Formal Verification of Molecular Computational Models in ACL2: A Case Study[⋆]

Francisco J. Martín-Mateos, José A. Alonso,
Maria José Hidalgo, and José Luis Ruiz-Reina

Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
`http://www.cs.us.es/{~fmartin,~jalonso,~mjoseh,~jruiz}`

Abstract. Theorem proving is a classical AI problem with a broad range of applications. Since its complexity is exponential in the size of the problem, many methods to parallelize the process has been proposed. One of these approaches is based on the massive parallelism of molecular reactions. ACL2 is an automated theorem prover especially adequate for algorithm verification. In this paper we present an ACL2 formalization of a molecular computational model: Adleman's restricted model. As an application of this model, an implementation of Lipton's experiment solving SAT is described. We use ACL2 to make a formal proof of the completeness and soundness properties of this implementation.

1 Introduction

In the last years the interest in developing new computational models based on biological models has increased [2, 13]. One of the main advantages of these models is the massive parallelism associated with some process. This reduces considerably the complexity of some problems (with respect to the elemental operations in the model). However, the biological implementation of these models is not often possible and, when it can be done, the cost of the experiments could force to increase our confidence in their correction.

ACL2 [7] is a programming language, a logic for reasoning about programs in the language, and a theorem prover supporting formal reasoning in the logic. Automated reasoning systems in general and ACL2 in particular, are usually used to build formal models of "digital systems", software and hardware [9, 15]. Using the proof techniques of these systems, we can prove properties of the formalized models. In this paper, we present an application of the ACL2 system to formalize and verify computational models based on biological models. In particular, we formalize a molecular computational model and one of the first biological experiment solving a NP-complete problem.

Adleman's first experiment [1] shows that NP-complete problems could be solved by means of manipulation of DNA molecules. Based on Adleman's ideas,

⋆ This work has been supported by project TIC2000-1368-C03-02 (Ministry of Science and Technology, Spain), cofinanced by FEDER funds.

R.J. Lipton [11] solved an instance of the satisfiability propositional problem. In this sense, new experiments has been done recently [3, 10]. In this paper we present our ACL2 formalization of Adleman's restricted model. This formalization is done in such a way that the subsequent development is generic: the specific operations are not important, but only their properties. In [14] a formalization of Lipton's experiment is given as an iterative algorithm based on the elemental operations of Adleman's restricted model. We define recursive functions implementing this formalization and we prove the completeness and soundness properties of these functions.

2 The ACL2 System

ACL2 [7] is a programming language, a logic for reasoning about programs in the language, and a theorem prover supporting formal reasoning in the logic. The ACL2 logic is a quantifier-free, first-order logic with equality, describing an extension of an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp and the logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference of the logic include those for propositional calculus, equality and instantiation. The ACL2 theorem prover mechanizes that logic, being particularly well suited for obtaining automatized proofs based on simplification and induction. For a detailed description of ACL2, we refer the reader to the book [6].

By the *principle of definition*, new function definitions are admitted as axioms only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that no inconsistencies are introduced by new definitions. Some higher order functionality is provided by means of the `encapsulate` mechanism [8] which allows the user to introduce new function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms. This mechanism behaves like an universal quantifier over a set of functions abstractly defined with it. So, any theorem proved about these functions is true for any functions with the same properties as the assumed in the `encapsulate`.

The user can start a proof attempt invoking the `defthm` command establishing the property she wants to prove. The ACL2 theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, the user can (and usually must) guide the prover by adding lemmas and definitions that are used in subsequent proofs as rewriting rules. A typical ACL2 proof effort consists of formalizing the problem in the logic and helping the prover to find a preconceived proof by means of a suitable set of rewriting rules. These rules can be found by inspecting the failed proofs. That is the methodology we followed in this case study.

For the sake of readability, the ACL2 expressions in this paper are presented using a notation closer to the usual mathematical notation than its original

Common Lisp syntax. Some of the functions are also used in infix notation. The complete files with definitions and theorems are available on the Web in `http://www.cs.us.es/~fmartin/acl2/molecular/`.

3 Adleman's Restricted Model

In [2] some abstract models for molecular computing are described. The first model proposed works with test tubes with a set of DNA molecules, i.e. a multiset of finite sequences over the alphabet $\{A, C, G, T\}$. Nevertheless, it may be preferable to use molecules other than DNA, using an alphabet Σ which is not necessarily $\{A, C, G, T\}$. Further, though DNA has a natural structure which allows to order the occurrence of elements and hence deal with sequences, this may not be true for other types of molecules. Then, the members of a tube will be multisets of elements from Σ. In the sequel, we consider an alphabet Σ and we call *aggregate* a multiset of elements from this alphabet.

The above considerations are the basis of the restricted model of molecular computation. This model works on test tubes with a multiset of aggregates (i.e. a multiset of multisets of elements from Σ). On these tubes, the following operations can be performed:

- $Separate(T, x)$: Given a tube T and an element $x \in \Sigma$, produces two new tubes, $+(T, x)$ and $-(T, x)$, where $+(T, x)$ is the tube consisting of every aggregate of T which contains the element x and $-(T, x)$ is the tube consisting of every aggregate of T which does not contain the element x:
$$+(T, x) = \{\gamma \in T : x \in \gamma\}$$
$$-(T, x) = \{\gamma \in T : x \notin \gamma\}$$
- $Merge(T_1, T_2)$: Given tubes T_1 and T_2, produces the new tube $T_1 \cup T_2$, which is the multiset union of the multisets T_1 and T_2.
- $Detect(T)$: Given a tube T, decides if T contains at least one aggregate; that is, returns "yes" if T contains at least one aggregate and returns "no" if it contains none.

These operations are performed in the laboratory in the following way. If a $Merge$ of tubes is required, this is accomplished by pouring the contents of one of the tubes into the other. If a $Separate$ or a $Detect$ operation is required on a tube then some technical operations (magnetic bead system, polymerase chain reaction, get electrophoresis, ...) are performed on it. This model is called "restricted" in the sense that the molecules themselves do not change in the course of a computation.

To formalize the restricted model in ACL2, we use lists to represent multisets. Then, a test tube is represented as a list of aggregates and an aggregate is represented as a list of elements from Σ. So, the functions associated with the molecular operations work on lists.

We consider two functions, `separate+` and `separate-`, associated with the *Separate* operation. The first one returning the value $+(T,x)$ and the second one the value $-(T,x)$. The *Merge* operation is associated with the function `tube-merge`. Finally, we consider the function `detect` associated with the *Detect* operation.

The definition of these functions is not so interesting as their properties. The properties of any algorithm built on the restricted model must be independent of the implementation of the operations. This will ensure the properties of the algorithm even when it was evaluated in a molecular laboratory. Therefore, we define them by means of the `encapsulate` mechanism, constraining them to have certain properties. These properties are the following:

ASSUMPTION: `member-separate+`
$\gamma \in \texttt{separate+}(T,x) \leftrightarrow x \in \gamma \wedge \gamma \in T$

ASSUMPTION: `member-separate-`
$\gamma \in \texttt{separate-}(T,x) \leftrightarrow x \notin \gamma \wedge \gamma \in T$

ASSUMPTION: `member-tube-merge`
$\gamma \in \texttt{tube-merge}(T_1,T_2) \leftrightarrow \gamma \in T_1 \vee \gamma \in T_2$

ASSUMPTION: `member-detect`
$\texttt{detect}(T) = \texttt{t} \leftrightarrow \exists e \in T$

If we want to test any algorithm built on the restricted model, we must provide concrete functions implementing the basic operations and prove the encapsulated properties for them. Anyway, introducing these properties by means of `encapsulate`, we ensure that the proof of subsequent properties are independent of these concrete implementations.

4 Lipton's Experiment

Adleman's experiment [1] solved an instance of the Hamiltonian path problem over a directed graph with two designated vertices, by implementing a brute force procedure in a laboratory of molecular biology. To solve the problem, an initial test tube with DNA molecules encoding all the paths in the graph was built. This tube was subjected to some operations based on DNA manipulation, and every aggregate encoding a path which was not a valid solution of the problem was removed.

Lipton shows in [11] how to solve an instance of the satisfiability problem for Propositional Logic, using the ideas of Adleman. To achieve this, he described every relevant assignment of a propositional formula by means of paths on a directed graph associated with the variable set of the formula. Specifically, given a propositional formula in conjunctive normal form, $F = c_1 \wedge \ldots \wedge c_p$, where the clauses $c_i = l_{i,1} \vee \ldots \vee l_{i,r_i}$, and the set of variables $Var(F) = \{x_1, \ldots, x_n\}$, the associated directed graph $G_n = (V_n, E_n)$ is defined as follows:

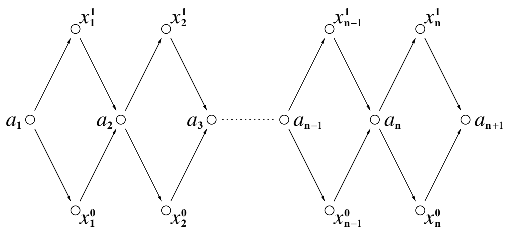

Fig. 1. Directed graph associated with a propositional formula with n variables

$$V_n = \{x_i^j : 1 \leq i \leq n, 0 \leq j \leq 1\} \cup \{a_i : 1 \leq i \leq n+1\}$$
$$E_n = \{(a_i, x_i^j), (x_i^j, a_{i+1}) : 1 \leq i \leq n, 0 \leq j \leq 1\}$$

This graph, shown in figure 1, verifies the following properties:

– There are 2^n paths from a_1 to a_{n+1}.
– There exists a natural bijection between the above set of paths and the relevant assignments of F, according to the following criteria: given a path from a_1 to a_{n+1}, $\gamma = a_1 x_1^{j_1} a_2 x_2^{j_2} \ldots x_n^{j_n} a_{n+1}$, then the assignment $\hat{\gamma}$ is associated with it, such as $\hat{\gamma}(x_i) = j_i,\ 1 \leq i \leq n$.

The initial test tube contains DNA molecules codifying the paths from a_1 to a_{n+1}, and so every relevant assignment of F. The alphabet and the initial test tube considered are the following:

$$\Sigma = \{a_i, x_i^j, a_{n+1} : 1 \leq i \leq n, 0 \leq j \leq 1\}$$

$$T_0 = \{\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, \ldots, x_n^{j_n}, a_{n+1}\} : 1 \leq i \leq n, j_i \in \{0,1\}\}$$

Lipton's experiment can be described as follows: for each clause in the initial formula, every aggregate representing an assignment falsifying this clause is removed. The way to work with clauses is the following: for each literal in the clause, every aggregate representing an assignment in which this literal is true is preserved, and the remaining aggregates are removed. This experiment has been formalized in [14], where it has been expressed as an iterative algorithm based on the elemental operations of Adleman's restricted model:

```
Input: $T_0$ (as described above)
  For $i \leftarrow 1$ to $p$ do
      $T_{i,0} \leftarrow \emptyset$
      $T''_{i,0} \leftarrow T_{i-1}$
      For $j \leftarrow 1$ to $r_i$ do
          $T'_{i,j} \leftarrow$ separate+$(T''_{i,j-1}, l^1_{i,j})$
          $T''_{i,j} \leftarrow$ separate-$(T''_{i,j-1}, l^1_{i,j})$
          $T_{i,j} \leftarrow$ tube-merge$(T_{i,j-1}, T'_{i,j})$
      $T_i \leftarrow T_{i,r_i}$
  detect$(T_p)$
```

where, for each literal $l_{i,j}$ in the initial formula:

$$l^1_{i,j} = \begin{cases} x^1_m \text{ if } l_{i,j} = x_m \\ x^0_m \text{ if } l_{i,j} = \neg x_m \end{cases}$$

In this formalization, the first loop deals with the clauses in the initial formula. The tube $T''_{i,0}$ is the set of aggregates before processing the clause c_i, and the tube $T_{i,0}$ acts as accumulator for the aggregates representing an assignment making c_i true. The second loop deals with the literals in a clause. The aggregates representing an assignment in which the literal is true (that is, the aggregates with the element $l^1_{i,j}$) are in the tube $T'_{i,j}$ which is merged with the accumulator, the remaining are in the tube $T''_{i,j}$ which is used with the next literal. When every literal in a clause has been processed, the tube T_{i,r_i} contains the aggregates from the tube $T''_{i,0}$ representing an assignment making true that clause.

It must be noticed that the complexity of this experiment, with respect to the basic molecular operations, is $O(k)$, where k is the number of literals. This low complexity is mainly due to the massive parallelism of molecular reactions. Of course, our simulation in ACL2 is sequential, and it loses this advantage. The basic molecular operations (with constant time cost) are performed in ACL2 by exhaustive analysis, and this dramatically increases the complexity.

Next, we present our implementation of Lipton's experiment in ACL2. First of all we must notice that the above algorithm depends on the initial test tube (T_0) and on the propositional formula (F), by means of the $l^1_{i,j}$. Then, we have defined a function with two arguments, the formula and the tube. On the other hand, the iterative formulation presented above is not adequate for its implementation in the functional language of ACL2. We have made a recursive formulation equivalent to the iterative version. In fact, we have defined two functions, one for each loop. The function dealing with the external loop works recursively on the number of clauses of F, and the other one works recursively on the number of literals of the selected clause.

Our implementation does not use the *Detect* operation, instead it returns the final tube in the external loop (T_p). This is useful to formulate the soundness and completeness properties of the functions implementing the experiment as we will show in the next section.

First of all we define the function `l-element`, that builds the element $l^1_{i,j}$ from the literal $l_{i,j}$. Literals are represented using integers, thus, for all $i > 0$, literal x_i is represented with the integer i, and $\neg x_i$ with $-i$. To represent the elements $l^1_{i,j}$ we use pairs: the element x^0_i is represented with the pair `(i . 0)` and the element x^1_i with the pair `(i . 1)`.

Definition:
```
l-element($L$) =
  **if** $L < 0$ **then** ($-L$ . 0)
  **else** ($L$ . 1)
```

Next, we define a function implementing the internal loop. Its inputs are a main tube T (corresponding to $T''_{i,j}$ in the iterative version presented above), an accumulator tube *acc* (corresponding to $T_{i,j}$) and a clause C (corresponding to c_i). The aggregates in the main tube containing the element $l^1_{i,1}$ are merged with the accumulator tube in a new one. The aggregates in the main tube that do not contain the element $l_{i,1}^1$ are poured in a new main tube. The new main and accumulator tubes are used in the recursive call on the rest of the literals:

Definition:
```
sat-lipton-clause($C$,$T$,$acc$) =
  **if** endp($C$) **then** $acc$
  **else let*** $T^+$ = separate+($T$,l-element(car($C$)))
             $T^-$ = separate-($T$,l-element(car($C$)))
             $Nacc$ = tube-merge($acc$,$T^+$)
      **in** sat-lipton-clause(cdr($C$),$T^-$,$Nacc$)
```

The main function deals with the external loop. Its inputs are a tube T (corresponding to the initial tube T_0 in the iterative version presented above) and a formula F in conjunctive normal form. This function applies the internal loop on this tube, an initially empty accumulator tube and the first clause of the formula. The result of this process is used as initial tube in the recursive call on the rest of clauses:

Definition:
```
sat-lipton-cnf-formula($F$,$T$) =
  **if** endp($F$) **then** $T$
  **else let** $NT$ = sat-lipton-clause(car($F$),$T$,**nil**)
      **in** sat-lipton-cnf-formula(cdr($F$),$NT$)
```

5 Using ACL2 to Prove Correctness

Once formalized in ACL2 the abstract model with its assumed properties, and defined the functions implementing the experiment in this formalization, we can prove in the system the termination, soundness and completeness properties of these functions. The termination property is straightforward (in the recursive calls the length of F or C decreases) and it is proved without additional help from the user. The soundness and completeness properties are the following:

1. Soundness: $\forall \gamma \in T_p, (\hat{\gamma}(F) = 1)$
2. Completeness: $\forall \gamma \in T_0, (\hat{\gamma}(F) = 1 \Rightarrow \gamma \in T_p)$

where F is a propositional formula in conjunctive normal form and $\hat{\gamma}(F)$ is the truth value of F in the assignment $\hat{\gamma}$ (the truth value of a formula in conjunctive normal form is extended as usual).

These properties of the algorithm have two hidden assumptions:

1. F is a formula in conjunctive normal form (`cnf-formula-p`).
2. γ is an aggregate with the form: $\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, \ldots, x_n^{j_n}, a_{n+1}\}$ with $1 \leq i \leq n$ and $j \in \{0, 1\}$

To deal with the first of these assumptions, we have formalized some concepts related to propositional logic. The functions `literal-p`, `clause-p` and `cnf-formula-p` characterize respectively literals (non-null integer numbers), clauses (lists of literals) and formulas in conjunctive normal form (lists of clauses). To represent assignments, we use association lists. In these lists a propositional variable can have associated any value; if this value is `1`, the variable is interpreted as true, otherwise it is interpreted as false. The functions `literal-value`, `clause-value` and `cnf-formula-value` compute respectively the truth value in an assignment of a literal, clause or formula in conjunctive normal form.

We use lists to represent aggregates in the following way: the aggregate $\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, \ldots, x_n^{j_n}, a_{n+1}\}$ is represented by the list

$$\texttt{((A . 1) (1 . } j_1\texttt{) (A . 2) ... (}n\texttt{ . }j_n\texttt{) (A . }n\texttt{))}$$

In this way, we use the same expression to represent an aggregate γ and the associated assignment $\hat{\gamma}$. The pairs `(A . `i`)` are ignored when we use this expression to represent an assignment.

We have checked that the following property is enough to characterize the aggregates: for each variable x_i in the original formula, there must exist one and only one x_i^j in the aggregate[1]. We have defined three functions checking this property. The first one (`literal-aggregate-p`) checks the property with respect to the variable of a literal, the second one (`clause-aggregate-p`) with respect to the variable set of a clause and the third one (`cnf-formula-aggregate-p`) with respect to the variable set of a formula in conjunctive normal form. In the sequel, when we say that γ is an aggregate w.r.t. a literal, a clause or a formula, we mean that γ is an aggregate with respect to its variable or its variable set.

Now, we can formulate the completeness property:

Theorem: completeness-sat-lipton-cnf-formula

$$\begin{array}{l} (\gamma \in T \wedge \texttt{cnf-formula-p}(F) \wedge \texttt{cnf-formula-aggregate-p}(\gamma,F) \\ \qquad \wedge\ \texttt{cnf-formula-value}(F,\gamma) = 1) \\ \quad \to \gamma \in \texttt{sat-lipton-cnf-formula}(F,T) \end{array}$$

Let us briefly describe the proof process of this theorem. The ACL2 prover tries to prove it by induction. Based on its heuristics, the system uses the induction scheme suggested by the function `sat-lipton-cnf-formula`. This produces the following subgoals:

1) $\texttt{endp}(F) \to P(\gamma, F, T)$
2) $\neg\texttt{endp}(F) \wedge P(\gamma,\texttt{cdr}(F),\texttt{sat-lipton-clause}(\texttt{car}(F),T,\textbf{nil})) \to P(\gamma, F, T)$

where $P(\gamma, F, T)$ denotes the property we want to prove.

As we can see, the first subgoal is straightforward (in this case the value of `sat-lipton-cnf-formula`(F,T) is T) and the second one is not easy. Using the simplification process, the system transforms the second subgoal obtaining the following:

[1] Therefore, the elements a_i in the aggregates are not necessary. Nevertheless, we have to consider them to faithfully reflect the original experiment.

$$\begin{aligned}&(\gamma \in T \wedge \texttt{consp}(F) \wedge \texttt{clause-p}(C) \wedge \texttt{cnf-formula-p}(F')\\ &\qquad \wedge \texttt{clause-aggregate-p}(\gamma,C) \wedge \texttt{cnf-formula-aggregate-p}(\gamma,F')\\ &\qquad \wedge \texttt{clause-value}(C,\gamma) \neq 0 \wedge \texttt{cnf-formula-value}(F',\gamma) = 1\\ &\qquad \wedge \gamma \notin \texttt{sat-lipton-clause}(C,T,\textbf{nil}))\\ &\quad \rightarrow \gamma \in \texttt{sat-lipton-cnf-formula}(F,T)\end{aligned}$$

where C is car(F) and F' is cdr(F).

In a first attempt, the proof of this subgoal fails. Inspecting the failed proof, we found that a very similar property should be proved about the function sat-lipton-clause. One possibility is the following:

Theorem: completeness-sat-lipton-clause

$$\begin{aligned}&(\gamma \in T \wedge \texttt{clause-p}(C) \wedge \texttt{clause-aggregate-p}(\gamma,C) \wedge \texttt{clause-value}(C,\gamma) = 1)\\ &\quad \rightarrow \gamma \in \texttt{sat-lipton-clause}(C,T,acc)\end{aligned}$$

Once again, the system tries to prove this theorem using the induction scheme suggested by the function sat-lipton-clause. Inspecting the proof attempt we can also conclude that some property about separate+ should be proved. This property is the following:

Theorem: completeness-separate+

$$\begin{aligned}&(\gamma \in T \wedge \texttt{literal-p}(L) \wedge \texttt{literal-aggregate-p}(\gamma,L) \wedge \texttt{literal-value}(L,\gamma) = 1)\\ &\quad \rightarrow \gamma \in \texttt{separate+}(T,\texttt{l-element}(L))\end{aligned}$$

This theorem is proved using elemental properties about aggregates and their associated assignments. Using this theorem, the system can prove the completeness property of sat-lipton-clause and, finally, the completeness property of sat-lipton-cnf-formula.

The proof of the soundness property is obtained in a similar way. The associated ACL2 event is the following:

Theorem: soundness-sat-lipton-cnf-formula

$$\begin{aligned}&(\texttt{cnf-formula-p}(F) \wedge \texttt{cnf-formula-aggregate-p}(\gamma,F)\\ &\qquad\qquad \wedge \gamma \in \texttt{sat-lipton-cnf-formula}(F,T))\\ &\quad \rightarrow \texttt{cnf-formula-value}(F,\gamma) = 1\end{aligned}$$

6 Conclusions

In this work we have presented a formalization of Adleman's restricted model, one of the first molecular computational models. This formalization has been done in a generic framework in which the concrete implementation of its operations is not important, but only their properties. Using this formalization we have defined functions simulating Lipton's experiment solving SAT. Finally, the completeness and soundness properties of these functions have been proved.

The formalization of unconventional models of computation is a suitable way of working with them when we do not have real models (e.g. we do not have a laboratory implementing molecular computational models). This formalization brings us the possibility of simulate real experiments or develop new ones.

Furthermore, using an automated reasoning system allows to formally prove properties of the simulated experiments. The automatic system helps to develop these proofs avoiding a hand development.

We have presented a recursive formalization, in opposite to the iterative version presented in [14]. This fact is due to the applicative nature of ACL2. This approach suggests the application of proof techniques based on induction (as usual in ACL2), to prove the correctness properties of the functions simulating the experiment, as opposed to the needed with an iterative version, based on Hoare logic. In [5] we have reproduced the development presented here in the PVS system [12], as part of a project about formal specification of molecular computational models in this system.

References

1. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science, 266:1021–1024, 1994.
2. Adleman, L.M.: On constructing a molecular computer. DNA Based Computers, DIMACS Series, 27, pp. 1–21. American Mathematical Society, 1996.
3. Braich, R.S., Chelyapov, N., Johnson, C., Rothemund, P.W.K. and Adleman, L.: Solution of a 20-Variable 3-SAT Problem on a DNA Computer. Science, 296:499–502, 2002.
4. Beaver, D.: A universal molecular computer. DNA Based Computers, DIMACS Series, 27, pp. 29–36. American Mathematical Society, 1996.
5. Graciani, C., Martín–Mateos, F.J. and Pérez–Jiménez, M.J.: Specification of Adleman's Restricted Model Using an Automated Reasoning System: Verification of Lipton's Experiment. LNCS vol. 2509 pp. 126–136, 2002.
6. Kaufmann, M., Manolios, P. and Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.
7. Kaufmann, M. and Moore, J S.: ACL2 Version 2.7, 2002. Homepage: `http://www.cs.utexas.edu/users/moore/acl2/`
8. Kaufmann, M. and Moore, J S.: Structured Theory Development for a Mechanized Logic. Journal of Automated Reasoning, 26(2): 161–203, 2001.
9. Moore, J S.: Piton: a mechanically verified assembly-level language. Kluwer Academic Publisher, 1996.
10. Lee, I.-H., Park, J.-Y., Jang, H.-M., Chai, Y.-G. and Zhang, B.-T.: DNA Implementation of Theorem Proving with Resolution Refutation in Propositional Logic. LNCS vol. 2568, pp. 156–167, 2003.
11. Lipton, R.J.: DNA solution of hard computational problems. Science, 268:542–545, 1995.
12. Owre, S., Rushby, J.M., Shankar, N. and Stringer–Calvert, D.W.J.: PVS System Guide. Homepage: `http://pvs.csl.sri.com/`
13. Paun, G.: Computing with membranes. Journal of Computer and System Sciences, 61(1):108–143, 2000.
14. Pérez–Jiménez, M.J., Sancho, F., Graciani, C. and Romero, A.: Soluciones moleculares del problema SAT (in spanish). Lógica, Lenguaje e Información, JOLL'2000, pp. 243–252. Ed. Kronos, 2000.
15. Russinoff, D.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS J. of Comp. Math., vol. 1, pp. 148–200, 1998.