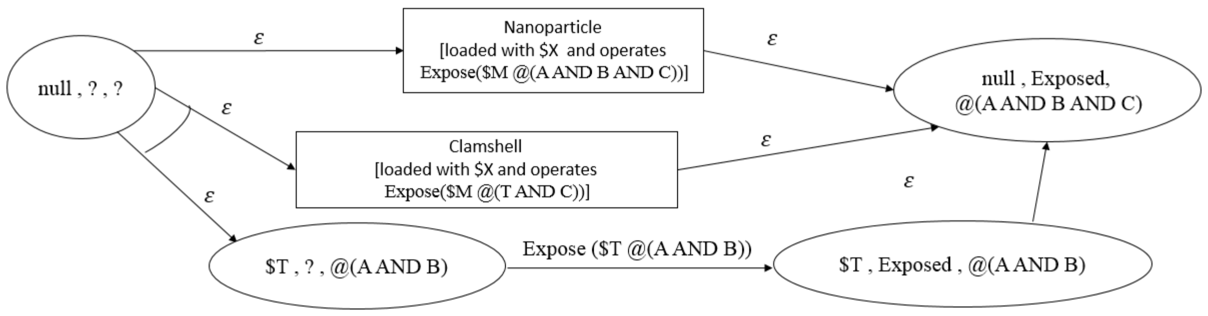


Figure 5: DECOMPOSE operation on the results of previous steps. The arc marks AND transitions in the AND/OR graph.

reasons:
a) R does not contain all the needed robots, thus is missing either the main robot or assistant robot(s). Based on the fact that no sub-process removes the main robot (thus it cannot be missing), and DECOMPOSE adds all assistant-robots and then loops thus they cannot be removed later, it follows that no robot can be left out of ConstructTree.
b) R contains unnecessary robots, and specifically harmful ones. However, we show that it is impossible to offer unnecessary robots, since only ones that implement a transition (that are necessary for the recipe) are offered by the algorithm.
c) The interaction between the different robots is problematic due to dependencies they share. However, the dependency-check goes over all possible combinations of robots that may depend on one another (directly or indirectly), thus this case is also impossible. □

Theorem 3. *Algorithm* ConstructTree *halts, and is complete and sound.*

Proof. Completeness and soundness of the algorithm are proven by Lemmas 1 and 2. In order to show that ConstructTree halts, note that the number of the FSMs in the FSM list which Algorithm ConstructTree works on is finite. Due to the fact that in each iteration at least one robot is created, the recursion's depth of construct-tree algorithm for each FSM is limited by the number of the robots in the longest recipe (which is bounded by the number of robots, which is finite). Therefore, the algorithm will necessarily halt. □

5 Experimental Results

The Bilbo compiler 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.

5.1 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 2). 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.

The Bilbo compiler's output is given in Figure 6 (we omit here the preparation protocols). 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.

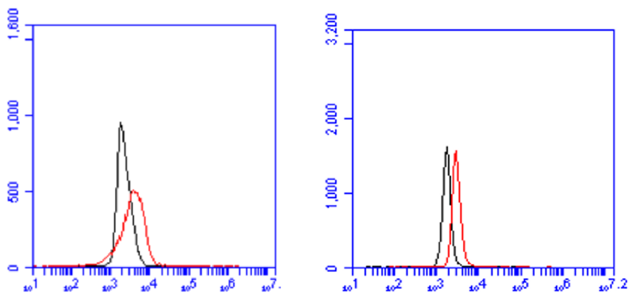
```
1: Clamshell: <$T, true, @A, @B>
2: Clamshell: <$., true, @T, @C>
```

Figure 6: 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 7(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.

5.2 AND/OR, Multiple Options

In a second experiment we demonstrate the compiler's capability to generate different implementation alter-



(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 7: Flow cytometry histograms of both experiments.

natives for the same program. To do this, we extended the nanobot library to also include the gold nanoparticles [Tonga *et al.*, 2015] previously described. We forced the compiler to generate all alternatives, by disabling the path selection stage. Thus all implementation alternatives are produced.

We compiled the Athelas program shown in Figure 8. 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 8: Rule used in the second experiment.

The compiler offers three different implementations. The first uses a nano-particle nanobot (marked `au` in Figure 9). The second, uses a combination of clamshell and nano-particles (Figure 10), and the third combines three types of clamshell nanobots (Figure 11).

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 9). 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.

As a second alternative implementation, the compiler

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

Figure 9: nanoparticle implementation for rule in Fig. 8.

proposes a swarm composed of a single clamshell and a particle nanobot (Figure 10). 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.

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

Figure 10: Gold particle and clamshell nanobot swarm for rule in Fig. 8.

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 9). The `(D AND E)` implementation uses a single clamshell, reacting to presence of both D and E (line 1).

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

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

Figure 11: Clamshell-only implementation of rule in Figure 8.

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 7(a)). Figure 7(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 Conclusions and future work

This paper presents a novel approach to programming nanobots for biomedical applications. Inspired by modern compilation paradigms, we 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. Concerns with safety, we prove the soundness and completeness of the Bilbo back-end, which is at the heart of the compilation process. We 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.

We believe this paper 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.

References

[Amir *et al.*, 2014] 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.

[Bachrach *et al.*, 2010] Jonathan Bachrach, Jacob Beal, and James McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19:825–847, 2010.

[Banerjee *et al.*, 2013] Anusuya Banerjee, Dhiraj Bhatia, Anand Saminathan, Saikat Chakraborty, Shaunak Kar, and Yamuna Krishnan. Controlled release of encapsulated cargo from a DNA icosahedron using a chemical trigger. *Angewandte Chemie International Edition*, 52(27):6854–6857, 2013.

[Cavalcanti *et al.*, 2009] Adriano Cavalcanti, Bijan Shirinzadeh, Toshio Fukuda, and Seiichi Ikeda. Nanorobot for brain aneurysm. *International Journal of Robotics Research*, 28(4):558–570, 2009.

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

[Douglas *et al.*, 2012] 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.

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

[Gupta *et al.*, 1986] 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.

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

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

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

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

[Park *et al.*, 2010a] J. H. Park, G. von Maltzahn, M. J. Xu, V. Fogal, V. R. Kotamraju, E. Ruoslahti, S. N. Bhatia, and M. J. Sailor. Cooperative nanomaterial system to sensitize, target, and treat tumors. *Proceedings of the National Academy of Science, USA*, 107(3):981–986, 2010.

[Park *et al.*, 2010b] 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.

[Pinciroli *et al.*, 2015] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. Available online at http://arxiv.org/abs/1507.05946, 2015.

[Ruoslahti *et al.*, 2010] Eric Ruoslahti, Sangeeta N. Bhatia, and Michael J. Sailor. Targeting of drugs and nanoparticles to tumors. *Journal of Cell Biology*, 188(6):759–768, 2010.

[Tonga *et al.*, 2015] Gulen Yesilbag Tonga, Youngdo Jeong, Bradley Duncan, Tsukasa Mizuhara, Rubul Mout, Riddha Das, Sung Tae Kim, Yi-Cheun Yeh, Bo Yan, Singyuk Hou, et al. Supramolecular regulation of bioorthogonal catalysis in cells using nanoparticle-embedded transition metal catalysts. *Nature chemistry*, 7(7):597–603, 2015.

[von Maltzahn *et al.*, 2011] 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.