Formalizing Simplicial Topology in ACL2

M. Andrés, L. Lambán, J. Rubio
Departamento Matemáticas y Computación
Universidad de La Rioja
Edificio Vives, Luis de Ulloa s/n
26004 Logroño, Spain
{mirian.andres,lalamban,
julio.rubio}@unirioja.es

J. L. Ruiz-Reina
Departamento de Ciencias de la Computación e
Inteligencia Artificial
Escuela Técnica Superior de Ingeniería
Informática. Universidad de Sevilla
Avda. Reina Mercedes s/n
41012 Sevilla, Spain
jruiz@us.es

ABSTRACT
This paper presents an approach to formally analyze concepts and algorithms in the mathematical domain of Simplicial Topology. Our aim is twofold. First to show, by means of an elementary example, that it can be feasible, and even natural, to undertake that formalization in ACL2, since most of the theorems in Simplicial Topology can be seen as theorems about list manipulation. It is also worth pointing out how this example can be also proved reusing some previously proved results about abstract reduction systems. Second, to sketch a methodology for using ACL2 to increase reliability of a previously existing symbolic computation system for Simplicial Topology, written in Common Lisp.

1. INTRODUCTION
Kenzo is a Common Lisp program [3] designed by F. Sergeraert, implementing his ideas on *Constructive Algebraic Topology* [8]. Several years ago, a project was launched to analyze the Kenzo system by means of formal methods. The first efforts were devoted to the Algebraic Specification of Kenzo (see, for instance, [5]). After that, these rather theoretical results were put into practice through *theorem provers*. The tactical assistant Isabelle [7] was chosen for the first studies [2].

The main limitation when using Isabelle for this task is the distance from the theories and proofs built in Isabelle, to the real Kenzo code. Kenzo is programmed in Common Lisp; thus, the idea of using ACL2 to verify the *actual* Kenzo programs is quite appealing. Nevertheless, this approach is quickly seen as incomplete: Kenzo intensively uses higher order functional programming, and therefore it cannot be simply translated to ACL2. Even so, one can imagine different strategies to face the problem of increasing the reliability of Kenzo by means of ACL2. One of them is to choose a convenient *first-order* fragment of Kenzo and to reprogram and verify it in ACL2. This is the proposal we are introducing in this paper, with respect to Simplicial Topology, an essential mathematical tool in Kenzo. (It is worth noting that the previous works in Isabelle are rather oriented to the *algebraic* part of Kenzo.) These new programs, written in ACL2, are very closely related to, but different from, its corresponding full-Common Lisp, Kenzo counterparts. Thus, a distance between our formal (ACL2) model and the *actual* running program is still present. In [1] a methodology to overcome this difficulty was introduced. This methodology is applied in this paper to the case of Simplicial Topology.

The organization of the paper is as follows. The next section introduces Simplicial Topology very briefly and explains how its rather abstract mathematical concepts can be smoothly formalized in ACL2. To illustrate this, Section 3 focuses on an example, giving the statement of an elementary property, and then, in Subsection 3.2, reporting on a direct proof in ACL2 of it. Then, in Subsection 3.3, an alternative ACL2 proof is commented on, which has the unexpected property of being based on abstract reduction systems. Section 4 presents our proposal to combine ACL2 certificates with previously written, and running, Common Lisp programs. The paper ends with conclusions and future work.

2. SIMPLICIAL TOPOLOGY IN ACL2
Simplicial Topology is a theory where *abstract* topological spaces are replaced by *combinatorial* artifacts, called *simplicial sets*. Then, topological spaces are recovered by means of a notion of *(geometrical) realization* of a simplicial set. The idea behind Simplicial Topology is that *algebraic* invariants associated to topological spaces are read (i.e. computed) in an easier way from combinatorial objects, as simplicial sets. (The main reference for the simplicial notions used in this paper is [6].)

Let us explain briefly the nature of this moving from the *abstract* world of topological spaces to the *discrete* world of simplicial objects. In Figure 1 we have drawn a topological space, which is composed by a sphere and other connected figures. The important thing to be stressed is that to define such a topological space, which is always of infinite cardinality, we should give some description indicating, not only the elements belonging to the space, but also the *topology* to be considered on them. This description can be done in different ways (for instance, giving a complete collection of its *o*pen subsets), but always in a cumbersome manner.

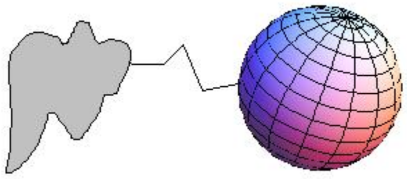

Figure 1: A topological space

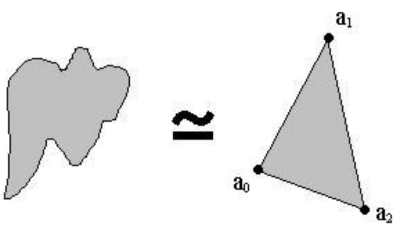

Figure 2: Model obtained by triangulation

The next aspect to be pointed out is that, since it is a *topological* space, we are only interested in its *topological* properties. For instance, *connectedness* is such a topological property: to study whether any two points in the space can be connected by a continuous path. From this point of view, the concrete size or the appearance of the space are not relevant. We could replace the space by any other, under the condition that one can be obtained from the other through a *homeomorphism* (informally speaking: a *continuous deformation*, i.e. a deformation which does not *disrupt* the space). Thus, we could look for different *models* of a given space. These models can be designed to make easier the study of some properties. For instance, it can be very difficult to decide whether a topological space (such as that of Figure 1) is connected, if the space is described by means of its infinitely many open subsets.

A first idea to look for more suitable models is *triangulating* the space. For instance, in Figure 2 we have illustrated that a part of Figure 1 can be modeled by means

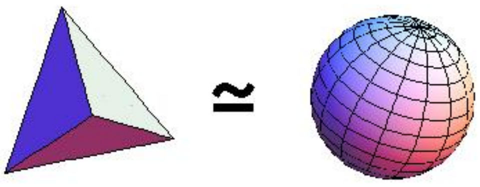

Figure 3: A tetrahedron is (topologically) a sphere

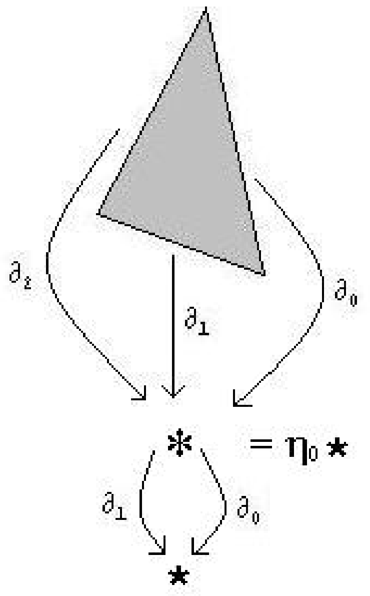

Figure 4: A smaller model for the sphere

of a triangle (without losing any topological information), and in Figure 3 that a sphere can be represented by means of a (hollow) tetrahedron. Once a space has been triangulated, it is very easy to pass to a *combinatorial* model. Coming back to Figure 2, the triangle can be described just by means of a list (a_0, a_1, a_2), understanding that its three faces (or three *sides*, in this case) are obtained in the following way: $\partial_0(a_0, a_1, a_2) = (a_1, a_2)$, $\partial_1(a_0, a_1, a_2) = (a_0, a_2)$ and $\partial_2(a_0, a_1, a_2) = (a_0, a_1)$. That is to say, the ∂_i is denoting the edge opposite to the i-th vertex. This structure is nothing but a natural generalization of the notion of a *graph*. Thus, the two "faces" of each edge are defined in an analogous way: $\partial_0(a_1, a_2) = (a_2)$, $\partial_1(a_1, a_2) = (a_1)$, and so on. The essential relationship explaining that this combinatorial representation (by means of lists of symbols) is really describing a triangle is captured by the formula: $\partial_i\partial_j = \partial_{j-1}\partial_i$, if $i < j$. For instance: $\partial_0\partial_1(a_0, a_1, a_2) = (a_2) = \partial_0\partial_0(a_0, a_1, a_2)$. These equalities describe the *adjacency* relations among the different (symbolic) elements in our combinatorial model.

Now, to go from the discrete to the continuous, we should choose for each vertex a_i a concrete point in a Euclidean space, for each edge a concrete segment (respecting the adjacency relations), and so on. This gives the previously evoked *geometrical realization* of our combinatorial model (see [6] for details). Let us remark that the problem on the connectedness of the space that was presented as difficult in the abstract context, is very easy in the combinatorial setting: a traversal algorithm allows us to compute the number of connected components.

Once it has been accepted that replacing topological spaces by combinatorial models opens the possibility of algorithmically computing topological invariants (as the number of connected components), the next question is efficiency. It is quite clear that the complexity of the algorithms depends on the number of elements in the model. From this point of view, the modelling of a sphere by means of a (hollow) tetrahedron, as in Figure 3, is quite verbose: 4 vertices, 6 edges, 4 triangles (14 elements). Since the topological notions are quite flexible, we could imagine a much more efficient way of representing a sphere. We could think in a triangle where all the edges and vertices are collapsed to just one point (this gives as figure a sort of "parachute"); see Figure 4. This would allows us to represent the sphere just with two essential elements (the triangle and the collapsing point), instead of the 14 elements needed in the thetrahedron.

The problem with this new respresentation of the sphere is there is a "dimension jump": there is one element of dimension 2 (the triangle) and another of dimension 0 (the point) but nothing in dimension 1. This seems to be contradictory with the faces system described previously, when commenting on Figure 2, because a face operator ∂_i decreases in one unit the dimension. The great idea of simplicial sets is that we can "invent" new combinatorial elements, without any topological meaning (that means that they produce nothing in the geometrical realization), but allowing us to maintain the "faces system", and thus the algorithmic treatment. In the example of Figure 4, the point $*$ is "repeated" in dimension 1, by means of the new expression $\eta_0(*)$. If the triangle is denoted by the symbol x, we would define its faces as $\partial_0 x = \partial_1 x = \partial_2 x = \eta_0(*)$. This *invented* element $\eta_0(*)$ is called a *degeneration* of the vertex $*$.

In order to discover which axiomatic presentation is necessary to keep coherent these new degenerated elements, which are included without any geometrical meaning, we can come back to the example of Figure 2. Let us define

$$\eta_0(a_0, a_1, a_2) := (a_0, a_0, a_1, a_2),$$

$$\eta_1(a_0, a_1, a_2) := (a_0, a_1, a_1, a_2),$$

$$\eta_2(a_0, a_1, a_2) := (a_0, a_1, a_2, a_2),$$

and similarly in other dimensions (that is to say, the operator η_i is repeating the i-th element in the list). Then the equalities relating ∂_i and η_i are summarized in the following definition of *simplicial set.*

Definition 1. A *simplicial set* K consists of a graded set $\{K_q\}_{q\in\mathbb{N}}$ and, for each pair of integers (i, q) with $0 \leq i \leq q$, *face* and *degeneracy* maps, $\partial_i : K_q \to K_{q-1}$ and $\eta_i : K_q \to K_{q+1}$, satisfying the *simplicial identities*:

$$\begin{aligned}
&\partial_i\partial_j = \partial_{j-1}\partial_i && \text{if } i < j\\
&\eta_i\eta_j = \eta_{j+1}\eta_i && \text{if } i \leq j\\
&\partial_i\eta_j = \eta_{j-1}\partial_i && \text{if } i < j\\
&\partial_i\eta_j = \mathrm{Id} && \text{if } i = j \text{ or } i = j+1\\
&\partial_i\eta_j = \eta_j\partial_{i-1} && \text{if } i > j+1
\end{aligned}$$

The elements of K_q are called *q-simplices.* A q-simplex x is *degenerate* if $x = \eta_i y$ with $y \in K_{q-1},\ 0 \leq i < q$; otherwise x is called *non-degenerate.* (For instance, 0-simplices can be tought as vertices, non-degenerate 1-simplices as edges, non-degenerate 2-simplices as (filled) triangles, non-degenerate 3-simplices as (filled) tetrahedra, etc.)

Let us note that, in fact, each operator as ∂_i or η_i depends on q (so, it should be more appropriate to denote them by ∂_i^q or η_i^q). Nevertheless, it is usual to skip the dimension q on notations, and we will see soon that it is convenient in our cases of interest, too.

Kenzo can deal with simplicial sets in a quite general way: infinite simplicial sets can be encoded, and operations on simplicial sets are built-in (for instance, given two simplicial sets K and L, Kenzo can construct its cartesian product $K \times L$, a new simplicial set). To this aim, Kenzo is based on higher-order functional programming, implementing each simplicial set as a record of lambda expressions.

Fortunately, in order to start formalizing Simplicial Topology in ACL2 it is not necessary to emulate this Common Lisp organization of Kenzo. The reason is theoretical: there exists a *universal* simplicial set Δ (see [6]) where the initial studies on Simplicial Topology can be carried out. This simplicial set Δ contains the minimal number of identifications from the equalities introduced in Definition 1. That is to say, any theorem proved on Δ by using *only* the equalities of Definition 1, will be also true for any other simplicial set K.

In our ACL2 context, the simplicial set Δ can be easily formalized: a q-simplex of Δ is any ACL2 list of length $q + 1$. The face operators are defined by means of a function `(del-nth i l)`, which eliminates the `i`-th element in the list `l`. The degeneracy operators are similarly defined by means of a function `(deg i l)`, which repeats the `i`-th element in the list `l`. Thus, as previously announced, the face and degeneracy operators can be defined in ACL2 regardless of the dimension of the simplices (i.e. the length of the lists). (Formally speaking, we are considering the simplicial set freely generated from the set of all ACL2 objects.)

In this manner, we could reduce the formalization of a fragment of the theory of Simplicial Topology to proving properties in ACL2, dealing with simple data structures such as lists. In the next section, we explore this possibility by means of a concrete and elementary example.

3. AN EXAMPLE

3.1 Statement of the theorem

One of the fundamental results in Simplicial Topology (it is so fundamental that... its proof is given as an exercise to the reader! [6]) is the following.

Theorem 1. Let K be a simplicial set. Any degenerate n-simplex $x \in K_n$ can be expressed in a unique way as a (possibly) iterated degeneracy of a non-degenerate simplex y in the following way:

$$x = \eta_{j_k} \dots \eta_{j_1} y$$

with $y \in K_r,\ k = n - r > 0,\ 0 \leq j_1 < \cdots < j_k < n.$

In order to model it in ACL2, let us first note that a non-degenerate simplex in Δ is a list where any two consecutive elements are different. Besides, a simplex in Δ can be represented in another way: as a pair of lists, the first element being a list of natural numbers (let us call it *degeneracy list*) and the second one any ACL2 list. Such a pair is representing the simplex obtained by repeatedly applying the degeneracy operators `deg` using as indexes those of the degeneracy list. The corresponding ACL2 function is `(degenerate dl l)`. With these definitions and notations, the previous theorem can be stated, in the case of Δ, in the following way.

Theorem 2. Any ACL2 list l can be expressed in a unique way as a pair (dl, l') such that:

$$l = degenerate(dl, l')$$

with l' without two consecutive elements equal and dl a strictly increasing degeneracy list.

3.2 A direct ACL2 proof

With the statement of the previous theorem in mind, we can define the following function `generate` to obtain the witnesses dl and l' from l.

```
(defun generate (l)
  (if (or (endp l) (endp (cdr l)))
      (cons nil l)
    (let ((gencdr (generate (cdr l))))
      (if (equal (first l) (second l))
          (cons (cons 0 (add-one (car gencdr)))
                (cdr gencdr))
        (cons (add-one (car gencdr))
              (cons (car l) (cdr gencdr)))))))
```

Note that the method of this function relies on eliminating consecutive repetitions in l, adding properly the corresponding indexes to dl (the function `add-one` adds one to every element of its input list.)

As for the existence part, we established the following, where the function `canonical` encodes all the properties required in Theorem 2 (strictly increasing degeneracy list, for example):

```
(defthm existence
  (let ((gen (generate l)))
    (and (canonical gen)
      (equal (degenerate (car gen) (cdr gen)) l))))
```

The proof is achieved by standard interaction with ACL2 by providing the proper lemmas and the suitable induction schemes.

In order to prove that the pair (dl, l') is unique, we establish the following lemma:

```
(defthm uniqueness-main-lemma
  (implies (canonical (cons l1 l2))
           (equal (generate (degenerate l1 l2))
                  (cons l1 l2))))
```

Problems appeared in the proof of the above lemma because the lists obtained after rewriting `(generate (degenerate l1 l2))` in

`(generate (degenerate (cdr l1) (deg (car l1) l2)))`,

do not satisfy the hypotheses of the theorem, so it was not possible to apply a simplified induction scheme on them. We solved it by means of some elaborated lemmas which transform the expressions until reaching a suitable situation where induction is applicable.

With the above lemma, it is now easy to prove the uniqueness part of Theorem 2, stated as follows:

```
(defthm uniqueness
  (implies
    (and (canonical p1) (canonical p2)
         (equal (degenerate (car p1) (cdr p1)) l)
         (equal (degenerate (car p2) (cdr p2)) l))
    (equal p1 p2)))
```

3.3 An abstract reduction systems approach

The direct proof of Theorem 2 which has just been outlined is not completely satisfactory since it does not explicitly use the face operators and is not directly based on the combinatorial properties which relate the face and the degeneracy maps. To take advantage of these relationships (and thus to carry out a formalization more closely related to the source concept), we present an alternative proof. The idea is to consider the elimination of a consecutive repetition in a list (by applying the corresponding face operator) as a simple reduction step; also, another type of reduction step can be used to "fix" disorders in the degeneracy list.

To formalize this idea, we define the following reduction relation $\rightarrow_S$, whose domain and rules are given as follows. The set of S-terms is the set of pairs (l_1, l_2) where l_1 is intended to be a list of natural numbers, and l_2 is any list. Two types of reduction rules are considered in $\rightarrow_S$:

- *o-reduction*: if the list l_1 has a "disorder" at position i, i.e., $l_1(i) \geq l_1(i+1)$, then $(l_1, l_2) \rightarrow_S (l'_1, l_2)$, where $l'_1(i) = l_1(i+1)$ and $l'_1(i+1) = l_1(i) + 1$ (here $l(j)$ denotes the j-th element of l).
- *r-reduction*: if at index i there is a repetition in l_2, i.e., $l_2(i) = l_2(i+1)$, then $(l_1, l_2) \rightarrow_S (l'_1, l'_2)$, where $l'_1 = cons(i, l_1)$ and $l'_2 = del\text{-}nth(i, l_2)$

These reductions are very closely linked to the equalities in Definition 1. Thus, *o-reductions* reflects exactly the equality $\eta_i\eta_j = \eta_{j+1}\eta_i$, if $i \leq j$. In an analogous way, *r-reductions* are related to $\partial_i\eta_i = \mathrm{Id}$. The iteration of these reductions gives us "paths" relating different representations of the same simplex in Δ. The "abstract reduction approach" allows us to concentrate just on local applications of the reductions instead of studying the whole class of representations as in the "direct approach". This feature is related to the well-known Newman's lemma, as explained in the sequel.

We modeled $\to_S$ in the framework of an existing ACL2 formalization about abstract reduction systems [9]. In our case, *operators* are represented as pairs of the form (t, i), where t is `'o` or `'r` (depending on the type of the reduction rule) and i is the position in the list where the corresponding reduction takes place. The relation $\to_S$ is then represented by means of two functions `(s-legal x op)` and `(s-reduce-one-step x op)`, defining, respectively, the conditions needed to apply a given operator `op` to the pair of lists `x`, and the result of applying a "legal" operator. See the supporting materials for their definitions. In [9], it is shown how these two functions suffice to represent a reduction and other related concepts such as noetherianity, equivalence closures, normal forms or confluence.

First, we proved that the reduction is noetherian (that is, there is no infinite sequence of S-reductions) using a suitable lexicographic measure. As a consequence, we can define the following function `s-normal-form`, that computes a normal form with respect to $\to_S$ (the function `s-reducible` returns a legal operator, whenever it exists, `nil` otherwise):

```
(defun s-normal-form (x)
  (let ((red (s-reducible x)))
   (if red
       (s-normal-form (s-reduce-one-step x red))
     x)))
```

Let $\overset{*}{\to}$ denote the reflexive-transitive closure of $\to$. A reduction is said to be *locally confluent* if whenever $y \leftarrow x \to z$ (a *local peak*) then $y \overset{*}{\to} u \overset{*}{\leftarrow} z$ (a *valley*). The reduction $\to_S$ is locally confluent, as established by the following theorem:

```
(defthm local-confluence
  (implies (and (s-equiv-p x y p) (local-peak-p p))
    (and (s-equiv-p x y (s-transform-local-peak p))
       (steps-valley (s-transform-local-peak p)))))
```

This theorem needs some explanation. The function called `s-equiv-p` formalizes the equivalence closure of $\to_S$ (usually denoted as $=_S$); that is, the connection of `x` and `y` by a sequence of (zero or more) $\to_S$ reduction steps, either applied from left to right or from right to left. This sequence of reductions is explicitly represented by its third argument `p`. The functions `local-peak-p` and `steps-valley` check the corresponding shape of a sequence of steps. Note that in this context, local confluence is justified by the definition of a function `s-transform-local-peak` that explicitly constructs a valley from a given local peak. This function is defined dealing with all the possible combinations of positions and types of the two S-reductions in a local peak.

As a consequence of Newman's lemma (a result on abstract reductions), every noetherian and locally confluent reduction is convergent [4], which in particular means that two equivalent elements (w.r.t. the equivalence closure of the reduction) have a common normal form. In our case, this is established by the following theorem:

```
(defthm s-reduction-convergent
  (implies (s-equiv-p x y p)
    (equal (s-normal-form x) (s-normal-form y))))
```

Now, the main relation between our reduction relation $\to_S$ and the function `degenerate` is given by the following two properties: (a) If $(l_1, l_2) \to_S (l_3, l_4)$, then $degenerate(l_1, l_2) = degenerate(l_3, l_4)$. And (b) If $degenerate(l_1, l_2) = l$, then $(nil, l) =_S (l_1, l_2)$.

For example the following is the ACL2 formalization of property (b) above (we omit some technical conditions, for the sake of clarity). Note that the function `degenerate-steps` explicitly constructs the sequence of reduction steps connecting (nil, l) and (l_1, l_2):

```
(defthm degenerate-s-equivalent
  (implies ....
    (s-equiv-p (cons l m)
               (cons nil (degenerate l m))
               (degenerate-steps l m))))
```

Now we simply define `(generate l)` as `(s-normal-form (cons nil l)))`. With this definition, and given the previous properties explained above, it is not difficult to prove the theorems `existence` and `uniqueness` exactly as stated in Subsection 3.2 (but w.r.t. this alternative definition of `generate`). As an interesting corollary, it can be proved that both definitions of `generate` are equivalent.

4. THE ROLE OF ACL2 FOR INCREASING RELIABILITY IN COMPUTER ALGEBRA PROGRAMS

In order to explain the differences between the ACL2 implementation sketched in the previous sections, and the real Kenzo implementation, let us introduce some terminology. In Kenzo, there is a distinction between *geometrical simplices* and *abstract simplices*. Roughly speaking, the first ones correspond to lists in Δ without two equal consecutive elements. The second ones are pairs (dg, l') where dg is a degeneracy list and l' is a geometrical simplex. The theorem of Section 3 explains that the representation of simplicial sets by means of abstract simplices is complete and faithful (i.e. every element of K can be represented in a unique way as an abstract simplex). As an example, and going back to the space described in Figure 4 (and with the notations introduced in Section 2), in that simplicial set there is just two *geometrical simplices*: the 2-simplex x and the 0-simplex $*$. There are infinitely many *abstract simplices* that can be denoted by $((), *)$, $((0), *)$, $((), x)$, $((0, 1), *)$, $((0), x)$, $((0, 1, 2), *)$, etc.

Operations with degeneracy lists are very frequent in Kenzo, and, in fact, this kind of manipulation is one of the sources of the exponential complexity of the algorithms implemented in Kenzo. Thus, to be efficient there is specially important. Sergeraert devised a smart implementation of degeneracy lists by coding them as binary numbers and then handling them by means of the fast Common Lisp functions `ash`, `logxor` and the like. This arithmetical implementation is very different from the more explicit one (more geometrical, let us say) presented here. Therefore, proving properties in ACL2 is still different from proving the correctness of Kenzo in this particular area.

Similar but more complicated situations can be found in other parts of Kenzo, where the occurrence of Common Lisp features which are not part of ACL2 (for instance, higher-order functional or object-oriented programming) precludes a direct modeling in ACL2 (let us note that it is not the case in the previous example: since the functions as `ash` and `logxor` are also in ACL2, we could undertake a direct modeling of if in ACL2). Such situations are reported on in [1]. There, we advocated by using ACL2 proofs to prepare an *automated testing* of some parts of Kenzo. The scenario is as follows, illustrated by means of examples from our simplicial setting. Given a program included in Kenzo (let us imagine, the function composing arithmetically two degeneracy lists), we establish an ACL2 *model* with the same *intended* behavior (for instance, we give in ACL2 a direct implementation of the composing, by means of lists of natural numbers). Then, we prove *in ACL2* the correctness of the ACL2 implementation (instead of proving the correctness of the arithmetical version, likely much harder to be verified in ACL2). And finally, we test the *actual* Kenzo program *against* the ACL2 certified model. The function achieving the testing should be, in general, a Common Lisp non-ACL2 program (so, without formal verification), but even if a proof of correctness is not considered, we think that a well-designed testing strategy could considerably increase the reliability of the Kenzo system. We think that this kind of approach is especially relevant in the parts of Kenzo where it is capable of finding algebraic invariants that have been not determined before, by any other means (theoretical or automatic).

5. CONCLUSIONS AND FURTHER WORK

In this short note, some ideas to apply ACL2 in Simplicial Topology have been presented. The main contributions are: an analysis of feasibility (by means of an elementary example) and a surprising remark relating ACL2 proofs in Simplicial Topology with abstract rewriting systems. The interest of this kind of research is due to its application to increase the reliability of a real Computer Algebra program called Kenzo [3].

The future work would be oriented to formalize and prove in ACL2 more difficult results from Simplicial Topology. Our main objective would be to give an ACL2 proof of the Eilenberg-Zilber theorem [6], which is essential in Kenzo since it establishes the bridge between Geometry and Algebra in Algebraic Topology.

6. ACKNOWLEDGMENTS

Partially supported by MEC, projects TIN2004-03884 and MTM2006-06513, and by Comunidad Autónoma de La Rioja, project ANGI-2005/19. We will like to thank Jónathan Heras for his design of the graphics in this paper.

7. REFERENCES

[1] M. Andrés, L. Lambán, and J. Rubio. Executing in Common Lisp, Proving in ACL2. In *Proceedings MKM/Calculemus 2007*, pages 1–12. LNAI **4573**, Springer, 2007.

[2] J. Aransay, C. Ballarin, and J. Rubio. Four Approaches to Automated Reasoning with Differential Algebraic Structures. In *Proceedings AISC 2004*, pages 221–234. LNCS **2349**, Springer, 2004.

[3] X. Dousson, F. Sergeraert, and Y. Siret. *The Kenzo Program, 1999–2007*. Institut Fourier, Grenoble, `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`, 2007.

[4] G. Huet. Confluent reduction : abstract properties and applications to term rewrite systems. *Journal of the Association for Computing Machinery*, 27(4):787–821, 1980.

[5] L. Lambán, V. Pascual, and J. Rubio. An Object-Oriented Interpretation of the EAT System. *Applicable Algebra in Engineering, Communication and Computing*, 14(3):187–215, 2003.

[6] J. P. May. *Simplicial Objects in Algebraic Topology*, volume 11 of *Mathematical Studies*. Van Nostrand, 1967.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logic*. Lecture Notes in Computer Science **2283**, Springer, 2002.

[8] J. Rubio and F. Sergeraert. Constructive Algebraic Topology. *Bulletin Sciences Mathématiques*, 126:389–412, 2002.

[9] J. L. Ruiz-Reina, J. A. Alonso, M. J. Hidalgo, and F. J. Martín-Mateos. Formal Proofs About Rewriting Using ACL2. *Annals of Mathematics and Artificial Intelligence*, 36(3):239–262, 2002.