Using Automated Reasoning Systems on Molecular Computing

Carmen Graciani Díaz and Mario J. Pérez-Jiménez

Research Group on Natural Computing,
Dpto. Ciencias de la Computación e Inteligencia Artificial,
Universidad de Sevilla (Spain)
{cgdiaz, marper}@us.es

Abstract. This paper is focused on the interplay between automated reasoning systems (as theoretical and formal devices to study the correctness of a program) and DNA computing (as practical devices to handle DNA strands to solve classical hard problems with laboratory techniques). To illustrate this work we have proven in the PVS proof checker, the correctness of a program, in a sticker based model for DNA computation, solving the pairwise disjoint families problem. Also we introduce the formalization of the Floyd–Hoare logic for imperative programs.

1 Introduction

One of the most active areas of research in Computer Science is the study and use of formal methods (applications of primarily discrete mathematics to software engineering problems). Its widely development and the complexity of interesting problems have given rise to automated reasoning. In this area, one of the main problems is the correctness [2]: developing specifications and proofs that ensures a program meets its specification. There is a previous work of formalization: expressing all definitions, theorems and proofs in a formal language without semantic ambiguity. This approximation has especial relevance in new computing paradigms such as the DNA based molecular computing. In many molecular models the data are *tubes* over an alphabet whose content encodes a collection of DNA strands. The operations considered are abstraction of different laboratory techniques to manipulate DNA strands.

This paper is organized as follows. It begins with a short presentation of the *Prototype Verification System (PVS)* and the sticker model. Then, how this model can be formalized in PVS, is briefly described. Section 4 introduces imperative programs and gives an overview of how we deal with them in PVS. Finally, as an example, a molecular solution of the pairwise disjoint families problem and a description of its formal verification obtained with PVS, is presented. The set of developed theories in PVS for this paper are available on the web at `http://www.cs.us.es/~cgdiaz/investigacion`.

2 The Prototype Verification System

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 [8]. In such a logic we 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 system. As an example we include in figure 1 the PVS theory `suc_finitas_def` which provides an alternative definition (to the type `finseq` given in the prelude) for sequences of a given length `n` for elements of a given type `V`.

```
suc_finitas_def[V: TYPE, n: nat]: THEORY
 BEGIN
% Finite sequence: S = {s_k}_{k < n+1}
  SUC_FINITAS: TYPE = [below[n] -> V]
  SF: TYPE = SUC_FINITAS
 END suc_finitas_def
```

Fig. 1. A PVS Theory

Before a theory may be used, it must be 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 an undecidable 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 $\vdash B$, where B is the 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 can also be combined into strategies.

3 The Sticker Model: A Description Through PVS

The sticker model used in this paper was introduced by S. Roweis et al. [9] (this model is completely different from the *sticker systems* introduced by L. Kari et al in [6]). It is an abstract model of DNA based molecular computing with random access memory in the following sense: some operations could modify the structure of the DNA molecules and so the information codified by them changes during the execution.

In this model, a *memory strand* (a single stranded DNA molecule) N bases in length subdivided into `k` non–overlapping *regions* each M bases long is considered to represent a string of `k` bits. Each region is identified with exactly one bit position. Also, `k` different *sticker strands* (single stranded DNA molecule) each of them M bases long and complementary with one and only one of the `k` memory regions are considered. If a sticker is annealed to its matching region then the corresponding bit is *on*. Otherwise, it is *off*. A memory strand together with its associated stickers, if any, is called a *memory complex* and represent one bit string. In this sense we consider memory complexes as finite sequence of bits in PVS:

```
BITS: TYPE = {on, off}
MEMORY_COMPLEX: TYPE = finseq[BITS]
```

Associated with this definition we consider the application σ from $\mathbb{N}$ into {`on, off`} defined as follows (where σ_i is the i-th element of the bit sequent σ):

$$\sigma(i) = \begin{cases} \sigma_i & \text{if } i < \texttt{k} \\ \texttt{off} & \text{otherwise} \end{cases}$$

```
appl(sigma: MEMORY_COMPLEX, i: nat): BITS =
  IF i < sigma`length THEN sigma`seq(i) ELSE off ENDIF
```

Within sticker model a *tube* is a collection of memory complexes representing a multiset of bit strings. All memory strands (underlying each complex) in a tube are identical and each one has stickers annealed only at the required bit positions.

In PVS we consider a general concept: a multiset of memory complexes:

```
GEN_TUBE: TYPE = MULTISETS[MEMORY_COMPLEX]
```

Then we restrict this definition to consider tubes containing only memory complexes of a given length, namely `k`.

```
MTUBE: TYPE =
  {T: GEN_TUBE | FORALL (sigma: MEMORY_COMPLEX):
                      ms_in(sigma, T) IMPLIES sigma`length = k}
```

The following are the molecular operations on tubes used in the sticker model and the corresponding implementation in PVS.

– To *combine* two tubes producing a new one containing all the memory complexes from both tubes.

```
combine(T1, T2: GEN_TUBE): GEN_TUBE =
    LAMBDA (sigma: MEMORY_COMPLEX): T1(sigma) + T2(sigma)
```

– To *separate* the content of a tube into two new tubes, one containing all the memory complexes with a particular sticker annealed (a particular bit *on*) and the other all those with that region free (that bit *off*).

```
separate(T: GEN_TUBE, oi: nat): [GEN_TUBE, GEN_TUBE] =
    (LAMBDA (sigma: MEMORY_COMPLEX):
       IF appl(sigma, oi) = on THEN T(sigma) ELSE 0 ENDIF,
     LAMBDA (sigma: MEMORY_COMPLEX):
       IF appl(sigma, oi) = off THEN T(sigma) ELSE 0 ENDIF)
```

– To *turn on* (*set*) a particular region annealing the appropriate sticker on every complex in a tube (turning the corresponding bit to *on*).
– To *turn off* (*clear*) a particular region removing the appropriate sticker, if any, on every complex in a tube (turning the corresponding bit to *off*).

Previously to the implementation of these operations we define the concept of modifying in a memory complex, σ, a particular bit, i, to $\mathtt{b} \in \{\mathtt{on},\ \mathtt{off}\}$.

$$\sigma_i^{\mathtt{b}} = \begin{cases} \{\sigma_0, \ldots, \sigma_{i-1}, \mathtt{b}, \sigma_{i+1}, \ldots, \sigma_{\mathtt{k}-1}\} & \text{if } i < \mathtt{k} \\ \sigma & \text{otherwise} \end{cases}$$

```
turn(sigma: MEMORY_COMPLEX, i: nat, b: BITS): MEMORY_COMPLEX =
  IF i < sigma‘length
    THEN sigma WITH [(seq) := sigma‘seq WITH [(i) := b]]
  ELSE sigma ENDIF
```

From this we consider a general operation that changes, in all memory complexes present in a tube, T, a particular bit, i, to $\mathtt{b} \in \{\mathtt{on},\ \mathtt{off}\}$.

$$\mathtt{Change(T,\ } i\mathtt{,\ b)} = \{\!\!\{\ \sigma_i^{\mathtt{b}} \mid\ \sigma \in \mathtt{T}\ \}\!\!\}$$

```
change(T: GEN_TUBE, i: nat, b: BITS): GEN_TUBE =
  LAMBDA (sigma: MEMORY_COMPLEX):
    IF i < sigma‘length AND sigma‘seq(i) = b
      THEN T(turn(sigma, i, off)) + T(turn(sigma, i, on))
    ELSIF i >= sigma‘length THEN T(sigma) ELSE 0 ENDIF
```

The implementation of the turn operations (set and clear) are as follows:

$$\mathtt{Set(T,\ i)} = \{\!\!\{\ \sigma_i^{\mathtt{on}} \mid\ \sigma \in \mathtt{T}\ \}\!\!\} \qquad \mathtt{Clear(T,\ i)} = \{\!\!\{\ \sigma_i^{\mathtt{off}} \mid\ \sigma \in \mathtt{T}\ \}\!\!\}$$

```
set(T: GEN_TUBE, i: nat): GEN_TUBE = change(T, i, on)

clear(T: GEN_TUBE, i: nat): GEN_TUBE = change(T, i, off)
```

Also a *read* operation is considered. This operation determines if a tube is empty and otherwise selects a complex from the tube and produce the associated string of bits. To implement it we consider the special memory complex of length 0 as the answer when there is no elements in the tube.

```
read(T: GEN_TUBE): MEMORY_COMPLEX =
  IF EXISTS (gamma: MEMORY_COMPLEX): ms_in(gamma, T)
    THEN choose({sigma: MEMORY_COMPLEX | ms_in(sigma, T)})
  ELSE empty_seq ENDIF
```

Usually we express the use of those operations as assignments. For example,

$$\texttt{T} \longleftarrow \texttt{Combine(}\texttt{T}_1\texttt{, }\texttt{T}_2\texttt{)}$$

The interpretation of a program in the sticker model as a sequence of such operations has taken us to consider them as imperative programs.

4 Imperative Programs

Following [3] we consider an *imperative program* as a sequence, $\texttt{I}_1$ `@@` $\texttt{I}_2$ `@@` ...`@@` $\texttt{I}_k$, of states transformers[1]. When such a program is executed on an initial state the first transformer is applied to it, the second is applied to the state obtained by the previous one and so on. A general work that shows how to deal in PVS with nontermination and nondeterministic state transformers can be found in [11].

A *state* is considered as a finite sequence of data in a given domain (we introduced the possibility of take a tuple of sequences over different domains to deal with elements of different nature). To access to the information stored in a state we have variables. Each variable is associated with a natural number in a one–to–one manner. The n–variable over a given state takes the value of the n–th element.

In general, a *term* is any function, `t`, that given a state, $\texttt{s} \in \texttt{S}$, produces an element over a given domain, `D`. Operations between elements of given domains are lifted to operations between terms using the function `l` we describe with an example. Suppose we have a binary operation $\texttt{op: D}_1 \times \texttt{D}_2 \rightarrow \texttt{R}$. With `l` we obtain a binary operation `l(op)`, that given two terms $\texttt{t}_1\texttt{: S} \rightarrow \texttt{D}_1$ and $\texttt{t}_2\texttt{: S} \rightarrow \texttt{D}_2$ produces the term $\texttt{l(op)(t}_1\texttt{, t}_2\texttt{): S} \rightarrow \texttt{R}$ where

$$\texttt{l(op)(t}_1\texttt{, t}_2\texttt{)(s) = op(t}_1\texttt{(s), t}_2\texttt{(s))}$$

This function `l` is generalized to consider constants. Given $\texttt{c} \in \texttt{D}$, we obtain the term $\texttt{l(c): S} \rightarrow \texttt{D}$ such that `l(c)(s) = c`.

In [5], Hoare introduced the $\{\varphi\}$ `P` $\{\psi\}$ notation to describe the behaviour of a program `P`. Those expressions are called *specifications of partial correctness* and have the following meaning: If φ and ψ are some conditions over states the specification is true if whenever the program `P` is executed over a state verifying φ and it halts, then it produces a state verifying ψ. As the considered notion of program only consider total functions those expressions are, in fact, *specifications of total correctness.*

[1] In order to save space, we do not include in this section the corresponding PVS implementations, see [3] and [4] for more details.

To construct a formal proof of a specification of partial correctness we use the Floyd–Hoare logic, a set of axioms and inference rules. Next we introduce the ones used to construct the correctness proof in the following section.

– The *consequence rule*:

$$\frac{\varphi \to \varphi',\ \{\varphi\texttt{'}\}\ \texttt{S}\ \{\psi\texttt{'}\},\ \psi' \to \psi}{\{\varphi\}\ \texttt{S}\ \{\psi\}}$$

– The *assignment instruction*, `X ⟵ t` (we denote ⟵ by `<<` in PVS) is a program that over a state `s` produces the state `s[t(s)/X]`, resulting from `s` after the substitution of the associated value for the variable `X` by `t(s)`. The *assignment axiom* is

$$\{\varphi\texttt{[t/X]}\}\ \texttt{X} \longleftarrow \texttt{t}\ \{\varphi\}$$

where φ`[t/X](s) = `φ`(s[t(s)/X])`.

– The *compose rule*:

$$\frac{\{\varphi\}\ \texttt{S}_1\ \{\psi\},\ \{\psi\}\ \texttt{S}_2\ \{\phi\}}{\{\varphi\}\ \texttt{S}_1\ \texttt{@@}\ \texttt{S}_2\ \{\phi\}}$$

– Given a program `P`, the following form

```
for X from 0 to t-1 do
    P
end for
```

(we write it `loop(X, t, P)` for short) has the following meaning

`X ⟵ 0 @@ P @@ ...@@ X ⟵ t-1 @@ P`

The loop rule is

$$\frac{\{\varphi \wedge \texttt{X < t}\}\ \texttt{P}\ \{\varphi\texttt{[X+1/X]}\}}{\{\varphi\texttt{[0/X]}\}\ \texttt{loop(X, t, P)}\ \{\varphi\texttt{[t/X]}\}}$$

no assignment to `X` or variables occurring in `t` is used in `P`

4.1 First Order Logic

To express conditions over states we consider a first order logic whose set of terms, `TERM`, is the inductive closure of the union of the set of variables mentioned above and the set of lifted constants under the constructors `l(op)` for every function `op`. The set of atomic formulas is the inductive closure of the pair of sets `TERM` and {`l(p)| p boolean constant`}, under the constructors `l(op)` for every predicate `op`.

Previous to the definition of the set of formulas we need the concept of the state, `s[d/X]`, resulting from a given one, `s`, after the substitution of the associated value for a variable `X` by `d`. That is, `s[d/X]` is a state such that for

any other variable different from `X` it has the same associated value than `s` and for `X` it has `d` as the associated value.

The set of formulas is the inductive closure of the set of atomic formulas under the constructors `l(∧)`, `l(∨)`, `l(¬)`, `foreach` and `exists`, where

`foreach(X, φ): S → bool` such that `foreach(X, φ)(s)` $\equiv \forall$ `d (φ(s[d/X]))`

`exists(X, φ): S → bool` such that `exists(X, φ)(s)` $\equiv \exists$ `d (φ(s[d/X]))`

5 The Pairwise Disjoint Families Problem

Let us consider the following problem:

Let $A = \{0, \ldots, p-1\}$. *Let* $\mathcal{F} = \{B_0, \ldots, B_{q-1}\}$ *a finite family of subsets of* A. *To determine all the ordered pairs* $(\mathcal{F}', \bigcup \mathcal{F}')$, *where* $\mathcal{F}'$ *is a subfamily of* $\mathcal{F}$ *and its elements are pairwise disjoint.*

To solve this problem in the sticker model we consider as initial tube T_0, a `(p+q, q)`–library (a tube containing, at least, a copy of any memory complex with `p+q` regions and the `p` last regions deactivated). The first `q` bits represent a subfamily of $\mathcal{F}$. Given a memory complex with `p+q` regions, σ, we consider that it codifies an ordered pair $(\mathcal{F}_\sigma, A_\sigma)$, where $\mathcal{F}_\sigma$ is the subfamily $\{B_j \mid \sigma(j) = \texttt{on}\}$ of $\mathcal{F}$ and A_σ is the subset $\{j \mid \sigma(j+q) = \texttt{on}\}$ of `A`.

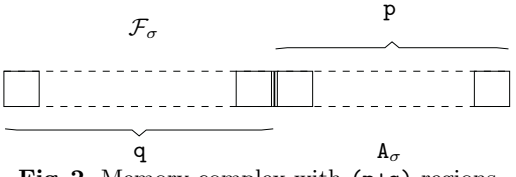

Fig. 2. Memory complex with `(p+q)` regions

The following is a program in the sticker model that solves the pairwise disjoint families problem (where b^i_j is the j-th element of B_i, the i-th subset of $\mathcal{F}$, and r_i is its size; that is, $B_i = \{b^i_0, \ldots, b^i_{r_i-1}\} \in \mathcal{F}$).

Note: Each instruction is labeled in order to make references.

```
            Procedure Disjoint
            INPUT: A family F of A subsets
I1                for I ⟵ 0 to q-1 do
L1                      (T*, T-) ⟵ Separate(T, I) @@
L2                      for J ⟵ 0 to r_I-1 do
l1                            (T+, T'-) ⟵ Separate(T*, b^I_J+q) @@
l2                            T* ⟵ Set(T'-, b^I_J+q)
                        end for @@
L3                      T ⟵ Combine(T*, T-)
                  end for
```

The following PVS expression implements the program:

```
disjoint(F: (FAMILY(p, q))): program =
LET eB = l(elemF(p,q,F)) IN
loop(VI, q,
     assig2((VTast, VTn), l(separate)(VT, VI)) @@
     loop(VJ, l(tam(p,q,F))(VI),
          assig2((VTm, VTnn), l(separate)(VTast, eB(VI, VJ) + l(q))) @@
          (VTast << l(set)(VTnn, eB(VI, VJ) + l(q)))) @@
     (VT << l(combine)(VTast, VTn)))

assig2(PT: [V1, V1], Pt: [term1, term1]): program =
      (PT‘1 << Pt‘1) @@ (PT‘2 << Pt‘2)
```

In order to stablish the correctness of this program we consider the formula:

$\Theta_{\mathcal{F}}$(T) $\equiv \forall \tau$ ($\tau \in$ T $\rightarrow \forall$ i_1< i_2< q ($\tau(i_1)$ = $\tau(i_2)$ = on $\rightarrow$ $B_{i_1} \cap B_{i_2} = \emptyset$))

expressing that the memory complexes of a given tube codifies a subfamily of $\mathcal{F}$ whose elements are pairwise disjoint.

```
correc_disjoint(F: (FAMILY(p, q)))(T: MTUBE[p + q]): bool =
  FORALL (tau: MEMORY_COMPLEX): (ms_in(tau, T) IMPLIES
      (FORALL (i1, i2: below[q]):
        (appl(tau, i1) = on AND appl(tau, i2) = on AND i1 < i2 IMPLIES
          disj(F‘seq(i1), F‘seq(i2)))))
```

The following specification stablish the correctness of the program (where `library?[p+q](q)` is a predicate over tubes characterizing a `(p+q, q)`–library):

$$\{\texttt{library?[p+q](q)(T)}\}\ \texttt{disjoint}(\mathcal{F})\ \{\Theta_{\mathcal{F}}\texttt{(T)}\}$$

To prove this specification we will use two formulas θ and δ, that will be invariants of the main loop (I_1) and inner loop (L_2), respectively. For these formulas we prove the following results:

1. library?[p+q](q)(T) $\rightarrow \theta$[0/I]
2. θ[q/I] $\rightarrow \Theta_{\mathcal{F}}$(T)
3. $\theta \wedge$ I < q $\rightarrow \delta$[0/J][+(T,I)/T*][-(T,I)/T-]
4. $\delta[r_I/J] \rightarrow \theta$[I+1/I][T* $\cup$ T-/T]
5. $\delta \wedge$ J < $r_I \rightarrow$
   δ[J+1/J][Set(T’-, b_J^I+q)/T*][-(T*, b_J^I+q)/T’-][+(T*, b_J^I+q)/T+]

From those results and using the appropriate axioms and inference rules from Floyd–Hoare logic we prove the following specifications:

– $\{\delta*\}$ l_1 @@ l_2 $\{\delta$[J+1/J]$\}$ where $\delta*$ is the formula
  δ[J+1/J][Set(T’-, b_J^I+q)/T*][-(T*, b_J^I+q)/T’-][+(T*, b_J^I+q)/T+]
  (using the assignment axiom and the compose rule).

- $\{\delta \wedge \texttt{J < r}_\texttt{I}\}\ \texttt{l}_1\ \texttt{@@}\ \texttt{l}_2\ \{\delta\texttt{[J+1/J]}\}$ (using 5 and the consequence rule).
- $\{\delta\texttt{[0/J]}\}\ \texttt{L}_2\ \{\delta\texttt{[r}_\texttt{I}\texttt{/J]}\}$ (using the loop for rule).
- $\{\delta\texttt{[0/J]}\}\ \texttt{L}_2\ \{\theta\texttt{[I+1/I][T*} \cup \texttt{T-/T]}\}$ (using 4 and the consequence rule).
- $\{\delta\texttt{[0/j][+(T,I)/T*][-(T,I)/T-]}\}\ \texttt{L}_1\ \texttt{@@}\ \texttt{L}_2\ \texttt{@@}\ \texttt{L}_3\ \{\theta\texttt{[I+1/I]}\}$ (with the assignment axiom and the compose rule).
- $\{\theta \wedge \texttt{I < q}\}\ \texttt{L}_1\ \texttt{@@}\ \texttt{L}_2\ \texttt{@@}\ \texttt{L}_3\ \{\theta\texttt{[I+1/I]}\}$ (with 3 and the consequence rule).
- $\{\theta\texttt{[0/I]}\}\ \texttt{I}_1\ \{\theta\texttt{[q/I]}\}\}$ (using the loop for rule).
- $\{\texttt{library?[p+q](q)(T)}\}\ \texttt{disjoint}(\mathcal{F})\ \{\Theta_\mathcal{F}\texttt{(T)}\}$ (using 1, 2 and the consequence rule).

The used formulas, θ and δ, are the following:

$$\theta \equiv \theta_D\texttt{(T,I)} \wedge \theta_R\texttt{(T,I)} \wedge \texttt{(I=0} \rightarrow \texttt{library?[p+q](q)(T)])}$$

$$\delta \equiv \delta_D\texttt{(T*,I,J)} \wedge \theta_D\texttt{(T-,I)} \wedge \texttt{carac(T-,I)} \wedge \delta_R\texttt{(T*,I,J)} \wedge \theta_R\texttt{(T-,I)}$$

where

- $\theta_D\texttt{(T,I)}$ is the formula:

  $\texttt{I} \leq \texttt{q} \rightarrow \forall\tau\ (\tau \in \texttt{T} \rightarrow \forall i_1 < i_2 < \texttt{I}\ (\tau(i_1) = \tau(i_2) = \texttt{on} \rightarrow \texttt{B}_{i_1} \cap \texttt{B}_{i_2} = \emptyset))$
  expressing that for each memory complex τ of a tube T, the elements of the subfamily $\mathcal{F}_\tau^\texttt{I} = \{\texttt{B}_i \mid \texttt{i < I} \wedge \tau(i) = \texttt{on}\}$ are pairwise disjoint.
- $\theta_R\texttt{(T, I)}$ is the formula:

  $$\texttt{I} \leq \texttt{q} \rightarrow \forall\tau\ (\tau \in \texttt{T} \rightarrow \forall k < \texttt{I}\ (\tau(k) = \texttt{on} \rightarrow \texttt{B}_k\texttt{+q} \subseteq \tau) \wedge \forall s < \texttt{p}\ (\tau(s\texttt{+q}) = \texttt{on} \rightarrow \exists k < \texttt{I}\ (\tau(k) = \texttt{on} \wedge s \in \texttt{B}_k)))$$
  that is, for each memory complex, τ, of a tube T, we have $\bigcup \mathcal{F}_\tau^\texttt{I} = \texttt{A}_\tau$.
- $\delta_D\texttt{(T,I,J)}$ is the formula:

  $$\texttt{I} < \texttt{q} \wedge \texttt{J} \leq r_\texttt{I} \rightarrow \forall\tau\,(\tau \in \texttt{T} \rightarrow (\tau(I) = \texttt{on} \rightarrow \forall i_1 < \texttt{I}\ (\tau(i_1) = \texttt{on} \rightarrow \texttt{B}_{i_1} \cap \texttt{B}_\texttt{I}^\texttt{J} = \emptyset)) \wedge \forall i_1 < i_2 < \texttt{I}\,(\tau(i_1) = \tau(i_2) = \texttt{on} \rightarrow \texttt{B}_{i_1} \cap \texttt{B}_{i_2} = \emptyset))$$
  expressing that for each memory complex, τ, in a tube T, if $\texttt{B}_\texttt{I} \in \mathcal{F}_\tau^{\texttt{I}+1}$, then the set $\texttt{B}_\texttt{I}^\texttt{J}$ (compose by the first J elements of $\texttt{B}_\texttt{I}$) is disjoint with the elements of the subfamily $\mathcal{F}_\tau^\texttt{I}$; and that the elements of the subfamily $\mathcal{F}_\tau^\texttt{I}$ are pairwise disjoint.
- $\delta_R\texttt{(T,I,J)}$ is the formula

  $$\texttt{I} < \texttt{q} \wedge \texttt{j} \leq r_\texttt{I} \rightarrow \forall\tau\,(\tau \in \texttt{T} \rightarrow (\tau\texttt{(I)} = \texttt{on} \wedge \forall k < \texttt{I}\ (\tau(k) = \texttt{on} \rightarrow \texttt{B}_k\texttt{+q} \subseteq \tau) \wedge \texttt{B}_\texttt{I}^\texttt{J}\texttt{+q} \subseteq \tau \wedge \forall s < \texttt{p}\,(\tau(s\texttt{+q}) = \texttt{on} \rightarrow \exists k < \texttt{I}\,((\tau(k) = \texttt{on} \wedge s \in \texttt{B}_k) \vee s \in \texttt{B}_\texttt{I}^\texttt{J}))))$$
  expressing that for each memory complex, τ, in a tube T, we have
  $$(\bigcup \mathcal{F}_\tau^\texttt{I}) \cup \texttt{B}_\texttt{I}^\texttt{J} = \texttt{A}_\tau$$
- $\texttt{carac(T, I)} \equiv \forall\tau\ (\tau \in \texttt{T} \rightarrow \tau\texttt{(I) = off)}$.
  This formula characterizes the contents of the second tube obtained after the use of the `Separate` operation.

6 Conclusions

A great part of our work within molecular computing is related to the formalization of the different models that have appeared. During this effort we have drawn the conclusion that 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. 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.

To develop the work presented we have provided not only an implementation of the sticker model in PVS. All the elements necessary to represent problems over finite sets of natural numbers has been described in the system. Also we have proved some usual properties over them and plan to complete this work for general purpose. Most of the proofs constructed with the system have been obtained using the basic commands and a previously elaborated hand written proof. This effort shows the utility of the system as a verification tool.

References

1. L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 1994.
2. R. S. Boyer and J S. Moore. *The correctness problem in computer science.* Academic Press, 1981.
3. P. Y Gloess. Imperative program verification in PVS. `http://www.labri.fr/Perso/~gloess/imperative/` (1999).
4. C. Graciani Díaz. *Especificación y verificación de programas moleculares en PVS.* Doctoral Thesis, University of Seville (2003).
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–583, 1969.
6. Lila Kari, Gheorghe Paun, Grzegorz Rozenberg, Arto Salomaa, and S. Yu. DNA computing, sticker systems and universality. *Acta Informatica*, 35:401–420, 1998.
7. S. Owre, N. Shankar and J. Rushby *The PVS specification and verification system.* `pvs.csl.sri.com`
8. S. Owre and N. Shankar *The formal semantics of PVS.* Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
9. Roweis, S.; Winfree, E.; Burgoyne, R.; Chelyapov, N. V.; Goodman, M. F.; Rothemund, P. W. K.; Adleman, L. M. A sticker based model for DNA computation. Landweber, L.; Baum, E., eds *DNA Based Computers II*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, 44, 1–27. American Mathematical Society (1999).
10. Sancho, F. *Verificación de programas en modelos de computación no convencionales.* Doctoral Thesis, University of Seville (2002).
11. H. Pfeifer, A. Dold, F. W. v. Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, Fakultät für Informatik, 1996.