Specification of Adleman's Restricted Model Using an Automated Reasoning System: Verification of Lipton's Experiment

C. Graciani Díaz, F.J. Martín Mateos, and Mario J. Pérez Jiménez

Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla, Spain Carmen.Graciani@cs.us.es

Abstract. The aim of this paper is to develop an executable prototype of an unconventional model of computation. Using the PVS verification system (an interactive environment for writing formal specifications and checking formal proofs), we formalize the *restricted model*, based on DNA, due to L. Adleman. Also, we design a formal molecular program in this model that solves **SAT** following Lipton's ideas. We prove using PVS the soundness and completeness of this molecular program. This work is intended to give an approach to the opportunities offered by mechanized analysis of unconventional model of computation in general. This approach opens up new possibilities of verifying molecular experiments before implementing them in a laboratory.

1 Introduction

Using formal notations does not ensure us that specifications will be correct. They still need to be validated by permanent reviews but, on the other hand, they support formal deduction; thus, reviews can be supplemented by mechanically checked analysis.

PVS[1] is a verification system: a specification language tightly integrated with a powerful theorem prover and other tools. We present in this paper a formalization, in the PVS verification system [3], of an abstract model of molecular computation: the *restricted model* due to L. Adleman [2]. This work is motivated by the results obtained using ACL2 in [6], where ACL2 is an automated reasoning system that provides both a programming language in which one can model computer systems and a tool that provides assistance to prove properties of these models.

In a molecular model the data are, in general, *tubes* (abstract structures representing a test tube in a laboratory) over a prefixed alphabet. The elements of these tubes encode a collection of DNA strands associating to each symbol of the alphabet an oligonucleotide, under certain conditions. The restricted model

* This work has been supported by DGES/MEC: Projects TIC2000-1368-C03-02 and PB96-1345

[1] The PVS Specification and Verification System http://pvs.csl.sri.com/

is based on filtering. In such a model, computations have as input an initial tube containing all possible solutions to the problem to be solved (a coding of them). Then, by performing separations, a tube with only the correct solutions of the problem is obtained.

In order to establish the formal verification of a program designed in the restricted model to solve a decision problem, we must prove two basic results:

• Every molecule in the output tube encodes a valid solution to the problem. That is, if the output is **YES** then the problem has a correct solution (*soundness* of the program).

• Each molecule in the input tube that encodes a correct solution to the problem is in the output tube. That is, if there is such a molecule in the input tube then the output is **YES** (*completeness* of the program).

The paper is organized as follows. In section 2, we briefly introduce the PVS verification system. In section 3, we present Adleman's restricted model and describe how this model is formalized in PVS. Section 4 sets up the SAT problem and how we deal with it in PVS. In section 5 we develop the molecular solution due to Lipton and we provide, in a compact way, a description of the formal verification of the program designed, obtained using PVS. The complete developed theories for this paper are available on the web at `http://www.cs.us.es/čgdiaz/investigacion`.

2 PVS

The *Prototype Verification System (PVS)* is a proof checker based on higher–order logic where types have semantics according to Zermelo–Fraenkel set theory with the axiom of choice [7]. In such a logic one can quantify over functions which take functions as arguments and return them as values.

Specifications are organized into *theories*. They can be parameterized with semantic constructs (constant or types). Also they can import other theories. A prelude for certain standard theories is preloaded into the PVS system. As an example we include in figure 1 the PVS theory `epsilons` (it appears in the prelude) which provides a "choice" function that does not have a nonemptiness requirement. Given a predicate over the type `T`, epsilon produces an element satisfying that predicate if one exists, and otherwise produces an arbitrary element of that type. Note that the type parameter is given as nonempty, which means that there is an nonempty `ASSUMPTION` automatically generated for this theory.

Before a theory may be used, it is typechecked. The PVS typechecker analyzes the theory for semantic consistency and adds semantic information to the internal representation built by the parser. Since this is not a decidable process, the checks which cannot be resolved automatically are presented to the user as assertions called type–correctness conditions.

The PVS prover is goal–oriented. Goals are sequents consisting of antecedents and consequents, e.g. $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$. The conjunction of the antecedents should imply the disjunction of consequents, i.e. $A_1 \wedge \cdots \wedge A_n \rightarrow B_1 \vee \cdots \vee B_m$. The proof starts with a goal of the form $\vdash A$, where A is the

```
epsilons [T: NONEMPTY_TYPE]: THEORY
 BEGIN
  p: VAR pred[T]
  x: VAR T
  epsilon(p): T
  epsilon_ax: AXIOM (EXISTS x: p(x)) => p(epsilon(p))
 END epsilons
```

Fig. 1. A PVS Theory

theorem to be proved. The user may type proof commands which either prove the current goal, or result in one or more new goals to prove. In this manner a proof tree is constructed. The original goal is proved when all leaves of the proof tree are recognized as true propositions. Basic proof commands are also combined into strategies.

3 Adleman's Restricted Model in PVS

In this section Adleman's restricted model is described and some considerations about the PVS version of the formalization are given.

Definition 1. *An **aggregate** over an alphabet Σ is a finite multiset of symbols from Σ. A **tube** is a multiset of aggregates over Σ.*

```
AGGREGATES: TYPE = MULTISETS[SIGMA]
TUBES: TYPE = MULTISETS[AGGREGATES]
```

The following are the basic molecular instructions in the Adleman's restricted model.

• `separate`(T, s): Given a tube T and a symbol $s \in \Sigma$ it produces two tubes: $+(T, s)$ (resp. $-(T, s)$) is the tube of all the aggregates of T containing (resp. not containing) the symbol s. As the `separate` operation produces two values, we use two functions to implement it in PVS.

```
sep_p(TT, symb): TUBES =
    LAMBDA (gamma): IF TT(gamma) = 0 OR NOT ms_in(symb, gamma)
                      THEN 0
                    ELSE TT(gamma) ENDIF
sep_n(TT, symb): TUBES =
    LAMBDA (gamma): IF TT(gamma) = 0 OR ms_in(symb, gamma)
                      THEN 0
                    ELSE TT(gamma) ENDIF
```

• `merge`(T_1, T_2): Given tubes T_1 and T_2 it produces their union $T_1 \cup T_2$, considered as multisets.

```
merge(T1, T2): TUBES = ms_union(T1, T2)
```

• `detect`(T): Given a tube T, says **YES** if T contains at least one aggregate, and says **NO** otherwise.

```
DECISION: TYPE = {YES, NO}
detect(TT): DECISION =
  IF ms_empty?(TT) THEN NO ELSE YES ENDIF
```

4 The Satisfiability Problem

SAT Problem: *Given a propositional formula in conjunctive normal form, to determine if there is a truth assignment, whose domain contains all the propositional variables occurring in the formula, such that this assignment satisfies the formula.*

Let φ be a formula in conjunctive normal form where $\varphi = c_1 \wedge \cdots \wedge c_p$ and each clause is a disjunction of literals, $c_i = l_{i,1} \vee \cdots \vee l_{i,r_i}$ for $1 \leq i \leq p$. A literal is a variable or the negation of a variable. Let $Var(\varphi) = \{x_1, \ldots, x_n\}$ the propositional variables occurring in φ.

Conjunctions of clauses and disjunctions of literals will be, in PVS, finite sequences of clauses and literals, respectively. To describe literals in PVS we represent them as a `(marker, propositional variable)` ordered pair. There will be two markers: `positive` and `negative`.

```
MARKERS: TYPE = {positive, negative}
PLIT: TYPE = [MARKERS, PVAR]
PCL: TYPE = finseq[PLIT]
PFORM_fnc: TYPE = finseq[PCL]
```

For a nonempty sequence $S = \{e_0, ..., e_n\}$ we denote $S = S' \circ \{e\}$ where $S' = \{e_0, ..., e_{n-1}\}$ and $e = e_n$. We denote, $e \in S \equiv \exists k(e = e_k)$.

The finite sequence of propositional variables occurring in φ is constructed recursively, in a natural way, over the finite sequence of variables that occurs in the clauses of φ. These sequences are constructed over the variables that appear in the literals occurring in each clause. We define in PVS the functions `PVar` to implement these constructions.

Definition 2.

• *If* $l = (mk, x)$ *then* $Var(l) = x$.

```
PVar((S, PV)): PVAR = PV
```

• *If* $c = \{l_1, \ldots, l_r\}$ *then* $Var(c) = \{Var(l_1), \ldots, Var(l_r)\}$.

```
PVar(PC): finseq[PVAR] = LET L = PC`length IN
      (# length := L,
         seq := LAMBDA (i: below[L]): PVar(PC`seq(i)) #)
```

- $Var(\varphi) = \begin{cases} \{\} & \text{if } \varphi = \{\} \\ Var(c) \circ Var(\varphi') & \text{if } \varphi = \varphi' \circ \{c\} \end{cases}$

```
PVar(PF): RECURSIVE finseq[PVAR] =
  IF PF`length = 0 THEN empty_seq
  ELSE fs_concat(PVar(fs_last(PF)), PVar(fs_red(PF)))
  ENDIF MEASURE PF`length
```

We define in PVS a truth assignment or valuation as an application between propositional variables and truth values, 0 or 1. The opposite value, $\bar{v}$, of a truth value, v, is defined as usual.

```
TRUTH_VALUES: TYPE = {zero, one}
VV: VAR TRUTH_VALUES
opp_value(VV): TRUTH_VALUES =
    IF VV = zero THEN one ELSE zero ENDIF

VALUATIONS: TYPE = [PVAR -> TRUTH_VALUES]
```

For a given valuation the truth value of a literal agrees with the truth value of its variable if the marker is `positive`; otherwise the truth value of the literal is the opposite one. The truth value of a clause in relation to a valuation can be computed recursively from the truth values of its literals. It will be 1 if there is, at least, one literal whose truth value is 1 and 0 otherwise. Similarly, the truth value for a given propositional formula can be computed recursively from the truth values of the clauses occurring in it. It will be 1 if all of them are 1; 0 otherwise.

We define the functions `PVal` in PVS to compute those values.

Definition 3.

- $\pi(l) = \begin{cases} \pi(Var(l)) & \text{if } l = (positive, x) \\ \neg\pi(Var(l)) & otherwise. \end{cases}$

```
PVal(PI, PL): TRUTH_VALUES =
    IF plit_positive?(PL) THEN PI(PVar(PL))
    ELSE opp_value(PI(PVar(PL))) ENDIF
```

- $\pi(c) = \begin{cases} 0 & \text{if } c = \{\} \\ \text{if } c = c' \circ \{l\} & \begin{cases} 1 & \text{if } \pi(l) = 1 \\ \pi(c') & otherwise. \end{cases} \end{cases}$

```
PVal(PI, PC): RECURSIVE TRUTH_VALUES =
  IF length(PC) = 0 THEN zero
  ELSIF PVal(PI, fs_last(PC)) = one THEN one
  ELSE PVal(PI, fs_red(PC)) ENDIF
  MEASURE length(PC)
```

- $\pi(\varphi) = \begin{cases} 1 & \text{if } \varphi = \{\} \\ \text{if } \varphi = \varphi' \circ \{c\} & \begin{cases} 0 & \text{if } \pi(c) = 0 \\ \pi(\varphi') & \text{otherwise.} \end{cases} \end{cases}$

```
PVal(PI, PF): RECURSIVE TRUTH_VALUES =
  IF length(PF) = 0 THEN one
  ELSIF PVal(PI, fs_last(PF)) = zero THEN zero
  ELSE PVal(PI, fs_red(PF))
  ENDIF MEASURE length(PF)
```

Nevertheless, to establish the value of a formula usually it is considered that the application domain is restricted to the set of variables that occur in the formula. We consider, in this sense the following approximation.

Definition 4. *A valuation σ, over a finite sequence S, is an application $\sigma : Ran(S) \to \{0, 1\}$, where $Ran(S)$ is the range of S.*

```
RES_VALUATIONS: TYPE =
  [# domain: finseq[PVAR],
     PVal: [{PV: PVAR | fs_in(PV, domain)} -> TRUTH_VALUES] #]
```

The truth value of a variable for a valuation σ over S is $\sigma(x)$ if $x \in S$, 0 otherwise. The definition of the truth value of a literal, clause or formula for a valuation σ over S is done in the same way as before.

5 Lipton's Solution to SAT Problem

In this regard we follow [5] closely. Given a propositional formula φ, we consider the following directed graph $G_\varphi = (V_\varphi, E_\varphi)$; where $V_\varphi = \{a_i, x_i^j, a_{n+1} |\ 1 \leq i \leq n \wedge (j = 0 \vee j = 1)\}$ and $E_\varphi = \{(a_i, x_i^j), (x_i^j, a_{i+1}) |\ 1 \leq i \leq n \wedge (j = 0 \vee j = 1)\}$.

There is a natural bijection between the set of simple paths starting at a_1 and ending at a_{n+1}, and the set of valuations defined over $Var(\varphi)$. Let $\gamma = a_1 x_1^{j_1} \ldots x_n^{j_n} a_{n+1}$ be such a path. The associated valuation σ_γ is characterized by the following relation: $\sigma_\gamma(x_i) = j_i$ for $1 \leq i \leq n$.

5.1 Design of the Molecular Program

Let us consider $\Sigma = \{a_i, x_i^j |\ i \in \mathbb{N} \wedge x \in \texttt{PVAR} \wedge (j = 0 \vee j = 1)\}$. Given a formula φ with $Var(\varphi) = \{x_1, \ldots, x_n\}$, the initial tube $T_\varphi = \{\!\{\{\!\{a_1 x_1^{j_1} \ldots x_n^{j_n} a_{n+1}\}\!\} |\ \forall i (1 \leq i \leq n \to j_i = 0 \vee j_i = 1)\}\!\}$ (we use the double braces to denote a multiset) can be generated in a laboratory. It is formed in the same way that the test tube of all paths to find the Hamiltonian Path is elaborated in [1].

To describe the symbols x^j for $j \in \{0, 1\}$ in PVS we represent them as a `(propositional variable, truth value)` ordered pair. Let us consider $\Sigma = \texttt{PVAR} \times \{0, 1\}$ the alphabet for the computation model. Given a propositional formula, we represent the aggregates for the initial tube by omitting the symbols a_i (they were used in the original experiment as auxiliary vertices of the directed graph considered to construct T_φ).

Initial tube: To construct the initial tube for a given finite sequence of propositional variables, in PVS, we define the following functions.

Definition 5.

$$\texttt{ins}(\texttt{x},\texttt{v},\gamma) = \begin{cases} \gamma & \textit{if } \exists v'((x,v') \in \gamma) \\ \gamma \cup \{(x,v)\} & \textit{otherwise.} \end{cases}$$

```
ins(PV, VV, gamma): AGGREGATES =
    IF EXISTS VV_p: ms_in((PV, VV_p), gamma) THEN gamma
    ELSE ms_incl((PV, VV), gamma) ENDIF
```

$$T_S = \begin{cases} \{\!\{\}\!\} & \textit{if } S = \{\} \\ \{\!\{\{\!\{(x,1)\}\!\}, \{\!\{(x,0)\}\!\}\}\!\} & \textit{if } S = \{x\} \\ \{\!\{\texttt{ins}(x,1,\gamma'), \texttt{ins}(x,0,\gamma') |\ \gamma' \in T_{S'}\}\!\} & \textit{otherwise } (S = S' \circ \{x\}). \end{cases}$$

```
make_tube(S): RECURSIVE TUBES =
  IF S`length = 0 THEN ms_empty
  ELSE LET SR = fs_red(S), SU = fs_last(S) IN
      IF SR`length = 0
        THEN LAMBDA (gamma):
             IF gamma = ms_incl((SU, one), ms_empty) OR
                 gamma = ms_incl((SU, zero), ms_empty)
               THEN 1 ELSE 0 ENDIF
      ELSE LET TT = make_tube(SR) IN LAMBDA (gamma):
         IF EXISTS gamma_p:
              ms_in(gamma_p, TT) AND
               (gamma = ins(SU, one, gamma_p) OR
                 gamma = ins(SU, zero, gamma_p))
           THEN 1 ELSE 0 ENDIF
      ENDIF
  ENDIF MEASURE S`length
initial_tube(PF): TUBES = make_tube(PVar(PF))
```

The specification of $\gamma|x$ states that x appears in γ associated with only one truth value.

Definition 6. $\gamma|x \equiv \exists v((x,v) \in \gamma \wedge (x,\bar{v}) \notin \gamma)$.

```
  pv?_aggregate(gamma, PV): bool =
      EXISTS VV: ms_in((PV, VV), gamma) AND
         NOT ms_in((PV, opp_value(VV)), gamma)
```

Lemma 1. $\gamma \in T_S \wedge x \in S \rightarrow \gamma|x$

```
 initial_tube_carac: LEMMA
  ms_in(gamma, make_tube(S)) AND fs_in(PV, S)
   IMPLIES pv?_aggregate(gamma, PV)
```

Definition 7. *Given $S \subseteq S'$ and $\gamma \in T_{S'}$ we define the associated valuation as $\sigma_\gamma^S : Ran(S) \to \{0,1\}$ where $\sigma_\gamma^S(x) = v$ for a truth value v such that $(x,v) \in \gamma$.*

```
asoc(S)(gamma: {gamma_p| EXISTS SL: fs_subseq(S, SL) AND
                          ms_in(gamma_p, make_tube(SL))}):
  RES_VALUATIONS =
      (# domain := S,
         PVal := LAMBDA (PV: {PV_p| fs_in(PV_p, S)}):
                   epsilon! VV: ms_in((PV, VV), gamma) #)
```

The following properties characterize the above definition.

Lemma 2. $\gamma \in T_S \wedge (x,v) \in \gamma \to \sigma_\gamma^S(x) = v$

```
asoc_carac: LEMMA
  ms_in(gamma, make_tube(S)) AND ms_in((PV, VV), gamma)
     IMPLIES asoc(S)(gamma)`PVal(PV) = VV
```

Lemma 3. *For each valuation $\sigma : Ran(S) \to \{0,1\}$ such that S is nonempty there exists an aggregate $\gamma \in T_S$ such that $\sigma_\gamma^S = \sigma$.*

```
make_tube_sound: LEMMA
  (sigma`domain = S AND S`length > 0) IMPLIES
     EXISTS gamma: ms_in(gamma, make_tube(S)) AND
       asoc(S)(gamma) = sigma
```

Following [4], for each literal $l_{i,j}$ occurring in φ we will denote $l_{i,j}^v = x_m^v$ if $l_{i,j} = x_m$; otherwise $l_{i,j}^v = x_m^{\bar{v}}$. That is, if an aggregate in the initial tube contains the $l_{i,j}^v$ symbol then, for the associated valuation of that aggregate, the literal $l_{i,j}$ has assigned the truth value v.

```
l_symb(PL, VV): [PVAR, TRUTH_VALUES] =
    IF plit_positive?(PL) THEN (PVar(PL), VV)
    ELSE (PVar(PL), opp_value(VV)) ENDIF
```

The operation over the initial tube is done as follows: Let T_1 be the tube whose aggregates encode a valuation that assigns the truth value 1 to the clause c_1. We construct this tube as follows:

Consider the tube that consists of the aggregates in T_0 whose associated valuation assign 1 to the literal $l_{1,1}$. Then, for those aggregates whose associated valuation assigns 0 to $l_{1,1}$, extract the ones that assign 1 to $l_{1,2}$. For the remainder aggregates (encoding a truth assignment that assigns 0 to $l_{1,1} \vee l_{1,2}$) extract those that assign 1 to $l_{1,3}$, and so on.

From T_1, in the same way as before, construct T_2, the tube whose aggregates assign 1 to the clause c_2, and so on.

According to this idea, a molecular program in the restricted model solving SAT is the following one.

```
Procedure sat_lipton(φ)
input: T ← T_φ
      for i = 1 to p do
          T' ← ∅
          for j = 1 to r_i do
              T'' ← +(T, l^1_{i,j})
              T ← −(T, l^1_{i,j})
              T' ← merge(T', T'')
          end for
          T ← T'
      end for
      detect(T)
```

To implement the program `sat_lipton` in PVS, we will define two recursive functions, `inner_l` and `main_l`, one for each `loop`.

Definition 8.

$$\texttt{inner\_l}(c, T, T') = \begin{cases} T' & \text{if } c = \{\} \\ \texttt{inner\_l}(c', -(T, l^1), T' \cup +(T, l^1)) & \text{if } c = c' \circ \{l\} \end{cases}$$

$$\texttt{main\_l}(\varphi, T) = \begin{cases} T & \text{if } \varphi = \{\} \\ \texttt{main\_l}(\varphi', \texttt{inner\_l}(c, T, \{\!\{\}\!\})) & \text{if } \varphi = \varphi' \circ \{c\} \end{cases}$$

```
inner_l(PC, TT, TRes): RECURSIVE TUBES =
  IF PC`length = 0 THEN TRes
  ELSE inner_l(fs_red(PC),
         sep_n(TT, l_symb(fs_last(PC), one)),
         merge(TRes, sep_p(TT, l_symb(fs_last(PC), one))))
  ENDIF MEASURE PC`length
inner_l(PC, TT): TUBES = inner_l(PC, TT, ms_empty)

main_l(PF, TT): RECURSIVE TUBES =
  IF PF`length = 0 THEN TT
  ELSE main_l(fs_red(PF), inner_l(fs_last(PF), TT))
  ENDIF MEASURE PF`length
main_l(PF): TUBES = main_l(PF, initial_tube(PF))

sat_lipton(PF): DECISION = detect(main_l(PF))
```

5.2 Completeness and Soundness in PVS

Next we present results needed to prove, in PVS, the completeness and soundness of the program.

Theorem 1 (completeness). *Given a formula φ such that $Var(\varphi)$ is nonempty we have that* $\exists\pi(\pi(\varphi) = 1) \rightarrow \texttt{sat\_lipton}(\varphi) = \texttt{YES}$

```
completeness: THEOREM
  PVar(PF)`length > 0 AND (EXISTS PI: PVal(PI, PF) = one))
      IMPLIES sat_lipton(PF) = YES
```

To prove this theorem it is sufficient to prove the next one.

Theorem 2 (sat_lipton_compl_gen). $Var(\varphi) \subseteq S \wedge T \subseteq T_S \wedge \gamma \in T \wedge \sigma_\gamma^S(\varphi) = 1 \rightarrow \gamma \in \texttt{main\_l}(\varphi, T)$.

```
sat_lipton_compl_gen: THEOREM
  fs_subseq(PVar(PF), S) AND ms_subset(TT, make_tube(S))
    AND ms_in(gamma, TT) AND PVal(asoc(S)(gamma), PF) = one
      IMPLIES ms_in(gamma, main_l(PF, TT))
```

A similar result for clauses is needed.

Theorem 3.
$Var(c) \subseteq S \wedge T \subseteq T_S \wedge \gamma \in T \wedge \sigma_\gamma^S(c) = 1 \rightarrow \gamma \in \texttt{inner\_l}(c, T, T')$

```
sat_lipton_compl_pcl: LEMMA
  fs_subseq(PVar(PC), S) AND ms_subset(TT, make_tube(S)) AND
  ms_in(gamma, TT) AND PVal(asoc(S)(gamma), PC) = one
    IMPLIES ms_in(gamma, inner_l(PC, TT, TRes))
```

Theorem 4 (soundness). $\texttt{sat\_lipton}(\varphi) = \texttt{YES} \rightarrow \exists \pi(\pi(\varphi) = 1)$.

```
soundness: THEOREM
 sat_lipton(PF) = YES IMPLIES EXISTS PI: PVal(PI, PF) = one
```

To prove this theorem it is sufficient to prove the next one.

Theorem 5. $Var(\varphi) \subseteq S \wedge T \subseteq T_S \wedge \gamma \in \texttt{main\_l}(\varphi, T) \rightarrow \sigma_\gamma^S(\varphi) = 1$.

```
sound_pform: LEMMA
  fs_subseq(PVar(PF), S) AND ms_subset(TT, make_tube(S)) AND
  ms_in(gamma, main_l(PF, TT))
     IMPLIES PVal(asoc(S)(gamma), PF) = one
```

Again a similar result for clauses is also needed.

Theorem 6. $Var(c) \subseteq S \wedge T \subseteq T_S \wedge \gamma \notin T' \wedge \gamma \in \texttt{inner\_l}(\varphi, T, T') \rightarrow \sigma_\gamma^S(c) = 1$.

```
sound_pcl: LEMMA
  fs_subseq(PVar(PC), S) AND ms_subset(TT, make_tube(S))
   AND (NOT ms_in(gamma, TRes)) AND
   ms_in(gamma, inner_l(PC, TT, TRes)))
     IMPLIES PVal(asoc(S)(gamma), PC) = one
```

6 Conclusions

A great part of our work within molecular computing is related to the formalization of the different models that have appeared. Also we have designed and verified programs within these models, to solve classical intractable problems. During this effort we have drawn the conclusion that the next step should be the implementation of these works in an automated reasoning system. Moreover, this approach gives us the possibility of obtaining executable models.

As stated before, the present formalization has been motivated by the results obtained in [6] using ACL2. One advantage of PVS is that it has sets and functions as types and that it is based on a higher–order logic so we gain expressiveness. We also have list data types and tools to deal with recursive definitions and proofs by induction over them.

In this paper a formalization in PVS of the restricted model has been presented. Also a verification using PVS of the Lipton's experiment solving the SAT problem has been obtained. We think that using automated reasoning systems, molecular experiments can be verified before realizing these experiments in a laboratory.

We plan to implement in PVS a more natural specification of this and other molecular models. We also want to establish the formal verification, in the mentioned terms, of the programs designed in those models. We believe and expect that this process would be especially useful to develop a fully or semi automatic strategy to prove the completeness and soundness of programs in unconventional models of computation.

References

1. Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.
2. Leonard M. Adleman. On constructing a molecular computer. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. DNA Based Computers.
3. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/papers/wift-tutorial/`.
4. M.J. Pérez Jiménez, F. Sancho Caparrini, M.C. Graciani Díaz, and A. Romero Jiménez. Soluciones moleculares del problema SAT de la Lógica Proposicional. In *Lógica, Lenguaje e Información, JOLL'2000*, pages 243–252. Ed. Kronos, 2000.
5. Richard J. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, April 1995.
6. F.J. Martín-Mateos, J.A. Alonso, M.J. Pérez-Jiménez, and F. Sancho-Caparrini. Molecular computation models in ACL2: a simulation of Lipton's experiment solving SAT. In *Third International Workshop on the ACL2 Theorem Prover and Its Applications*, Grenoble, 2002.
7. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.