Chapter 1

Preliminaries

In this chapter we introduce some preliminaries that we will use in the rest of this memoir. The first section is devoted to the main mathematical notions employed in this work. The Kenzo system, a successful Common Lisp program developed by Francis Sergeraert and some coworkers devoted to perform computations in Algebraic Topology, is presented in Section 1.2. Finally, the *deduction* machinery employed in this memoir, the ACL2 system, is briefly presented in Section 1.3.

1.1 Mathematical preliminaries

Algebraic Topology is a vast and complex subject, in particular mixing Algebra and (combinatorial) Topology. Algebraic Topology consists in trying to use as much as possible "algebraic" methods to attack topological problems. For instance, one can define some special groups associated with a topological space, in a way that respects the relation of homeomorphism of spaces. This allows us to study properties about topological spaces by means of statements about groups, which are often easier to prove.

1.1.1 Homological Algebra

The following basic definitions can be found, for instance, in [Mac63].

Definition 1.1. Let R be a ring with a unit element $1 \neq 0$. A *left R-module M* is an additive abelian group together with a map $p : R \times M \to M$, denoted by $p(r, m) \equiv rm$, such that for every $r, r' \in R$ and $m, m' \in M$

$$\begin{aligned}(r + r')m &= rm + r'm \\ r(m + m') &= rm + rm' \\ (rr')m &= r(r'm) \\ 1m &= m\end{aligned}$$

A similar definition is given for a *right R-module.*

For $R = \mathbb{Z}$ (the integer ring), a $\mathbb{Z}$-module M is simply an abelian group. The map $p : \mathbb{Z} \times M \to M$ is given by

$$p(n, m) = \begin{cases} m + \overset{n}{\cdots} + m & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ (-m) + \overset{n}{\cdots} + (-m) & \text{if } n < 0 \end{cases}$$

Definition 1.2. Let R be a ring and M and N be R-modules. An *R-module morphism* $\alpha : M \to N$ is a function from M to N such that for every $m, m' \in M$ and $r \in R$

$$\begin{aligned} \alpha(m + m') &= \alpha(m) + \alpha(m') \\ \alpha(rm) &= r\alpha(m) \\ \alpha(0_M) &= 0_N \end{aligned}$$

Definition 1.3. Given a ring R, a *graded module* M is a family of left R-modules $(M_n)_{n\in\mathbb{Z}}$.

Definition 1.4. Given a pair of graded modules M and M', a *graded module morphism* f of degree k between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ such that $f_n : M_n \to M'_{n+k}$ for all $n \in \mathbb{Z}$.

Definition 1.5. Given a graded module M, a *differential* $(d_n)_{n\in\mathbb{Z}}$ is a family of module endomorphisms of M of degree -1 such that $d_{n-1} \circ d_n = 0$ for all $n \in \mathbb{Z}$.

From the previous definitions, the notion of chain complex can be defined. Chain complexes are the central notion in Homological Algebra and can be seen as an algebraic means to study properties of topological spaces in several dimensions.

Definition 1.6. A *chain complex* C_* is a family of pairs $(C_n, d_n)_{n\in\mathbb{Z}}$ where $(C_n)_{n\in\mathbb{Z}}$ is a graded module and $(d_n)_{n\in\mathbb{Z}}$ is a differential of C_*.

The module C_n is called the module of *n-chains.* The image $B_n = \operatorname{Im} d_{n+1} \subseteq C_n$ is the (sub)module of *n-boundaries.* The kernel $Z_n = \operatorname{Ker} d_n \subseteq C_n$ is the (sub)module of *n-cycles.*

In many situations the ring R is the integer ring, $R = \mathbb{Z}$. In this case, a chain complex C_* is given by a graded abelian group $\{C_n\}_{n\in\mathbb{Z}}$ and a graded group morphism of degree -1, $\{d_n : C_n \to C_{n-1}\}_{n\in\mathbb{Z}}$, satisfying $d_{n-1} \circ d_n = 0$ for all n. From now on in this memoir, we will work with $R = \mathbb{Z}$.

Let us present some examples of chain complexes.

Example 1.7.

- The *unit chain complex* has a unique non null component, namely a $\mathbb{Z}$-module in degree 0 generated by a unique generator and $d_n = 0$ for all $n \in \mathbb{Z}$.

- A chain complex to model the *circle* is defined as follows. This chain complex has two non null components, namely a $\mathbb{Z}$-module in degree 0 generated by a unique generator and a $\mathbb{Z}$-module in degree 1 generated by another generator; and the differential is the null map.
- The *diabolo* chain complex has associated three chain groups:
  - C_0, the free $\mathbb{Z}$-module on the set $\{s_0, s_1, s_2, s_3, s_4, s_5\}$.
  - C_1, the free $\mathbb{Z}$-module on the set $\{s_{01}, s_{02}, s_{12}, s_{23}, s_{34}, s_{35}, s_{45}\}$.
  - C_2, the free $\mathbb{Z}$-module on the set $\{s_{345}\}$.

  and the differential is provided by:
  - $d_0(s_i) = 0$,
  - $d_1(s_{ij}) = s_j - s_i$,
  - $d_2(s_{ijk}) = s_{jk} - s_{ik} + s_{ij}$.

  and it is extended by linearity to the combinations $c = \sum_{i=1}^{m} \lambda_i x_i \in C_n$.

We can construct chain complexes from other chain complexes, applying constructors such as the direct sum or the tensor product.

Definition 1.8. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}, D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes. The *direct sum* of C_* and D_* is the chain complex $C_* \oplus D_* = (M_n, d_n)_{n\in\mathbb{Z}}$ such that, $M_n = (C_n, D_n)$ and the differential map is defined on the generators (x, y) with $x \in C_n$ and $y \in D_n$ by $d_n((x, y)) = (d_{C_n}(x), d_{D_n}(y))$ for all $n \in \mathbb{Z}$. The notion of direct sum can be generalized to a collection of chain complexes.

Definition 1.9. Let M be a right R-module, and N a left R-module. The *tensor product* $M \otimes_R N$ is the abelian group generated by the symbols $m \otimes n$ for every $m \in M$ and $n \in N$, subject to the relations

$$\begin{aligned}(m + m') \otimes n &= m \otimes n + m' \otimes n\\ m \otimes (n + n') &= m \otimes n + m \otimes n'\\ mr \otimes n &= m \otimes rn\end{aligned}$$

for all $r \in R,\ m, m' \in M$, and $n, n' \in N$.

If $R = \mathbb{Z}$ (the integer ring), then M and N are abelian groups and their tensor product will be denoted simply by $M \otimes N$.

Definition 1.10. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes of right and left $\mathbb{Z}$-modules respectively. The *tensor product* $C_* \otimes D_*$ is the chain complex of $\mathbb{Z}$-modules $C_* \otimes D_* = ((C_* \otimes D_*)_n, d_n)_{n\in\mathbb{Z}}$ with

$$(C_* \otimes D_*)_n = \bigoplus_{p+q=n} (C_p \otimes D_q)$$

where the differential map is defined on the generators $x \otimes y$ with $x \in C_p$ and $y \in D_q$, according to the Koszul rule for the signs, by

$$d_n(x \otimes y) = d_{C_p}(x) \otimes y + (-1)^p x \otimes d_{D_q}(y)$$

Let us present now, one of the most important invariants used in Homological Algebra. Given a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$, the identities $d_{n-1} \circ d_n = 0$ mean the inclusion relations $B_n \subseteq Z_n$: every boundary is a cycle (the converse in general is not true). Thus the next definition makes sense.

Definition 1.11. Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex of R-modules. For each degree $n \in \mathbb{Z}$, the *n-homology module* of C_* is defined as the quotient module

$$H_n(C_*) = \frac{Z_n}{B_n}$$

It is worth noting that the homology groups of a space X are the ones of its associated chain complex $C_*(X)$; the way of constructing the chain complex associated with a space X is explained, for instance, in [Mau96]. In an intuitive sense, homology groups measure "n-dimensional holes" in topological spaces. A 0-dimensional hole is a pair of points in different path components; and so H_0 measures the number of connected components of a space. The homology groups H_n measure higher dimensional connectedness. For instance, the n-sphere, S^n, has exactly one n-dimensional hole and no m-dimensional holes if $m \neq n$.

Moreover, it is worth noting that homology groups are an algebraic invariant, see [Mau96]. That is to say, if two topological spaces are homeomorphic, this means that all their homology groups coincide.

Let us finish this section with some additional definitions related to chain complexes.

Definition 1.12. A chain complex $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ is *acyclic* if $H_n(C_*) = 0$ for all n, that is to say, if $Z_n = B_n$ for every $n \in \mathbb{Z}$.

Definition 1.13. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}, D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes, a *chain complex morphism* between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ of degree 0 between $(C_n)_{n\in\mathbb{Z}}$ and $(D_n)_{n\in\mathbb{Z}}$ such that $d'_n \circ f_n = f_{n-1} \circ d_n$ for each $n \in \mathbb{Z}$.

Definition 1.14. Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex. A chain complex $D_* = (D_n, d'_n)_{n\in\mathbb{Z}}$ is a *chain subcomplex* of C_* if

- D_n is a submodule of C_n, for all $n \in \mathbb{Z}$
- $d'_n = d_n \mid_{D_*}$

The condition $d'_n = d_n \mid_{D_*}$ means that the boundary operator of the chain subcomplex is just the differential operator of the larger chain complex restricted to its domain. We denote $D_* \subset C_*$ if D_* is a chain subcomplex of C_*.

Definition 1.15. A short exact sequence is a sequence of modules:

$$0 \leftarrow C'' \overset{j}{\leftarrow} C \overset{i}{\leftarrow} C' \leftarrow 0$$

which is exact, that is, the map i is injective, the map j is surjective and Im $i =$ Ker j.

From now on in this memoir, we will work with non-negative chain complexes, that is to say, $\{C_n\}_{n\in\mathbb{Z}}$ such that $C_n = 0$ if $n < 0$. A non-negative chain complex C_* will be denoted by $C_* = \{C_n\}_{n\in\mathbb{N}}$. Moreover, the chain complexes we work with are supposed to be free.

Definition 1.16. A chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ of $\mathbb{Z}$-modules is said to be *free* if C_n is a free $\mathbb{Z}$-module (a $\mathbb{Z}$-module which admits a basis) for each $n \in \mathbb{N}$.

1.1.2 Simplicial Topology

1.1.2.1 Simplicial Sets

Simplicial sets were first introduced by Eilenberg and Zilber [EZ50], who called them *semi-simplicial complexes*. They can be used to express some topological properties of spaces by means of combinatorial notions. A good reference for the definitions and results of this section is [May67].

Definition 1.17. A *simplicial set* K, is a union $K = \bigcup_{q\geq 0} K^q$, where the K^q are disjoints sets, together with functions:

$$\begin{array}{lll} \partial_i^q : K^q \to K^{q-1}, & q > 0, & i = 0, \dots, q, \\ \eta_i^q : K^q \to K^{q+1}, & q \geq 0, & i = 0, \dots, q, \end{array}$$

subject to the relations:

$$\begin{array}{llcccc} (1) & \partial_i^{q-1}\partial_j^q & = & \partial_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (2) & \eta_i^{q+1}\eta_j^q & = & \eta_j^{q+1}\eta_{i-1}^q & \text{if} & i > j, \\ (3) & \partial_i^{q+1}\eta_j^q & = & \eta_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (4) & \partial_i^{q+1}\eta_i^q & = & \textit{identity} & = & \partial_{i+1}^{q+1}\eta_i^q, \\ (5) & \partial_i^{q+1}\eta_j^q & = & \eta_j^{q-1}\partial_{i-1}^q & \text{if} & i > j+1. \end{array}$$

The ∂_i^q and η_i^q are called *face* and *degeneracy* operators respectively.

The elements of K^q are called *q-simplexes*. A simplex x is called *degenerate* if $x = \eta_i y$ for some simplex y and some degeneracy operator η_i; otherwise x is called *non degenerate*. An example of simplicial sets, that can be useful to clarify some notions, is the *standard simplicial set of dimension* m, $\Delta[m]$.

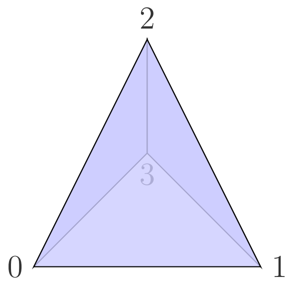


Figure 1.1: non degenerate simplexes of the standard simplicial set $\Delta[3]$

Definition 1.18. For $m \geq 0$, the *standard simplicial set of dimension* m, $\Delta[m]$, is a simplicial set built as follows. An n-simplex of $\Delta[m]$ is any $(n+1)$-tuple $(a_0, \dots, a_n)$ of integers such that $0 \leq a_0 \leq \cdots \leq a_n \leq m$, and the face and degeneracy operators are defined as

$$\partial_i(a_0, \dots, a_n) = (a_0, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$
$$\eta_i(a_0, \dots, a_n) = (a_0, \dots, a_{i-1}, a_i, a_i, a_{i+1} \dots, a_n)$$

In Definition 1.18 the super-indexes in the degeneracy and face maps have been skipped, since they can be inferred from the context. It is a usual practice and will be freely used in the sequel, both for degeneracy and for face maps.

Example 1.19. Figure 1.19 shows the non degenerate simplexes of the standard simplicial set $\Delta[3]$:

- 0-simplexes (vertices): (0), (1), (2), (3);
- non degenerate 1-simplexes (edges): (0 1), (0 2), (0 3), (1 2), (1 3), (2 3);
- non degenerate 2-simplexes (triangles): (0 1 2), (0 1 3), (0 2 3), (1 2 3); and
- non degenerate 3-simplex (filled tetrahedra): (0 1 2 3).

It is worth noting that there is not any non degenerate n-simplex with $n > 3$.

Once we have presented the non degenerate simplexes, let us introduce the behavior of the face and degeneracy maps. For instance, if we apply the face and degeneracy maps over the 2-simplex (0 1 2) (for the rest of simplexes is analogous) we will obtain:

$$\partial_i((0\ 1\ 2)) = \begin{cases} (1\ 2) & \text{if } i = 0 \\ (0\ 2) & \text{if } i = 1 \\ (0\ 1) & \text{if } i = 2 \end{cases}$$

$$\eta_i((0\ 1\ 2)) = \begin{cases} (0\ 0\ 1\ 2) & \text{if } i = 0 \\ (0\ 1\ 1\ 2) & \text{if } i = 1 \\ (0\ 1\ 2\ 2) & \text{if } i = 2 \end{cases}$$

Let us note that the face operator applied over the 2-simplex (0 1 2) produces simplexes with geometrical meaning (that are the three edges of the simplex (0 1 2)). On the contrary, the simplexes obtained from applying the degeneracy operator do not have any geometrical meaning.

In the rest of the memoir a non degenerate simplex will be called *geometric* simplex, to stress that only these simplexes really have a geometrical meaning; the degenerate simplexes can be understood as *formal* artifacts introduced for technical (combinatorial) reasons. This becomes clear in the following discussion.

The next essential result, which follows from the commuting properties of degeneracy maps in the definition of simplicial sets provides a way to represent any simplex of a simplicial set in a unique manner.

Proposition 1.20. Let K be a simplicial set. Any 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 $y \in K^r$, $k = n - r \geq 0$, and $0 \leq j_1 < \dots < j_k < n$.

This proposition allows us to encode all the elements (simplexes) of *any* simplicial set in a generic way, by means of a structure called *abstract simplex*. More concretely, an *abstract simplex* is a pair (*dgop gmsm*) consisting of a sequence of degeneracy maps *dgop* (which will be called a *degeneracy operator*) and a geometric simplex *gmsm*. The indexes in a degeneracy operator *dgop* must be in strictly decreasing order. For instance, if σ is a non degenerate simplex, and σ' is the degenerate simplex $\eta_1\eta_2\sigma$, the corresponding abstract simplexes are respectively $(\emptyset\ \ \sigma)$ and $(\eta_3\eta_1\ \ \sigma)$, as $\eta_1\eta_2 = \eta_3\eta_1$, due to equality (2) in Definition 1.17.

In a similar way that we defined chain subcomplex we can define the notion of simplicial subcomplex.

Definition 1.21. Let $K = (K^n, \partial_i, \eta_i)_{n\in\mathbb{Z}}$ be a simplicial set. $L = (L^n)_{n\in\mathbb{Z}}$ is a *simplicial subcomplex* of K if

- $L^n \subset K^n$ for all $n \in \mathbb{Z}$
- $(L^n, \partial_i, \eta_i)_{n\in\mathbb{Z}}$ is a simplicial set

Let us show now, other important examples of simplicial sets that are (simplicial) models of well-known topological spaces.

Definition 1.22. For $m \geq 1$, the *sphere of dimension* m, S^m, is a simplicial set built as follows. There are just two geometric simplexes: a 0-simplex, let us denote it by $\star$, and an m-simplex, let us denote it by sm. The faces of sm are the degeneracies of $\star$; that is to say $\partial_i(sm) = \eta_{m-1}\eta_{m-2}\dots\eta_0\star$ for all $0 \leq i \leq m$.

Definition 1.23. Let $m_1, \ldots, m_n$ natural numbers such that $m_i \geq 1$ for all $1 \leq i \leq n$, the *wedge of spheres of dimensions* $m_1, \ldots, m_n$, $S^{m_1} \vee \ldots \vee S^{m_n}$, is a simplicial set built as follows. It has the following geometric simplexes: in dimension 0 a 0-simplex, let us denote it by $\star$, and in dimension p as many simplexes as the number of m_j, $1 \leq i \leq n$, such that $m_j = p$. The faces of every p-simplex are the degeneracies of $\star$; that is to say, let x a p-simplex, then $\partial_i(x) = \eta_{p-1}\eta_{p-2}\ldots\eta_0\star$ for all $0 \leq i \leq p$.

Definition 1.24. For $n > 0$ and $p > 1$, the *Moore space of dimensions* p, n, $M(\mathbb{Z}/p\mathbb{Z}, n)$, is a simplicial set built as follows. There are just three geometric simplexes: a 0-simplex, let us denote it by $\star$, an n-simplex, let us denote it by Mn, and an $n+1$-simplex, let us denote it by Mn'. The faces of Mn are the degeneracies of $\star$; moreover, p of the faces of Mn' are identified with Mn and the rest are the degeneracies of $\star$.

Definition 1.25. The *real projective plane*, $P^\infty\mathbb{R}$, is a simplicial set built as follows. In dimension n, this simplicial set has only one geometric simplex, namely the integer n. The faces of this non degenerate simplex n are given by the following formulas: $\partial_0 n = \partial_n n = n - 1$ and for $i \neq 0$ and $i \neq n$, $\partial_i n = \eta_{i-1}(n-2)$.

The *real projective plane n-dimensional*, $P^n\mathbb{R}$, is a simplicial set analogous to the model of $P^\infty\mathbb{R}$ but without simplexes in dimensions $m \geq n$.

We can also construct simplicial models of truncated real projective planes. Let $n > 1$, $P^\infty\mathbb{R}/P^{n-1}\mathbb{R}$ is a simplicial set analogous to the model of $P^\infty\mathbb{R}$ but without simplexes in dimensions $1 \leq m < n$. The faces of the n-simplex n are the degeneracies of the 0-simplex 0.

Let $n > 1$ and $l \geq n$, $P^l\mathbb{R}/P^{n-1}\mathbb{R}$ is a simplicial set analogous to the model of $P^l\mathbb{R}$ but without simplexes in dimensions $1 \leq m < n$.

The four above definitions are related to simplicial models of initial topological spaces. Moreover, we also have models for topological constructors that are applied to some spaces to obtain new ones.

Definition 1.26. Given two simplicial sets K and L, the *Cartesian product* $K \times L$ is a simplicial set with n-simplexes:

$$(K \times L)^n = K^n \times L^n$$

and for all $(x, y) \in K^n \times L^n$ the face and degeneracy operators are defined as follows:

$$\begin{aligned} \partial_i(x,y) &= (\partial_i x, \partial_i y) \quad \text{for } 0 \leq i \leq n \\ \eta_i(x,y) &= (\eta_i x, \eta_i y) \quad \text{for } 0 \leq i \leq n \end{aligned}$$

Let K be a simplicial set and $\star \in K_0$ a chosen 0-simplex (called the *base point*). We will also denote by $\star$ the degenerate simplices $\eta_{n-1}\ldots\eta_0\star \in K_n$ for every n.

Definition 1.27. A simplicial set K is said to be *reduced* (or 0-*reduced*) if K has only one 0-simplex. Given $m \geq 1$, K is *m-reduced* if there is an n-simplex per each $n \leq m$.

Definition 1.28. Given a reduced simplicial set X, the *suspension* $\mathbf{S}(X)$ is a simplicial set with a unique 0-simplex and whose n-simplexes are the $(n-1)$-simplexes of X for $n \geq 1$. We define inductively $\mathbf{S}^n(X) = \mathbf{S}(\mathbf{S}^{n-1}(X))$ for all $n \geq 1$, $\mathbf{S}^0(X) = X$.

If we impose some condition on the sets of simplicial sets we will obtain new kinds of simplicial objects.

Definition 1.29. An *(abelian) simplicial group* G is a simplicial set where each G^n is an (abelian) group and the face and degeneracy operators are group morphisms.

Examples of *(abelian) simplicial groups* are provided as follows.

Definition 1.30. Given a reduced simplicial set X, the *loop space* simplicial version of X, $G(X)$, is a simplicial group defined as follows: $G^n(X)$ is the free group generated by X^{n+1} with the relations $\{\eta_0 x; x \in X^n\}$. If e_n is the identity element of $G^n(X)$, we have the canonical application $\tau : X^{n+1} \to G^n(X)$ defined by:

$$\tau(y) = \begin{cases} e_n & \text{if exists } x \in X^n \text{ such that } y = \eta_0 x, \\ y & \text{otherwise.} \end{cases}$$

If ∂, η are the face and degeneracy operators of X, we define the face and degeneracy operators, denoted by $\overline{\partial}, \overline{\eta}$, over the generators of $G^n(X)$ as follows:

$$\begin{aligned} \overline{\partial_0}\tau(x) &= [\tau(\partial_0 x)]^{-1}\tau(\partial_1 x) \\ \overline{\partial_i}\tau(x) &= \tau(\partial_{i+1} x) \text{ if } i > 0 \\ \overline{\eta_i}\tau(x) &= \tau(\eta_{i+1} x) \text{ if } i \geq 0 \end{aligned}$$

In the sequel, we will denote $G(X)$ as $\Omega(X)$. We define inductively $\Omega^n(X) = \Omega(\Omega^{n-1}(X))$ for all $n \geq 1$, $\Omega^0(X) = X$.

An important example of abelian simplicial groups is the model for an Eilenberg MacLane space.

Definition 1.31. Let π be an abelian group. The abelian simplicial group $K = K(\pi, 0)$ is given by $K^n = \pi$ for all $n \geq 0$, and with face and degeneracy operators $\partial_i : K^n = \pi \to K^{n-1} = \pi$ and $\eta_i : K^n = \pi \to K^{n+1} = \pi$, $0 \leq i \leq n$, equal to the identity map of the group π. This abelian simplicial group is the Eilenberg MacLane space of type $(\pi, 0)$.

In order to construct the spaces $K(\pi, n)$'s several methods can be used, although the results are necessarily isomorphic [May67]. The space $K(\pi, n)$ can be built recursively by means of the classifying space constructor.

Definition 1.32. Let G be an abelian simplicial group. The *classifying space* of G, written $B(G)$, is the abelian simplicial group built as follows. The n-simplexes of $B(G)$ are the elements of the Cartesian product:

$$B(G)^n = G^{n-1} \times G^{n-2} \times \cdots \times G^0.$$

In this way $B(G)^0$ is the group which has only one element that we denote by [ ]. For $n \geq 1$, an element of $B(G)^n$ has the form $[g_{n-1}, \dots, g_0]$ with $g_i \in G^i$. The face and degeneracy operators are given by

$$\begin{aligned}
\eta_0[\,] &= [e_0] \\
\partial_i[g_0] &= [\,], \quad i = 0, 1 \\
\partial_0[g_{n-1}, \dots, g_0] &= [g_{n-2}, \dots, g_0] \\
\partial_i[g_{n-1}, \dots, g_0] &= [\partial_{i-1} g_{n-1}, \dots, \partial_1 g_{n-i+1}, \partial_0 g_{n-i} + g_{n-i-1}, g_{n-i-2}, \dots, g_0], \quad 0 < i \leq n \\
\eta_0[g_{n-1}, \dots, g_0] &= [e_n, g_{n-1}, \dots, g_0] \\
\eta_i[g_{n-1}, \dots, g_0] &= [\eta_{i-1} g_{n-1}, \dots, \eta_0 g_{n-i}, e_{n-i}, g_{n-i-1}, \dots, g_0], \quad 0 < i \leq n
\end{aligned}$$

where e_n denotes the null element of the abelian group G^n.

We define inductively $B^n(G) = B(B^{n-1}(G))$ for all $n \geq 1$, $B^0(G) = G$.

Theorem 1.33. [May67] Let π be an abelian group and $K(\pi, 0)$ as explained before. Then $B^n(K)$ is a $K(\pi, n)$.

1.1.2.2 Link between Homological Algebra and Simplicial Topology

Up to now, we have given a brief introduction to Simplicial Topology and Homological Algebra; let us present now the link between these two subjects that will allow us to compute the homology groups of simplicial sets.

Definition 1.34. Let K be a simplicial set, we define the *chain complex associated with* K, $C_*(K) = (C_n(K), d_n)_{n \in \mathbb{N}}$, in the following way:

- $C_n(K) = \mathbb{Z}[K^n]$ is the free $\mathbb{Z}$-module generated by K^n. Therefore an n-chain $c \in C_n(K)$ is a combination $c = \sum_{i=1}^m \lambda_i x_i$ with $\lambda_i \in \mathbb{Z}$ and $x_i \in K^n$ for $1 \leq i \leq m$;
- the differential map $d_n : C_n(K) \to C_{n-1}(K)$ is given by

$$d_n(x) = \sum_{i=0}^{n} (-1)^i \partial_i(x) \text{ for } x \in K^n$$

  and it is extended by linearity to the combinations $c = \sum_{i=1}^m \lambda_i x_i \in C_n(K)$.

The following statement is an immediate consequence of the previous definition.

Proposition 1.35. Let X, Y simplicial sets, where X is a simplicial subcomplex of Y, then $C_*(X)$ is a chain subcomplex of $C_*(Y)$.

From the link between simplicial sets and chain complexes we can define the homology groups of a simplicial set as follows.

Definition 1.36. Given a simplicial set K, the *n-homology group* of K, $H_n(K)$, is the n-homology group of the chain complex $C_*(K)$:

$$H_n(K) = H_n(C_*(K))$$

To sum up, when we want to study properties of a topological space which admits a triangulation, we can proceed as follows. We can associate a simplicial model K with a topological space X which admits a triangulation. Subsequently, the chain complex $C_*(K)$ associated with K can be constructed. Afterwards, properties of this chain complex are computed; for instance, homology groups. Eventually, we can interpret the properties of the chain complex as properties of the topological space X.

1.1.3 Effective Homology

As we have seen, a central problem in our context consists of computing *homology groups* of topological spaces. By definition, the homology group of a space X is the one of its associated chain complex $C_*(X)$. If $C_*(X)$ can be described as a graded free abelian group with *finitely* many generators at each degree; then, computing each homology group can be translated to a problem of diagonalizing certain integer matrices (see [Veb31]). So, we can assert that homology groups are computable in this finite type case.

However, things are more interesting when a space X is not of finite type (in the previously invoked sense) but it is known that its homology groups *are* of finite type. Then, it is natural to study if these homology groups are computable. The *effective homology method*, introduced in [RS02] and [RS06], provides a framework where this computability question can be handled.

In this subsection, we present some definitions and fundamental results about the effective homology method. More details can be found in [RS02] and [RS06].

In the context of effective homology, we can distinguish two different kinds of objects: *effective* and *locally effective* chain complexes.

Definition 1.37. An *effective chain complex* is a *free* chain complex of $\mathbb{Z}$-modules $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ where each group C_n is finitely generated and:

- an algorithm returns a (distinguished) $\mathbb{Z}$-basis in each degree n, and
- an algorithm provides the differential maps d_n.

If a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ is effective, the differential maps $d_n : C_n \to C_{n-1}$ can be expressed as finite integer matrices, and then it is possible to know *everything* about C_*: we can compute the subgroups $\operatorname{Ker} d_n$ and $\operatorname{Im} d_{n+1}$, we can determine whether an n-chain $c \in C_n$ is a cycle or a boundary, and in the last case, we can obtain $z \in C_{n+1}$ such that $c = d_{n+1}(z)$. In particular an elementary algorithm computes its homology groups using, for example, the Smith Normal Form technique (for details, see [Veb31]).

Definition 1.38. A *locally effective chain complex* is a *free* chain complex of $\mathbb{Z}$-modules $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ where each group C_n consists of an infinite number of generators.

In this case, no *global* information is available. For example, it is not possible in general to compute the subgroups $\operatorname{Ker} d_n$ and $\operatorname{Im} d_{n+1}$, which can have infinite nature. However, "local" information can be obtained: we can compute, for instance, the boundary of a given element.

In general, we can talk of *locally effective objects* when only "local" computations are possible. For instance, we can consider a locally effective simplicial set; the set of n-simplexes is not finite, but we can compute the faces of any specific n-simplex.

The effective homology technique consists in combining locally effective objects with effective chain complexes. In this way, we will be able to compute homology groups of locally effective objects.

The following notion is one of the fundamental notions in the effective homology method, since it will allow us to obtain homology groups of locally effective chain complexes in some situations.

Definition 1.39. A *reduction* ρ (also called *contraction*) between two chain complexes C_* and D_*, denoted in this memoir by $\rho : C_* \Rrightarrow D_*$, is a triple $\rho = (f, g, h)$

$$\begin{tikzcd} C_* \arrow[loop left, "h"] \arrow[r, bend left, "f"] & D_* \arrow[l, bend left, "g"] \end{tikzcd}$$

where f and g are chain complex morphisms, h is a graded group morphism of degree $+1$, and the following relations are satisfied:

1) $f \circ g = \operatorname{Id}_{D_*}$;

2) $d_C \circ h + h \circ d_C = \operatorname{Id}_{C_*} - g \circ f$;

3) $f \circ h = 0; \quad h \circ g = 0; \quad h \circ h = 0.$

The importance of reductions lies in the following fact. Let $C_* \Rrightarrow D_*$ be a reduction, then C_* is the direct sum of D_* and an acyclic chain complex; therefore the graded homology groups $H_*(C_*)$ and $H_*(D_*)$ are canonically isomorphic.

Very frequently, the *small* chain complex D_* is effective; so, we can compute its homology groups by means of elementary operations with integer matrices. On the other hand, in many situations the *big* chain complex C_* is locally effective and therefore its homology groups cannot directly be determined. However, if we know a reduction from C_* over D_* and D_* is effective, then we are able to compute the homology groups of C_* by means of those of D_*.

Given a chain complex C_*, a *trivial reduction* $\rho = (f, g, h) : C_* \Rrightarrow C_*$ can be constructed, where f and g are the identity map and $h = 0$.

As we see in the next proposition, the composition of two reductions can be easily constructed.

Proposition 1.40. Let $\rho = (f, g, h) : C_* \Rrightarrow D_*$ and $\rho' = (f', g', h') : D_* \Rrightarrow E_*$ be two reductions. Another reduction $\rho'' = (f'', g'', h'') : C_* \Rrightarrow E_*$ is defined by:

$$\begin{aligned} f'' &= f' \circ f \\ g'' &= g \circ g' \\ h'' &= h + g \circ h' \circ f \end{aligned}$$

Another important notion that provides a connection between locally effective homology chain complexes and effective chain complexes is the notion of equivalence.

Definition 1.41. A *strong chain equivalence* (from now on, *equivalence*) ε between two chain complexes C_* and D_*, denoted by $\varepsilon : C_* \Lleftarrow\!\!\Rrightarrow D_*$, is a triple (B_*, ρ_1, ρ_2) where B_* is a chain complex, and ρ_1 and ρ_2 are reductions from B_* over C_* and D_* respectively:

$$\begin{tikzcd} & B_* \arrow[dl, Rightarrow, "\rho_1"'] \arrow[dr, Rightarrow, "\rho_2"] & \\ C_* & & D_* \end{tikzcd}$$

Very frequently, D_* is effective; so, we can compute its homology groups by means of elementary operations with integer matrices. On the other hand, in many situations both C_* and B_* are locally effective and therefore their homology groups cannot directly be determined. However, if we know a reduction from B_* over C_*, a reduction from B_* over D_*, and D_* is effective, then we are able to compute the homology groups of C_* by means of those of D_*.

Once we have introduced the notion of equivalence, it is possible to give the definition of *object with effective homology*, which is the fundamental idea of the effective homology technique. These objects will allow us to compute homology groups of locally effective objects by means of effective chain complexes.

Definition 1.42. An *object with effective homology* X is a quadruple $(X, C_*(X), HC_*, \varepsilon)$ where:

- X is a locally effective object;
- $C_*(X)$ is a (locally effective) chain complex associated with X, that allows us to study the homological nature of X;
- HC_* is an effective chain complex;
- ε is an equivalence $\varepsilon : C_*(X) \Lleftarrow\!\!\Rrightarrow HC_*$.

Then, the graded homology groups $H_*(X)$ and $H_*(HC_*)$ are canonically isomorphic, then we are able to compute the homology groups of X by means of those of HC_*.

The main problem now is the following one: given a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$, is it possible to determine its effective homology? We must distinguish three cases:

- First of all, if a chain complex C_* is by chance effective, then we can choose the trivial effective homology: ε is the equivalence $C_* \Lleftarrow C_* \Rrightarrow C_*$, where the two components ρ_1 and ρ_2 are both the trivial reduction on C_*.
- In some cases, some theoretical results are available providing an equivalence between some chain complex C_* and an *effective* chain complex. Typically, the Eilenberg MacLane space $K(\mathbb{Z},1)$ has the homotopy type of the circle S^1 and a reduction $C_*(K(\mathbb{Z},1)) \Rrightarrow C_*(S^1)$ can be built.
- The most important case: let $X_1,\dots,X_n$ be objects with effective homology and Φ a constructor that produces a new space $X = \Phi(X_1,\dots,X_n)$ (for example, the Cartesian product of two simplicial sets, the classifying space of a simplicial group, etc). In *natural* "reasonable" situations, there exists an effective homology version of Φ that allows us to deduce a version with effective homology of X, the result of the construction, from versions with effective homology of the arguments $X_1,\dots,X_n$. For instance, given two simplicial sets K and L with effective homology, then the Cartesian product $K \times L$ is an object with effective homology too; this is obtained by means of the Eilenberg-Zilber Theorem, see [RS06].

Two of the most basic (in the sense of fundamental) results in the effective homology method are the two *perturbation lemmas*. The main idea of both lemmas is that given a reduction, if we *perturb* one of the chain complexes then it is possible to perturb the other one so that we obtain a new reduction between the *perturbed* chain complexes. The first theorem (the Easy Perturbation Lemma) is very easy, but it can be useful. The Basic Perturbation Lemma is not trivial at all. It was discovered by Shih Weishu [Shi62], although the abstract modern form was given by Ronnie Brown [Bro67].

Definition 1.43. Let $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ be a chain complex. A *perturbation* δ of the differential d is a collection of group morphisms $\delta = \{\delta_n : C_n \to C_{n-1}\}_{n\in\mathbb{N}}$ such that the sum $d+\delta$ is also a differential, that is to say, $(d+\delta)\circ(d+\delta) = 0$.

The perturbation δ produces a new chain complex $C'_* = (C_n, d_n + \delta_n)_{n\in\mathbb{N}}$; it is the *perturbed* chain complex.

Theorem 1.44 (Easy Perturbation Lemma, EPL)**.** Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{N}}$ be two chain complexes, $\rho = (f,g,h) : C_* \Rrightarrow D_*$ a reduction, and δ_D a perturbation of d_D. Then a new reduction $\rho' = (f',g',h') : C'_* \Rrightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from C_* by replacing the old differential d_C by the perturbed differential $(d_C + g\circ\delta_D\circ f)$;

2) the new chain complex D'_* is obtained from the chain complex D_* only by replacing the old differential d_D by $(d_D + \delta_D)$;

3) $f' = f$;

4) $g' = g$;

5) $h' = h$.

The perturbation δ_D of the *small* chain complex D_* is naturally transferred (using the reduction ρ) to the *big* chain complex C_*, obtaining in this way a new reduction ρ' (which in fact has the same components as ρ) between the perturbed chain complexes. On the other hand, if we consider a perturbation d_C of the top chain complex C_*, in general it is not possible to perturb the small chain complex D_* so that there exists a reduction between the perturbed chain complexes. As we will see, we need an additional hypothesis.

Theorem 1.45 (Basic Perturbation Lemma, BPL)**.** [Bro67] Let us consider a reduction $\rho = (f, g, h) : C_* \Rrightarrow D_*$ between two chain complexes $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{N}}$, and δ_C a perturbation of d_C. Furthermore, the composite function $h \circ \delta_C$ is assumed *locally nilpotent*, in other words, given $x \in C_*$ there exists $m \in \mathbb{N}$ such that $(h \circ \delta_C)^m(x) = 0$. Then a new reduction $\rho' = (f', g', h') : C'_* \Rrightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from the chain complex C_* by replacing the old differential d_C by $(d_C + \delta_C)$;

2) the new chain complex D'_* is obtained from D_* by replacing the old differential d_D by $(d_D + \delta_D)$, with $\delta_D = f \circ \delta_C \circ \phi \circ g = f \circ \psi \circ \delta_C \circ g$;

3) $f' = f \circ \psi = f \circ (\mathrm{Id}_{C_*} - \delta_C \circ \phi \circ h)$;

4) $g' = \phi \circ g$;

5) $h' = \phi \circ h = h \circ \psi$;

with the operators ϕ and ψ defined by

$$\phi = \sum_{i=0}^{\infty} (-1)^i (h \circ \delta_C)^i$$
$$\psi = \sum_{i=0}^{\infty} (-1)^i (\delta_C \circ h)^i = \mathrm{Id}_{C_*} - \delta_C \circ \phi \circ h,$$

the convergence of these series being ensured by the locally nilpotency of the compositions $h \circ \delta_C$ and $\delta_C \circ h$.

It is worth noting that the effective homology method is not only a theoretical method but it has also been implemented in a software system called *Kenzo* [DRSS98]. In this system, the BPL is a central result and has been intensively used.

1.2 The Kenzo system

Kenzo is a 16000 lines program written in Common Lisp [Gra96], devoted to Symbolic Computation in Algebraic Topology. It was developed by Francis Sergeraert and some co-workers, and is www-available (see [DRSS98] for documentation and details). It works with the main mathematical structures used in Simplicial Algebraic Topology, [HW67], (chain complexes, differential graded algebras, simplicial sets, morphisms between these objects, reductions and so on) and has obtained some results (for example, homology groups of iterated loop spaces of a loop space modified by a cell attachment, see [Ser92]) which have not been confirmed nor refuted by any other means.

The fundamental idea of the Kenzo system is the notion of *object with effective homology* combined with *functional programming*. By using functional programming, some techniques in Algebraic Topology are encoded in the form of algorithms. Moreover, the possibility to implement in a computer some infinitely generated objects and do computations with them is obtained thanks to the effective homology theory.

In addition, not only *known* algorithms were implemented but also new methods were developed to *transform* the main "tools" of Algebraic Topology, mainly the spectral sequences, not at all *algorithmic* in the traditional organization, into actual *computing* methods.

The computation process in the Kenzo system requires a great amount of algebraic structures and equivalences to be built, and thus a lot of resources.

This section is devoted to present some important features of Kenzo that will be relevant onward.

1.2.1 Kenzo mathematical structures

The data structures of the Kenzo system are organized in two layers. As algebraically modeled in [LPR03, DLR07], the first layer is composed of algebraic data structures (chain complexes, simplicial sets, and so on) and the second one of standard data structures (lists, trees, and so on) which are representing *elements* of data from the first layer.

Here we give an overview of how mathematical structures (the first layer) are represented in the Kenzo system, using object oriented (*CLOS*) and functional programming features simultaneously.

Most often, an object of some *type* in mathematics is a structure with several components, frequently of functional nature. Let us consider the case of a user who wants to handle groups; this simple particular case is sufficient to understand how CLOS gives the right tools to process mathematical structures.

We can define a `GRP` class whose instances correspond to concrete groups as follows.

```
> (DEFCLASS GRP ()
    ((elements :type list :initarg :elements :reader elements)
     (mult :type function :initarg :mult :reader mult1)
     (inv :type function :initarg :inv :reader inv1)
     (nullel :type function :initarg :nullel :reader nullel)
     (cmpr :type function :initarg :cmpr :reader cmpr1))) ✠
#<STANDARD-CLASS GRP>
```

In this organization, a group is made of five slots which represent a list of the elements of the group (`elements`), the product of two elements of the group (`mult`), the inverse of an element of the group (`inv`), the identity element of the group (`nullel`) and a comparison test (`cmpr`) between the elements of the group. It is worth noting that `mult`, `inv`, `nullel` and `cmpr` slots are functional slots.

The most important difference between the mathematical definition and its implementation is that no axiomatic information appears at all. Kenzo, being a system for computing, does not need any information about the properties of the operations, only their behavior is relevant. As a consequence of this, abelian groups can be implemented in Common Lisp similarly to groups, even when their mathematical definitions differ.

Let us construct now the group $\mathbb{Z}/2\mathbb{Z}$; that is to say, the cyclic group of dimension 2. In our context we define the comparison, the product and the inverse functions with the help of the `mod` function, a predefined Lisp function.

```
> (MAKE-INSTANCE 'GRP
    :elements '(0 1)
    :mult #'(lambda (g1 g2) (mod (+ g1 g2) 2))
    :inv #'(lambda (g) (mod (- 2 g) 2))
    :nullel #'(lambda () 0)
    :cmpr #'(lambda (g1 g2) (= 0 (mod (- g1 g2) 2)))) ✠
#<GROUP @ #x26f4bc12>
```

In this way, we can define mathematical structures in Common Lisp using object oriented (*CLOS*) and functional programming features. We urge the interested reader to consult [Ser01] where a detailed explanation about how to define mathematical structures in Common Lisp is explained.

The previously explained organization is moved to the Kenzo context in order to define the main mathematical structures used in Simplicial Algebraic Topology, [HW67]. Figure 1.2 shows the Kenzo class diagram where each class corresponds to the respective mathematical structure.

The lefthand part of the class diagram is made of the main mathematical categories that are used in combinatorial Algebraic Topology. As we said previously, a *chain complex* is a graded differential module; an *algebra* is a chain complex with a compatible multiplicative structure, the same for a *coalgebra* but with comultiplicative structure. If a multiplicative and a comultiplicative structures are added and if they are compatible

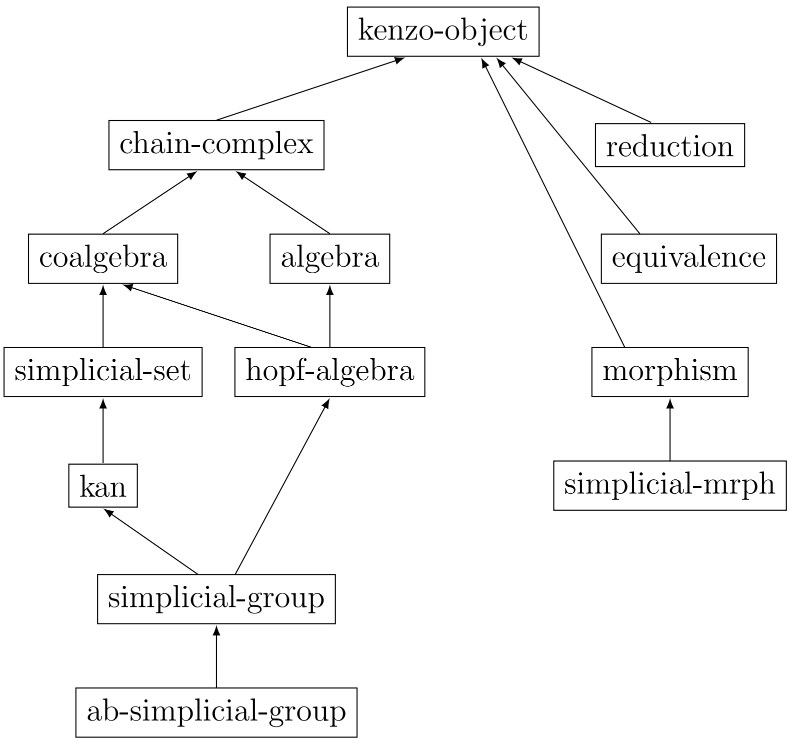


Figure 1.2: Kenzo class diagram

with each other in a natural sense, the it is a *Hopf algebra*, and so on. The righthand part of the class diagram is made of the operations over the mathematical structures of the lefthand. It is worth noting that all the mathematical structures in the Kenzo system are graded structures.

The following class definition corresponds to the simplest algebraic structure implemented in Kenzo, the chain complexes:

```
(DEFCLASS CHAIN-COMPLEX ()
    ((cmpr  :type cmprf :initarg :cmpr  :reader cmpr1)
     (basis :type basis :initarg :basis :reader basis1)
     ;; BaSe GeNerator
     (bsgn :type gnrt :initarg :bsgn :reader bsgn)
     ;; DiFFeRential
     (dffr :type morphism :initarg :dffr :reader dffr1)
     ;; GRound MoDule
     (grmd :type chain-complex :initarg :grmd :reader grmd)
     ;; EFfective HoMology
     (efhm :type homotopy-equivalence :initarg :efhm :reader efhm)
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
     ;; ORiGiN
     (orgn :type list :initarg :orgn :reader orgn)))
```

The relevant slots are `cmpr`, a function coding the equality between the generators of the chain complex; `basis`, the function defining the basis of each group of n-chains, or the keyword `:locally-effective` if the chain complex is not effective; `dffr`, the differential morphism, which is an instance of the class `MORPHISM`; `efhm`, which stores information about the effective homology of the chain complex; and `orgn`, is used to keep record of information about the object.

The class `CHAIN-COMPLEX` is extended by inheritance with new slots, obtaining more elaborate structures. For instance, extending it with an `aprd` (algebra product) slot, we obtain the `ALGEBRA` class. Multiple inheritance is also available; for example, the class `SIMPLICIAL-GROUP` is obtained by inheritance from the classes `KAN` and `HOPF-ALGEBRA`.

It is worth emphasizing here that simplicial sets have also been implemented as a subclass of `CHAIN-COMPLEX`. To be precise, the class `SIMPLICIAL-SET` inherits from the class `COALGEBRA`, which is a direct subclass of `CHAIN-COMPLEX`, with a slot `cprd` (the coproduct). The class `SIMPLICIAL-SET` has then one slot of its own: `face`, a Lisp function computing any face of a simplex of the simplicial set. The `basis` is in this case (when working with effective objects) the list of non degenerate simplexes, and the differential map of the associated chain complex is given by the alternate sum of the faces, where the degenerate simplices are canceled.

1.2.2 Kenzo way of working

In Kenzo there is a one higher-level objective: compute groups of topological spaces. This main objective can be broken in two actions: (1) computing groups, and (2) constructing spaces. Note that the second task is necessary to carry out the first one.

When a user has decided to construct a space in Kenzo, he should decide which type he wants to build: a simplicial set, a simplicial group and so on; namely, an object of one of the types of the lefthand part of the class diagram of Figure 1.2. Therefore, the user has to construct an instance of one of those classes.

As this task can be quite difficult for a non expert user; Kenzo provides useful functions to create interesting objects of regular usage, which belong to four types: chain complexes, simplicial sets, simplicial groups and abelian simplicial groups.

These functions can be split in two different groups: (1) functions to construct initial spaces and, (2) functions to construct spaces from other ones applying topological constructors.

The following elements, gathered by the types of the constructed object, represent the main spaces that can be constructed in Kenzo from scratch (that is to say, which belong to the first group):

- Chain Complexes:
    - *Unit chain complex*: the `zcc` function, with no arguments, constructs the unit chain complex, see Example 1.7.

- *Circle*: the `circle` function, with no arguments, constructs the circle chain complex, see Example 1.7.

- Simplicial Sets:

  - *Standard simplicial set*: the `delta` function, with a natural number n as argument, constructs $\Delta[n]$, see Definition 1.18.
  - *Sphere*: the `sphere` function, with a natural number n as argument, constructs S^n, see Definition 1.22.
  - *Sphere Wedge*: the `sphere-wedge` function, with a sequence of natural numbers $n1, \ldots, nk$ as arguments, constructs $S^{n1} \vee \ldots \vee S^{nk}$, see Definition 1.23.
  - *Moore space*: the `moore` function, with two natural number n, p as arguments, constructs $M(\mathbb{Z}/n\mathbb{Z}, p)$, see Definition 1.24.
  - *Projective space*: the `r-proj-space` function has two optional arguments k and l. If neither k and l are provided, then the function constructs $P^\infty\mathbb{R}$. If k is provided but l is not, then the function constructs $P^\infty\mathbb{R}/P^{k-1}\mathbb{R}$. If both k and l are provided, then the function constructs $P^l\mathbb{R}/P^{k-1}\mathbb{R}$. In the latter case, if $k = 1$, then the function constructs $P^l\mathbb{R}$; see Definition 1.25.
  - *Finite simplicial set*: the `build-finite-ss` function, with a list of simplexes as argument, constructs a simplicial set, see [DRSS98].

- Abelian Simplicial Group:

  - *Eilenberg MacLane space* type $(\mathbb{Z}, n)$: the `k-z` function, with a natural number n as argument, constructs $K(\mathbb{Z}, n)$, see Definition 1.31.
  - *Eilenberg MacLane space* type $(\mathbb{Z}/2\mathbb{Z}, n)$: the `k-z2` function, with a natural number n as argument, constructs $K(\mathbb{Z}/2\mathbb{Z}, n)$, see Definition 1.31.

On the contrary, the following elements, gathered by types, represent the main spaces that can be constructed in Kenzo from other spaces applying topological constructors (that is to say, which belong to the second group):

- Chain Complexes:

  - *Tensor Product*: the `tnsr-prdc` function, with two chain complexes C_*, D_* as arguments, constructs the tensor product $C_* \otimes D_*$, see Definition 1.10.

- Simplicial Sets:

  - *Cartesian product*: the `crts-prdc` function, with two simplicial sets K, L as arguments, constructs the Cartesian product $K \times L$, see Definition 1.26.
  - *Suspension*: the `suspension` function, with a simplicial set X and a natural number n as arguments, constructs the suspension $\mathbf{S}^n(X)$, see Definition 1.28.

- Simplicial Group:
  - *Loop space*: the `loop-space` function, with a simplicial set X and a natural number n as arguments, constructs the loop space $\Omega^n(X)$, see Definition 1.30.
  - *Classifying space*: the `classifying` function, with a simplicial group X, constructs the classifying space $B(X)$, see Definition 1.32.

Eventually, once we have constructed some spaces in our Kenzo session, the Kenzo user can perform computations. Namely, the `homology` function with a chain complex X (or an instance of one of its subclasses: simplicial set, simplicial group and so on) and a natural number n as arguments computes $H_n(X)$.

To sum up, the usual way of working with the Kenzo system is as follows. As a first step, the user constructs some initial spaces by means of some built-in Kenzo functions (as spheres, Moore spaces, Eilenberg MacLane spaces and so on); then, in a second step, he constructs new spaces by applying topological constructions (as Cartesian products, loop spaces, and so on); as a third, and final, step, the user asks Kenzo for computing the homology groups of the spaces.

1.2.3 Kenzo in action

Let us show a didactic example to illustrate the interaction with the Kenzo program. The homology group $H_5(\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4)))$ is "in principle" reachable thanks to old methods, see [CM95], but experience shows even the most skilful topologist meet some difficulties to determine it, see [RS02]. With the Kenzo program, you construct the Moore space $M(\mathbb{Z}/2\mathbb{Z},4)$ in the following way:

```
> (setf m4 (moore 2 4)) ✠
[K1 Simplicial-Set]
```

A Kenzo display must be read as follows. The initial `>` is the Lisp prompt of this Common Lisp implementation. The user types out a Lisp statement, here `(setf m4 (moore 2 4))` and the maltese cross ✠ (in fact not visible on the user screen) marks in this text the end of the Lisp statement, just to help the reader: the right number of closing parentheses is reached. The Return key then asks Lisp to *evaluate* the Lisp statement. Here the Moore space $M(\mathbb{Z}/2\mathbb{Z},4)$ is constructed by the Kenzo function `moore`, taking into account of the arguments 2 and 4, and this Moore space is *assigned* to the Lisp symbol `m4` for later use. Also evaluating a Lisp statement *returns* an object, the result of the evaluation, in this case the Lisp object implementing the Moore space, displayed as `[K1 Simplicial-Set]`, that is, the Kenzo object #1, a `Simplicial-Set`. The internal structure of this object, made of a rich set of data, in particular many functional components, is not displayed. The identification number printed by Kenzo allows the user to recover the whole object by means of a function called simply `k` (for instance, the evaluation of `(k 1)` returns the Moore space $M(\mathbb{Z}/2\mathbb{Z}, 4)$, in our running example). In addition, another function (called `orgn` and which is one of the slots of the `Chain-Complex` class) allows the user to obtain the origin of the object (i.e. from which function and with which arguments has been produced), and thus the printed information is enough to get a complete control of the different objects built with Kenzo.

It is then possible to construct the third loop space of the Moore space, $\Omega^3(M(\mathbb{Z}/2\mathbb{Z}, 4))$, as a simplicial *group*.

```
> (setf o3m4 (loop-space m4 3)) ✠
[K30 Simplicial-Group]
```

The combinatorial version of the loop space is *highly* infinite: it is a combinatorial version of the space of *continuos* maps $S^3 \to M(\mathbb{Z}/2\mathbb{Z}, 4)$, but functionally codes as a small set of functions in a `Simplicial-Group` object.

Eventually, the user can compute the fifth homology group of this space.

```
> (homology o3m4 5) ✠
Homology in dimension 5:
Component Z/Z2
Component Z/Z2
Component Z/Z2
Component Z/Z2
Component Z/Z2
---done---
```

To be interpreted as stating $H_5(\Omega^3(M(\mathbb{Z}/2\mathbb{Z}, 4))) = \mathbb{Z}_2^5$. In this way, Kenzo computes the homology groups of *complicated* spaces. This is due to the Kenzo implementation of the effective homology method which is explained in the following subsection.

1.2.4 Effective Homology in Kenzo

As we have previously said, the central idea of the Kenzo system is the notion of *object with effective homology*. In this subsection, we are going to show how this notion explained in Subsection 1.1.3 is used in Kenzo.

As we stated in Subsection 1.1.3, the main problem is the following one: given an object, determine its effective homology version. Three cases have been distinguished.

First of all, let an object X if the chain complex $C_*(X)$ is by chance effective, then we can choose the trivial effective homology: ε is the equivalence $C_*(X) \twoheadleftarrow\!\!\Leftarrow C_*(X) \Rrightarrow\!\!\!\!\!\!\!\!\!\!\twoheadrightarrow C_*(X)$, where the both reductions are the trivial reduction on C_*. This situation happens, for instance, in the case of the sphere S^3.

```
> (setf s3 (sphere 3)) ✠
[K1 Simplicial-Set]
```

We can ask for the effective homology of S^3 as follows:

```
> (efhm s3) ✠
[K9 Homotopy-Equivalence K1 <= K1 => K1]
```

An homotopy equivalence is automatically constructed by Kenzo where both reductions are the trivial reduction on $C_*(S^3)$ (let us note that the `K1` object not only represents the simplicial set S^3 but also the chain complex $C_*(S^3)$ due to the heritage relationship between simplicial sets and chain complexes). In this case, the effective homology technique does not provide any additional tool to the computation of the homology groups of S^3. On the contrary, we will see the power of this technique when the initial space X is locally effective.

The second feasible situation happened when given a locally effective object X some theoretical result was available providing an equivalence between the chain complex $C_*(X)$ and an *effective* chain complex.

According to Subsubsection 1.1.2.1, the "minimal" simplicial model of the Eilenberg MacLane space $K(\mathbb{Z},1)$ is defined by $K(\mathbb{Z},1)_n = \mathbb{Z}^n$; an infinite number of simplexes is required in every dimension $n \geq 1$; that is to say, we have a locally effective object. This does not prevent such an object from being installed and handled by Kenzo.

```
> (setf kz1 (k-z 1)) ✠
[K1 Abelian-Simplicial-Group]
```

The `k-z` Kenzo function construct the standard Eilenberg MacLane space. In ordinary mathematical notation (as seen in Subsubsection 1.1.2.1), a 3-simplex of `kz1` could be for example $[3,5,-5]$, denoted by $(3\ 5\ -5)$ in Kenzo. The faces of this simplex can be determined as follows.

```
> (dotimes (i 4) (print (face kz1 i 3 '(3 5 -5)))) ✠
<AbSm - (5 -5)>
<AbSm - (8 -5)>
<AbSm 1 (3)>
<AbSm - (3 5)>
nil
```

The faces are computed as explained in Subsubsection 1.1.2.1. Then, *local* computations are possible, so, the object `kz1` is *locally effective.* But no global information is available. For example, if we try to obtain the list on non degenerate simplexes in dimension 3, we obtain an error.

```
> (basis kz1 3) ✠
Error: The object [K1 Abelian-Simplicial-Group] is locally-effective
```

This basis in fact is $\mathbb{Z}^3$, an infinite set whose element list cannot be explicitly store nor displayed. So, the homology groups of `kz1` cannot be elementarily computed. However, $K(\mathbb{Z},1)$ has the homotopy type of the circle S^1 and the Kenzo program knows this fact.

```
> (efhm kz1) ✠
[K22 Homotopy-Equivalence K1 <= K1 => K16]
```

A reduction `K1` $= K(\mathbb{Z},1) \Rrightarrow$ `K16` is constructed by Kenzo. What is `K16`?

```
> (orgn (k 16)) ✠
(circle)
```

`K16` is the expected object, the circle S^1 which is an effective chain complex; so we can compute its homology groups by means of tradicional methods. Therefore, we can compute the homology groups of the space $K(\mathbb{Z},1)$ by means of the effective homology method.

Eventually, the last situation happened when given $X_1,\ldots,X_n$ objects with effective homology and Φ a constructor that produced a new space $X = \Phi(X_1,\ldots,X_n)$, we wanted to compute the effective homology version of X. Let us present an example.

The Cartesian product of two locally effective simplicial sets produces another locally effective simplicial set.

```
> (setf kz1xkz1 (crts-prdc kz1 kz1)) ✠
[K6 Simplicial-Set]
> (basis kz1xkz1 3) ✠
Error: The object [K6 Simplicial-Set] is locally-effective
```

So, the homology groups of `kz1xkz1` cannot be elementarily computed. However, Kenzo is able to construct an equivalence between this object and an effective chain complex.

```
> (efhm kz1xkz1) ✠
[K54 Homotopy-Equivalence K6 <= K44 => K34]
```

An equivalence `K6` $= K(\mathbb{Z},1) \times K(\mathbb{Z},1) \Lleftarrow$ `K44` $\Rrightarrow$ `K34` is constructed by Kenzo. What is `K34`?

```
> (orgn (k 44)) ✠
(TNSR-PRDC (circle) (circle))
```

The object `K44` is the tensor product of two circles $S^1 \otimes S^1$ (the reduction is obtained from the Eilenberg-Zilber Theorem, see [RS06]), an effective chain complex; so we can compute its homology groups by means of tradicional methods. Therefore, we can compute the homology groups of the space $K(\mathbb{Z}, 1) \times K(\mathbb{Z}, 1)$ by means of the effective homology method.

It is worth noting that a Kenzo user does not need to explicitly construct the equivalence to compute the homology groups of a locally effective object. This task is automatically performed by the Kenzo system which constructs the necessary objects without any additional help.

For instance, let us consider a fresh Kenzo session where we have constructed the space $\Omega^3(M(\mathbb{Z}/2\mathbb{Z}, 4))$, the example of the previous subsection:

```
> (setf m4 (moore 2 4)) ✠
[K30 Simplicial-Group]
> (setf o3m4 (loop-space m4 3)) ✠
[K30 Simplicial-Group]
```

At this moment, we can check the number of objects constructed in Kenzo (this information is stored in a global variable called `*idnm-counter*`).

```
> *idnm-counter* ✠
41
```

Subsequently, after computing the third homology, we ask again the number of objects constructed in Kenzo and we obtain the following result.

```
> (homology o3m4 3) ✠
Homology in dimension 3 :
Component Z/4Z
Component Z/2Z
> *idnm-counter* ✠
404
```

This means that Kenzo has constructed 363 intermediary objects in order to compute the homology groups of $\Omega^3(M(\mathbb{Z}/2\mathbb{Z}, 4))$; namely, in order to construct an equivalence between the locally effective object $\Omega^3(M(\mathbb{Z}/2\mathbb{Z}, 4))$ and an effective chain complex.

1.2.5 Memoization in Kenzo

The Kenzo program is certainly a functional system. It is frequent that several thousands of functions are present in memory, each one being dynamically defined from other ones, which in turn are defined from other ones, and so on. In this quite original situation, the same calculations are frequently asked again. To avoid repeating these calculations, it is better to store the results and to systematically examine for each calculation whether the result is already available (*memoization* strategy).

As a consequence, the state of a space evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a homology group, depends on the concrete state of the space involved in the calculation (in the more explicit case, to re-calculate the homology group of a space could be negligible in time, even if in the first occasion this was very time consuming).

Let us shown an example, of the Kenzo memoization. We want to compute the fifth homology group of the space $\Omega^3(S^4 \times S^4)$ and see how much time this computation takes (using the `time` function); therefore, we proceed as usual.

```
> (setf s4 (sphere 4)) ✠
[K1 Simplicial-Set]
> (setf s4xs4 (crts-prdc s4 s4)) ✠
[K6 Simplicial-Set]
> (setf o3s4xs4 (loop-space s4xs4 3)) ✠
[K35 Simplicial-Group]
> (time (homology o3s4xs4 5)) ✠
;; some lines skipped
; real time  1,139,750 msec (00:18:59.750)
```

The first time that we compute $H_5(\Omega^3(S^4 \times S^4))$, Kenzo takes almost 20 minutes to obtain the result. However, if we ask again for the same computation:

```
> (time (homology o3s4xs4 5)) ✠
;; some lines skipped
; real time  262,484 msec (00:04:22.484)
```

in this case Kenzo only needs 4 minutes. It is worth noting that Kenzo does not store the final result (that is to say, the group $H_5(\Omega^3(S^4 \times S^4)))$ but intermediary computations related to the differential of the generators of the space. Then, when a Kenzo user asks a computation previously computed, Kenzo does not simply look up and returned it, but it uses some previously stored computations to calculate the result faster.

Moreover, it is very important not to have several copies of the same function; otherwise it is impossible for one copy to guess some calculation has already been done by another copy. This is a very important question in Kenzo, so that the following idea has been used. Each Kenzo object has a rigorous definition, stored as a list in the `orgn` slot of the object (`orgn` stands for origin of the object). This is the main reason of the top class `kenzo-object`: making this process easier. The actual definition of the `kenzo-object` class is:

```
(DEFCLASS KENZO-OBJECT ()
  ((idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
   (orgn :type list :initarg :orgn :reader orgn))) ✠
```

Then, when any `kenzo-object` is to be considered, its *definition* is constructed and the program firstly looks in `*k-list*` (a list which stores the already constructed `kenzo-object` instances) whether some object corresponding to this definition already exists; if yes, no `kenzo-object` is constructed, the already existing one is simply returned. Look at this small example where we construct the second loop space of S^3, then the first loop space, and then again the second loop space. In fact the initial construction of the second loop space required the first loop space, and examining the identification number `K??` of these objects shows that when the first loop space is later asked for, Kenzo is able to return the already existing one.

```
> (setf s3 (sphere 3)) ✠
[K372 Simplicial-Set]
> (setf o2s3 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (setf os3 (loop-space s3 1)) ✠
[K374 Simplicial-Group]
> (setf o2s3-2 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (eq o2s3 o2s3-2) ✠
T
```

The last statement shows the symbols `o2s3` and `o2s3-2` points to the same machine address. In this way we are sure any kenzo-object has no duplicate, so that the memory process for the values of numerous functions cannot miss an already computed result.

1.2.6 Reduction degree

Working with Kenzo, a constructor for the space X has associated a number $g(X) \in \mathbb{Z}$. When, we apply an operation O over the space X, a set of rules are used to compute $g(O(X))$. We will call *reduction degree of* X to this number, $g(X)$, which is attach to our concrete representation of the spaces. This number is a lower bound of the simply connectedness degree; that is to say, the homotopy groups of the space X are null at least from 1 to $g(X)$. We list as follows the set of rules for the initial spaces and topological operators which are going to be employed in this memoir.

Tensor product ($X \otimes Y$): $g(X \otimes Y) = min\{g(X), g(Y)\}$.

Suspension ($S^n(X)$): $g(S^n(X)) = \begin{cases} g(X)+n & \text{if } g(X) \geq 0 \\ g(X) & \text{if } g(X) < 0 \end{cases}$

Sphere (S^n): $g(S^n) = n-1$.

Moore space ($M(\mathbb{Z}/p\mathbb{Z}, n)$): $g(M(\mathbb{Z}/p\mathbb{Z}, n)) = n-1$.

Standard simplicial set (Δ^n): $g(\Delta^n) = 0$.

Sphere wedge ($S^{n_1} \vee S^{n_2} \vee \ldots \vee S^{n_k}$):

$$g(S^{n_1} \vee S^{n_2} \vee \ldots \vee S^{n_k}) = min\{g(S^{n_1}), g(S^{n_2}), \ldots, g(S^{n_k})\}.$$

Projective space ($P^\infty\mathbb{R}$) : $g(P^\infty\mathbb{R}) = 0$.

Projective space ($P^\infty\mathbb{R}/P^n\mathbb{R}$) : $g(P^\infty\mathbb{R}/P^n\mathbb{R}) = n-1$.

Projective space ($P^l\mathbb{R}$) : $g(P^l\mathbb{R}) = 0$.

Projective space ($P^l\mathbb{R}/P^n\mathbb{R}$) : $g(P^l\mathbb{R}/P^n\mathbb{R}) = n-1$.

$K(\mathbb{Z}, n)$: $g(K(\mathbb{Z}, n)) = n-1$.

$K(\mathbb{Z}/2\mathbb{Z}, n)$: $g(K(\mathbb{Z}/2\mathbb{Z}, n)) = n-1$.

Loop space ($\Omega^n(X)$): $g(\Omega^n(X)) = g(X) - n$.

Cartesian product ($X \times Y$): $g(X \times Y) = min\{g(X), g(Y)\}$.

Classifying space ($B^n(X)$): $g(B^n(X)) = \begin{cases} g(X)+n & \text{if } g(X) \geq 0 \\ g(X) & \text{if } g(X) < 0 \end{cases}$

The reduction degree of a space X provides us information about the capabilities of the Kenzo system to obtain the homology groups of X. When $g(X) < 0$, a Kenzo attempt to compute the homology groups of X will raise in an error; otherwise the Kenzo system can compute the homology groups of X.

For instance, the reduction degree of $\Omega^2 S^2$ is -1; then, let us show what happens if we try to compute the third homology group of this space.

```
> (setf s2 (sphere 2)) ✠
[K1 Simplicial-Set]
> (setf o2s2 (loop-space s2 2)) ✠
[K18 Simplicial-Group]
> (homology o2s2 3) ✠
Error: ‘NIL’ is not of the expected type ‘NUMBER’
[condition type: TYPE-ERROR]
```

As we can see an error is produced; then, the Kenzo system cannot compute the homology groups for $\Omega^2(S^2)$.

1.2.7 Homotopy groups in Kenzo

Up to now, we have been working with one of the most important algebraic invariants in Algebraic Topology: homology groups. We can wonder what happens with the other main algebraic invariant: *homotopy groups*.

The n-homotopy group of a topological space X with a base point x_0 is defined as the set of homotopy classes of continuous maps $f : S^n \to X$ that map a chosen base point $a \in S^n$ to the base point $x_0 \in X$. A more detailed description and results about homotopy groups can be found in [Hat02, May67].

Homotopy groups were defined by Hurewicz in [Hur35] and [Hur36] as a generalization of the fundamental group [Poi95]. It is worth noting that except in special cases, homotopy groups are hard to be computed. For instance, whereas the homology groups of spheres are easily computed, computing their homotopy groups remains as a difficult subject, see [Tod62, Mah67, Rav86].

For the general case, Edgar Brown published in [Bro57] a *theoretical* algorithm for the computation of homotopy groups of simply connected spaces such that their homology groups are of finite type. However Brown himself explained that his "algorithm" has not practical use.

An interesting algorithm based on the effective homology theory was developed by P. Real, see [Rea94]. In that paper, an algorithm that computes the homotopy groups of a 1-reduced simplicial set was explained. Here, we just state the algorithm.

Algorithm 1.46 ([Rea94])**.**
Input: a 1-reduced simplicial set with effective homology X and a natural number n such that $n \geq 1$.
Output: the n-th homotopy group of the underlying simplicial set X.

This algorithm is based on the Whitehead tower process [Hat02], a method which allows one to reach any homotopy group of a 1-reduced simplicial set.

It is worth noting that Kenzo, in spite of not providing a function called `homotopy`, like in the case of homology groups, implements all the necessary tools to use the algorithm presented in [Rea94]. A detailed explanation about how to use this method in Kenzo was explained in Chapter 21 of the Kenzo documentation [DRSS98]. However, it is worth noting that in the current Kenzo version, homotopy groups of a 1-reduced simplicial set X can only be computed, if the first non null homology group of X is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$; this is due to the fact that the algorithm presented in [Rea94] needs the calculation of homology groups of Eilenberg-MacLane spaces $K(G, n)$, and in the original Kenzo distribution only the homology of spaces $K(\mathbb{Z}, n)$ and $K(\mathbb{Z}/2\mathbb{Z}, n)$ are built-in.

1.3 ACL2

The ACL2 Theorem Prover has been used to verify the correctness of some Kenzo programs, results which will be presented in this memoir (see Chapters 5 and 6). This section is devoted to provide a brief description of ACL2. In spite of being a glimpse introduction to this system, this description provides enough information to read the sections dedicated to ACL2 topics. A complete description of ACL2 can be found in [KMM00b, KM].

In addition, an interesting ACL2 feature, that will be really important in our developments, is described in this section, too. Some other necessary concepts about ACL2 will be introduced later on for a better understanding of this memoir.

1.3.1 Basics on ACL2

Information in this section has been mainly extracted from [KMM00b].

ACL2 stands for A Computational Logic for Applicative Common Lisp. ACL2 is a programming language, a logic and a theorem prover. Thus, the system constitutes an environment in which algorithms can be defined and executed, and their properties can be formally specified and proved with the assistance of a mechanical theorem prover.

The first version of ACL2 was developed by B. Boyer and J S. Moore in 1989, using the functional and efficient programming language Common Lisp. ACL2 is the successor to the Nqthm [BM97] (or Boyer-Moore) logic and proof system and its Pc-Nqthm interactive enhancement. ACL2 was born as response to the problems Nqthm users faced in applying that system to large-scale proof projects. Namely, the main drawback of the Nqthm appears when the logic specification were used not only for reasoning about the modeling system but also for executing. The main difference between ACL2 and Nqthm lies in the chance of executing the models of the ACL2 logic in Common Lisp.

ACL2 and Nqthm have already shown their effectiveness for implementing (non-trivial) large proofs in mathematics. One of these examples can be found in [Kau00], where the Fundamental Theorem of Calculus is formalized; another development can be found in [BM84], where a mechanical proof of the unsolvability of the halting problem is implemented; other examples of theorems certified with ACL2 and Nqthm are: the Gauss's Law of Quadratic Reciprocity [Rus92], the Church-Rosser theorem for lambda calculus [Sha88], the Gödel's incompleteness theorem [Sha94], and so on.

Besides, ACL2 has been used for a variety of important formal methods projects of industrial and commercial interest, including (for example) the verification of the RTL code which implements elementary floating point operations of the AMD Athlon processor [Rus98] and ROM microcode programs of CAP digital signal processor of Motorola [BKM96].

More references about these and other developments related to ACL2 and Nqthm are www-available in [KM, BM97].

After this brief historical introduction, let us devote some lines to explain the programming language, the logic and the theorem prover of ACL2.

As a programming language, ACL2 is an extension of an applicative subset of Common Lisp. The logic considers every function defined in the programming language as a first-order function in the mathematical sense. For that reason, the programming language is restricted to the applicative subset of Common Lisp. This means, for example, that there is no side-effects, no global variables, no destructive updates and all the functions must be total and terminate. Even with these restrictions, there is a close connection between ACL2 and Common Lisp: ACL2 primitives that are also Common Lisp primitives behave exactly in the same way, and this means that, in general, ACL2 programs can be executed in any compliant Common Lisp.

The ACL2 logic is a first-order logic, in which formulas are written in *prefix notation*; they are quantifier free and the variables in it are implicitly universally quantified. The logic includes axioms for propositional logic (with connectives `implies`, `and`, ...), equality (`equal`) and those describing the behavior of a subset of primitive Common Lisp functions. Rules of inference include those for propositional logic, equality and instantiation of variables. The logic also provides a principle of proof by induction that allows the user to prove a conjecture splitting it into cases and inductively assuming some instances of the conjecture that are smaller with respect to some well founded measure.

An interesting feature of ACL2 is that the same language is used to define programs and to specify properties of those programs. Every time a function is defined with `defun`, in addition to define a program, it is also introduced as an axiom in the logic (whenever it is proved to terminate for every input). Theorems and lemmas are stated in ACL2 by the `defthm` command, and this command also starts a proof attempt in the ACL2 theorem prover. In the ACL2 jargon a file containing definitions and statements that have been certified as admissible by the system is called a *book*.

The main proof techniques used by ACL2 in a proof attempt are simplification and induction. The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, in a deeper sense the system is interactive: very often non-trivial proofs are not found by the system in a first attempt and then it is needed to guide the prover by adding lemmas, suggested by a preconceived hand proof or by inspection of failed proofs. These lemmas are then used as rewrite rules in subsequent proof attempts. This kind of interaction with the system is called "The Method" by ACL2 authors.

In the following subsection, an introduction to an interesting ACL2 feature, that will be critical in our developments, is presented.

1.3.2 ACL2 encapsulates

In this subsection, an introduction to the ACL2 encapsulates is given. Encapsulates are a powerful tool which allow the structural development of theories. The encapsulation facility is much more general than sketched here, see [KMM00b, KMM00a, BGKM91, KM01a]. They will be used, in this memoir, in Section 4.3 and onward.

The *encapsulate principle* allows the introduction of function symbols in ACL2, without a completely specification of them, but just assuming some properties which partially define them. To be admissible and to preserve the consistency, there must exists functions (called "local witness") verifying the properties assumed as axioms. Once this is proved, the local witness definitions can be neglected, and the function symbols introduced with the encapsulate principle are partially specified by means of the assumed properties. Then, we can assume a property described by the formula `F` for the functions `f1`, . . ., `fn` if:

- `f1`, . . ., `fn` are new function symbols.
- There exist admissible definitions of n functions whose names are `f1`, . . ., `fn` such that the formula `F` can be proved for that functions.

If the encapsulate is admissible, the function symbols `f1`, . . ., `fn` are added to the ACL2 language and the formula `F` is added as axiom to the logic. It is worth noting that the local witness are only used to ensure the admissibility of the encapsulate, since the axioms associated with their definitions are not kept. Events introduced by means of an encapsulate preserve the consistency of the ACL2 logic, see [KM01b] for a rigorous proof of this fact.

In order to clarify this ACL2 mechanism an example will be presented. Let us suppose that we want to assume the existence of a binary function which is associative and commutative. The ACL2 command in charge of this task is the following one.

```
(encapsulate
    ; the signatures
    (((op * *) => *))

    ; the witnesses
    (local (defun op (a b) (+ a b)))

    ; the axioms
    (defthm op-associative
      (equal (op (op x y) z) (op x (op y z))))

    (defthm op-commutative
      (equal (op x y) (op y x))))
```

The first expression is a list with the arity description of the functions which are introduced with the encapsulate principle (called *signature*). In this case, we indicate that `op` is a function with two arguments which returns a unique value. The local witnesses are defined using `defun`, but they are declared as local by means of `local`. The properties assumed over the encapsulate functions are stated by means of `defthm`. The encapsulate can contain definitions and properties declared as local; however, once the admissibility of the encapsulate has been proved, just the non local definitions and properties are added to the ACL2 logic.

The functions defined by means of an encapsulate *cannot be executed*, since its partial specification can be not enough to deduce the value for every input. However, there exists a great advantage from the verification point of view; due to the fact that we are formalizing and verifying a generic result which can be instantiated for concrete cases later on. This issue will be presented in Subsection 6.1.1.

Chapter 2

A framework for computations in Algebraic Topology

Traditionally, Symbolic Computation systems, and Kenzo is no exception, have been oriented to research. This implies in particular, that development efforts in the area of Computer Algebra systems have been focussed on aspects such as the improvement of the efficiency or the extension of the applications scope. On the contrary, aspects such as the interaction with other systems or the development of friendly user interfaces are pushed into the background (it is worth noting that most of Computer Algebra systems use a command line as user interface). Things are a bit different in the case of widely spread commercial systems such as *Mathematica* or *Maple*, where some attention is also payed to connectivity issues or to special purpose user interfaces. But even in these cases the central focus is on the results of the calculations and not on the interaction with other kind of (software or human) agents.

The situation is, in any sense, similar in the area of interoperability among Symbolic Computation systems (including here both Computer Algebra systems and Proof Assistants tools). In this case, the emphasis has been put in the universality of the middleware (see, for instance, [CH97]). Even if important advances have been achieved, severe problems have appeared, too, such as difficulties in reusing previous proposals and the final obstacle of the speculative existence of a *definitive mathematical interlingua*. The irruption of XML technologies (and, in our context, of MathML [A$^+$08] and OpenMath [Con04]) has allowed standard knowledge management, but they are located at the *infrastructure* level, depending always on higher-level abstraction devices to put together different systems. Interestingly enough, the initiative SAGE [Ste] producing an integrated environment seems to have no use for XML standards, intercommunication being supported by ad-hoc SAGE mechanisms.

To sum up, in the symbolic computation area, we are always looking for *more powerful* systems (with more computation capacities or with more general expressiveness). However, it is the case that our systems became so powerful, that we can lose some interesting kinds of users or interactions. This situation was encountered in the design and development of *TutorMates* [GL$^+$09]. TutorMates is aimed at linking an educational front-end (based on Java) with the *Maxima* system [Sch09] (a Common Lisp Computer Algebra system specialized in symbolic operations but that also offers numerical capabilities such as arbitrary-precision arithmetic). The purpose of TutorMates was educational, so it was clear that many outputs given by Maxima were unsuitable for final users (students, and teachers, at high school level) depending on the degree and the topic learned in each TutorMates session. To give just an example, an imaginary solution to a quadratic equation has meaning only in certain courses. In this way, a *mediated* access to Maxima was designed. The central concept is an intermediary layer that communicates, by means of an extension of MathML, the front-end and Maxima. This approach is now transferred to the field of Symbolic Computation in Algebraic Topology, where the Kenzo system provides a complete set of calculation tools, which can be considered difficult to use by a non-Common Lisp trained user (typically, an Algebraic Topology student, teacher or researcher).

The most elaborated approach to increase the usability and accessibility of Kenzo was reported in [APRR05]. There, a remote access to Kenzo was devised, using *CORBA* [Gro] technology. An XML extension of MathML played a role there too, but just to give genericity to the connection (avoiding the definition in the CORBA Interface Description Language [Gro] of a different specification for each Kenzo class and datatype). There was no intention of taking profit from the semantics possibilities of MathML. Being useful, this approach ended in a prototype, and its enhancement and maintenance were difficult, due both to the low level characteristics of CORBA and to the pretentious aspiration of providing *full* access to Kenzo functionalities. We could classify the work of [APRR05] in the same line as [CH97] or the initiative [IAM], where the emphasis is put into powerful and generic access to symbolic computation engines.

Now, we have undertaken the task of devising a framework, from now on called Kenzo framework, which provides a *mediated* access to the Kenzo system, constraining the Kenzo functionality, but providing guidance to the user in his navigation on the system. The rest of this chapter is organized as follows. A brief overview of the architecture of the Kenzo framework is presented in Section 2.1. On the contrary, Section 2.2 is devoted to provide a detailed description of each one of the Kenzo framework components. Finally, the Kenzo framework execution flow is presented by means of an example in Section 2.3. Two ongoing works devoted to increase the computation capabilities of the Kenzo framework by means of ideas about remote and distributed computations are presented respectively in Section 2.4 and 2.5.

2.1 Framework architecture

When starting the project of developing a framework for Kenzo, several requirements were stated. Some of them were simply natural specifications, others were of a more problematic nature. Those issues are presented here as challenges to be fulfilled. These challenges largely determined the design decisions presented in this section. The most important challenges we faced were:

1. *Functionality.* The system should provide access to the Kenzo capabilities for constructing topological spaces and computing (homology and homotopy) groups.
2. *Extensibility of Kenzo.* The system design should be capable of evolving at the same time as the Kenzo system.
3. *Integration with other systems.* In spite of having Kenzo as main computation kernel, the system should be designed in such a way that it could support different connections to other symbolic manipulation systems (GAP [GAP] in computational algebra, for instance, or ACL2 [KM] from the theorem proving side).
4. *Interaction with different clients.* The system should be designed in such a way that it could support several ways of interaction (graphical user interfaces, web services, web applications, Computer Algebra systems, and so on).
5. *Efficiency.* The framework should be roughly equivalent to Kenzo in time and space efficiency.
6. *Error handling.* Our framework should forbid the user some manipulations raising errors, from both structural and semantical points of view.
7. *Representation of mathematical knowledge.* The representation of the data of our framework should be independent of the modules of the framework, that can be programmed in different programming languages.
8. *Communication of the mathematical knowledge.* The encoded data should be easily transferable both between the different modules of the framework and also outside the framework.

Let us observe that, as it is usual in system design, some decisions aimed to fulfill a concrete requirement could compromise other ones. The most important trade-off in our previous list is between requirements 2, 3, 4 (these three requirements are three different extensibility nuances) and 5. A layered architecture with complex mediators could produce poorer performance. A careless treatment of intermediary documents and files could also imply a great memory waste. The error handling (item 6) is closely related to the functionality included in the system (item 1), since the pure Kenzo system allows some instructions that are not desirable from the point of view of error managing (since they produce runtime errors), then, the system should avoid this kind of situations. Besides, error handling (item 6) could be in a conflict with efficiency, too, because dealing with semantical information at the external layers of an architecture can slow down the system as a whole. In addition, it would be very useful if the chosen encoding for the data, related to requirements 7 and 8, was able to include some knowledge in order to help the management of error handling. We have tried to deal with all these constraints while respecting the requirements guided by already proved methodologies and *patterns*.

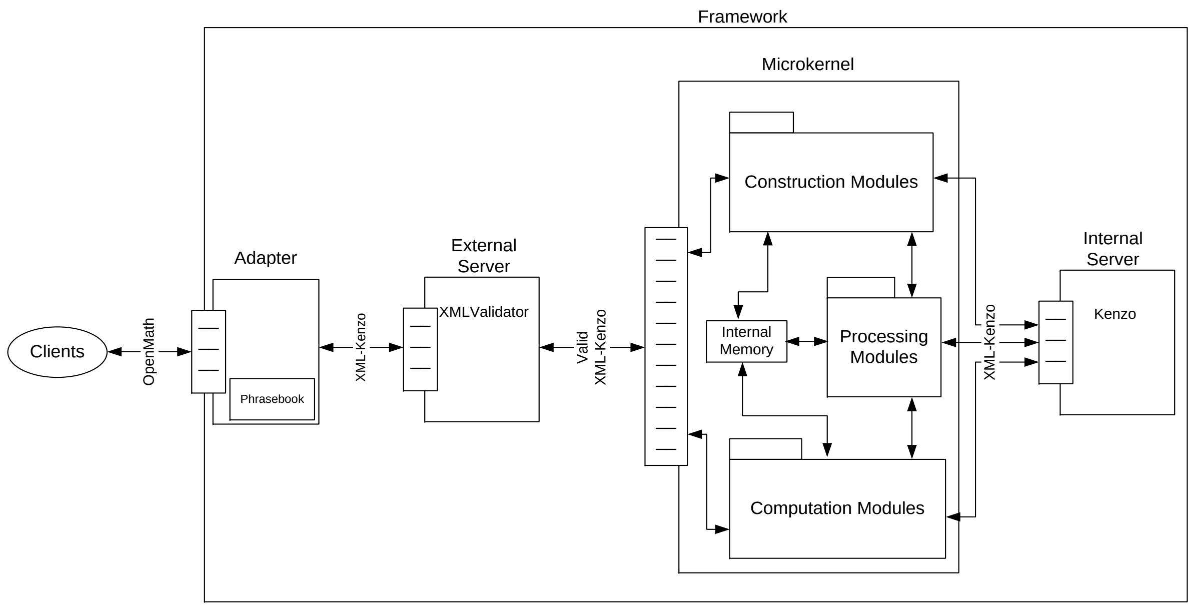


Figure 2.1: Architecture of the Kenzo framework based on the Microkernel pattern

The previous discussion led us to choose the *Microkernel architectural pattern* [B$^+$96, B$^+$07] to organize the system and XML to encode the mathematical and systemic knowledge.

The Microkernel pattern gives a global view as a *platform*, in terminology of [B$^+$96], which implements a virtual machine with applications running on top of it, namely a *framework* (in the same terminology). The *Microkernel* architectural pattern applies to software systems that must be able to adapt to changing systems requirements. It separates a minimal functional core from extended functionality and customer-specific parts. This pattern defines five kinds of participating *components*: *internal servers*, *external servers*, *adapters*, *clients* and the *microkernel*.

The *microkernel* is the main component and includes functionality that enables other components running in separate processes to communicate with each other. It is also in charge of maintaining system-wide resources such as files or processes. Core functionality that cannot be implemented within the microkernel is separated in *internal servers*. An *external server* is a component that uses the microkernel for implementing its own view of the underlying application domain. The external server receives requests from client applications using the communication facilities provided by the *adapter*.

A high level perspective of the architecture of our system, based on this pattern, as a whole is shown in Figure 2.1.

Let us present a brief overview of the framework components, a more detailed description of each one of them will be provided in Section 2.2.

First of all, let us give some flavor about the concept of mediated access in our framework. It is worth noting that the mediated access is not provided just by one of the framework components but by the whole system.

In our framework the mediated access is the knowledge which guides a user to interact correctly with the system avoiding errors, that was one of the challenges of our system. There are different kinds of knowledge included in our framework, and they categorize the errors managed in our system as follows:

- Related to the construction of spaces:
  - mathematical restrictions:
    - *type restrictions*: a space constructor can only be applied over objects of a concrete type (and, of course, all its subtypes); for instance, the "classifying space" constructor can only be applied over a space which is a simplicial group; and,
    - restrictions of the arguments of space constructors:
      - *independent argument restrictions*: the value of an argument of the constructor must satisfy some properties which are independent from the rest of the arguments of the space constructor; for instance, the "Moore" constructor takes p and n as arguments to construct the moore space $M(\mathbb{Z}/p\mathbb{Z}, n)$, both p and n are restricted to be natural numbers and p must be higher than 1; those are examples of independent argument restrictions;
      - *functional dependent restrictions*: the value of an argument of the space constructor depends on the value of another one; for instance, in the "Moore" constructor the value of n must be higher or equal than $2p - 4$; that is a functional dependent restriction.
  - *Kenzo implementation argument restrictions*: some constrains are imposed by the Kenzo implementation of spaces; for instance, the "Sphere" constructor takes as argument a natural number that Kenzo constraints to be lower than 15.
- Related to the computation of (homology and homotopy) groups:
  - *restriction of the computation dimension*: in the case of computing $H_n(X)$ the value of n must be higher or equal than 0; and in the case of computing $\pi_n(X)$ the value of n must be higher or equal than 1; and,
  - *reduction degree restrictions*: the notion of reduction degree was explained in Subsection 1.2.6. In the case of computing $H_n(X)$ the reduction degree of X must be higher or equal than 0; and in the case of computing $\pi_n(X)$ the reduction degree of X must be higher or equal than 1.

All this knowledge is spread throughout the Kenzo framework in its components. These components are going to be briefly explained in the following paragraphs.

As we have said previously, XML is the chosen technology to encode the data of our framework since it perfectly fulfills requirements 7 and 8, and also let us encode some mathematical knowledge that will be useful to manage error handling. In particular, we have defined an XML language called *XML-Kenzo*. This language is used for data interchange among the different components of the framework.

The XML-Kenzo specification is employed to represent some of the knowledge included in the framework, namely the knowledge which allows us to manage the following restrictions:

1. type restrictions,
2. independent argument restrictions of the space constructors,
3. implementation restrictions of the space constructors arguments, and
4. restriction of the dimension in computations.

On the contrary, the rest of constraints (functional dependent restrictions of the arguments of the constructors and reduction degree restrictions) cannot be represented in XML-Kenzo; so, they have been included in a different way in the framework.

Let us present now the rest of the system components.

Kenzo itself, wrapped with an interface based on XML-Kenzo, is acting as *internal server* and is used as the core to perform computations in our framework. It is worth noting that the functionality available from the internal server is a subset of the Kenzo one. Namely, the functionality that allows us to construct spaces of regular usage and to perform computations of groups is accessible through the Internal Server.

Due to the Microkernel pattern organization new computations and deduction engines can be incorporated as internal servers; solving the third requirement, the integration with other systems.

The main component of this architecture, the microkernel, is responsible for managing all system resources, maintains information about resources and allows access to them in a coordinated and systematic way. The microkernel acting as intermediary layer is based on an XML-Kenzo processor, allowing both a link with Kenzo and including the management of constraints not handled in the XML-Kenzo specification. The functionality exposed by the microkernel is related to the Kenzo way of working, providing, on the one hand, the way of constructing spaces (construction modules) and, on the other hand, the functionality to perform computations (computation modules). Besides, the fifth requirement (efficiency) has been solved at this level by programming a *memoization* strategy, a technique also used in Kenzo, see Subsection 1.2.5. This has required to include an improvement of our own in the Microkernel pattern: an internal memory used to the optimization tasks. As a result, the waiting time is to a great extent similar to that of the original Kenzo system. Moreover, as we have already said, the sixth requirement (error handling) is partially fulfilled at the microkernel level. To be more concrete, the modules of the microkernel are in charge of validating the restrictions that were not included in the XML-Kenzo specification; that is to say, functional dependent restrictions of the arguments of the constructors and reduction degree restrictions. In addition, processing modules provide several enhancements to the Kenzo system. The functionality of the processing modules is not exposed by the microkernel but it is used by both construction and computation modules.

The external server exports the functionality of the microkernel, that is, the mechanisms to construct spaces and to compute groups. Moreover, the external server is in charge of validating the knowledge included in the XML-Kenzo specification. Namely, the external server is in charge of checking: type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, and restriction of the dimension in computations. Therefore, *requests* arriving to the microkernel always satisfy these restrictions; that is to say, they satisfied the XML-Kenzo specification, this kind of requests are called valid XML-Kenzo requests.

Finally, the *adapter* exposes the functionality provided by the external server to clients. However, due to the fact that XML-Kenzo is ad-hoc for our framework is not sensible to use this language to communicate with the outside. In order to grapple with this problem, the OpenMath XML standard [Con04] has been employed. Then, the *adapter* converts the interface provided by the external server, based on XML-Kenzo, into a more suitable interface, based on OpenMath, which can be used by different clients (for instance, Graphical User Interfaces, web applications or web services) without knowing the internal representation of the data of our framework, then, the fourth aspect requested to our framework, the interaction with different clients, is achieved. Besides, as OpenMath is also an XML language, requirements 7 and 8 are also satisfied at this level. The process to convert from/to OpenMath requests to/from XML-Kenzo requests is tackled by a program included in the adapter called *Phrasebook* [Con04].

The communication between the different components of the system is based on a message style model, that is, the modules of the framework are communicated in a direct and synchronous way.

The main complaint to this framework could be the restriction of the full capabilities of the Kenzo system, since just the main Kenzo functionality is included, however the interaction with it is easier and enriched.

2.2 Framework components

This section is devoted to present a detailed description of each one of the Kenzo framework components.

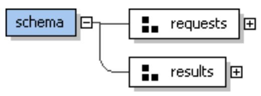


Figure 2.2: Main elements of XML-Kenzo

2.2.1 XML-Kenzo

One of the most important decisions in the development of our framework was the language employed to represent the mathematical data inside the framework. The representation language should be independent of the implementation language used, and the represented data must be easily interchangeable among the different components of the framework, both current components and future extensions. These requirements oriented us towards our solution: using XML technology [B$^+$08]. Once we chose XML to represent data inside our framework, we should decide whether we extended a mathematical XML language, MathML or OpenMath, or if we defined a fresh one. *MathML* provides several facilities to represent data in the web but is not suitable to represent content, see [KSN10] for a survey about this question. *OpenMath* is very useful in the communication with the outside of the framework, but, we have to extend it with the problems associated to this task, since OpenMath is a general purpose standard which cannot be fully adapted to our needs. So, we opted for defining from scratch an XML language adapted to our needs. The new XML language was called *XML-Kenzo*.

It is worth noting that the XML-Kenzo language is employed inside the framework; on the contrary, to communicate the framework with the outside we use OpenMath since it is more suitable for that task.

The specification of XML-Kenzo is based both on Kenzo and on mathematical conventions and is provided by means of an *XML schema* definition (XSD), see [E$^+$07]. The formal specification of XML-Kenzo defines the structure of the objects indicating their restrictions and providing valid combinations.

There are two types, *groups* in terminology of XML schema definitions, of elements in the XML-Kenzo specification based on the usual interaction between a client and a software system. To be more concrete, the interaction between a client and a software system consists of the user sending *requests* to the system and the system returning *results* to the client. Therefore, we have defined two types: `requests` and `results`, see Figure 2.2.

Let us focus first on the `requests` XML-Kenzo group. As we explained in Subsection 1.2.2, Kenzo includes functions with two different aims: construct spaces and perform computations. This situation is reflected in the specification of XML-Kenzo, where two elements belong to the `requests` group. Namely, the elements: `operation` to represent the computation functions and `constructor` to represent the different spaces that can be built, see Figure 2.3.

Now, let us focus on the `constructor` element. When a user has decided to con-

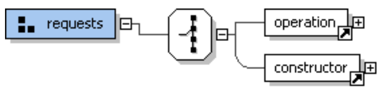


Figure 2.3: Requests XML-Kenzo group

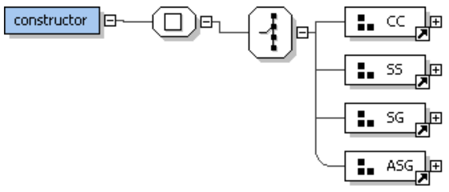


Figure 2.4: Types of spaces in XML-Kenzo

struct a space in Kenzo, he should decide which type he wants to build: a simplicial set, a simplicial group and so on (see all the possible types in Subsection 1.2.1). As was explained in Section 1.2.2, the Kenzo system provides useful functions to create interesting objects of regular usage, which can belong to four types: chain complexes, simplicial sets, simplicial groups and abelian simplicial groups. All these functions of regular usage are represented in our XML-Kenzo specification and have a unique XML-Kenzo representation. So, four types are defined in the XML-Kenzo specification: `CC` (Chain Complex), `SS` (Simplicial Set), `SG` (Simplicial Group) and `ASG` (Abelian Simplicial Group). Then the child of the `constructor` element must be an element of one of these types, see Figure 2.4. The following elements, gathered by types, represent the spaces that can be constructed in our framework (the complete list of spaces that can be constructed and their type can be seen in Figure 2.5):

- The `SS` type contains the elements that represent most of the constructors of Kenzo. Some of them construct spaces from scratch such as `sphere`, `moore-space`, `build-finite-ss`, and so on. Others construct spaces from other ones; for instance, `crts-prdc` which represents the $\underline{C}a\underline{r}t\underline{e}\underline{s}ian$ $\underline{pr}o\underline{d}u\underline{c}t$.

- The `SG` type contains the elements whose arguments are simplicial groups; that is to say, both loop spaces, `loop-space`, and classifying spaces, `classifying-space`.

- `ASG` contains the elements to construct Eilenberg MacLane spaces of type $K(\mathbb{Z}, n)$, `k-z`, and $K(\mathbb{Z}/2\mathbb{Z}, n)$, `k-z2`.

- The `CC` type contains elements that correspond with the constructors defined at the algebraic level but not at the simplicial one. The element `chain-complex` constructs a simple chain complex such as the unit chain complex or the circle. The rest of the elements of this type construct spaces from other ones, for instance `tnsr-prdc` which represents the $\underline{t}e\underline{ns}o\underline{r}$ $\underline{pr}o\underline{d}u\underline{c}t$.

As we said previously, XML-Kenzo can be used to represent knowledge and restrictions about its elements. We commented in Section 2.1 that there are three kinds of

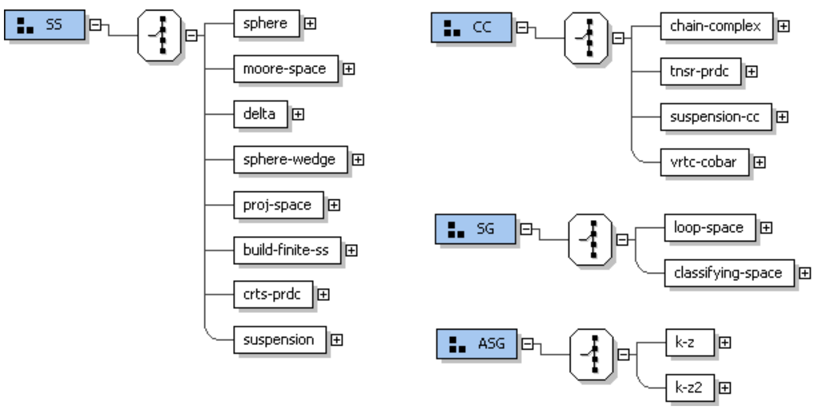


Figure 2.5: Constructor groups in XML-Kenzo

restrictions related to the construction of spaces that are dealt with in the XML-Kenzo specification; namely, type restrictions, independent argument restrictions of the space constructors and implementation restrictions of the space constructors. Let us present how we handle these restrictions in the specification of the XML-Kenzo language.

Let us focus first on type restrictions. These constraints are applied over the constructors of spaces from other ones. As we explained in Subsection 1.2.1, Kenzo is, in its pure mode, an untyped system (or rather, a dynamically typed system), inheriting its power and its weakness from Common Lisp. Thus, for instance, in Kenzo a user could apply a constructor to an object without satisfying its input specification. For instance, the method constructing the classifying space of a simplicial group could be called with an argument which is a simplicial set without a group structure over it. Then, at run-time, Common Lisp would raise an error informing the user of this restriction. This is shown in the following fragment of a Kenzo session.

```
> (setf s4 (sphere 4)) ✠
[K1 Simplicial-Set]
> (classifying-space s4) ✠
Error: No methods applicable for generic function #<STANDARD-GENERIC-FUNCTION
CLASSIFYING-SPACE> with args ([K1 Simplicial-Set]) of classes (SIMPLICIAL-SET)
[condition type: PROGRAM-ERROR]
```

With the first command, we construct the sphere of dimension `4`, a simplicial set. Thus, when in the second command we try to construct the classifying space of a simplicial set, the Common Lisp Object System (*CLOS*) raises an error.

This kind of error is controlled in our framework thanks to the XML-Kenzo specification, since the inputs for the operations between spaces can be only selected among the spaces with suitable characteristics (Figure 2.6 shows the specification of the `classifying-space` element in XML-Kenzo, this element only allows as child an element either from the `SG` or the `ASG` group). This enriches Kenzo with a small (semantical) type system.

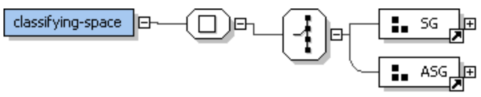


Figure 2.6: Classifying specification in XML-Kenzo

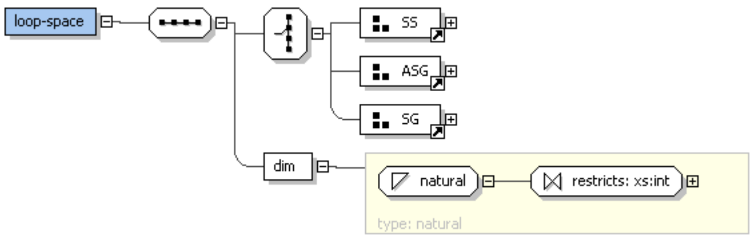


Figure 2.7: Loop space specification in XML-Kenzo

It is worth noting that the inheritance relations between Kenzo types, see Section 1.2.1, cannot be directly specified using an XML schema, so, we tackle this situation in the following way. If we have the types A and B, represented in the XML schema as the groups `A` and `B` respectively, where B inherits from A and a function f that can be applied over the objects of the type A, and of course also over the objects of type B, then the element of the XML schema that represents the function f has as child an element that belongs either to the `A` or the `B` group, and this must be included in the schema in an explicit way. For instance, the `loop-space` Kenzo function is applied over a simplicial set, but, as can be seen in Figure 1.2 of Subsection 1.2.1, simplicial groups and abelian simplicial groups are subtypes of simplicial sets, so the `loop-space` element of XML-Kenzo has as child an element belonging either to the `SS`, the `SG` or the `ASG` group. Moreover, it also has a child that represents the dimension of the loop space, as can be seen in Figure 2.7.

Both second and third kinds of restrictions, that are, independent argument restrictions of the space constructors and implementation restrictions of the space constructors, are also coped with in the XML-Kenzo specification. These restrictions are always applied over objects constructed from scratch, such as spheres, Moore spaces, Eilenberg MacLane spaces and so on. The restrictions over the arguments of those functions are translated into the restrictions over the elements encoding them. For instance, spheres only have sense if their dimension is a natural number (an independent argument restriction). In addition, the function that constructs a sphere in Kenzo has as argument a natural number n, such that $0 < n < 15$; this restriction is included in the XML-Kenzo specification of the `sphere` element as is shown in Figure 2.8. This kind of restrictions can be included in the specification of the XML-Kenzo language without any special hindrance.

On the contrary, functional dependent restrictions of the arguments cannot be imposed in the XML-Kenzo specification, since restrictions about the value of an element depending on the value of other elements cannot be defined in XML schemas.

Once we have presented the way of encoding Kenzo constructors in XML-Kenzo, we

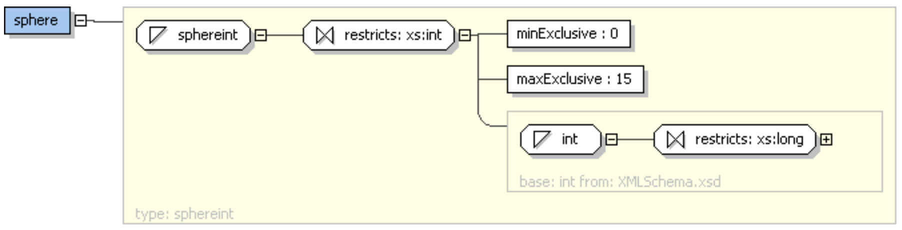


Figure 2.8: Sphere specification in XML-Kenzo

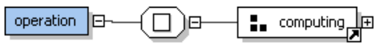


Figure 2.9: Operation element in XML-Kenzo

can construct different XML-Kenzo objects such as the space $\Omega^3(S^4)$ which is represented as the following XML-Kenzo object:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

It is worth noting that the type of the elements (`requests` in the case of the `constructor` element, `SG` in the case of the `loop-space` element and `SS` in the case of the `sphere` element) is not included in the XML-Kenzo object since it is implicit to the element (an element defined in an XML schema only belongs to one type). From now on, we will call *construction requests* to the XML-Kenzo objects whose root element is `constructor`.

Let us focus now on the other element of the `requests` group: `operation`. The `operation` element represents requests which ask to the system the computation of homology and homotopy groups. The `operation` element has as child one element of the `computing` group, see Figure 2.9. The computation of homology and homotopy groups is represented by means of the elements `homology` and `homotopy` respectively. Both elements belong to the `computing` group, see Figure 2.10. The structure of both `homology` and `homotopy` elements is the same, they have two children, the first one is a space of one of the groups `CC`, `SS`, `SG` or `ASG`; and the second one is an element called `dim`.

As we commented in Section 2.1, we can handle one of the restrictions related to computations, that is the restriction of the dimension in homology and homotopy computations. This is translated in the XML-Kenzo specification into a constraint of the value of the `dim` element of both `homology` and `homotopy` elements. Nevertheless, the other restriction related to computations (the reduction degree restriction of the space) cannot be managed at this level since its value must be computed from the description of the space.

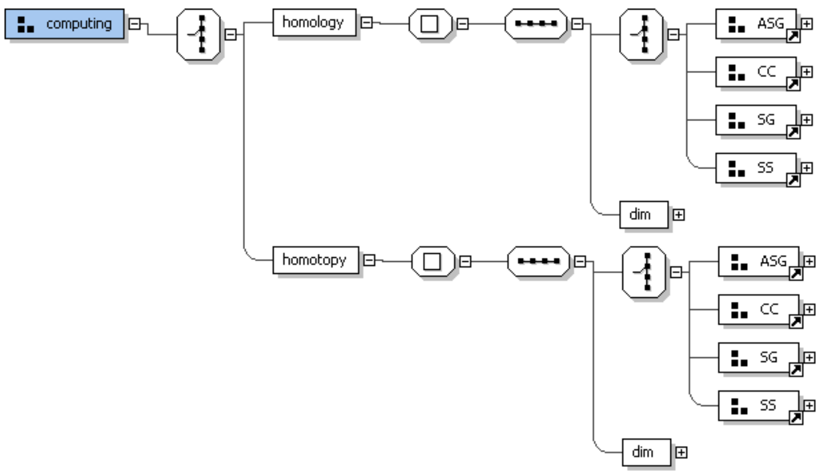


Figure 2.10: Computing group of XML-Kenzo

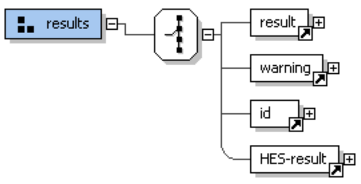


Figure 2.11: Elements of XML-Kenzo results group

Once we have presented the way of encoding Kenzo operations in XML-Kenzo, we can construct different XML-Kenzo objects such as the operation $H_5(\Omega^3(S^4))$ which is represented by the following XML-Kenzo object

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  </homology>
</operation>
```

From now on, we will call *computation requests* to the XML-Kenzo objects whose root element is `operation`. We will call *valid XML-Kenzo requests* to the XML-Kenzo objects of `requests` type which satisfy the XML-Kenzo specification. This finishes the description of the elements of the `requests` XML-Kenzo group.

Let us present now the `results` XML-Kenzo group. We have defined four elements belonging to this group: `id`, `warning`, `result` and `HES-result`, see Figure 2.11.

In our framework, objects have associated a unique identification number. To communicate this identification among the modules of the framework we use the `id` element whose value is a natural number (the unique identification number).

```
<id> 1 </id>
```

The `warning` element is used to represent errors returned by the framework and its value is a string with the correspondent error in natural language.

```
<warning> The dimension of the sphere must be a natural number </warning>
```

The `result` element is used to return the result of a computation by means of an undefined number of `components` whose value is a natural number.

```
<result>
   <component> 2 </component>
   <component> 3 </component>
</result>
```

If the above `result` element is returned, that means that the result is $\mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}/3\mathbb{Z}$.

Finally, `HES-result` provides a result, by means of an undefined number of components whose value is a natural number, and an explanation of the reasoning followed to obtain the result, by means of a string.

```
<HES-result>
 <component>0</component>
 <explanation>
    The space was a contractible space since it was the cartesian product of
    two contractible spaces. The homotopy groups of a contractible space are
    always null. Then, the homotopy group of the space in dimension 3 is null.
 </explanation>
</HES-result>
```

To sum up, the XML-Kenzo schema specifies two kind of objects: requests and results. There are two kinds of requests: to construct spaces and to compute groups. Moreover, most of the restrictions of the functions devoted to construct spaces and compute groups are imposed in XML-Kenzo, in this way some knowledge is provided with the XML-Kenzo specification.

As we will see throughout the next subsections, XML-Kenzo played a key role in the development of the rest of the framework components.

The complete specification of the XML-Kenzo language can be seen in [Her11].

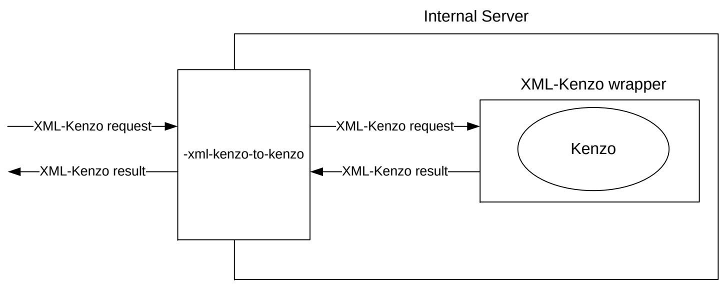


Figure 2.12: Internal Server

2.2.2 Internal Server

The *internal server* is a Common Lisp *component* that provides access to the Kenzo functionality through an XML-Kenzo interface. This module is split in three parts (see Figure 2.12): the full Kenzo system, an XML-Kenzo wrapper for Kenzo, that is a bunch of Common Lisp files that defines a Kenzo interface based on XML-Kenzo, and also the transformation from XML-Kenzo objects to their Kenzo encoding and viceversa; and last but not least, an *interface* which offers a *service* to use Kenzo through the XML-Kenzo wrapper.

We have talked at length about Kenzo, see Section 1.2; so let us focus on both the interface and the XML-Kenzo wrapper. The interface provides a service called `xml-kenzo-to-kenzo` which allows one to access to the functionality exported by the XML-Kenzo wrapper. The XML-Kenzo wrapper provides access to a subset of the Kenzo functionality, namely the functionality specified in the XML-Kenzo language by means of the elements of the `requests` group. Hence, from this interface we can construct the topological spaces of regular usage in Kenzo (such as spheres, Moore spaces, loop spaces and so on), and compute homology and homotopy groups.

The workflow when the `xml-kenzo-to-kenzo` service is invoked is as follows. If the request is a construction request, the internal server transforms the XML-Kenzo request into a Kenzo instruction by means of a function called `xml-kenzo-to-kenzo`. Afterwards, the Kenzo instruction is executed in the Kenzo kernel, and then as a result a Kenzo object is obtained. The result returned by the internal server is the unique identifier of the Kenzo object, the value of the slot `idnm` of a Kenzo instance (see Section 1.2.1). This identification number is returned by means of an `id` XML-Kenzo object. For instance, if the internal server receives the request:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

the instruction `(loop-space (sphere 4) 3)` is executed in the Kenzo kernel. As a result an object of the Simplicial Group class is constructed, and the identifier of that object is returned, in a fresh Kenzo session the result will be:

```
<id>30</id>
```

In the case of computation requests, the internal server transforms the XML-Kenzo request into a Kenzo instruction by means of the `xml-kenzo-to-kenzo` function. Afterwards, the Kenzo instruction is executed in the Kenzo kernel, and as a result a group is obtained. The result returned by the internal server is the group codified in a `result` XML-Kenzo object. For instance, if the internal server receives the request:

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  </homology>
</constructor>
```

the instruction `(homology (loop-space (sphere 4) 3) 5)` is executed in the Kenzo kernel. As a result, the group $\mathbb{Z}/2\mathbb{Z}$ is obtained and returned by the internal server using an XML-Kenzo `results` object whose root is `result`.

```
<result>
  <component>2</component>
</result>
```

The answer returned by the internal server is always an identifier, in the case of a *construction request*, or a group, in the case of a *computation request*. What we want to highlight is that errors never happen since all the dangerous situations are stopped in a more external level of the framework (some of them were dealt with in the specification of XML-Kenzo, others will be handled in the microkernel), as we will see in the following subsections, and therefore, the requests received by the internal server are always safe.

The main disadvantage of the internal server with respect to Kenzo is the restriction of the full capabilities of the Kenzo kernel. Let us note that the restriction is only related to the construction of spaces where, as we explained in Subsection 2.2.1, only the spaces that can be constructed using the Kenzo ad-hoc functions are available; on the contrary, the Kenzo capability to perform computations remains untouched. Nevertheless, thanks to the combination of the internal server and XML-Kenzo the interaction between the modules of the framework, that can be programmed in a language different from Common Lisp, and Kenzo is easier since the communication is performed by means of XML-Kenzo objects and the internal server is in charge of converting them from/to Kenzo instructions.

2.2.3 Microkernel

The *microkernel* is the main component of the framework, it is devoted to the management of resources and includes some of the knowledge of the framework (the rest of knowledge was provided by the XML-Kenzo specification). The microkernel is split in four constituents (see Figure 2.1): the construction modules, the computation modules, the processing modules and the internal memory. The interaction with the microkernel is provided by means of an *interface* based on XML-Kenzo. This *interface* only gives access to the functionality of the construction and computation modules.

2.2.3.1 Internal Memory

The first component of the microkernel that we are going to describe is the *internal memory*. The *internal memory* is a Common Lisp module which stores both the constructed spaces and the computed results. That is to say, the internal memory stores the state of the microkernel.

The storage of constructed spaces and computed results avoids unnecessary communications between the microkernel and the internal server. Therefore, the efficiency of the framework is improved. In this component, for both constructed spaces and computed results, a *memoization* strategy has been implemented.

Let us focus first on the spaces constructed in the microkernel.

At this moment, the Kenzo framework just interacts with an internal server that is the Kenzo system. However, in the future we want to include different internal servers. This means that objects of very different nature (such as spaces or groups) are going to coexist in the Kenzo framework and, in particular, in the microkernel.

Taking this question into account, we decided to design a class hierarchy with a main class; and specialize this class with different subclasses for the objects coming from the different internal servers.

Therefore, we have defined the main class of microkernel objects, `MK-OBJECT`, whose definition is:

```
(DEFCLASS MK-OBJECT ()
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *number-of-objects*) :reader idnm)
     ;; ORiGiN
     (orgn :type string :initarg :orgn :reader orgn)))
```

This class has two slots (which are common for all the microkernel objects):

- `idnm`, an integer being the object identifier for this system. This is generated by the microkernel in a sequential way, each time a new object is created.
- `orgn`, a string containing the XML-Kenzo object that is the *origin* of the space. This comment is unique and is important, because when a module constructs a new `mk-object` instance, it uses the XML-Kenzo string to search in a specific list, `*object-list*`, if the object has not been already built. So, one avoids the duplication of instances of the same object.

The `*object-list*` list stores the state of the microkernel related to the constructed objects.

As we have said, we are going to have different specializations of the `MK-OBJECT` class, but at this moment we just have one devoted to represent Kenzo spaces in the microkernel. Namely, we have defined a subclass of the `MK-OBJECT` class, the class `MK-SPACE-KENZO`, whose definition is:

```
(DEFCLASS MK-SPACE-KENZO (MK-OBJECT)
    (;; REduction DEgree
     (rede :type fixnum :initarg :rede :reader rede)
     ;; Kenzo IDentification NuMber
     (kidnm :type fixnum :initarg :kidnm :reader idnm)
```

This class has two additional slots to the ones of the `MK-OBJECT` class:

- `rede`, an integer giving the reduction degree, see Subsection 1.2.6, of the space.
- `kidnm`, an integer being the object identifier inside the Kenzo system.

Both `idnm` and `kidnm` can be used to identify the space, however as we have seen in Subsection 1.2.4, the number of objects in Kenzo grows incredibly fast when Kenzo performs computations. Then, we have decided to use a fresh identifier for the objects constructed in the microkernel hiding the intermediary objects built in Kenzo. Then, the value of the slot `kidnm` identifies the space in the Kenzo component of the internal server, and the value of the slot `idnm` identifies the space in the microkernel.

To sum up, Kenzo spaces are represented in the microkernel as instances of the `MK-SPACE-KENZO` class. In addition, as objects of the microkernel they are stored in the `*object-list*` list.

Let us explain now the memoization strategy implemented in the internal memory to handle computed results. As we have seen in Subsection 1.2.5, computing some groups in Kenzo can require a substantial amount of time. Kenzo is the kernel of our framework, so this situation also happens in our framework. Therefore, it seems desirable to store the computations of groups somewhere once they have been computed, in order to avoid the re-computation in Kenzo. The microkernel stores the groups associated with a concrete space in the internal memory when a computation is executed for the first time, and if this computation is executed again later, the group is simply looked up and returned, without further execution.

This means that the behavior of the functions which compute the groups depends on whether the asked group for an space has been already computed. Otherwise, the group must be stored after it has been calculated. These two extra tasks are done by two pair of functions that are implemented in the computation modules, namely the *testers* and the *setters*. The testers take as arguments the identifier of the object, that is stored in the `idnm` slot of the `mk-space-kenzo` instance, and the dimension `n` of the group and return the result or `nil` according to whether the `n`-th homology (or homotopy) group of the space whose identifier is `idnm` had already been stored. The setters take as arguments the identifier `idnm` of the object, the dimension `n`, and the result `result` of computing the `n`-th homology (or homotopy) group of the object `idnm` and put the result into the internal memory, where the result can directly look it up. The main procedures, implemented in the computing modules, are called the *getters*, and from the preceding discussion it is seen that there must really be at least two methods for the getters. One method is used when the tester returns `nil`; it is the method which first does the real computation by means of the Kenzo functions and then executes the setter with the computed value. A second method is used when the tester does not return `nil`; it simply returns the stored value.

Up to now, we have explained the implemented strategy but not the way of storing the results. This is an important issue for efficiency reasons and has been tackled by means of efficient data structures, namely *hash tables* and arrays.

To store computed results, we use two hash tables, one for the computations related to homology groups and another one for the computations of homotopy groups. The identification number of each object, slot `idnm` of the `mk-object` class, acts as key into the hash tables. The value associated to each key is an array where the n-th homology (or homotopy) group of the object associated with the identification number is stored in the position n.

Then, if we try to compute several times the fifth homology group of the space $\Omega^3(S^4 \times S^4)$, we can see the profit of using the memoization technique in the microkernel. In the first computation, the microkernel takes almost 22 minutes, but in a further trial the result is returned in just a second since it has been stored in the internal memory.

It is worth noting that the values stored in the hash tables are not kept from one session to another. The reader can wonder the reason to not reuse computations from a previous session storing the results in a persistent way. At least two different approaches can be considered to manage results from other sessions.

- We could keep on using our hash tables to store results, the main improvement would consist of saving the results stored in the hash tables at the end of the session and loading the stored results at the beginning of each session. Advantage: we use hash tables that are efficient data structures. Disadvantage: when the number of results increase, the time to load all the results in memory and the amount of memory used increase, too.

- We could use a database to store in a persistent way the results instead of keeping the results in hash tables. Advantage: efficient persistence storage of results. Disadvantage: since we are working with XML data, our database should be an XML database, and this kind of databases are usually big and the search of results in these databases is less efficient than the search in hash tables.

Thus, additional research is needed to achieve an efficient storage of results to be reused in different sessions. Therefore, at this moment we use the approach previously presented without saving results from one session to another.

Thanks to this memoization strategy implemented in the internal memory the framework efficiency increases.

The state of the microkernel is obtained gathering both the `*object-list*` list of constructed spaces and the two hash tables of computed results.

2.2.3.2 Processing Modules

The processing modules are three Common Lisp modules that are in charge of providing some enhancements to the construction of spaces and the computation of groups. The functionality of these modules is not available outside the microkernel but it is used by both construction and computation modules.

2.2.3.2.1 Reduction degree module The reduction degree module is in charge of managing the reduction degree of spaces, see Subsection 1.2.6. Kenzo allows the user to construct spaces whose reduction degree is lower than 0 (as the loop space iterated three times of the sphere of dimension 2). In these spaces some operations (for instance, the computation of the set of faces of a simplex) can be achieved without any problems. On the contrary, theoretical results ensure that the homology groups are not of finite type, and then they cannot be computed. In pure Kenzo, the user could ask for a homology group of such space, catching a runtime error, as is shown in the following fragment of a Kenzo session:

```
> (setf o2s2 (loop-space (sphere 2) 3)) ✠
[K18 Simplicial-Group]
> (homology o2s2 1) ✠
Error: ‘NIL’ is not of the expected type ‘NUMBER’
[condition type: TYPE-ERROR]
```

The reduction degree module is used to avoid this kind of errors in our framework. To this aim, the reduction degree module is implemented as a small expert system, computing, in a symbolic way (that is to say, working with the XML-Kenzo description of the spaces, and not with the spaces themselves), the reduction degree of the space. The set of rules that allows the reduction degree module to obtain the reduction degree of a space was given in Subsection 1.2.6, and the computational price of using this small rule-based system is negligible with respect to ordinary computations in Algebraic Topology.

As we have seen in Subsubsection 2.2.3.1, the reduction degree of a space will be assigned to the `rede` slot of the `mk-space-kenzo` instance in the construction of the object.

The relevance of the reduction degree, and hence of this module, is due to the computation modules, since when a computation is demanded, the system monitors if the reduction degree of the space allows the computation of the homology or homotopy group, or whether a warning must be returned as answer. In this way, the dangerous requests related to the reduction degree are always stopped at the microkernel and never arrive to the internal server, avoiding operations which would raise errors.

2.2.3.2.2 Homotopy algorithm module The homotopy algorithm module (from now on, HAM) is in charge of chaining methods in order to compute some homotopy groups. In pure Kenzo, there is no final function allowing the user to compute homotopy groups. Instead, there is a number of complex algorithms, allowing a user to chain them to get some homotopy groups. The HAM is in charge of chaining the different algorithms presented in Kenzo to reach the final objective.

Moreover, Kenzo, in its current version, has limited capabilities to compute homotopy groups (depending on the homology of Eilenberg MacLane spaces that are only partially implemented in Kenzo), so the *chaining* of algorithms cannot be universal. Thus the homotopy algorithm module processes the call for a homotopy group, making some requests to the Kenzo kernel (computing some intermediary homology groups, for instance) before deciding if the computation is possible or not.

Let us present the procedure implemented in HAM. The underlying mathematics of the algorithm were explained in a detailed way in [Rea94].

Algorithm 2.1.

1. Find the lower dimension $0 < r \leq n$, such that the r-th homology group of X is not null.

   (a) If r does not exist (i.e., $H_q(X) = 0$ for all $0 < q \leq n$), $\pi_n(X)$ is null (applying the Hurewicz theorem, see [Whi78]).

   (b) If $r = n$, $\pi_n(X) = \pi_r(X) = H_r(X)$ (applying the Hurewicz theorem, see [Whi78]).
   (c) If $r < n$,
      i. If the r-th homology group is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$ go to step 2.
      ii. Otherwise, return a `warning` XML-Kenzo object informing that the homotopy group is not computable in the current version.

2. Compute the r-th fundamental cohomology class of X, $h_r = [\chi_r] \in H^r(X, \pi_r)$, where $\pi_r = \pi_r(X) = H_r(X)$.

3. Build the fibration over X canonically associated to the above cohomology class:

$$\begin{tikzcd}
K(\pi_r, r-1) \arrow[d] \\
T \equiv X' \equiv X^{(r+1)} = X \times_{\tau_{\chi r}} K(\pi_r, r-1) \arrow[d] \\
X
\end{tikzcd}$$

4. Get the total space $T = X^{(r+1)}$ associated to the above twisting operator.
   (a) If $r + 1 = n$, $\pi_n(X) = H_n(T)$.
   (b) If $n > r + 1$ and $H_{r+1}(T) = \mathbb{Z}$ or $H_{r+1}(T) = \mathbb{Z}/2\mathbb{Z}$, increase one unit the value of r and go to step 2 but using the space T instead of X.

This is the algorithm implemented in HAM.

2.2.3.2.3 Homotopy expert system As we explained in the previous paragraph, the HAM is able to compute some homotopy groups of some spaces by means of the algorithm presented in [Rea94]. Nevertheless, there are several homotopy groups that are not reachable by the Kenzo framework using the HAM (the implementation of the algorithm is limited to the spaces which first non null homology group is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$). To overcome this gap, we undertook the task of developing a homotopy *expert system* (from now on, HES) from the extensive literature about this topic; taking profit of theoretical knowledge contained in theorems. The knowledge (that is, the theorems about homotopy groups) can be found in different books, for instance, [CM95, Mau96, Hat02].

The HES is a rule-based system [GR05]. Rule-based systems do not represent knowledge in a declarative and static way (as a bunch of things that are true), but they represent knowledge in terms of a bunch of rules that tell what you can conclude in different situations. Rule-based systems have been employed in a wide variety of contexts, such as the discovery of molecular structures [LBFL80], the identification of bacterias which cause severe infections [BS84] or to configure computer systems [McD82].

The structure of a rule-based expert system, see [GDR05], consists of, and the HES is no exception, the following components (see Figure 2.13):

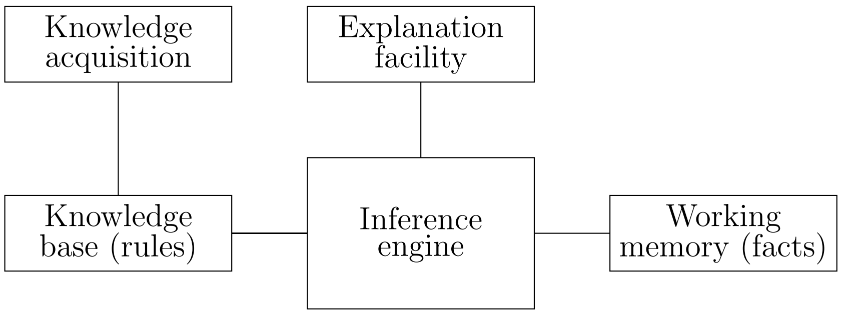


Figure 2.13: Structure of a rule-based expert system

- the *Working memory (the facts)*, representing what we know at any time about the problem we are working at,
- the *Knowledge base (the rules)*, containing the domain specific problem-solving knowledge,
- the *Inference engine*, a general program that activates the knowledge in the knowledge base. This program depending on the facts applies different rules to obtain a conclusion.
- a *Knowledge acquisition* module, allowing one to acquire and edit the knowledge base,
- an *Explanation facility* module, allowing the user to understand how the expert system obtains the results.

Let us present each component for the particular case of the HES.

The working memory represents what we know at any time about the problem we are working at by means of facts. There exist two kinds of facts in the HES: *static* and *dynamic*.

Static facts are properties associated with the spaces. These properties are known at the moment of the construction of the object. Some examples of this kind of facts are:

Fact 1.
$$\forall n \in \mathbb{N}: \quad \Delta^n \text{ is a contractible space.}$$

Fact 2. If $X = A \times B$ where A and B are contractible spaces, then,
$$X \text{ is a contractible space.}$$

Fact 3.
$$\forall n \in \mathbb{N} \text{ and } G \text{ group}: \quad K(G,n) \text{ is an Eilenberg MacLane space of type } (G,n)$$

Fact 4. If $X = B(Y)$ where Y is an Eilenberg MacLane space of type (G,n), then,
$$X \text{ is an Eilenberg MacLane space of type } (G,n+1)$$

At this moment, the static facts determine the spaces that are *contractible spaces*, *spheres*, *Eilenberg MacLane spaces* and *Loop spaces of spheres.* The way of implementing this issue in the microkernel is by means of four subclasses of the `mk-space-kenzo` class, presented in Subsubsection 2.2.3.1; this class hierarchy will be used to determine which rules can be applied.

To store spaces which represent spheres, we use a new subclass called `MK-SPACE-SPHERE`, whose definition is:

```
(DEFCLASS MK-SPACE-SPHERE (MK-SPACE-KENZO)
    (;; DIMension
     (dim :type fixnum :initarg :dim :reader dim))
```

this class has an additional slot, to the ones of the `MK-SPACE-KENZO`, called `dim`, which represents the dimension of the sphere.

To store contractible spaces, we use a new subclass called `MK-SPACE-CONTRACTIBLE`, whose definition is:

```
(DEFCLASS MK-SPACE-CONTRACTIBLE (MK-SPACE-KENZO) ())
```

this class does not have any additional slot to the ones of the `MK-SPACE-KENZO` class.

To store spaces which represents Eilenberg MacLane spaces, we use a new subclass called `MK-SPACE-K-G`, whose definition is:

```
(DEFCLASS MK-SPACE-K-G (MK-SPACE-KENZO)
    (;; ITERation
     (iter :type fixnum :initarg :iter :reader iter))
    (;; GROUP
     (group :type fixnum :initarg :group :reader group))
```

this class has two additional slots to the ones of `MK-SPACE-KENZO`: (1) `iter`, which represents the number of iterations of the Eilenberg MacLane space, and (2) `group`, which represents the group of the Eilenberg MacLane space.

To store spaces which represent loop spaces of spheres, we use a new subclass called `MK-SPACE-LS-SPHERE`, whose definition is:

```
(DEFCLASS MK-SPACE-LS-SPHERE (MK-SPACE-KENZO)
    (;; ITERation
     (iter :type fixnum :initarg :iter :reader iter))
    (;; DIMension Sphere
     (dims :type fixnum :initarg :dims :reader dims))
```

this class has two additional slots to the ones of `MK-SPACE-KENZO`: (1) `iter`, which represents the number of iterations of the Loop space, and (2) `dims`, which represents the dimension of the sphere. When, the `Loop Space` module constructs a `MK-SPACE-LS-SPHERE` instance the value of the `iter` slot is the number of iterations of the Loop Space and the value of `dims` is the value of the `dim` slot of the `MK-SPACE-SPHERE` object component of the Loop Space.

In the future, if we include new static facts which determine other types of spaces, we only need to define a new subclass of the `mk-space-kenzo` class for that concrete kind of spaces.

As we have said, the static facts are properties that are known at the moment of the construction of the objects. On the contrary, *dynamic* facts are properties that are obtained from computations. For instance:

Fact 5.

$$\text{For } n = 4: \quad \pi_n(S^3) \text{ has been computed previously and its value is } \mathbb{Z}/2\mathbb{Z}.$$

Fact 6.

$$\text{For } 0 \leq n < 4: \quad H_n(S^4) \text{ is null.}$$

The storage of dynamic facts was already explained in this memoir, namely in Subsubsection 2.2.3.1 where we talked at length about the storage of results in the internal memory of the microkernel. Then, when a new result is obtained, it is automatically saved in the internal memory to avoid re-computations. The HES has access to this internal memory, and therefore to dynamic facts.

Let us present now, the *knowledge base* of our HES. The rules in the HES represent possible results to take when specified conditions hold on items of the working memory. They are sometimes called condition-action rules, since they are `IF-THEN` rules.

The current knowledge base consists of 23 rules (the complete list can be seen in [Her11]) which are related to *contractible spaces*, *spheres*, *Eilenberg MacLane spaces* and *Loop spaces of spheres*. Let us present here some of them:

Rule 1.

$$\begin{array}{ll} \textbf{IF} & X \text{ is a contractible space} \\ \textbf{AND} & n \geq 1 \\ \textbf{THEN} & \pi_n(X) = 0 \end{array}$$

Rule 2.

$$\begin{array}{ll} \textbf{IF} & X \text{ is an Eilenberg MacLane space of type } (G, n) \\ \textbf{AND} & n = r \\ \textbf{THEN} & \pi_n(X) = G \end{array}$$

Rule 3.

$$\begin{array}{ll} \textbf{IF} & X \text{ is the sphere } S^2 \\ \textbf{AND} & \pi_n(S^3) \text{ has been computed previously} \\ \textbf{THEN} & \pi_n(X) = \pi_n(S^3) \end{array}$$

Both Rules 1 and 2 are applied over static facts; however, the application of Rule 3 involves a dynamic fact, namely the knowledge of a previously computed homotopy group. Therefore, some rules can only be applied when some computations have been previously performed.

These rules are used by the *inference engine* of the HES which uses the *forward chaining* method for reasoning [GR05]. This method starts with the available data and uses inference rules to extract more data until a goal is reached. Let us consider an example. We want, for instance, to compute $\pi_3(\Delta^4 \times \Delta^5)$. Then the inference engine proceeds as follows:

1	Δ^4 is a contractible space,	Fact 1
2	Δ^5 is a contractible space,	Fact 1
3	$\Delta^4 \times \Delta^5$ is a contractible space,	Fact 2 (1,2)
4	$3 = n \geq 1$	
$\therefore$	$\pi_3(\Delta^4 \times \Delta^5) = 0$	Rule 1 (3,4)

In the case of having conflicts between some rules, the method which has been used to solve this problem consists of using the rule with highest priority based on our mathematical knowledge. The priority is established by placing the rules in an appropriate order in the knowledge base. This strategy works properly for small expert systems, see [BS84], as our HES. This conflict resolution strategy has worked in our system; however, we are aware of the necessity of a more sophisticated strategy to deal with the conflicts in the HES when the number of rules increases.

The inference engine has been implemented by means of a very powerful tool of Common Lisp: the combination of generic functions and methods [Gra96]. When the class system and the functional organization of Common Lisp are considered, the notions of *generic functions* and *methods* are normally used. A *generic function* is a functional object whose behavior will depend on the class of its arguments; a generic function is defined by a `defgeneric` statement. The code for a generic function corresponding to a particular class of its arguments is a *method* object; each method is defined by a `defmethod` statement. We have a generic function:

```
(DEFGENERIC homotopy-HES (object n))
```

which has assigned concrete methods for the different spaces with special characteristics. Namely, we have four methods for the generic `homotopy` function, one for each one of the subclasses of the `mk-space-kenzo` class, extending their functionality:

```
(DEFMETHOD homotopy-HES  ((contractible mk-space-contractible) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((sphere mk-space-sphere) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((k-g mk-space-k-g) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((ls-s mk-space-ls-sphere) n) ...)
```

Each one of these methods uses the concrete rules of the HES for each one of the spaces with special characteristics to obtain their homotopy groups. As we have said previously, other subclasses can be defined in the future to represent other kinds of spaces; then, if we add rules for that kinds of spaces, we only need to define new methods (one per each new kind of space) to manage the computation of homotopy groups for them.

The body of those methods is a *conditional* statement using the Common Lisp `cond` instruction. Each one of the `cond` options represents a rule of the HES. For instance, the code of the method related to the `mk-space-k-g` objects is as follows:

```
(defmethod homotopy ((k-g mk-space-k-g) n)
  (cond ((equal (iter k-g) n)
         (explanation-facility-module k-g n
                    (format nil "<component>~A</component>" (group k-g))))
        ((not (equal (iter k-g) n))
         (explanation-facility-module k-g n "<component>0</component>"))))
```

Each one of the clauses of the conditional statement represents one rule of the HES. Then, if we want to add new rules to the knowledge base, we only need to include more clauses at the end of the conditional statement. This is the knowledge acquisition mechanism of our HES. We are aware of the necessity of including a different *knowledge acquisition mechanism* in order to provide a way of adding new rules at runtime and without modifying the source code. But, at this moment, this elementary organization has been shown sufficient.

Last but not least, we have an *explanation facility module* that is a mechanism to explain the reasoning which the expert system has followed in order to get a conclusion. This mechanism is provided to the user and is implemented as a generic function.

```
(DEFGENERIC explanation-facility-module (space n result))
```

This generic function takes as argument an object X, a natural number n, and the value of the $\pi_n(X)$ group. For each one of the subclasses of the `mk-space-kenzo` class we have a concrete method which is able to explain the followed reasoning by the HES to obtain the result. With that information, the `explanation-facility-module` function generates a `HES-result` XML-Kenzo object which is the returned result by the HES.

Gathering the functionality of the HES and the HAM, our framework is able to compute some homotopy groups. In addition, we can also combine the HES and the HAM to obtain other homotopy groups as we are going to show in the following paragraph.

2.2.3.2.4 Integration of the HES and the HAM It is worth noting that if we disable whether the HES or the HAM in the microkernel, our framework can keep on computing homotopy groups with the other one. Then, they can be considered as independent modules. However, if we enable both modules, the HAM and the HES cooperate to compute homotopy groups of spaces.

On the one hand, the HAM is used as a last appeal tool in the HES. That is to say, if none of the rules of the HES can be applied to obtain a homotopy group, the HES invokes the homotopy algorithm module to check if it is able to obtain some result applying the algorithm.

On the other hand, the homotopy algorithm module is used to obtain some results that are used as dynamic facts of the objects. For instance, if we want to compute $\pi_i(S^3)$ for $i = 4, 5, 6$ the HES does not have any rules which can help. Then, the HES invokes the HAM to perform the computations and stores the results in the internal memory of the microkernel. Subsequently, if we want to compute $\pi_i(S^2)$ for $i = 4, 5, 6$, the HES can use Rule 3 and the previously computed homotopy groups of the sphere S^3 to obtain the homotopy groups of the sphere S^2; in this way the HES and the HAM interact. Of course, the microkernel could have also used the HAM to compute $\pi_i(S^2)$ for $i = 4, 5, 6$; but the option of using the HES system is better because of the complexity of the algorithm implemented in HAM.

2.2.3.3 Construction Modules

Construction modules provide the functionality to construct topological spaces in the microkernel. In particular, they implement the procedures to construct the spaces associated with the elements that are children of the `constructor` element of the XML-Kenzo specification. Moreover, they are in charge of checking the Kenzo restrictions that were not included in the XML-Kenzo specification for the constructors of spaces. That is to say, the functional dependent restrictions over the arguments.

It is worth noting that there are two kinds of functional dependent restrictions over the arguments. On the one hand, if there are at least two arguments and the value of one of them depends on the value of another one, then, we have a functional dependent restriction (for instance, in the “Moore” constructor the value of its argument n must be higher or equal than two times plus four its argument p). On the other hand, if there is just one argument, but with the singularity of being compound, and the value of one of the elements of this compound argument depends on the value of another one, then, we also have a functional dependent restriction (for instance, in the “finite-ss” constructor the value of its argument is a list of elements which must determine a simplicial set;

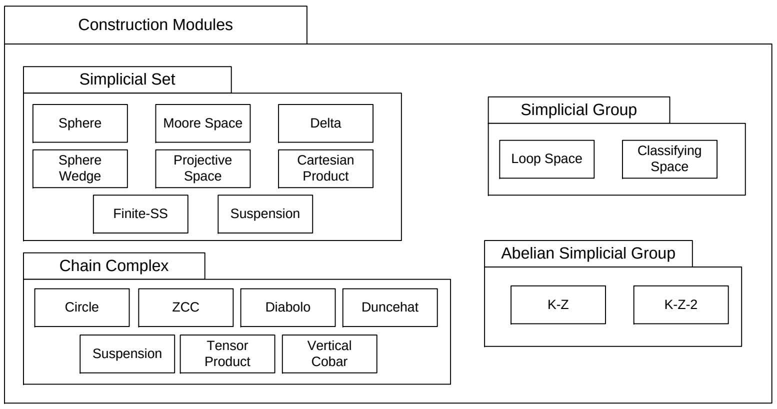


Figure 2.14: Construction modules

then, this compound argument is restricted to the values of its components). Both kinds of functional dependent restrictions are dealt with in the microkernel, namely in the construction modules.

Following the XML-Kenzo specification of the `constructor` element, we have organized construction modules in four blocks, one block per each type of constructor specified in XML-Kenzo (`CC`, `SS`, `SG` and `ASG`). Besides, each one of these blocks has a module for each one of the elements of the respective type of constructor, see Figure 2.14. All these modules are implemented in Common Lisp.

When the microkernel receives a construction request the functionality of the module associated with the child of the `constructor` element is invoked. For instance, if the microkernel receives the request:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

the functionality of the `Loop Space` module is invoked.

When a construction module is invoked three situations are feasible: (1) a new space is created in the microkernel, (2) the object was previously built and is simply returned or (3) a warning is produced.

The procedure to construct the spaces in the construction modules is very similar in all of them.

1. Check the functional dependent restrictions (this step depends on each particular constructor):

   (a) If the object does not fulfill the restrictions a `warning` XML-Kenzo object is returned.
   (b) Otherwise, go to step 2.

2. Search in the `*object-list*` list if the object was built previously:
   (a) If the object was built previously, its identification number, in an `id` XML-Kenzo object, is returned.
   (b) Otherwise, go to step 3.

3. Construct an instance of the `mk-space-kenzo` class where:
   - the value of the `rede` slot is computed by the reduction degree module; see paragraph 2.2.3.2.1,
   - `idnm` is automatically generated,
   - `kidnm` is the value obtained from requesting the internal server the construction of the space, and
   - the XML-Kenzo object received (from the external server) as input is assigned to `orgn`.

4. Push the object in the `*object-list*` list of already created spaces.

5. Return the `idnm` of the object in an `id` XML-Kenzo object.

The above procedure is followed by most of the construction modules of the micro-kernel. However some of them do not construct instances of the `mk-space-kenzo` class but of some of its subclasses (see Paragraph 2.2.3.2.3) in order to use the additional information provided by the subclasses in the computation of homotopy groups (see Paragraph 2.2.3.2.3). However, the values of the slots which come from the `mk-space-kenzo` class of those instances are the same explained in the above procedure. To be more concrete, these construction modules are `Sphere`, `Delta`, `K-Z`, `K-Z-2`, `Cartesian Product`, `Suspension`, `Classifying Space` and `Loop Space`. The procedure implemented in these modules is different since they are the modules which can produce contractible spaces, spheres, Eilenberg MacLane spaces and loop spaces of spheres; that is to say, the spaces whose homotopy groups can be handled in the HES.

The `Sphere` module constructs instances of the `MK-SPACE-SPHERE` subclass, where the value of the `dim` slot is the dimension of the sphere (this dimension is obtained from the XML-Kenzo object).

The `Delta` module constructs instances of the `MK-SPACE-CONTRACTIBLE` subclass.

Both `K-Z` and `K-Z2` modules construct instances of the `MK-SPACE-K-G` subclass where the value of the `iter` slot is the number of iterations of the Eilenberg MacLane space (this number is obtained from the XML-Kenzo object) and the value of the `group` slot is 1 if is `K-Z` the working module and 2 if is `K-Z2`.

The `Cartesian Product` module constructs a `MK-SPACE-CONTRACTIBLE` instance if both components of the cartesian product are `MK-SPACE-CONTRACTIBLE` objects; otherwise, it constructs a `MK-SPACE-KENZO` instance with the general procedure.

The `Suspension` module constructs a `MK-SPACE-SPHERE` instance when the component of the suspension is a sphere, S^n, the value of the `dim` slot of the `MK-SPACE-SPHERE` instance constructed by the `Suspension` module is 1 plus n; otherwise, the `Suspension` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

The `Classifying Space` module constructs a `MK-SPACE-K-G` instance when the component of the Classifying Space is an Eilenberg MacLane space $K(G, n)$, the value of the `iter` slot of the `MK-SPACE-K-G` object constructed by the `Classifying Space` module is 1 plus n and the value of the `group` slot is the representation of G (1 in the case of $K(\mathbb{Z}, n)$ and 2 in the case of $K(\mathbb{Z}/2\mathbb{Z}, n)$); otherwise, if the component of the Classifying space is not an Eilenberg MacLane space, then the `Classifying Space` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

Finally, the `Loop Space` module constructs a `MK-SPACE-K-G` instance when the component of the Loop Space is an Eilenberg MacLane space, $K(G, n)$, the value of the `iter` slot of the `MK-SPACE-K-G` object constructed by the `Loop Space` module is n minus the number of iterations of the loop space and the value of the `group` slot in the `MK-SPACE-K-G` object constructed by the `Loop Space` module is is the representation of G. When the component of the Loop Space is a sphere S^n, the `Loop Space` module constructs a `MK-SPACE-LS-SPHERE` instance, where the value of the `iter` slot is the number of iterations of the Loop Space and the value of `dims` is n. Otherwise, the `Loop Space` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

2.2.3.4 Computation Modules

Computation modules provide the functionality to perform computations through the microkernel. As we have seen in the XML-Kenzo specification, the `operation` element only has a child, either the `homology` or the `homotopy` element. Hence, the system only allows the computation of homology and homotopy groups. This situation is reflected in the computation modules, two Common Lisp modules that compute homology and homotopy groups respectively.

2.2.3.4.1 Homology module The *homology module* implements a procedure in Common Lisp that allows us to compute homology groups through the microkernel. The procedure not only calls the homology function of the Kenzo system but also wraps it to enhance the computation of homology groups in the framework, avoiding computations that would raise errors and optimizing them.

When the microkernel receives a computation request where the child of the `operation` element is `homology` the homology module is invoked. For instance, the request

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  <homology>
</operation>
```

activates the homology module. The procedure implemented in the homology module is as follows:

1. Extract the space whose homology wants to be computed.
2. Search in the `*object-list*` list if the space was built previously:
   - If the space was built previously, go to step 3.
   - Otherwise, the correspondent construction module builds the space and stores it in the `*object-list*` list.
     - If the construction module returns a warning then return the warning as result in a `warning` XML-Kenzo object.
     - Otherwise, go to step 4.
3. Search in the internal memory if the computation was performed previously; see Subsubsection 2.2.3.1.
   - If the computation was stored then the result by means of a `result` XML-Kenzo object is returned.
   - Otherwise, go to step 4.
4. Monitor the value of the reduction degree of the space, that is stored in the `rede` slot.
   - If the reduction degree is higher or equal than 0, go to step 5.
   - Otherwise, inform with a `warning` XML-Kenzo object that the system cannot compute the homology group since the reduction degree of the space is lower than 0.
5. Ask to the internal server the homology group of the space.
6. Store the result obtained from the internal server in the internal memory; see Subsubsection 2.2.3.1.
7. Return the result by means of a `result` XML-Kenzo object.

The result returned by the homology module when receives the above request is

```
<result>
    <component>3</component>
    <component>2</component>
    <component>1</component>
</result>
```

that must be read as $\mathbb{Z}/3\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}$ that is the fifth homology group of $\Omega^3 S^4$.

2.2.3.4.2 Homotopy module The *homotopy module* implements a procedure in Common Lisp that allows us to compute some homotopy groups through the microkernel in two different ways depending on the space. The homotopy module uses the HES (see Paragraph 2.2.3.2.3) if we want to compute homotopy groups of spheres, contractible spaces, Eilenberg MacLane spaces or loop spaces of spheres; otherwise, the homotopy module uses the HAM (see Paragraph 2.2.3.2.2).

When the microkernel receives a computation request where the child of the `operation` element is `homotopy` the homotopy module is invoked. For instance, the request

```
<operation>
  <homotopy>
    <sphere>3</sphere>
    <dim>6</dim>
  </homotopy>
</operation>
```

activates the homotopy module. The procedure implemented in the homotopy module is as follows.

1. Extract the space whose homotopy wants to be computed.
2. Search in the `*object-list*` list if the space was built previously.
    - If the object was built previously, go to step 3.
    - Otherwise, the correspondent construction module builds the space and stores it in the `*object-list*` list.
        - If the construction module returns a warning then the warning is returned as result in a `warning` XML-Kenzo object.
        - Otherwise, go to step 4.
3. Search in the internal memory if the computation was performed previously, see Subsubsection 2.2.3.1.

- If the computation was stored then the result using a `result` XML-Kenzo object is returned.
- Otherwise, go to step 4.

4. Monitor the value of the reduction degree of the space.
    - If the reduction degree is higher or equal than 1, go to step 5.
    - Otherwise, inform with a `warning` XML-Kenzo object that the system cannot compute the homotopy group since the reduction degree of the space is lower than 1.
5. If the space whose homotopy wants to be computed is an instance of the classes: `mk-space-sphere`, `mk-space-contractible`, `mk-space-k-g` or `mk-space-ls-sphere`:
    - Invoke the HES; see Paragraph 2.2.3.2.3.
    - Otherwise, invoke the HAM; see Paragraph 2.2.3.2.2
6. Store the obtained result in the internal memory; see Subsubsection 2.2.3.1.
7. Return the result using a `result` XML-Kenzo object if the result was obtained by the HAM or a `HES-result` XML-Kenzo object if the result was obtained by the HES.

The result returned by the homotopy module when receives the above request is

```
<result>
    <component>12</component>
</result>
```

that must be read as $\mathbb{Z}/12\mathbb{Z}$ that is the sixth homotopy group of S^3.

It is worth noting that the last restriction handled in the framework, that is the reduction degree restriction, is dealt with in the computation modules; namely, in step 4 of the procedures.

To sum up, computation modules (with the help of processing modules) not only use the functionality of Kenzo to obtain homology and homotopy groups, but also execute some test that enhance the original computation capabilities of Kenzo.

2.2.4 External Server

The *external server* is a Common Lisp *component* providing access to all *services* available in the microkernel interface through an XML-Kenzo interface. The external server receives XML-Kenzo requests, analyzes them, invokes the microkernel and sends the results back to the caller.

The analysis performed in the external server consists of validating the XML-Kenzo requests against the specification given in the XML-Kenzo language. This means that restrictions imposed in the XML-Kenzo language (type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, and restriction of the dimension in computations) are checked here.

To validate these constraints an XML validator has been implemented. It takes as input an XML object, accesses to the XML-Kenzo schema definition and checks that the XML object is valid against the XML-Kenzo specification. It is worth noting that if we modify the XML-Kenzo schema definition, for instance adding a new kind of constructor, the external server evolves without modifying the source code.

As we have seen previously, other restrictions which were not included in the XML-Kenzo specification are checked at the microkernel level (namely, functional dependent restrictions of the arguments of the constructors and reduction degree restrictions). Besides, as we are working with XML objects as data, they must be *well-formed XML*, but the task of verifying this aspect is responsibility of the *adapter*.

2.2.5 Adapter

The *adapter* is a Common Lisp module that provides access to the *services* available in the external server through an *OpenMath* [Con04] interface.

As we explained in Subsection 2.2.1, the main reason to define the fresh XML-Kenzo language instead of using the standard OpenMath language was the general purpose of this standard which makes its adaptation to our context a bit hard. However, the need of providing a suitable interface, uncoupled of the internal representation used in the framework, for different clients turns the tables in the most external part of the framework where we use OpenMath instead of using XML-Kenzo.

The OpenMath standard defines several mathematical objects of general purpose, such as linear algebra operators or arithmetic functions. However, specific objects, such as the mathematical structures used in Algebraic Topology, have not been declared; but, they can be defined following the guidelines given in [Dav00].

In particular, we have extended the OpenMath standard defining the objects specified in XML-Kenzo. To declare new objects in OpenMath we defined *Content Dictionaries*. A Content Dictionary is the declaration of a collection of objects, their names, descriptions, and rules. Namely, we have defined five Content Dictionaries, one for each one of the groups defined in XML-Kenzo (`CC`, `SS`, `SG`, `ASG` and `Computing`). Each one of these Content Dictionaries contains the declaration of the elements that belong to that group in XML-Kenzo; see Figures 2.5 and 2.10.

The definition of an object in a Content Dictionary includes its name, its description, examples of the object and sometimes its formal properties specified using OpenMath (in which case the equivalent commented property using natural language should also be given). For instance, the definition of the sphere element in OpenMath starts as follows:

```
<CDDefinition>
```

then the name of the object

```
<Name>sphere</Name>
```

the description

```
<Description>
  This symbol is a function with one argument, which should be a natural number
  n between 1 and 14. When applied to n it represents the sphere of dimension n.
</Description>
```

an example (namely S^3)

```
<Example>
  <OMOBJ>
    <OMA>
      <OMS cd="SS" name="sphere"/>
      <OMI>3</OMI>
    </OMA>
  </OMOBJ>
</Example>
```

the commented property in natural language

```
<CMP>The dimension of the sphere must be a natural number between 1 and 14 </CMP>
```

the formal property using the OpenMath standard

```
<FMP>
  <OMA>
    <OMS cd="logic1" name="and"/>
    <OMA>
      <OMS cd="set1" name="in"/> <OMV name="n"/> <OMS name="N" cd="setname1"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMI>1</OMI> <OMV name="n"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMV name="n"/> <OMI>14</OMI>
    </OMA>
  </OMA>
</FMP>
```

are given and finally the definition is closed.

```
</CDDefinition>
```

In this way, all the objects represented in the XML-Kenzo language are specified in OpenMath. Moreover, another Content Dictionary called `Aux` to specify the objects belonging to the `results` XML-Kenzo group, such as the elements to return warnings or results, has been defined. The complete definition of the Content Dictionaries can be seen in [Her11].

At first glance, formal properties could seem a suitable way to represent the restrictions imposed in the XML-Kenzo language and this is true for some of them. However some restrictions (namely, type restrictions) can only be given here by means of the commented properties that are just properties in natural language, and this is not enough to our purpose.

Therefore, although OpenMath is not strong enough to represent all the mathematical knowledge that we need in our framework, we can use it in the most external part of our framework to represent our mathematical objects and exchange them with other systems, a task that suits perfectly to OpenMath.

Once we have defined the OpenMath objects, we need a software in charge of converting from XML-Kenzo representation to OpenMath representation and viceversa, this component is called *Phrasebook* and is programmed as a Common Lisp module of the adapter.

To sum up, the adapter receives OpenMath requests, checks that the OpenMath request is *well-formed XML* by means of a XML package of Common Lisp [Inc05], converts the OpenMath request into a well-formed XML-Kenzo request, invokes the external server with the XML-Kenzo request and sends the result received from the external server back to the caller using the OpenMath format.

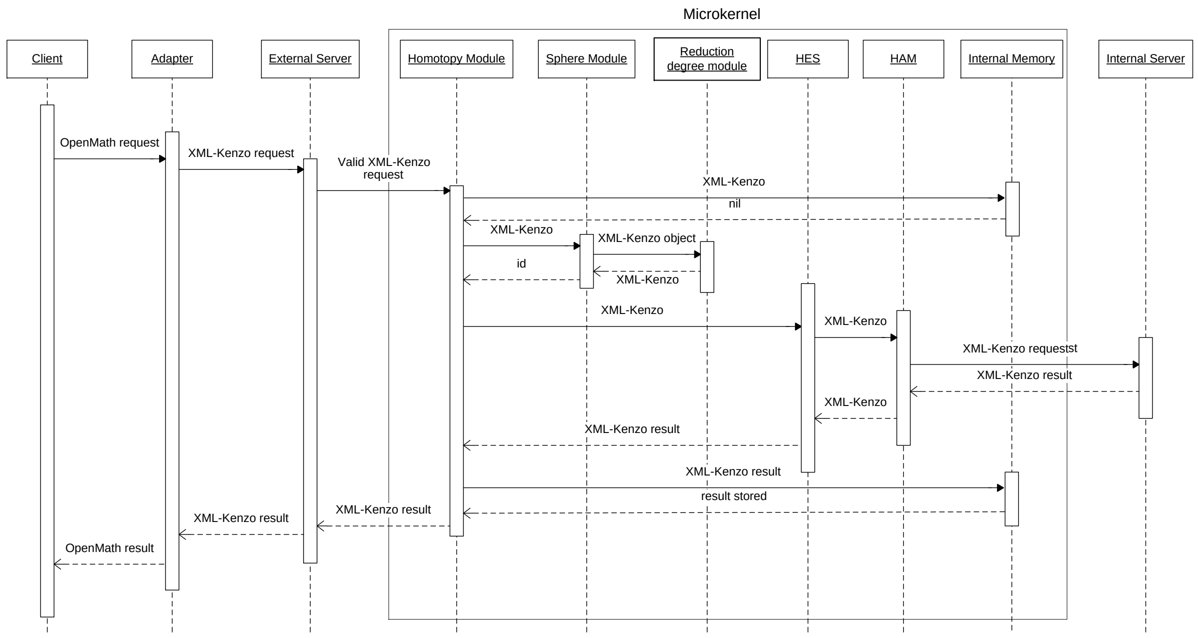


Figure 2.15: UML sequence diagram

2.3 Execution Flow

Once we have presented all the components of our framework, let us illustrate the execution flow of the system with a detailed scenario: the computation of the sixth homotopy group of the sphere of dimension 3, $\pi_6(S^3)$, in a fresh session of the Kenzo framework; that is, neither objects were constructed or computations were performed in the Kenzo framework previously. The execution flow of this scenario is depicted in Figure 2.15 with a UML-like sequence diagram.

The OpenMath representation of $\pi_6(S^3)$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
            <OMS cd="SS" name="sphere"/>
            <OMI>3</OMI>
        </OMA>
        <OMI>6</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath data from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
    <homotopy>
        <sphere>3</sphere>
        <dim>6</dim>
    </homotopy>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification, in this case as the root element is `operation` it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homotopy` element has two children. ✓
3. The first child of the `homotopy` element belongs to one of the groups `CC`, `SS`, `SG` or `ASG`. ✓
4. The value of the `sphere` element is a natural number higher than 0 and lower than 15. ✓
5. The second child of the `homotopy` element is the `dim` element. ✓
6. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, a valid request is sent to the microkernel. In the microkernel the `homotopy` module is invoked. When the `homotopy` module is invoked, the procedure explained in Paragraph 2.2.3.4.2 is executed. First, the `homotopy` module search in `*object-list*` list of the internal memory if the sphere of dimension 3 was constructed previously, as this space was not constructed, the `sphere` module is invoked to construct it, this module in turn calls the reduction degree module to obtain the reduction degree of the sphere. Subsequently, once constructed the `mk-space-sphere`, as the reduction degree of the space S^3 is two, the `homotopy` module can call the HES. The HES tries to apply its rules; however, as all the rules which can be applied to `mk-space-sphere` instances depend on dynamic facts, it cannot apply any rule (since we are working with a fresh session, then dynamic facts have not been computed yet). Therefore, the HES invokes the HAM which performs some intermediary computations and, in this case as the intermediary tests succeed, calls the internal server to compute $\pi_6(S^3)$. The result returned by the internal server is:

```
<result>
    <component>12</component>
</result>
```

This result is stored in the internal memory, to avoid re-computations or to use it as dynamic fact, by the `homotopy` module and sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>12</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

2.4 Integration of local and remote Kenzo computations

As we have said several times throughout this memoir, some Kenzo computations can be very time and space consuming (requiring, typically several days of CPU time on powerful dedicated computers). Then, we realized that we could use a dedicated server to perform those heavy computations instead of overloading the Kenzo framework user's computer. This section is an ongoing work, so just some ideas are provided in order to give a flavor of the way of integrating local and remote Kenzo computations.

As a first step to integrate local and remote Kenzo computations, we can consider the framework depicted in Figure 2.16 where the computations are not longer performed locally but in a remote server.

There are two new components in this framework. A powerful and external *Kenzo server* and a *Kenzo server client*.

The external Kenzo server is a dedicated sever which receives XML-Kenzo computation requests and returns XML-Kenzo results. The Kenzo server client is a program which is able to invoke the services which the Kenzo sever offers and receive the results returned by the Kenzo server. It is worth noting that the Kenzo server client can be seen as a new internal server of the Kenzo framework.

In this new organization we are keeping in our framework the internal sever since both construction and processing modules use it. However, the computations are not longer performed in the internal server but in the Kenzo server; then, the computation modules are not connected with the internal server but with the Kenzo server through the Kenzo server client.

Therefore, we have two different ways of performing computations of groups of spaces in the Kenzo framework. On the one hand, the original way of working, where all the computations are locally performed by means of the Kenzo internal server. On the

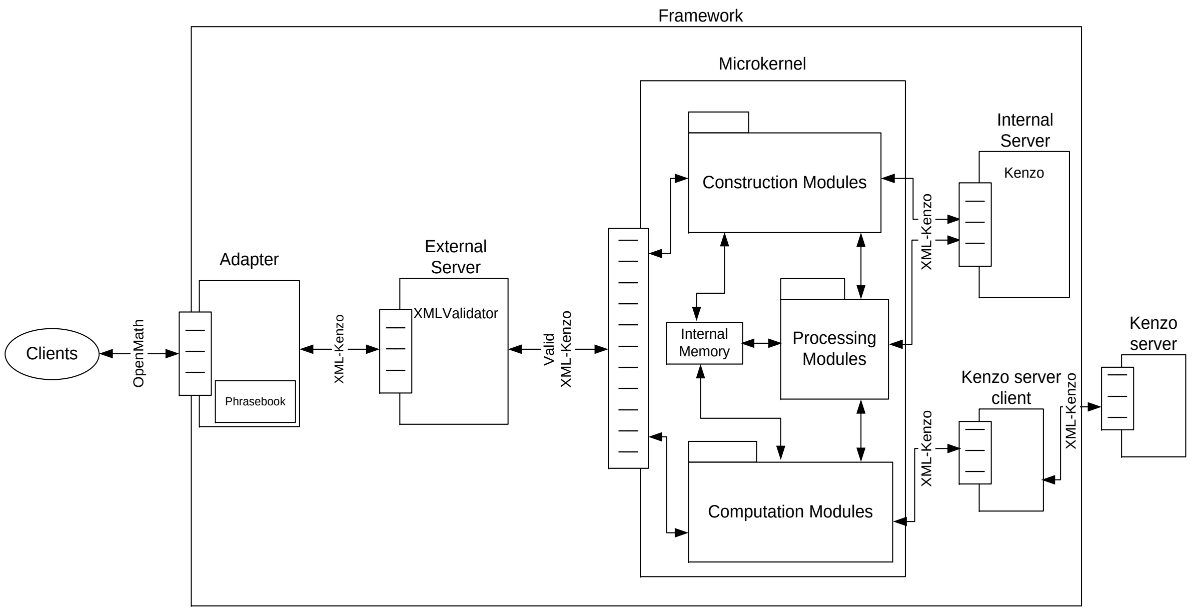


Figure 2.16: Kenzo framework with remote Kenzo computations

other hand, the new approach presented in this section where all the computations are performed in an external Kenzo server.

The second mode has obvious drawbacks related to the reliability of Internet connections, to the overhead of management where several concurrent users are allowed, and so on. But the original way of working is not fully satisfactory since interesting Kenzo computations used to be very time and space consuming. Thus a mixed strategy should be convenient: the computation modules of the microkernel of the Kenzo framework should decide if a concrete calculation can be done in the local computer or it deserves to be sent to the Kenzo server. In this case, as it is not sensible to maintain open an Internet connection for several days waiting for the end of a computation, a subscription mechanisms, allowing the Kenzo framework to disconnect and to be subscribed to the process of computation in the remote server, should be used.

The difficulties of how to mix local and remote computations have two sources: (1) the knowledge here is not based on well-known theorems since it is context-dependent (for instance, it depends on the computational power of a local computer), and so it should be based on heuristics; and (2) the technical problems to obtain an optimal performance are complicated, due, in particular, to the necessity of maintaining a shared state between two different computers.

With respect to the kind of heuristic knowledge to be managed into the intermediary level, there is some part of it that could be considered obvious: for instance, to ask for a homology group $H_n(X)$ where the degree n is big, should be considered harder than if n is small, and then one could wonder about a limit for n before sending the computation to the remote server. Nevertheless, this simplistic view is to be moderated by some expert knowledge: it is the case that in some kinds of spaces, difficulties decrease when the degree increases. The heuristics should consider each operation individually. For instance, it is true that in the computation of homology groups of iterated loop spaces, difficulties increase with the degree of iteration. Another measure of complexity is related to the number of times a computation needs to call the Eilenberg-Zilber algorithm (see [DRSS98]), where a double exponential complexity bound is reached. Further research is needed to exploit the expert knowledge in the area suitably, in order to devise a systematic heuristic approach to this problem.

In order to understand the nature of the problem of sharing the Kenzo state between two computers it is necessary to consider that there are two kinds of state in our context. Starting from the most simple one, the state of a session can be described by means of the spaces that have been constructed so far. Then, to encode (and recover) such a state, a session file containing a sequence of constructors is enough.

Moreover, there exists another kind of state. A space in Kenzo consists of a number of methods describing its behavior (explaining, for instance, how to compute the faces of its elements), it was already explained in Subsection 1.2.1. Due to the high complexity of the algorithms involved in Kenzo, a strategy of memoization has been implemented (see Section 1.2.5). As a consequence, the state of a space evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a homology group, depends on the concrete states of every space involved in the calculation (in the more explicit case, to re-calculate a homology group on a space could be negligible in time, even if in the first occasion this was very time consuming). This notion of state of a space is transmitted to the notion of state of a session. We could speak of two states of a session: the one sallow evoked before, that is essentially static and can be recovered by simply re-constructing the spaces; and the other deep state which is dynamic and depends on the computations performed on the spaces.

To analyze the consequences of this Kenzo organization, we should play with some scenarios. Imagine during a local session a very time consuming calculation appears; then we could simply send the sallow state of an object to the remote server, because even if some intermediary calculations have been stored in local memory, they can be re-computed in the remote server (finally, if they are cheap enough to be computed on the local computer, the price of re-computing them in the powerful remote server would be low). Once the calculation is remotely finished, there is no possibility of sending back the deep state of the remote session to the local computer because, usually, the memory used will exhaust the space in the local computer. Thus, it could seem that to transmit the sallow state would be enough. But, in this picture, we are losing the very reason why Kenzo uses the memoization (dynamic programming) style. Indeed, if after obtaining a difficult result (by means of the remote server) we resume the local session and ask for another related difficult calculation, then the remote server will initialize a new session from scratch, being obligated to re-calculate every previous difficult result, perhaps making the continuation of the session impossible. Therefore, in order to take advantages of all the possibilities Kenzo is offering now on powerful scientific servers, we are faced with some kind of state sharing among different computers (the local computers and the server), a problem known as difficult in the field of distributed object-oriented programming.

The same problem above presented appears in the organization of the Kenzo server. At this moment, we are working with the *Broker* architectural pattern, see [B$^+$96, B$^+$07], in order to find a natural organization of the Kenzo server. In a broker based server, there would be several Kenzo kernels and a component called *broker*. The broker component will be responsible for the distribution of requests arriving to the Kenzo server across the different Kenzo kernels, and the returning of results. One of the main advantages of using the *Broker* pattern is the fact that Kenzo kernels not only can work individually but also can collaborate to obtain results. However, to achieve the cooperation of different Kenzo kernels we find the problem of sharing the state among the different kernels presented in the previous paragraph.

Therefore, more work is necessary to find a plausible solution to integrate local and remote Kenzo computations.

2.5 Towards a distributed framework

In the previous section, an ongoing work devoted to increase the computational capabilities of the Kenzo framework through the distribution of computations across local and remote Kenzo kernels was presented. On the contrary, this section, also related to a work in progress, presents an alternative to the Kenzo framework which will take advantage of the collaborative work of different clients by means of a system located not in user's computer but in a shared server. The new framework will be called *server framework*.

The server framework has some common concerns with the Kenzo framework, since both frameworks provide a mediated access to the Kenzo system. There are, however, important differences. The Kenzo framework is installed in the local computer of its clients. On the contrary, the server framework is located in a remote server, and users of this system only need a light client able to communicate with the server framework; due to this fact, we have a collaborative work among different clients of the server framework, since a client can take advantage of the computations performed by other one. However, no reward comes without its corresponding price, and several problems, which do not exist in the Kenzo framework (for instance, the reliability of Internet connections or the management of concurrency problems), appear in the development of the server framework.

The most important challenges that we faced in the development of our server framework were:

1. *Functionality.* The system should provide access to the Kenzo capabilities for computing (homology and homotopy) groups.

2. *Interaction with different clients.* The server framework should be designed in such a way that it could support different kinds of clients (graphical user interfaces, web applications and so on).

3. *Reusability the Kenzo framework components.* As far as possible, we want to reuse the different components implemented in the Kenzo framework, in order to take advantage of the enhancements included in that framework.

4. *Error handling.* The server framework should forbid the user some manipulations raising errors. This question is easily handled due to the reuse of Kenzo framework components.

5. *Decoupling components.* In spite of partly reusing the modules implemented for the Kenzo framework, we are aware that the message passing communication style of the Kenzo framework is not suitable for a server architecture. Then, a kind of asynchronous communication among the components must be provided in order to decouple them.

6. *Storage of results.* The results must be stored in a persistent way to avoid recalculations.

7. *Multithreading.* The server framework should be designed in such a way that it could support the process of several requests at the same time.

8. *Management of concurrency problems.* The control of concurrency issues is one of the most important questions when a system can process several requests at the same time.

9. *Subscription mechanism.* It is not sensible to keep open an internet connection for several days waiting for the end of a computation; then, some reactive mechanism should be implemented, allowing the client to disconnect and to be subscribed in some way to the process of computation in the remote server.

The previous requirements led us, inspired by the work of [MlBR07], to choose both the *Linda model* [Gel85] and the *Shared Repository* architectural pattern [B$^+$96, B$^+$07] to reorganize the Kenzo framework components in a server framework. The communication between the server framework and its clients will be provided by means of *web services* [C$^+$07] based on OpenMath. In addition, a mail subscription mechanism will be implemented to avoid the problem of keeping open a connection during too much time.

The rest of this section is organized as follows. First of all, Subsection 2.5.1 is devoted to introduce the Linda model and the Shared Repository pattern. Our server framework based on the Linda model and the Shared Repository pattern is presented in Subsection 2.5.2.

2.5.1 The Linda model and the Shared Repository pattern

Our server architecture has been inspired by both the *Linda model* [Gel85] and the *Shared Repository* architectural pattern [B$^+$96, B$^+$07].

The *Linda model* is based on *generative communication*, a mechanism for asynchronous communication among processes based on a shared data structure. The asynchronous communication is performed by means of the insertion and extraction of data over the shared space.

The shared space is called *tuple space*, since it contains a set of *tuples* produced by the different processes. As soon as a tuple is inserted in the tuple space, it has an independent existence.

Tuples are represented by means of a list of items, the items are split by means of commas and closed between brackets. Tuples can contain both *actual* and *formal* items. An actual item contains a specific value, like 3 or *foo*. A formal item acts like a placeholder (a character "?" denotes a variable which has to be treated as formal).

Example 2.2. Some examples of tuples are as follows:

$$\begin{array}{l} (Paris, Toronto, ?) \\ (Peter, 15) \\ (car1, 10, 15.35) \end{array}$$

In the Linda model, processes access the tuple space using five simple operations:

out: adds a tuple from the process to the tuple space.

in: deletes a tuple from the tuple space and returns it to the process. The process is blocked if the tuple is not available.

rd: returns a copy of one tuple of the tuple space. The process is blocked if the tuple is not available.

inp: a non blocking version of the *in* operation.

rdp: a non blocking version of the *rd* operation.

By means of the last four operations tuples can be retrieved from a tuple space. The arguments of these functions are tuple *templates*, possibly containing formal items (placeholders or wildcards).

The *Shared Repository* pattern is based on a very similar idea to the Linda model. It adds the nuance that a component has no knowledge of both what components have produced the data it uses, and what components will use its outputs.

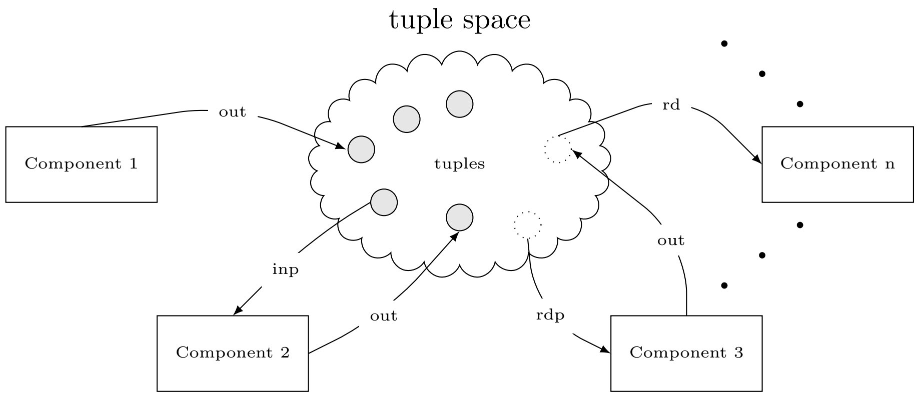


Figure 2.17: Feasible framework

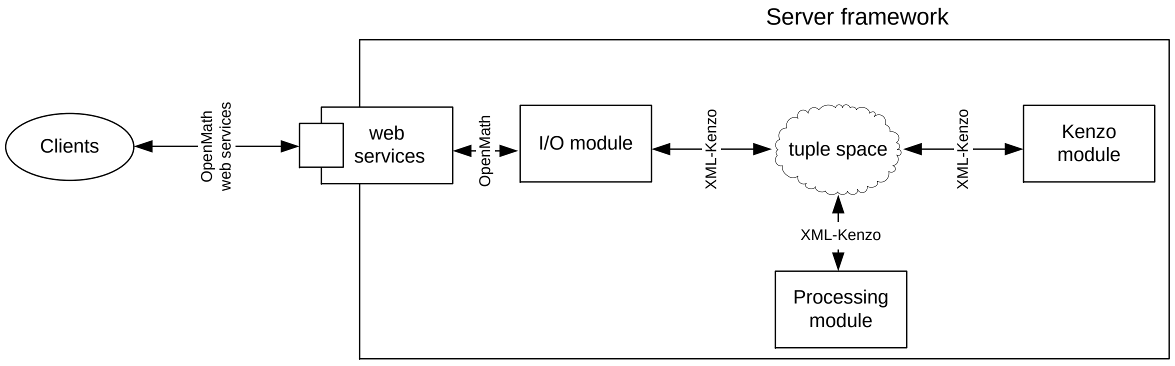


Figure 2.18: Schema of the server framework architecture

Figure 2.17 shows a feasible framework which brings together both Linda model and the Shared Repository pattern.

An architecture based on the combination of the Linda model and the Shared Repository pattern solves two of the challenges that were stated at the beginning of this section. On the one hand, the communication among the modules of our framework will be performed through the tuple space; in this way, the different components of our system will be decoupled. On the other hand, the question of storing results is easily solved using the Linda model due to the fact that all the computations can be stored as tuples in the tuple space avoiding re-computations.

2.5.2 A server framework based on the Linda model

A high level perspective of the server framework, based on both the Linda model and the Shared Repository pattern, as a whole is shown in Figure 2.18.

The rest of this subsection is devoted to present the components of this framework, as well as some important issues tackled in its development.

2.5.2.1 Components of the server framework

As we said at the beginning of this section, we decided to reuse, as much as possible, the components implemented in the Kenzo framework in our server framework. This decision was taken due to the fact that we did not want a powerful Kenzo server which can be used just by the Kenzo framework to perform long computations (as in the case presented in Section 2.4), but a server framework which can be employed for different clients taking advantage of the Kenzo enhancements implemented in the Kenzo framework and the collaborative work with other clients.

XML keeps on being the chosen technology to encode the data of our framework. Namely, XML-Kenzo is used for data interchange among the different components of the framework, and OpenMath is the language employed to communicate the server framework with the outside world.

The main innovation of the server framework with respect to the Kenzo framework is the tuple space of the Linda model. In order to use the Linda model in our context, a running implementation of it must be available. There are different Linda model implementations for Java [FHA99], C++ [Slu07], and even one for Common Lisp [Bra96]. Instead of using any of them, we have preferred to develop a fresh Allegro Common Lisp implementation adapted to our very concrete situation since the Common Lisp version presented in [Bra96] is out-of-date and the integration of either Java or C++ versions in our framework could be difficult.

We have developed our implementation of a tuple space holding the following properties: it must be shared (different modules can access it concurrently), persistent (once a tuple is stored in the tuple space, it stays in it until a module deletes it) and should comply with the *ACID* (Atomicity, Consistency, Isolation and Durability) rules.

Two possibilities were considered when designing our Linda model implementation. The first one consists of implementing the Linda model from scratch, without using any kind of external tool (for instance, the tuple space could be installed in computer memory and we could store the data as lists). This option would be very time consuming since it must solve well known problems in the domain of concurrent systems. Additionally, since our data are expressed as XML data, something like *XQuery* or *XPath* [E$^+$07] should be implemented in our case to search tuples inside our system, too.

These problems oriented us towards the second possibility: using already existing technology. Our previous remarks gave us the clue: XML databases [CRZ03] could be a good support in our case.

There are two kinds of XML databases: *native XML databases* and *XML enabled databases*. Native XML databases has XML documents as its fundamental unit of storage; on the contrary, XML enabled databases work with XML objects as data and use as internal model a tradicional (relational or object oriented) database; then, we chose XML enabled databases because we are going to work with XML-Kenzo objects. In XML enabled databases, each XML object is mapped to a datum of the internal

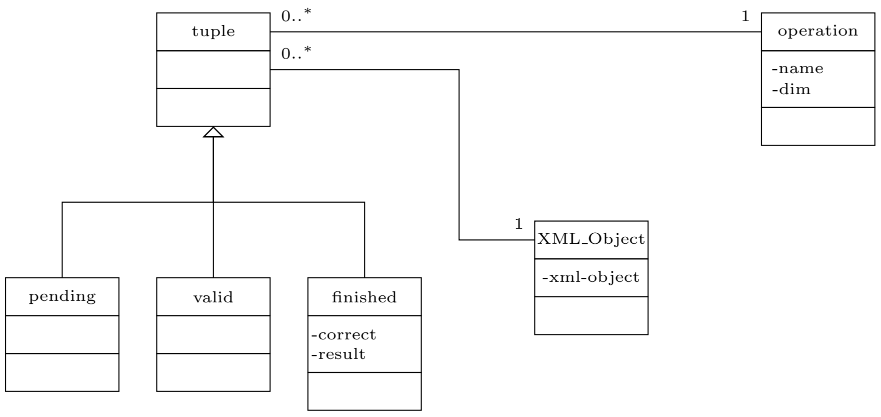


Figure 2.19: Class diagram

(relational or object oriented) model of the database giving access to all the features and the performance that can be found in the corresponding database manager. These databases include two kind of processes: map XML data to objects or tables, and convert database elements into XML data.

The chosen database was AllegroCache [Aas05]. AllegroCache is an object oriented database of Allegro Common Lisp. One of the main advantages of working with AllegroCache is that the data are always stored persistently but one can work directly with objects as if they were in standard memory. Besides it supports a full transactional model with long and short transactions, meets the classic ACID requirements for a database and maintains referencial integrity of complex data objects. Persistent classes located by AllegroCache are just usual CLOS classes, where the metaclass `persistent-class` is declared.

In order to complete our Linda model implementation, we still have to deal with both tuples and the operations to interact with the tuple space. Since we have decided that tuples rely on XML-Kenzo data but the repository (to be understood as a tuple space) is based on AllegroCache, it is necessary to convert from XML-Kenzo objects to instances of persistent classes stored in the database and viceversa. To this aim, we have devised a class system represented with UML notation in Figure 2.19.

As we have said previously, the server framework is devoted to perform computations; then, the requests which arrive to this framework are computation requests. Each computation request is made up of both a topological space and an operation over it, so two classes, `XML-Object` (with just one slot whose value is the XML-Kenzo representation of the space) and `operation` (with two slots, `name` and `dim`, which provide respectively the name of the operation, homology or homotopy, and the dimension) have been defined to model the two components of a computation request. The `tuple` class binds these two components of a request.

Moreover, in our server framework a tuple can have three different states: (1) the tuple is pending of being validated, (2) the tuple has been validated, and (3) the system has obtained the result of the computation request associated with the tuple, and, then, the tuple has finished its work. Then, a tuple is implemented as an instance of the `pending` subclass in the first case, as a `valid` instance in the second one; and as a `finished` instance in the last one. It is worth noting that instances of the `finished` class have two additional slots: (1) the `correct` slot indicates if some problem was found when processing the tuple by means of a boolean value, and (2) the `result` slot stores a `warning` XML-Kenzo object if some problem was found when processing the tuple or a `result` XML-Kenzo object otherwise.

Finally, as our tuple space is an AllegroCache database, the Linda model operations are implemented in a natural way from the `select`, `insert` and `delete-instance` functions of AllegroCache. For instance, the *inp* operation of the Linda model can be programmed by combining a `select` operation and a `delete-instance` one. Thus, we have built five methods with the following signatures:

$$\begin{array}{ll}
\texttt{writetuple:} & \texttt{tuple} \rightarrow Boolean, \\
\texttt{taketuple:} & \texttt{tuple} \rightarrow \texttt{tuple}, \\
\texttt{readtuple:} & \texttt{tuple} \rightarrow \texttt{tuple}, \\
\texttt{taketuplep:} & \texttt{tuple} \rightarrow \texttt{tuple} \vee \texttt{nil} \text{ and} \\
\texttt{readtuplep:} & \texttt{tuple} \rightarrow \texttt{tuple} \vee \texttt{nil},
\end{array}$$

These are, respectively, our versions for *out*, *in*, *rd*, *inp* and *rdp*.

This finishes the description of our Linda model implementation. The rest of the server framework components are three Common Lisp modules based on the components of the Kenzo framework and the interface which allows the communication with the outside. All the modules are organized in two layers: the business logic layer and the persistence layer. The persistence layer is in charge of communicating the module with the tuple space through the operations (`writetuple`, `taketuple` and so on) of our Linda model implementation. The business logic layer always works with XML-Kenzo objects produced by the persistence layer from the tuples of the tuple space.

Then, our modules are as much independent as possible from the chosen technology used to implement the Linda model, since the business logic layers always work with the same representation of objects; then, if we change our implementation of the Linda model, we only need to change the persistence layers; remaining untouched the business logic layers.

Let us explain the three modules.

2.5.2.1.1 The I/O module The *I/O module* business logic layer consists of the Kenzo framework adapter, presented in Subsection 2.2.5. The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `pending` tuples and the reading of `finished` tuples. When, a new OpenMath computation request arrives to the server framework this module is activated and proceeds as follows.

1. Check that the OpenMath request is well formed:
    (a) If the request is not well formed a `warning` OpenMath object is returned.
    (b) Otherwise, go to step 2.
2. Transform the OpenMath object into an XML-Kenzo object.
3. The persistence layer searches among the `finished` tuples of the tuple space if the result for that request was previously computed:
    (a) If the result was stored, the `finished` tuple which stores the result is read by the persistence layer and an XML-Kenzo object result is returned to the business logic layer. Go to step 6.
    (b) Otherwise, go to step 4.
4. The persistence layer constructs a `pending` tuple from the XML-Kenzo request and writes it in the tuple space.
5. When the server framework obtains a result for the computation request, the `finished` tuple which stores the result for that request is read by the persistence layer and an XML-Kenzo object result is returned to the business logic layer.
6. Transform the XML-Kenzo result into an OpenMath result.
7. Return the OpenMath result to the client.

2.5.2.1.2 The Processing module The *Processing module* business logic layer is a combination of the external server, presented in Subsection 2.2.4 and part of the functionality of the microkernel (namely, the knowledge included in that component to manage errors), presented in Subsection 2.2.3. This module is in charge of validating the correctness of requests; namely, all the knowledge included in both the external server and the microkernel to handle errors is gathered in this module. To be more concrete all the restrictions managed in the Kenzo framework (that is to say; type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, functional dependent restrictions of the arguments of the constructors, reduction degree restrictions and restriction of the dimension in computations) are handled in the processing module. Let us note that in spite of only working with computing requests, the server framework also has to check the constraints related to the construction of spaces in order to avoid errors.

The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `valid` and `finished` tuples and the reading of `pending` tuples. When, a `pending` tuple is written in the tuple space the processing module is activated and proceeds as follows.

1. The persistence layer takes the `pending` tuple.

2. The persistence layer transforms the `pending` tuple into an XML-Kenzo object which is sent to the business logic layer.

3. The business logic layer checks all the constraints about the XML-Kenzo object:

   (a) the business logic layer indicates to the persistence layer to write a `valid` tuple in the tuple space if all the constraints are fulfilled.

   (b) Otherwise, indicates to the persistence layer to write a `finished` tuple in the tuple space with `nil` as value of the `correct` slot and the warning produced as value of the `result` slot.

4. The persistent layer writes the indicated tuple in the tuple space.

2.5.2.1.3 The Kenzo module The Kenzo module business logic layer is the internal server of the Kenzo framework, presented in Subsection 2.2.2. The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `finished` tuples and the reading of `valid` tuples. When, a `valid` tuple is written in the tuple space the Kenzo module is activated and proceeds as follows.

1. The persistence layer takes the `valid` tuple.

2. The persistence layer transforms the `valid` tuple into an XML-Kenzo object which is sent to the business logic layer.

3. The business logic computes the value of the request and sends an XML-Kenzo result to the persistence layer.

4. The persistence layer constructs a `finished` tuple from the XML-Kenzo result and writes it in the tuple space.

At this moment the server framework has only a Kenzo kernel; however, as we said in Section 2.4, we should use several Kenzo kernels in order to deal with different Kenzo computations. To this aim, we could use the *Broker* pattern [B$^+$96, B$^+$07] which will allow us to distribute different parts of a computation among different Kenzo kernels. Nevertheless, this task remains as further work.

2.5.2.1.4 Relations among the modules and the tuple space The relations among the modules and the tuple space are shown in Figure 2.20.

It is worth noting that `pending` and `valid` tuples are respectively removed from the tuple space by the processing module and the Kenzo module; on the contrary, `finished` tuples remain in the tuple space to avoid re-computations achieving the challenge of storing results.

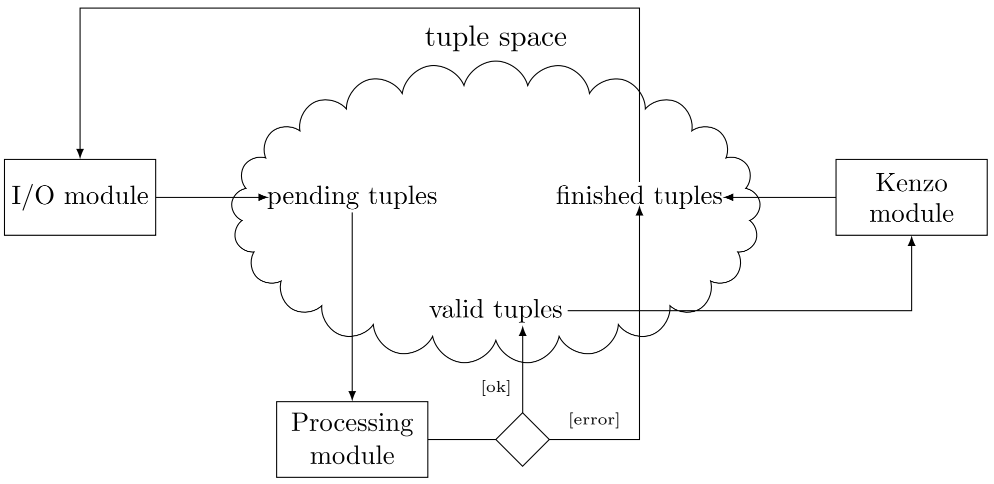


Figure 2.20: Relations among modules and the tuple space

2.5.2.1.5 Web Services The previous paragraphs have been devoted to present the different components of the framework and their relations with the tuple space. Let us focus now on other innovation with respect to the Kenzo framework: the use of web services to communicate the server framework with different clients.

In order to offer transparent access to our server computing infrastructure, three *web services* have been developed (briefly, a web service is a program, based on XML, which provides access to an application over the network). These three web services allow the use of the server framework only knowing the OpenMath format of the computation requests (no knowledge about Lisp or Kenzo is needed). So, a developer could build an application (in platforms with support for web services, such as Java, .Net, Lisp and so on) for our web services only knowing the OpenMath syntax, which is described in the OpenMath Content Dictionaries, and, of course, the web services technology of the concrete programming language. We have implemented these web services using the SOAP 1.1 API for Allegro Common Lisp [Incd]. This API allows us to open the access to our server on the internet via the *SOAP* protocol [G$^+$] (a web services protocol). Let us explain the differences among the three web services.

The first web service makes a request to the server framework and waits until the result is returned. This web service can be useful in the implementation of a light client where all the computations are done in the server. However, if the computation asked to the server framework takes too much time, the connection must be kept open all that time.

On the contrary, in the second web service, the connection is not maintained, and therefore it needs not only the OpenMath computation request but also an e-mail address where the result will be returned. In this case, to send mails, we have chosen to use Java joins with the *JavaMail* API [Mic99]. The JavaMail API is a set of abstract APIs modeling a mail system and providing the required technology. In order to do this, we need to communicate our framework (developed in Lisp) with a Java application. There exists different solutions to this problem: we could use the middleware CORBA [Gro], or use the *jLinker* package [Incc] provided by Allegro Common Lisp; nevertheless, these options are too general for our concrete aim. Then, we have used a solution based on one of the *jLinker* ideas. *jLinker* requires two open socket connections between Lisp and Java, one for calls from Lisp to Java, and another one for the inverse communication. In our case we only need one socket for communicating Lisp with Java, because our Lisp system does not require information from our Java system. In Java we have a passive socket waiting for the creation of a connection by means of an active socket (it is the same idea presented in Subsubsection 2.5.2.2 to activate the server framework components). In the Lisp system, whenever a new result is created, the web service creates an active socket which connects with the passive socket of Java and sends both the e-mail address of the client and the result of the computation requested. Finally, the Java program sends an e-mail to the client with the result of the computation asked for the client. However, this web service is not fully satisfactory, since even if the computation asked to the server framework only taking a few microseconds, the client always has to check his e-mail to obtain the result.

As we have said, neither of the two previous web services is fully satisfactory, since they implement two extreme interactions: the first one always keeps open the connection until the result is returned; on the contrary, the second one never does it. Then, to solve this problem a third web service has been implemented mixing the good properties of the others. This web service receives an OpenMath computation request and an e-mail address. When the result is obtained the system checks if the connection with the user is still open, in that case the web service directly returns the result to the user; otherwise it sends the result to the e-mail address. Clients of this web service must decide how much time the connection remains open, an issue which depends on each concrete client.

These web services solve two of the challenges stated at the beginning of this section. On the one hand, web services and OpenMath are general enough to be used for different clients. On the other hand, a subscription mechanism is implemented, then, instead of waiting for the end of a computation, this reactive mechanisms sends an e-mail with the result to the user if the user connection is not longer available.

2.5.2.2 Communication among modules

Up to now, how the different modules are communicated has not been explained. This is one of the most important issues, and has been tackled by means of a reactive mechanism, to be more concrete by means of a *publish-subscribe* machinery.

This mechanism is based on the existence of both *subjects*, which can be modified, and *observers* subscribed to any possible subjects modification. This design has been implemented following the *Publish-Subscriber* pattern [B^+96, B^+07]. This design pattern, also called *Observer*, helps to keep synchronized the state of cooperating components, providing a mechanism for asynchronous communication.

In this pattern, each subject has associated a component which takes the role of publisher. All the components which depend on changes in the subject are its subscribers (or observers). The publisher component maintains a registry of currently-subscribed components. Moreover, subscribers can be interested only in a kind of changes of the subject, so, this information is also kept by the publisher. Then, whenever the state of a subject changes, its publisher sends a notification to all the subscribers interested in that change.

In our context, we only have one subject that is our tuple space (the AllegroCache database) which has associated a publisher component (a bunch of Common Lisp functions), and the subscribers are the three modules wrapped with a notification mechanism (several Common Lisp functions) which allow them to receive the notifications from the publisher.

To store the subscriptions, an AllegroCache database, called from now on *subscription database*, is associated with the publisher of the tuple space. This database contains instances of the `subscription` class which is defined as follows:

```
(defclass subscription ()
  ((host :type string  :initarg :host :accessor host )
   (port :type integer :initarg :port :accessor port )
   (type-tuples :type symbol :initarg :type :accessor type :index any))
  (:metaclass persistent-class))
```

This class has three slots:

- `host`, a string being the *host* of a subscriber.
- `port`, an integer being the port of a subscriber.
- `type-tuples`, a symbol (`'pending`, `'valid` or `'finished`) indicating the kind of tuples which are interesting for the subscriber. In addition this slot is an *index slot*. Index slots are the way of filtering and ordering the results of a search in AllegroCache.

Moreover we have defined this class as persistent by means of the metaclass `persistent-class`. This implies that every instance of this class will be permanently stored (until a module explicitly deletes it) in the database.

The subscription database always contains, at least, three instances: one per each module of the server framework. To be more concrete, the instance associated with the I/O module includes its host, port and the interesting tuples for this module, that are `finished` tuples:

```
(make-instance 'subscription :host "localhost" :port 8001 :type-tuples 'finished)
```

the instance associated with the processing module includes its host, port and the interesting tuples for this module, that are `pending` tuples:

```
(make-instance 'subscription :host "localhost" :port 8002 :type-tuples 'pending)
```

and, finally, the instance associated with the Kenzo module includes its host, port and the interesting tuples for this module, that are `valid` tuples:

```
(make-instance 'subscription :host "localhost" :port 8003 :type-tuples 'valid)
```

It is worth noting that in this case the host of all the modules is the same. That is to say, currently, all the modules are located in the same computer. However, we could install each module in a different computer, and devote a dedicated computer for each one of the modules.

In the future, if we want to include a new module as subscriber of the tuple space, we only need to define an instance with its host, port and the interesting tuples. In this way, whenever an interesting tuple for the module is written in the tuple space, a notification is sent to the module by the publisher.

Let us explain now the notification mechanism which is based on *sockets* technology and is implemented with the Allegro Common Lisp socket package [Inca]. This package provides all the necessary tools to communicate our modules and the publisher by means of sockets technology.

Each one of the modules of our server framework has a *passive socket* in the port specified in its subscription instance. This passive socket is waiting its activation by means of the following function:

```
(defun observer-<module> ()
  (loop
    (let ((sock (make-socket :connect :passive :local-port <port>)))
      (wait-for-input-available sock)
      (close sock) (validate))))
```

where `<module>` and `<port>` are replaced with the name of the module and the port specified in the subscription database for that module. The rest of the function must be read as follows. We have a loop instruction that repeats always the same process: (1) creates a passive socket, (2) waits until the passive socket is activated, (3) closes the socket, and, finally, (4) does the job associated with the module.

Let us stress in the `observer-<module>` definition the use of the `wait-for-input-available` function. This function waits the connection of an *active socket* with the passive socket `sock`. Therefore, the well-known *busy-waiting* concurrency problem is solved (a good description of the concurrency issues presented in this memoir can be consulted in [Sch97]). *Busy-waiting* happens when a process repeatedly checks if a condition is true (for instance, the availability of a shared resource), the problem is that the processor spends all its time waiting for some condition. A sensible solution to this problem consists of using *semaphores* which is the approach followed with the `wait-for-input-available` function. Note that this function acts as a *binary semaphore*, to be precise as a `P` operation (the `P` operation sleeps the process until the resource controlled by the semaphore becomes available). Then, the module is not repeatedly checking if a new tuple has been written in the tuple space, but it is slept until someone activates it (this activation is produced when an interesting tuple for the module is written in the tuple space).

After the description of the implementation of the notification mechanism from the modules side, let us present now the publisher side. In the publisher we have the following function definition.

```
(defun notify (type-tuple)
  (open-network-database "localhost" 8010)
  (let ((subscrites
         (retrieve-from-index 'subscription 'type-tuple type-tuple :all t)))
    (dolist (temp subscrites)
      (let ((socket (make-socket :remote-host (host temp) :remote-port (port temp))))
        (close socket))))
  (close-database))
```

This function is called when a new tuple is written in the tuple space with the type of the written tuple, `type-tuple`, as argument. The workflow followed by this function when it is activated is as follows: (1) open the subscription database, (2) search all the modules which are subscribed to the writing of a tuple of `type-tuple` type in the tuple space, (3) creates an active socket for each one of the subscribers, and, finally, (4) close the database. It is worth noting that this function is acting as the `V` operation in a binary semaphore (the `V` operation is the inverse of the `P` operation: it makes a resource available). In this way, all the subscribers to the tuples of `type-tuple` type are activated.

Therefore, the tuple space and the modules are communicated achieving the challenge of an asynchronous communication among the modules of the framework.

2.5.2.3 An example of complete computation

Once we have described our server architecture, the communication among the modules and the communication between the server framework and its clients by means of web services, let us illustrate the execution flow of the server framework with a detailed scenario: the computation of the sixth homotopy group of the sphere of dimension 3, $\pi_6(S^3)$, in a *fresh* session of the server framework; that is, computations were not previously performed. The execution flow of this scenario is depicted in Figure 2.21 with a UML-like sequence diagram.

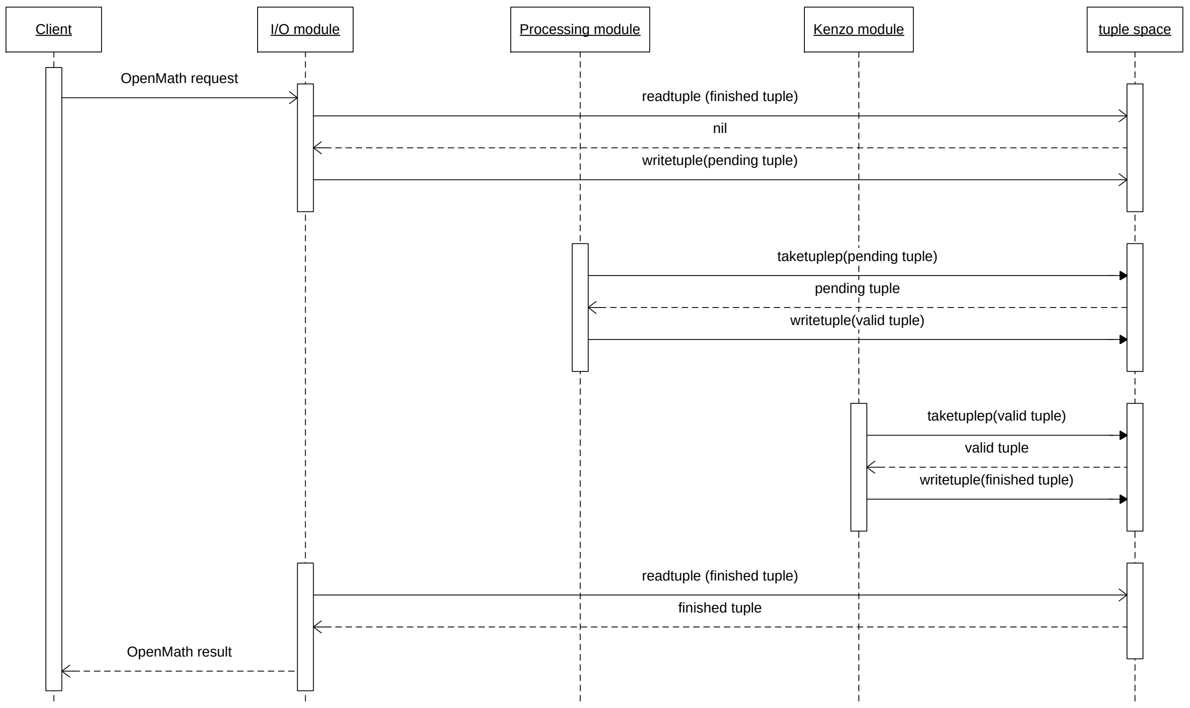


Figure 2.21: UML sequence diagram

The server has received the OpenMath request by means of anyone of the three web services presented in the previous subsubsection. The process of the request is the same for all of them, the only difference is the way of returning the result; but that was explained in the previous subsubsection. The OpenMath representation of $\pi_6(S^3)$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
            <OMS cd="SS" name="sphere"/>
            <OMI>3</OMI>
        </OMA>
        <OMI>6</OMI>
    </OMA>
</OMOBJ>
```

The I/O module receives the previous OpenMath data from a client through one of the web services. This module checks that the OpenMath instruction is well-formed and converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
  <homotopy>
    <sphere>3</sphere>
    <dim>6</dim>
  </homotopy>
</operation>
```

Subsequently, the I/O module, which has access to the `finished` tuples, asks if a `finished` tuple with the result of the request exists in the tuple space. To this aim, a `finished` template object is created from the XML-Kenzo datum in order to query the tuple space as follows (the wildcard `'?` allows us to define a search template on tuples):

```
> (readtuplep
   (make-instance 'finished
     :operation (make-instance 'operation
                 :name "homotopy"
                 :dim 6)
     :XML_Object (make-instance 'XML_Object
                  :xml-object "<sphere>3</sphere>")
     :correct '? :result '?)) ✠
NIL
```

In this case, `NIL` is returned, meaning the result is not in the database (as we have said, we are working in a fresh session, so, no computation has been performed previously). Then the I/O module writes a `pending` tuple (corresponding to the request) in the tuple space (from now on, we will not include the attribute values of the instances if they are not different from the previous ones):

```
> (writetuple (make-instance 'pending :operation (...) :XML_Object (...))) ✠
T
```

Due to the existence of a new pending tuple the processing module is activated, and it asks for a pending tuple:

```
> (taketuplep (make-instance 'pending :operation '? :XML_Object '?)) ✠
#<PENDING @ #x2143918a>
```

With `(make-instance 'pending :operation '?  :XML_Object '?)`, the processing module asks for a general pending tuple. The `taketuplep` method deletes the element of the database and creates an XML-Kenzo object used by the processing module to validate the request. In this case, the request is considered sensible so the processing module writes a valid tuple in the tuple space:

```
> (writetuple (make-instance 'valid :operation (...) :XML_Object (...))) ✠
T
```

Afterwards, the Kenzo module is activated due to the writing of a new valid tuple in the tuple space. Then it asks for the new tuple:

```
> (taketuplep (make-instance 'valid :operation '? :XML_Object '?)) ✠
#<VALID @ #x214be502>
```

and computes the result of the request writing the result as a finished tuple:

```
> (writetuple
    (make-instance 'finished :operation (...) :XML_Object (...)
                             :correct t :result "<component>12</component>")) ✠
T
```

Then the I/O module is activated and asks again for the result of the request.

```
> (readtuplep
    (make-instance 'finished :operation (...) :XML_Object (...)
                             :correct '? :result '?)) ✠
#<FINISHED-PERSISTENT oid: 5022 @ #x214cff02>
```

In this case, the result is not deleted from the tuple space because a `readtuplep` operation is used. Then, the result can be used again, without recomputing it.

Eventually, the I/O module converts the result into its OpenMath representation

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>12</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

2.5.2.4 Concurrency issues

In the previous subsubsection, we have presented an execution scenario where there is only one request in the system. In general, in a server several requests coexist at the same time; then, some *concurrency* problems can appear and must be dealt with. Let us remember that the management of concurrency problems was one of the challenges that we faced in the development of the server framework.

It is worth noting that most of the typical concurrency problems are solved thanks to the organization of our framework, based on the Linda model. As the server is based on this model, if a process wants to modify a tuple, it must extract it, modify it and then insert it again in the tuple space (remark that the extraction and the insertion operations are atomic). Therefore different processes keep independent among themselves, avoiding the occurrence of several problems related to concurrency.

Concurrency appears in two different ways in our server architecture. On the one hand, different modules can work at the same time; that is to say, while the Kenzo module is computing a homology group, the processing module can validate the correctness of a request and the I/O module can receive new requests and write them in the tuple space. On the other hand, several processes of the same module can work at the same time; that is to say, the Kenzo module can perform several computations simultaneously.

In the former case (different modules working at the same time), the most important concurrency features which must be considered in our server framework are:

- *Absence of deadlocks.* A deadlock happens when two *processes* are waiting for the other to finish, and thus neither ever does. In our server framework this situation never happens since the execution of each one of the modules is independent from the others.
- *Mutual exclusion.* Mutual exclusion occurs when some algorithms avoid the simultaneous modification of a shared resource. In our server, the shared resource is the tuple space; however, each server framework component only access to a kind of tuples. Therefore, in spite of concurrently accessing to the tuple space, the server framework components access to different kinds of tuples; then, inconsistencies are avoided and the good property of mutual exclusion is achieved.
- *Absence of starvation.* Starvation describes a situation where a process is unable to access to shared resources and is unable to make progress. This situation is not feasible in our server framework, since the modules are only activated when new resources are included in the tuple space. Moreover, the modules access to the tuple space by means of non blocking operations; then, if a module asks data to the tuple space but there is no datum, then the module backs to sleep. Therefore, our server framework components never starve.

Let us focus now on the latter case (several processes of the same module working at the same time). Every time that a module is activated a new *thread* (in charge of executing the instruction of the correspondent module) is built from the main process. In order to do this, a multiprocessing package of Common Lisp which provides all the server framework modules with the main tools for working in a concurrent way (management of *threads, locks, queues* and son on) has been used. In this situation, the most important concurrency features which must be considered in our server framework are:

- *Absence of deadlocks.* Each thread of a module is independent from the rest of threads of the module. Then, none waits for the others to finish and, then, deadlocks never happen.

- *Absence of starvation.* A module launches a thread only when the module has received some data to process and this thread is killed when if finishes its task. Therefore, starvation never happens since the thread does not access to the shared resource to receive the data, because they are created with the data that must process.

- *Absence of race condition problems.* Race conditions arise in software systems when separate threads of execution depend on some shared state. It is worth noting that if two processes read (`readtuplep`) the same tuple, two different tuples will be created when each process writes its result, then, the *race condition* problem is avoided.

This short analysis shows that some typical inconsistencies produced by concurrency problems will be avoided in our framework.

Chapter 3

Extensibility and Usability of the Kenzo framework

The previous chapter was devoted to present a framework which provides a mediated access to the Kenzo system, constraining its functionality, but providing guidance to the user in his navigation on the system. However, we cannot consider that the Kenzo framework is enough to cover fully our aims.

On the one hand, we want to be able to increase the capabilities of the system by means of new Kenzo functionalities or the connection with other systems such as Computer Algebra systems or Theorem Prover tools. On the other hand, the XML interface (to be more concrete, the OpenMath interface) is not desirable for a human user; then, a more suitable way of interacting with the Kenzo framework must be provided.

To cope with the extensibility challenge, we have deployed the Kenzo framework as a client of a *plug-in* framework that we have developed. This allows us to add new functionality to the Kenzo framework without accessing to the original source code. Moreover, as a client of both Kenzo and plug-in frameworks, an extensible friendly front-end has been developed. By combining the frameworks and the front-end we are able to improve the usability and the accessibility of the Kenzo system, increasing the number of users who can take profit of the Kenzo computation capabilities. The whole system, that is to say, the combination of the two frameworks and the front-end, is called *fKenzo*, an acronym for f̲riendly K̲e̲n̲z̲o̲ [Her09].

The rest of this chapter is organized as follows. The *plug-in* framework, which the Kenzo framework is a client of, is explained in Section 3.1. The main client of the Kenzo framework, an extensible user interface, is introduced in Section 3.2.

3.1 A plug-in framework

Architectures providing plug-in support are used for building software systems which are extensible and customizable to the particular needs of an individual user. Several interesting plug-in approaches exist. One of the first plug-in platforms was Emacs ("Editor MACroS") [Sta81], whose extensions are written in *elisp* (a Lisp dialect) and can be added at runtime. The Eclipse platform [OTI03] is certainly the most prominent representative of those plug-in platforms and has driven the idea to its extreme: "Everything is a plug-in". Plug-ins for Eclipse are written in Java. Other examples are OSGi [Ini03], a Java-based technology for developing components, or Mozilla [Moz], a web browser with a great amount of extensions.

Kenzo is also a plug-in system: to add new functionality to the Kenzo system, a file with the new functions must be included; remaining untouched the Kenzo main code. In general, all the systems implemented in Common Lisp are extensible. Therefore, to extend the functionality of a Common Lisp program, we, usually, only need to load new functions by means of Common Lisp files.

One of the most important challenges that we faced in the development of the Kenzo framework was the management of its extensibility. On the one hand, we wanted that the Kenzo framework could evolve at the same time as Kenzo. On the other hand, the framework should be enough flexible to integrate new tools, such as other Computer Algebra systems and Theorem Provers tools. In addition, we wanted that the original source code of the Kenzo framework remained untouched when the system was extended. Therefore, we decided to include a *plug-in* support in our framework by means of a *plug-in* framework developed by us.

The rest of this section is organized as follows. Subsection 3.1.1 is devoted to present the architecture of the plug-in framework. Moreover, explanations about the way of adding new functionality to the Kenzo framework through the plug-in framework are provided in Subsection 3.1.2.

3.1.1 Plug-in framework architecture

Extensibility is very important because an application is never really finished. There are always improvements to include and new features to implement.

With the aim of being able to extend different systems we have developed a plug-in framework. This framework may have systems of very different nature as clients. Therefore, the way of extending a client can be different to the way of extending the rest of them. This issue has been taken into account in the design of the plug-in framework.

The implementation of the plug-in framework has been based on the *plug-in* pattern presented in [MMS03]. This pattern distinguishes two main components: the *plug-ins* and the *plug-in manager*. Moreover, we have included two improvements of our own, the

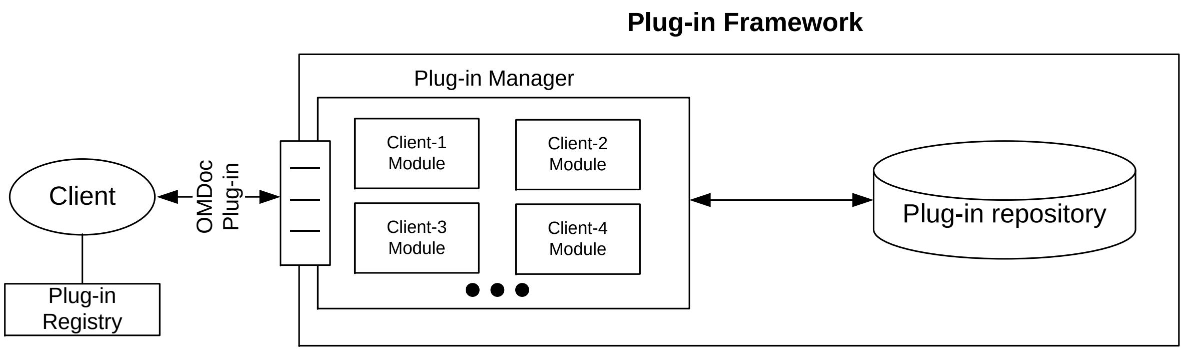


Figure 3.1: Plug-in framework

plug-in repository and the *plug-in registries* (each client of the *plug-in framework* will have associated a *plug-in registry*). A high level perspective of the plug-in framework is depicted in Figure 3.1. Let us explain each one of the constituents of our plug-in framework.

A *plug-in* in our system consists of different resources used to extend a concrete client. Since the clients of the plug-in framework are very different, the nature of the resources included in them too. However, it would be desirable that the format of all the *plug-ins* (not the format of the resources which obviously depends on each client) was common for the different clients.

Then, we have chosen *OMDoc* [Koh06] as format to codify our *plug-ins*. *OMDoc* is an open markup language for mathematical documents, and the knowledge encapsulate in them. The OMDoc plug-ins are documents that wrap the necessary resources to extend a concrete client of the plug-in framework.

Essentially, an OMDoc plug-in is an XML document, that stores the metadata about the plug-in (authorship, title, and so on) and wraps the resources (by means of references to other files that, of course, depend on the concrete client) which extend a client of the plug-in framework.

For instance, let us suppose that we have a client, called `client-1`, of our plug-in framework; and we want to extend that client by means of the resources stored in a file called `client-1-resources`. Then, the OMDoc plug-in, called `new-module-client-1`, will contain, in addition to the metadata about the *plug-in*, the following XML fragment (all the OMDoc plug-ins follow the same pattern).

```
<code id="new-module-client-1">
   <data format="client-1"> client-1-resources </data>
</code>
```

The above XML code must be read as follows. The `id` argument of the `code` tag indicates the name of the new module. Inside this tag the different resources to extend `client-1` with the new plug-in are referenced by means of the `data` tag. This tag has as `format` attribute the name of the concrete client, in this case `client-1`, and the value of the `data` tag is the reference to the concrete resource. This information is very useful for the *plug-in manager*.

When the plug-in framework receives as input an OMDoc plug-in, the plug-in manager is invoked. The *plug-in manager* is a Common Lisp program that consists of several modules, one per client of the plug-in framework. As we have said, each one of these modules is related to a concrete client, and then, is implemented depending on the extensibility needs of each one of them. The plug-in manager, depending on the information stored in the plug-in (namely the value of the `format` attribute of the `data` tag), invokes the corresponding client module. Subsequently, this module extends the client with the indicated resources.

The plug-ins and the resources referenced by them are stored in a folder included in the plug-in framework called the *plug-in repository*.

Finally, each client of the plug-in framework has associated a file called *plug-in registry*. Each one of these files stores a list of the *plug-ins* that were added to the corresponding client. When a new plug-in is included in a client, the information about that plug-in is stored in its plug-in registry. Moreover, when a client is started its first task consists of sending the information of the *plug-in registry* to the *plug-in manager* in order to achieve the same state of the last configuration of the client.

If some problem appears during the loading of a plug-in the plug-in framework informs of the error and constructively suggests a solution. The way of returning the error is by means of a `warning` OpenMath object whose value is the produced error and the feasible solution.

Once we have presented the plug-in framework, let us explain how one of its clients, that is the Kenzo framework, uses it. Another client of this framework is a customizable front-end that will be presented in Section 3.2.

3.1.2 The Kenzo framework as client of the plug-in framework

As it was discussed earlier, one of the most important issues tackled in the design of the Kenzo framework was the deployment of an extensible architecture. A good approach to solve this question consists of having a component-based architecture as base-system, and then equipping it with different components. This is the approach followed in the Kenzo framework. As was explained in the previous chapter, the base-system of the Kenzo framework is based on the *Microkernel* architectural pattern, therefore, we have a component-based architecture. Moreover, to be able to include new improvements and features, the Kenzo framework has been implemented as a client of the plug-in framework.

As we explained previously, all the Kenzo framework components are Common Lisp modules, see Section 2.2, and since Common Lisp programs are designed to be extensible, we only need to load new functions in each component by means of Common Lisp files. This is the same method followed in Kenzo to extend its functionality.

The plug-in manager of the plug-in framework contains a component that is devoted to load new functionality in the Kenzo framework. This module extends the functionality of the Kenzo framework components with two different aims. On the one hand, to provide access to new Kenzo functionality through the Kenzo framework. On the other hand, to interact with other systems, such as Computer Algebra systems or Theorem Provers tools, by means of the Kenzo framework. Let us present the integration of new functionality in the Kenzo framework.

Let us suppose that we have developed a new module for Kenzo that allows us to construct a kind of spaces that were not included in the Kenzo system, we will see concrete examples in Chapter 5, and, of course, we want to include the new functionality in our framework. Then, we need to extend all the components of the framework by means of the following files:

- a file (let us called it `new-constructor.lisp`) with the functionality to include the new constructor developed for the Kenzo system in the Kenzo component of the internal server,
- a file (let us called it `new-constructor-is.lisp`) with the functionality to expand the interface of the internal server to support the access to the new Kenzo functionality,
- a file (let us called it `new-constructor-m.lisp`) with the functionality for the microkernel to include a new construction module. This functionality follows the pattern explained in Subsubsection 2.2.3.3 for constructions modules. Moreover, this file also expands the microkernel interface to provide access to the functionality of the new construction module,
- the new specification of the XML-Kenzo language including the new constructor. As we have explained previously in Subsection 2.2.4, this will extend the capabilities of the external server. The file containing the XML-Kenzo specification is the `XML-Kenzo.xsd` file, and
- a file (let us called it `new-constructor-a.lisp`) with the functionality which increases the adapter functionality to transform from OpenMath requests to XML-Kenzo requests.

The `new-constructor` plug-in, is an OMDoc document called `new-constructor.omdoc`, which contains some metadata about the document, and references the different files in the following way.

```
<code id="new-constructor">
   <data format="Kf/internal-server"> new-constructor.lisp </data>
   <data format="Kf/internal-server"> new-constructor-is.lisp </data>
   <data format="Kf/microkernel"> new-constructor-m.lisp </data>
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/adapter"> new-constructor-a.lisp </data>
</code>
```

Each `data` element has specified in the `format` attribute one of the Kenzo framework components (internal server, microkernel, external server and adapter). The value of a `data` element is a file which extends the Kenzo framework. The indicated component in the `format` attribute of a `data` element is extended by means of the file indicated with the value of the `data` element. For instance in the case of the first `data` element, the functionality of the internal server is extended by means of the `new-constructor.lisp` file.

The plug-in framework receives as input the `new-constructor.omdoc` plug-in. Subsequently, the Kenzo framework module of the plug-in framework is invoked. This module is split in four constituents, one per Kenzo framework component. The behavior of the constituents in charge of the internal server, the microkernel and the adapter consists of loading the new functionality in the corresponding Kenzo framework component. The constituent for the external server overwrites the XML-Kenzo specification with the new one (no additional interaction is needed to extend the functionality of the external server, since when the XML-Kenzo specification is changed the external server is automatically upgraded).

Finally, the information of the new plug-in is stored in the plug-in registry of the Kenzo framework to store the configuration for further uses.

This was related to the inclusion of new Kenzo functionality in the Kenzo framework; the case of widening the Kenzo framework through the addition of an internal server (a Computer Algebra system or a Theorem Prover tool) is practically analogous. The main difference is the file related to the internal server which, instead of adding new Kenzo functionality to the current internal server, shows how to connect with the new system. Examples of the addition of new internal servers will be presented in Chapter 4.

The features associated to the Kenzo framework owing to its implementation as a client of the plug-in framework are listed below.

- Extensibility: the Kenzo framework can be extended by plug-ins. Each new functionality can be realized as an independent plug-in.
- Flexibility: each unnecessary plug-in can be removed and each necessary plug-in can be loaded at run-time. Therefore the Kenzo framework can be configured in such a way that it has only the needed functionality.
- Storage of configuration: thanks to the registry associated to the Kenzo framework,

the configuration of a session is stored for the future.

- Easy to install: the installation of plug-ins is friendly from the plug-in folder.

3.2 Increasing the usability of the Kenzo framework

The design of a client for our framework was one of the most important issues tackled in our development. The client should not only take advantage of all the enhancements included in the Kenzo framework but also be designed to increase the usability and accessibility of the Kenzo system. In this context, the program designers in Symbolic Computation always meet the same decision problem: two possible organizations[1].

1. Provide a package of procedures in the programming language L, allowing a user of this language to load this package in the standard L-environment, and to use the various functions and procedures provided in this package. The interaction with the procedures is by means of a command line interface.

2. Provide a graphical user interface (GUI) with the usual buttons, menus and other widgets to give to an inexperienced user a direct access to the most simple desired calculations, without having to learn the language L.

The main advantage of the former alternative is that the total freedom given by the language L remains available; however, the technicalities of the language L remain present as well. Moreover, the *prompt* of a command line interface do not usually provide adequate information to the user about the correct command syntax. Besides, since the user has to memorise the syntax and options of each command, it often takes considerable investment in time and effort to become proficient with the program. Therefore, command line interfaces may not be appealing to new or casual users of a software system. The vast majority of Computer Algebra systems fall in this category, including GAP, CoCoA, Macaulay, and Kenzo. On the contrary, graphical interfaces (the second alternative) are easier to use, since instead of relying on commands, the GUI communicates with the user through objects such as menus, dialog boxes and so on. These objects are a means to provide a more intuitive user interface, and supply more information than a simple prompt. However, no reward comes without its corresponding price, and GUIs can slow down expert users.

Since the final users of our framework are Algebraic Topology students, teachers or researchers that usually do not have a background in Common Lisp, we opted for implementing a user-friendly front-end allowing a topologist to use the Kenzo program guiding his interaction by means of the Kenzo framework, without being disconcerted by the Lisp technicalities which are unavoidable when using the Kenzo system. This GUI is communicated with the Kenzo framework through the OpenMath interface of the

[1]These are, of course, two extreme positions: many other possibilities can be explored between them.

adapter and can be customized by means of the plug-in framework. The whole system, that is to say the two frameworks and the GUI, is called *fKenzo*, an acronym for friendly Kenzo [Her09]. We have talked at length about the Kenzo framework in Chapter 2 and also about the plug-in framework in Section 3.1, so, let us present now the GUI of the *fKenzo* system.

The *fKenzo* GUI was implemented with the *IDE* of Allegro Common Lisp [Incb]. Various features have been implemented in this GUI. They improve the interaction with the Kenzo system from the user point of view. The main features of the *fKenzo* GUI are listed below.

1. *Easy to install*: the installation process is based on a typical windows installation.
2. *No external dependencies*: the *fKenzo* GUI does not need any additional installation to work.
3. *Functionality*: the *fKenzo* GUI allows the user to construct topological spaces of regular usage and compute homology and (some) homotopy groups of these spaces.
4. *Error handling*: this GUI is a client of the Kenzo framework; and all the enhancements included in the framework are inherited by the GUI. In this way, the user is guided in his interaction with the system and some errors are avoided.
5. *Consistent metaphors*: an advanced user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured in the GUI.
6. *Interaction styles*: the user can interact with the GUI by means of the mouse and also using keyboard shortcuts.
7. *Mathematical rendering*: the *fKenzo* GUI shows results using standard mathematical notation.
8. *Storage of sessions*: session files include the spaces constructed during a session, and can be saved and loaded in the future. Moreover, these session files can be exported, used to communicate them to other users and rendered in browsers.
9. *Storage of results*: result files storing the results obtained during a session, as in the case of session files, can be saved, exported, used to communicate them to other users and rendered in browsers.
10. *Customizable*: the *fKenzo* GUI can be configured to the needs of its users.
11. *Extensibility*: the *fKenzo* GUI can evolve with the Kenzo framework using the plug-in framework.

The rest of this section is devoted to present the GUI of the *fKenzo* program, trying to cover both the user (Subsections 3.2.1 and 3.2.2) and the developer (Subsections 3.2.3 and 3.2.4) perspectives.

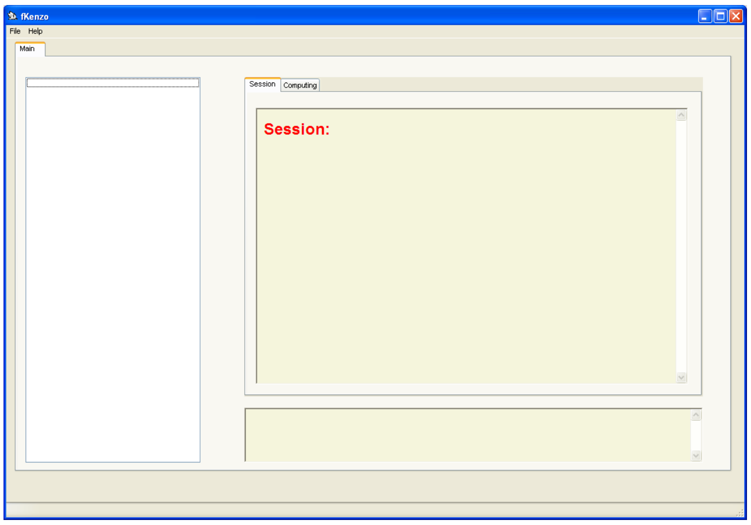


Figure 3.2: Initial screen of *fKenzo*

3.2.1 *fKenzo* GUI: a customizable user interface for the Kenzo framework

To use *fKenzo*, one can go to [Her09] and download the installer. After the installation process, the user can click on the *fKenzo* icon, accessing to an "empty" interface, shown in Figure 3.2.

Then, the first task of the user consists of loading some functionality in the *fKenzo* GUI through the different *modules* included in the distribution of *fKenzo*. From the user point of view, a module is a file which loads functionality in the *fKenzo* GUI.

The main toolbar of the *fKenzo* GUI is organized into two menus: *File* and *Help*. The *File* menu has the following options: *Add Module*, *Delete Module*, and *Exit*. The aim of the *Add Module* option consists of loading the functionality of a module. The user can configure the interface by means of five modules: *Chain Complexes*, *Simplicial Sets*, *Simplicial Groups*, *Abelian Simplicial Groups* and *Computing*. Each one of these modules corresponds with one of the XML-Kenzo groups explained in Subsection 2.2.1. In addition, some other *experimental* modules can be installed, such as a possibility of interfacing the GAP Computer Algebra system or the ACL2 Theorem Proving tool; these modules will be presented in Chapter 4.

When one of the "construction modules" (*Chain Complexes*, *Simplicial Sets*, *Simplicial Groups* or *Abelian Simplicial Groups*) is loaded, a new menu appears in the toolbar allowing the user to construct the spaces specified in the XML-Kenzo schema for that type, see Figure 2.5 of Subsection 2.2.1. Moreover, two new options called *Save Session* and *Load Session* are added to the *File* menu. When saving a session a file is produced containing the spaces which have been built in that session. These session files are saved using the OpenMath format and can be rendered in different browsers. These session files can be loaded, from the *Load Session* option, and allow the user to resume a saved session.

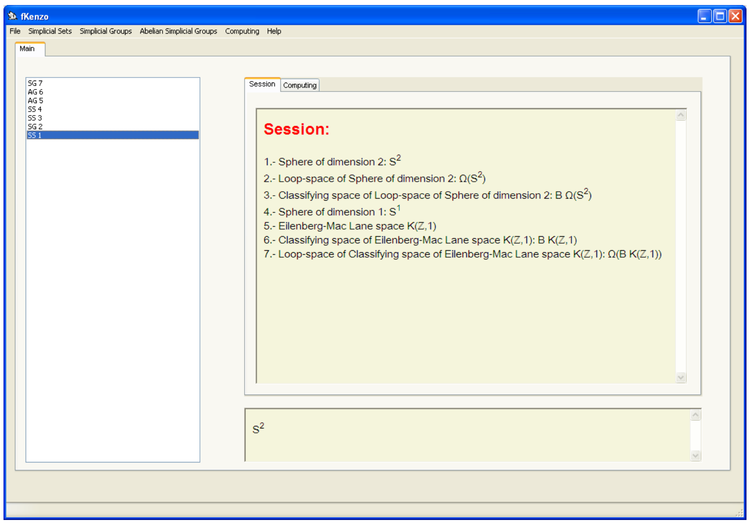
When the *Computing* module is loaded, a new menu appears in the toolbar allowing the user to compute homology and homotopy groups of the constructed spaces. Besides, a new option called *Save Computing* appears in the *File* menu. This option works in a similar way to *Save Session* but instead of saving the spaces built during the current session, it saves the computations also using the OpenMath format. However, these results cannot be reloaded into the *fKenzo* GUI, since they cannot be re-used in further computations. It is worth noting that computations depend on the state of the system, and this state cannot be exported, so we cannot re-use computations performed in a different session. When the user exits *fKenzo*, its configuration is saved for future sessions.

The visual aspect of the *fKenzo* panel is as follows. The "Main" tab contains, at its left side, a list of the spaces constructed in the current session, identified by its type (CC = Chain Complex, SS = Simplicial Set, SG = Simplicial Group, AG = Abelian simplicial Group) and its internal identification number (the `idnm` slot of the `mk-object` instance of the microkernel). When selecting one of the spaces in this list, its standard notation appears at the bottom part of the right side. At the upper part, there are two tabs "Session" (containing a textual description of the constructions made, see an example of session in Figure 3.3) and "Computing" (containing the homology and homotopy groups computed in the session; see an example in Figure 3.4). In both "Session" and "Computing" tabs the results are rendered using again standard mathematical notation.

In the *fKenzo* GUI, focus concentrates on the object (space) of interest, as in Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it (in particular, with respect to computing results). It is separated by means of tabs not only because on the division between spaces and computed results (the system moves from one to the other dynamically, putting the focus on the last user action), but also due to the extensibility of *fKenzo*. To be more precise, our system is capable of evolving to integrate with other systems (computational algebra systems or theorem proving tools), so playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns. Examples of the integration of a Computer Algebra system and a Theorem Prover tool in *fKenzo* will be presented in Chapter 4.

3.2.2 *fKenzo* in action

To illustrate the performance of the *fKenzo* program, let us consider a hypothetical scenario where a graduate course is devoted to *fibrations*, in particular introducing the functors *loop space* Ω and *classifying space* B. In the simplicial framework, [May67] is a good reference for these subjects. In this framework, if X is a connected *space*, its loop space ΩX is a simplicial group, the structural group of a universal fibration $\Omega X \hookrightarrow PX \to X$. Conversely, if G is a simplicial group, the classifying space BG is the base space also of a universal fibration $G \hookrightarrow EG \to BG$. The obvious symmetry



Session:

1.- Sphere of dimension 2: S^2
2.- Loop-space of Sphere of dimension 2: $\Omega(S^2)$
3.- Classifying space of Loop-space of Sphere of dimension 2: $B\ \Omega(S^2)$
4.- Sphere of dimension 1: S^1
5.- Eilenberg-Mac Lane space $K(\mathbb{Z},1)$
6.- Classifying space of Eilenberg-Mac Lane space $K(\mathbb{Z},1)$: $B\ K(\mathbb{Z},1)$
7.- Loop-space of Classifying space of Eilenberg-Mac Lane space $K(\mathbb{Z},1)$: $\Omega(B\ K(\mathbb{Z},1))$

S^2

Figure 3.3: Example of session in *fKenzo*

between both situations naturally leads to the question: in an appropriate context, are the functors Ω and B inverse of each other?

For the composition $B\Omega$, let us compare for example the first homology groups of S^2 with the homology groups of $B\Omega S^2$, and the homology groups of ΩS^2 and $\Omega B\Omega S^2$.

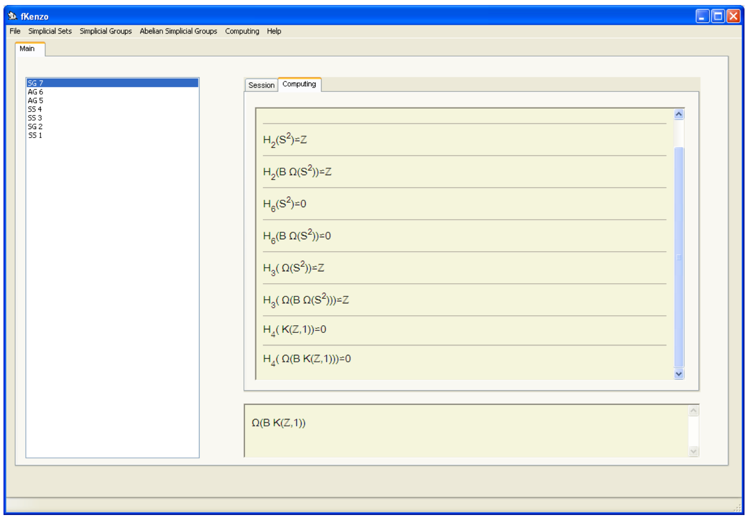
Since we are planning to work with simplicial sets and simplicial groups, we can load the necessary functionality using *File* $\rightarrow$ *Add Module*, and then choosing `Simplicial-Groups.omdoc`. The interface changes including now a new menu called *Simplicial Sets* and another one called *Simplicial Groups*. When selecting *Simplicial Sets* $\rightarrow$ *sphere*, *fKenzo* asks for a natural number, as we saw in the definition of XML-Kenzo, limited to 14. Then (see Figure 3.3) the space denoted by `SS 1` appears in the left side of the screen; when selecting it, the mathematical notation of the space appears in the bottom part of the right side of the panel. The history of the constructed spaces is shown in the "Session" tab.

If we try to construct a classifying space from the *Simplicial Group* menu, *fKenzo* informs us that it needs a simplicial group (thus likely an error is avoided). We can then construct the space ΩS^2. Since ΩS^2 is the only simplicial group in this session, when using *Simplicial Groups* $\rightarrow$ *classifying space*, ΩS^2 is the only available space appearing in the list which *fKenzo* shows. Finally, we can construct the space $\Omega B\Omega S^2$.

When loading the `Computing.omdoc` file, the menu *Computing*, where we can select "homology", becomes available. In this manner we can compute the homology groups of the spaces as can be seen in Figure 3.4.

These results give some plausibility to the relations $B\Omega = id$ and $\Omega B = id$.

The reader could wonder why the simpler case of S^1 has not been considered. Using again *fKenzo* this time a warning is produced; yet the result $B\Omega S^1 \sim S^1$ is true. But the Eilenberg-Moore spectral sequence cannot be used in this case to compute $H_*\Omega S^1$,



$H_2(S^2)=Z$

$H_2(B\ \Omega(S^2))=Z$

$H_6(S^2)=0$

$H_6(B\ \Omega(S^2))=0$

$H_3(\ \Omega(S^2))=Z$

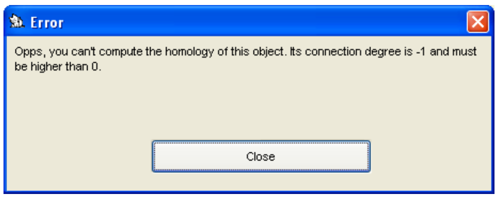
$H_3(\ \Omega(B\ \Omega(S^2)))=Z$

$H_4(\ K(Z,1))=0$

$H_4(\ \Omega(B\ K(Z,1)))=0$

$\Omega(B\ K(Z,1))$

Figure 3.4: Example of computations in *fKenzo*



Opps, you can't compute the homology of this object. Its connection degree is -1 and must be higher than 0.

Figure 3.5: Reduction degree error in *fKenzo*

because S^1 is not simply connected, and *fKenzo* checks this point (see Figure 3.5). The symmetry comparison between S^1 and ΩBS^1 fails too, for another reason: the "standard" S^1 is a topological group, but the standard simplicial representation of S^1 cannot be endowed with a structure of simplicial group. This situation is reflected in the construction of classifying spaces in *fKenzo* since the sphere S^1 does not appear in the list which *fKenzo* shows.

This is a good opportunity to introduce the Eilenberg MacLane space $K(\mathbb{Z}, 1)$, the "minimal" Kan model of the circle S^1, a simplicial group. In order to work with Eilenberg MacLane spaces in *fKenzo*, the *Abelian Simplicial Group* module should also be loaded, in this way the space $K(\mathbb{Z}, 1)$ can be built, as can be seen in Figure 3.3. The *fKenzo* comparison between the first homology groups of $K(\mathbb{Z}, 1)$ and $\Omega BK(\mathbb{Z}, 1)$ does give the expected result.

We think this illustrates how *fKenzo* can be used as a research tool, precisely a specialized computer tool, for Algebraic Topology.

Up to now, we have presented the user point of view. In the following subsections, some explanations on the development of the *fKenzo* GUI are given.

3.2.3 Customization of the *fKenzo* GUI

The most important challenge that we have faced in the development of the *fKenzo* GUI was the deployment of an extensible and modular user interface. Modularity has two aims in *fKenzo*. One of them is related to the separation of concerns in the user interface. The second one allows us to design a dynamically extensible GUI, where modules are plugged in.

3.2.3.1 Declarative programming of User Interfaces

In all the graphical user interfaces exist a separation of concerns, hence if we want to extend a GUI we need to extend it at all its levels. With respect to this aspect, our inspiration comes from [HK09], where a proposal for declarative programming of user interfaces was presented. In [HK09], the authors distinguished three constituents in any user interface: *structure*, *functionality* and *layout*.

Structure: Each *UI* has a specific hierarchical *structure* which typically consists of basic elements (like text input fields or selection boxes) and composed elements (like dialogs).

Functionality: When a user interacts with a UI, some events are produced and the UI must respond to them. The event handlers are functions associated with events of some widget and that are called whenever such event occurs (for instance clicking over a button).

Layout: The elements of the structure are put in a layout to achieve a visually appealing appearance of the UI. In some approaches layout and structural information were mixed, however in order to obtain clearer and reusable implementations these issues should be distinguished.

The approach presented in [HK09] used the *Curry* language [Han06] to declare all the ingredients. Instead of doing a similar work, but using the Common Lisp language (recall that our GUI is implemented in Common Lisp) to all the ingredients, we have preferred to employ different technologies devoted to each one of the constituents. Let us present each one of these ingredients in our context using a concrete example, that is the window used to construct a sphere in the *fKenzo* GUI, see Figure 3.6.

The *structure* of our GUI is provided by *XUL* [H^+00]. XUL, XML User Interface, is Mozilla's XML-based user interface language which lets us build feature rich cross-platform applications defining the structure of all the elements of a UI. Then, a XUL description must be provided in order to define the structure of the elements of our GUI. The main reason to codify the structure of our GUI by means of XUL is the reusability of this language. The XUL code can be used in order to build forms in different environments for different applications without designing new interfaces.

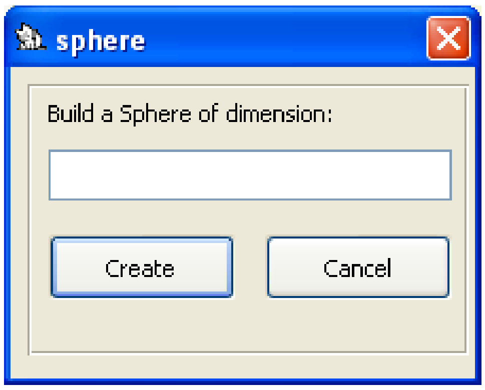


Figure 3.6: Sphere dialog

Let us examine the structure of the window of Figure 3.6. That window is called "sphere" and gathers in a groupbox the following elements: the text "Build a Sphere of dimension:", a textbox to introduce a natural number between 1 and 14, and a row that contains both "Create" and "Cancel" buttons. Moreover, each button has associated an event when its state is changed, namely when the button is clicked. We can specify that structure in the following XUL code:

```
<window name="sphere">
 <groupbox>
     <label value="Build a Sphere of dimension:"/>
     <textbox id="n" type="number" min="1" max="14"/>
     <hbox>
       <button label="Create" name="create" event="on-change"/>
       <button label="Cancel" name="cancel" event="on-change"/>
     </hbox>
 </groupbox>
</window>
```

As we have said previously, we can use the above XUL code in different clients. For instance, the previous XUL is presented in a Mozilla browser [Moz] as shown in Figure 3.7.

As can be thought, providing the XUL description of an element of a GUI can be a tedious task due to the XML nature of XUL. To make this task easier, an interpreter which is able to convert from the Allegro Common Lisp *IDE* forms to their XUL representation, and viceversa, has been developed. This is a more comfortable way of working because we define the forms in the Allegro *IDE* using a graphical interface, and then, the interpreter automatically generates the XUL code.

The *functionality* of the elements of our GUI, that is the set of *event handlers*, has been programmed in Common Lisp keeping the following convention in order to define the names of the *event handlers* functions:

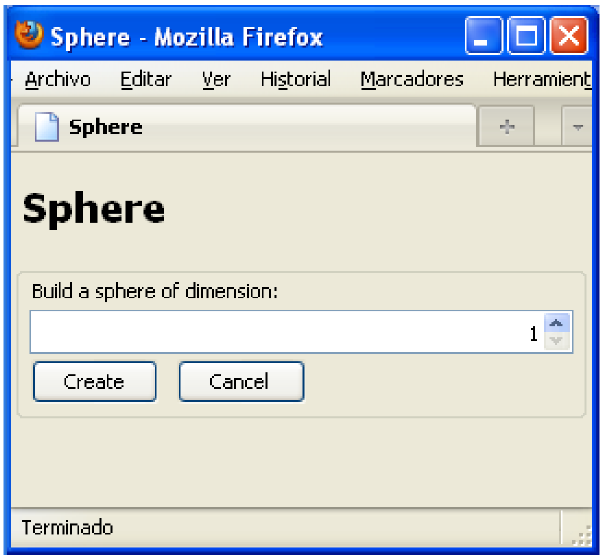


Figure 3.7: Sphere form in Firefox browser

```
(defun <window-name>-<element-name>-<event> (params)
  ;; event handler code
)
```

where `<window-name>`, `<element-name>` and `<event>` must be replaced with the name of the dialog, the name of the element and the event, respectively. For instance, the event associated with the “Create” button of the sphere dialog, see Figure 3.6, is codified as follows:

```
(defun sphere-create-on-change (params)
  ;; event handler code
)
```

Finally, the *layout* of the elements of our GUI, that is the visual appearance of the GUI, can be configured by means of a *stylesheet* [K$^+$07]. For instance, if we define the following (fragment of a) stylesheet:

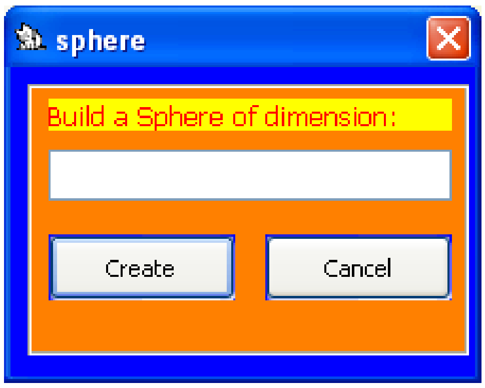


Figure 3.8: Sphere dialog with a stylesheet

```
<xsl:template name="window">
     <xsl:param name="color">blue</xsl:param>
</xsl:template>

<xsl:template name="groupbox">
     <xsl:param name="color">orange</xsl:param>
</xsl:template>

<xsl:template name="label">
     <xsl:param name="background-color">yellow</xsl:param>
     <xsl:param name="font-color">red</xsl:param>
</xsl:template>
```

the sphere dialog would have the aspect shown in Figure 3.8. That is to say, the stylesheet is used to modify the visual attributes of the elements of the GUI. If we do not provide a stylesheet the default values of the visual aspect attributes are used.

In this way, all the graphical constituents of the interface can be defined.

The next subsubsection is devoted to present how the information related to the different constituents is stored in our modules.

3.2.3.2 *fKenzo* GUI modules

A *fKenzo* GUI module is an OMDoc document that references at least two resources: a file that contains the structure of the graphical elements of the module and another one containing the functionality of those elements. In addition, a *fKenzo* GUI module can reference a file with the layout. Besides, a *fKenzo* GUI module provides some metadata (authorship, title and so on). The following conventions have been followed in the four *fKenzo* GUI construction modules and also in the computation one.

A *fKenzo* GUI module follows the schema presented for the rest of plug-ins of the plug-in framework, see Subsection 3.1.1. For instance, in the case of the "Simplicial

Groups" module:

```
<code id="Simplicial Groups">
 <data format="fKenzo/GUI/structure"> simplicial-groups-structure </data>
 <data format="fKenzo/GUI/functionality"> simplicial-groups-functionality </data>
</code>
```

The first reference corresponds to the structure of the graphical constituents of that module. This document is called "<Module>-structure" where "<Module>" is the name of the correspondent module. This file is an OMDoc document. To introduce XUL code in these documents we have used an OMDoc feature called *OpenMath foreign objects* (the `<OMForeign>` tag) which allows us to introduce non-OpenMath XML in OMDoc files. For instance, the module of Simplicial Groups includes a new menu with two menu items: one to construct Loop spaces (which has associated the shortcut "Ctrl+L" and the event `show-loop-space`) and another one to Classifying spaces (which has associated the shortcut "Ctrl+Y" and the event `show-classifying`). In this case the structure of the new menu is stored in the document "Simplicial-Groups-structure" with the following XUL code.

```
<OMForeign>
  <toolbarbutton type="menu" label="Simplicial Groups">
    <menupopup>
      <menuitem label="Loop Space" acceltext="Ctrl" accesskey="L"
                command="show-loop-space"/>
      <menuitem label="Classifying Space" acceltext="Ctrl" accesskey="Y"
                command="show-classifying"/>
    </menupopup>
  </toolbarbutton>
</OMForeign>
```

The functionality related to a concrete module is encoded in an OMDoc document called "<Module>-functionality" where "<Module>" is the name of the corresponding module. To this aim, we use an OMDoc feature which allows us to introduce code (in our case Common Lisp functions) in OMDoc files by means of the `code` tag. For instance, the functionality of the event called `show-loop-space` associated to the `Loop Space` menu item of the `Simplicial Groups` menu is encoded in the "Simplicial-Groups-functionality" document as follows.

```
<code id="show-loop-space">
    <metadata>
        <description> The event associated to the Loop Space menuitem </description>
    </metadata>
    <data format="application/fKenzo">
        <![CDATA[ (defun show-loop-space ()
                    ;; code ) ]]>
    </data>
</code>
```

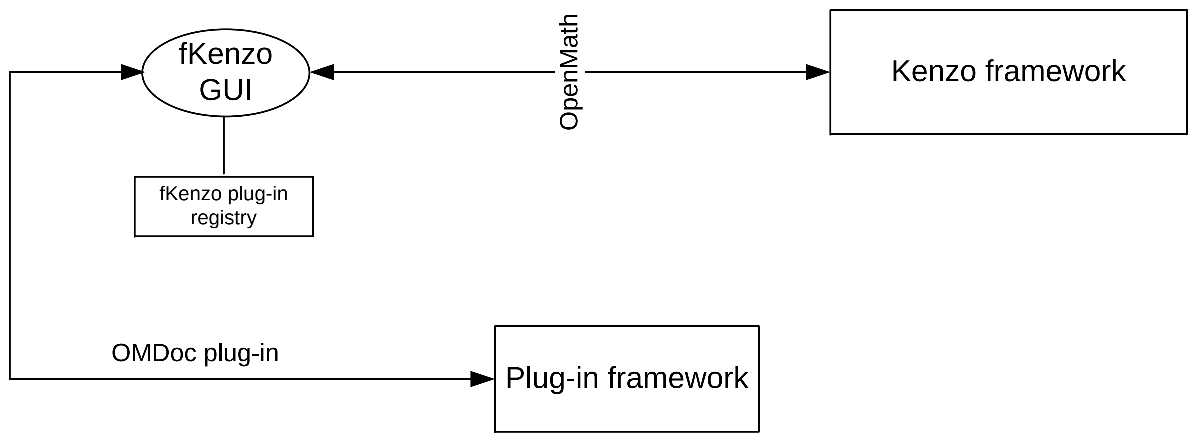


Figure 3.9: Plug-in framework and *fKenzo* GUI

Finally, the resource related to the layout is optional, and in particular the *fKenzo* GUI modules use the default layout, so they never reference any layout file. Anyway, if we want to provide a different appearance for the graphical elements of a module we can define an OMDoc document where we encode the stylesheet, which customizes the visual aspect of the elements of the module, using the same feature employed in the case of structure documents, that is the `<OMForeign>` tag.

The organization presented here allows us to deal with the design of a dynamically extensible GUI, where modules are plugged in. Our front-end becomes *extensible* thanks to the plug-in framework since each user interface unit is encoded in a unique OMDoc file, with its inner modular organization: structure, functionality and (optionally) layout.

3.2.3.3 *fKenzo* GUI as client of the plug-in framework

To tackle the extensibility question in our user interface, the *fKenzo* GUI has been designed not only as a client of the Kenzo framework, but also as a client of the plug-in framework presented in Section 3.1. In this way, the *fKenzo* GUI can be extended in an easy way by means of modules, described in the previous subsubsection. In this context, instead of using the term *plug-in* we prefer the term *module* which is more appropriate (an application which is extended by means of modules does not have any functionality, apart from the one which allows us to load modules, if we have not added any module, as the *fKenzo* GUI; on the contrary, an applications which is extended by means of plug-ins can work without adding them, as the Kenzo framework). The relations among the *fKenzo* GUI, the Kenzo framework, and the plug-in framework are depicted in Figure 3.9.

The plug-in manager (see Subsection 3.1.1) of the plug-in framework contains a module in charge of extending the *fKenzo* GUI. This module extends the GUI providing access to a concrete part of the functionality of the Kenzo framework by means of the five modules explained for the *fKenzo* GUI (the four construction modules and the computing one).

In particular, the plug-in manager of the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. This module is split in two constituents. The first one is an interpreter in charge of converting from XUL code to Common Lisp code the different graphical components. In addition if a layout file is specified, then the layout properties are applied to the graphical components. The second one associates the functionality of the event handlers to the elements defined in the structure document.

Then, it is enough to produce an OMDoc file with the suitable components, and then it can be interpreted and added in our GUI. It is exactly what happened when in Subsection 3.2.2 we described the way of working with *fKenzo*: using *File* $\rightarrow$ *Add Module* with the `Simplicial-Groups.omdoc` module sends this module to the plug-in framework. Subsequently, the plug-in manager invokes the *fKenzo* module (one of the subcomponents of the plug-in manager) that extends the user interface.

The implementation of the *fKenzo* GUI as a client of the plug-in framework shows the feasibility and usefulness of this framework.

The *fKenzo* GUI features partly owing to the implementation of the user interface as a client of the plug-in framework are listed below.

- Modularity: the GUI is organized in different modules, each one devoted to a concrete concern.
- Extensibility: the GUI can be extended by the modules. Each new functionality can be realized as an independent module.
- Flexibility: each unnecessary module can be removed and each necessary module can be loaded at run-time. Therefore the GUI can be configured in such a way that it has only the needed functionality.
- Easy to install: the installation of modules is friendly (only select a file with the option `Add Module`) from the plug-in folder.
- Internet based update: the GUI supports an update mechanism from the `Help` menu. This allows the GUI to download new modules or updates.
- Storage of configuration: the GUI configuration is automatically saved for further sessions.

3.2.3.4 Extending the Kenzo framework from the *fKenzo* GUI

The previous subsubsections have been devoted to explain how the *fKenzo* GUI can be customized by means of the plug-in framework. Moreover, the plug-in framework can also be employed to increase the functionality of the Kenzo framework as we presented in Subsubsection 3.1.2. Besides, in the same way that we wanted that the Kenzo framework could evolve at the same time as Kenzo, we also hope that the *fKenzo* GUI will be able to evolve at the same time that the Kenzo framework.

Up to now, the *fKenzo* modules that we have presented (the four construction modules and the computation one) to customize the *fKenzo* GUI do not suppose any improvement in the Kenzo framework. However, it is worth noting that the modules for the GUI not only can extend the GUI but also the Kenzo framework.

Let us retake the example presented in Subsubsection 3.1.2 where a plug-in called `new-constructor` was defined to increase the functionality of the Kenzo framework by means of a new constructor. Now, following the guidelines of Subsubsection 3.2.3.2, we can define three files (structure, functionality and layout) to customize the GUI to interact with the new constructor. Finally, we define a *fKenzo* GUI module which not only references the three files (structure, functionality and layout) to customize the GUI but also the plug-in which adds new functionality to the Kenzo framework.

```
<code id="new-constructor">
 <data format="fKenzo/GUI/structure"> new-constructor-structure </data>
 <data format="fKenzo/GUI/functionality"> new-constructor-functionality </data>
 <data format="fKenzo/GUI/layout"> new-constructor-layout </data>
 <data format="fKenzo/GUI/Kf"> new-constructor-plug-in </data>
</code>
```

When the user selects this new module from the `Add Module` option, the plug-in framework will extend both the *fKenzo* GUI and the Kenzo framework. In Subsubsection 3.2.3.3, we explained that the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. We said that this module is split in two constituents but we have included a new constituent devoted to invoke the Kenzo framework module of the plug-in framework for the cases explained in this subsubsection. This last constituent is only invoked if the *fKenzo* GUI module references a Kenzo framework plug-in, in that case the Kenzo framework module of the plug-in manager is also called.

In addition, the *fKenzo GUI plug-in registry* and the *Kenzo framework plug-in registry* must be coherent in order to avoid inconsistencies in the whole system.

A high level perspective of the interaction between the two frameworks and the GUI can be seen in Figure 3.10.

This extensibility principle makes very easy to us the incorporation of experimental features to the system, without interfering with the already running modules, as we will see in Chapters 4 and 5.

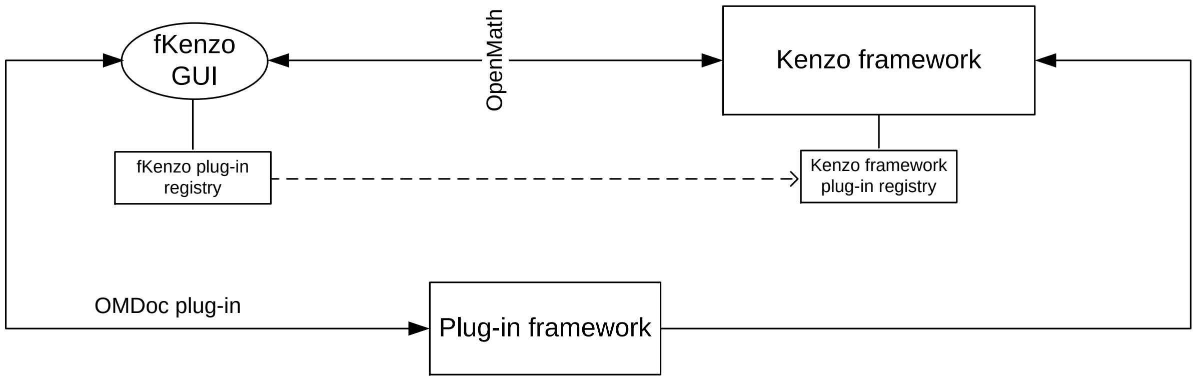


Figure 3.10: Plug-in framework, *fKenzo* GUI and Kenzo framework

3.2.4 Interaction design

In the previous subsection we have explained the design decisions that we took to develop an extensible and modular interface, probably the most important feature of the *fKenzo* GUI from the developer perspective. However, other decisions were taken on the *fKenzo* user interface design.

3.2.4.1 Task model

The first idea guiding the construction of a user interface must be the objectives of the interaction. In *fKenzo* there is only one higher-level objective: to compute groups of spaces. This main objective is later on broken in several subobjectives, trying to emulate the way of thinking of a typical Kenzo user. Once this first objective analysis is done, the next step is to design a task model. That is to say, a hierarchical planning of the main actions the user should undertake to get his objectives. This is a previous step before devising the navigation of the user, which will give the concrete guidelines needed to implement the interface.

In our case, the two main actions of the system are: (1) computing groups, and (2) constructing spaces. Note that the second task is necessary to carry out the first one. In turn, the task of constructing spaces can be separated into: (a) constructing new fresh spaces and (b) loading spaces from a previous session. Thus, the notion of session comes on the scene. With respect to the construction of fresh spaces, once the user has decided to go for it, he should decide which type of space he wants to build: simplicial set, simplicial group, and so on. Note that the construction of a space of a type can involve the construction of other space of whether its same type or a different type. This third layer of tasks gives us the module organization of the interface, while computing produces a separated module.

The task design is organized hierarchically by diagrammatic means. See in Figure 3.11 a first decomposition layer depicted by means of a package diagram [Gro09].

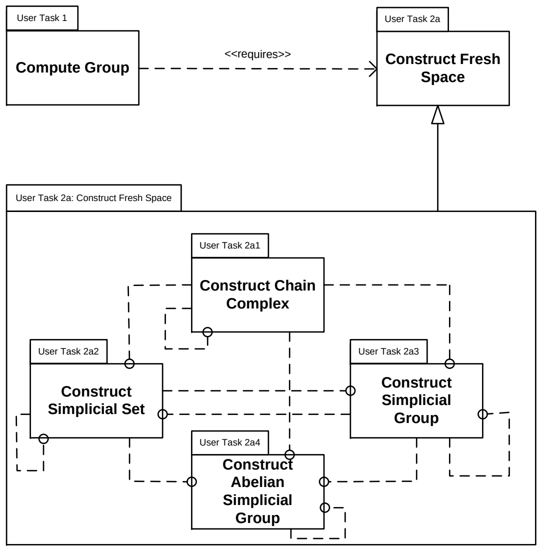


Figure 3.11: Hierarchical decomposition of the "Construct Fresh Space" user task

Each task (each frame) is linked to auxiliary tasks (giving a horizontal dependency structure). Then, each frame is described in more detail (vertical structure) by making explicit its subtasks graph.

Task modeling provides us with both the high level modular structure and the different steps needed to reach a user subobjective. The concrete actions a user should perform to accomplish the tasks are devised in the control and navigation models.

3.2.4.2 Control and navigation model

The design of the interaction between a user and a computer program involves well-known challenges (use of convenient metaphors, consistency of the control through the whole application, and so on). In order to avoid some frequent drawbacks we have followed the guidelines of the Noesis method (see [DZ07] for the general theory, and [CMDZ06] for the design of reactive systems). In particular, our development has been supported by the Noesis models for control and navigation in user interfaces. These graph-supported models enable an exhaustive traversal study of the interfaces, allowing the analyst to detect errors, disconnected areas, lack of uniformity, etc. before the programming phase. Figure 3.12 shows the control and navigation submodel describing the construction of a loop space $\Omega^n X$. Different kinds of interactions are graphically represented in Figure 3.12 by different icons. For instance, selecting from a list is depicted with a form icon; directly writing an input is depicted with a pen, and so on. These pictures help the programmer to get a quick overall view of the different controls to be implemented.

Let us observe that this diagrammatic control model is *abstract*, in the sense that nothing is said about the concrete way the transitions should be translated into the user

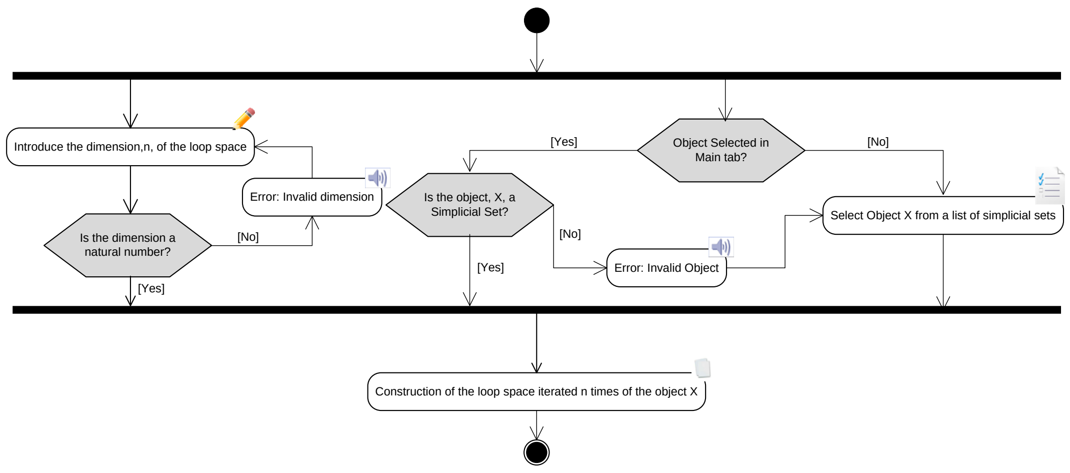

Introduce the dimension,n, of the loop space
Error: Invalid dimension
Is the dimension a natural number?
[No]
[Yes]
Object Selected in Main tab?
[Yes]
[No]
Is the object, X, a Simplicial Set?
[No]
[Yes]
Error: Invalid Object
Select Object X from a list of simplicial sets
Construction of the loop space iterated n times of the object X

Figure 3.12: Control graph for the construction of $\Omega^n X$

interface. In fact, in the *fKenzo* GUI this model is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through control-keys. The second style has been included thinking of advanced users, who want to use shortcuts to access the facilities of the interface. The adaptation to different kinds of users is one of the principles for design usability in [Nie94], and has been considered, as the rest of principles, in our development.

3.2.4.3 Challenges in the design of the *fKenzo* GUI

User interface design is a central issue for the usability of a software system. Ideally, the design of a user interface should be done following certain rules, such as those listed in guidelines documents, see for instance [KBN04]. However these guidelines have hundreds of rules, then instead of strictly following those rules the design of a user interface abides by heuristics rules based on common sense. A small set of heuristic principles more suited as the basis for practical design of user interfaces was given in [Nie94]. We have used the nine principles given in [Nie94] for guiding the design decisions of the *fKenzo* GUI.

1. *Visibility of system status.* the *fKenzo* GUI should always keep users informed about what is going on. As we have said previously, the *fKenzo* GUI can be used to construct spaces and compute groups. In the case of the spaces constructed in the *fKenzo* GUI this first principle is achieved, since the *fKenzo* GUI shows a list with the spaces constructed in the current session to the user. Related to the computation of groups, some calculations in Algebraic Topology may need several hours, then to dealt with the *visibility of the fKenzo status*, a message informs the user of this situation when a computation is performed. Besides, the *fKenzo* GUI allows the user to interrupt the current computation and keep on working with his session.

2. *Match between system and the real world.* the *fKenzo* GUI should show results

using well-known Algebraic Topology mathematical notation. This second aspect has been solved by means of combining OpenMath and stylesheets. When selecting one of the spaces of the left list of the main *fKenzo* GUI window, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. A *stylesheet* has been defined using *XSLT* [K$^+$07]. This *stylesheet* is in charge of rendering using mathematical notation the object represented with an OpenMath instruction. In both "Session" and "Computing" tabs the results are also rendered using mathematical notation thanks to the same *stylesheet*.

3. *Consistency and standards.* A user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured. Then, the *fKenzo* GUI is *consistent* with respect to Kenzo. It is worthwhile noting that this is the most influential requirement with respect to the visual aspect of our interface. In addition to the menu bar, there are three main parts in the screen: a left part, with a listing of the objects already constructed in the current session, a right panel with several tabs, and a bottom part with the standard mathematical representation of the object selected. Thus, focus concentrates on the object (space) of interest, as in Common Lisp/Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it. It is separated by means of tabs not only because on the division among spaces and computing results (the system moves dynamically from one to the other), but also due to the capability of integrating other systems, playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns.

4. *Error prevention.* The *fKenzo* GUI should forbid the user the manipulations raising errors. The most important design decision related to this point is the use of the GUI as client of the Kenzo framework. In this way, all the enhancements included in the framework are inherited by the GUI forbidding the user some manipulations raising errors and guiding his interaction with the system.

5. *Recognition rather than recall.* The *fKenzo* GUI should minimize the user's memory load. This design principle is fulfilled thanks to the combination of stylesheets and OpenMath that are used to inform the user about the selected space. Moreover, this principle was important for the design of the dialogs used to construct spaces from other spaces and compute groups. For example, if a space is selected in the screen of the *fKenzo* GUI the dialogs used to construct spaces from other spaces and compute groups take that space by default as input, then the user does not need to select the space in the dialog.

6. *Flexibility and efficiency of use.* The *fKenzo* GUI should provide shortcuts that speed up the interaction for the expert user and also suit the needs of each user. To handle this question, the interaction with the GUI is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through

control-keys used as accelerators. Moreover, thanks to the organization of the system by means of modules, a user can load the functionality that he needs.

7. *Minimalist design.* The *fKenzo* GUI should not contain irrelevant information; to that aim, the dialogs showed to the user only contain the key information.

8. *Good error messages.* the *fKenzo* GUI should indicate precisely the problems. Thanks to the use of the GUI as client of the Kenzo framework, the warnings obtained from the framework are used in the GUI. These warnings express in plain language the problem, and constructively suggest a solution.

9. *Help and documentation.* The *fKenzo* GUI should include a good documentation. Even though the *fKenzo* GUI can be used without documentation, help and documentation are provided in the `Help` menu. This help is always available, is focused on the user's tasks, lists concrete steps to be carried out, and contains both information about the use of the *fKenzo* GUI and the underlying mathematical theory.

In summary, design principles has been followed in the *fKenzo* GUI obtaining in this way a usable interface for the Kenzo system through the Kenzo framework.

Chapter 4

Interoperability for Algebraic Topology

When working in Mathematics, a researcher or student uses different sources of information to solve a problem. Typically, he can consult some papers or textbooks, make some computations with a Computer Algebra system, check the results against some known tables or, more rarely, try some conjectures with a Proof Assistant tool. That is to say, mathematical knowledge is dispersed among several sources.

Our aim consists of mechanizing, in some particular cases, the management of these multiple-sources information systems by means of *fKenzo*. Since it would be too pretentious to try to solve fully this problem, we work in a very concrete context. Thematically, we restrict ourselves to (a subset of) Algebraic Topology. With respect to the sources, in order to have a representation wide enough, we have chosen two Computer Algebra systems (Kenzo and GAP), and a Theorem Prover (ACL2).

This aim has some common concerns with the well known SAGE project [Ste] and the Software Composability Science project [F$^+$08], since all of us are trying to join several mathematical software packages. There are, however, important differences. SAGE is an integrated system in which users interact with the SAGE front-end and the Computer Algebra systems are used as back-end servers. The representation of mathematical objects in SAGE is based on an internal representation. On the other hand, the Science project provides a framework that allows services to be both provided and consumed by any Computer Algebra system. The Science project uses OpenMath as representation for mathematical objects.

Our approach combines some of the characteristics of both SAGE and Science projects but also has some significant differences. As in the SAGE initiative, we provide a common front-end to use the different systems that are integrated in the Kenzo framework as internal servers; however, the SAGE front-end is a command line interface, with the problems that this approach presents for a non expert user (we discussed this question in Section 3.2); whereas, in our case we provide a friendly graphical user interface, the *fKenzo* GUI. Moreover, when a user wants to invoke a system from SAGE, he must do it explicitly; on the contrary our front-end hides the details about the system employed at each moment. To communicate the mathematical objects between our system and other systems we use both OpenMath, as in the Science project, and our XML-Kenzo language. In addition, we use some of the programs developed by the Science initiative in our development.

It is worth noting that we do not only want to integrate Computer Algebra systems in our framework (as in the case of SAGE and Science projects), but also Theorem Prover tools.

In addition, our final aim not only consists of having several Computer Algebra systems and Theorem Prover tools, and use them individually by means of a common GUI, but also making them work in a coordinate and collaborative way to obtain new tools and results not reachable if we use severally each system.

In general, the integration of Computer Algebra systems and systems for mechanized reasoning tries to overcome their weak points: efficiency in the case of Theorem Provers and consistency in the case of Computer Algebra systems. There are several possibilities to interface Computer Algebra and Theorem Prover systems, let us cite only three of them: (1) use a Computer Algebra system as a hint engine for a Theorem Prover, (2) use a Computer Algebra system as a proof engine for a Theorem Prover, and (3) prove in the Theorem Prover the correctness of Computer Algebra algorithms.

Both first and second cases involve a certain *degree of trust* of the prover to the Computer Algebra system; several experiments have been performed in these lines, see for instance the interaction between HOL and Maple [HT98] or the communication between Coq and GAP [KKL]. The last track (prove in the Theorem Prover the correctness of Computer Algebra algorithms) allows us to build more reliable and accurate components for a Computer Algebra system, for instance Buchberger's algorithm for computing Gröbner basis (one of the most important algorithms in Computer Algebra) has been formalized in [MPRR10] using the ACL2 theorem prover. In the work presented in this memoir, we have focussed on the third aspect.

The rest of this chapter is mainly organized in two parts devoted to present how the GAP Computer Algebra system and the ACL2 Theorem Prover were integrated in our system as new internal servers.

First of all, the integration and composability of the GAP Computer Algebra system in *fKenzo* is presented. Namely, first things first, to achieve the composability of GAP in our framework we need first its integration; then, Section 4.1 presents the integration of GAP. How Kenzo and GAP work in a coordinate and collaborative way is explained in Section 4.2.

In the second part, some explanations about how ACL2 is integrated in *fKenzo* are given. The integration of ACL2 in the system is shown in Section 4.3. The coordinate way of working of Kenzo, GAP and ACL2 is presented in Section 4.4.

Finally, Section 4.5 is devoted to present a methodology to integrate different systems as internal servers in our system.

4.1 Integration of the GAP Computer Algebra system

The second Computer Algebra system that we have integrated in our framework (the first one was Kenzo) is GAP [GAP] with its HAP package [Ell09]. This decision was taken inspired by the work presented in [RER09] where Kenzo and GAP (and, namely, its HAP package) have been communicated to create new tools. From now on, GAP/HAP refers to the GAP Computer Algebra system where its HAP package has been loaded.

In this first stage of the integration of GAP/HAP in our system, we have provided support for the construction of cyclic groups and the computation of their homology groups.

The rest of this section is organized as follows. Subsection 4.1.1 introduces the basic background about group homology; in Subsection 4.1.2 the GAP Computer Algebra system and its HAP package are presented; Subsection 4.1.3 explains how our framework is extended to include the GAP/HAP functionality. Moreover, an enhancement of the framework to deal with the properties of their objects is presented in Subsection 4.1.4. Finally the way of broadening the *fKenzo* GUI to include the GAP/HAP Computer Algebra system is detailed in Subsection 4.1.5.

4.1.1 Mathematical preliminaries

The following definitions and important results about homology of groups can be found in [Bro82].

Definition 4.1. Let G be a group and $\mathbb{Z}G$ be the *integral group ring* of G (see [Bro82]). A *resolution* F_* for a group G is an acyclic chain complex of $\mathbb{Z}G$-modules

$$\ldots \to F_2 \xrightarrow{d_2} F_1 \xrightarrow{d_1} F_0 \xrightarrow{\varepsilon} F_{-1} = \mathbb{Z} \to 0$$

where $F_{-1} = \mathbb{Z}$ is considered a $\mathbb{Z}G$-module with the trivial action and $F_i = 0$ for $i < -1$. The map $\varepsilon : F_0 \to F_{-1} = \mathbb{Z}$ is called the *augmentation.* If F_i is free for all $i \geq 0$, then F_* is said to be a free resolution.

Given a free resolution F_*, one can consider the chain complex of $\mathbb{Z}$-modules (that is to say, abelian groups) $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ defined by

$$C_n = (\mathbb{Z} \otimes_{\mathbb{Z}G} F_*)_n, \quad n \geq 0$$

(where $\mathbb{Z} \equiv C_*(\mathbb{Z}, 0)$ is the chain complex with only one non-null $\mathbb{Z}G$-module in dimension 0) with differential maps $d_{C_n} : C_n \to C_{n-1}$ induced by $d_n : F_n \to F_{n-1}$.

Although the chain complex of $\mathbb{Z}G$-modules F_* is acyclic, $C_* = \mathbb{Z} \otimes_{\mathbb{Z}G} F_*$ is, in general, not exact and its homology groups are thus not null. An important result in homology of groups claims that the homology groups are independent from the chosen resolution for G.

Theorem 4.2. Let G be a group and F_*, F'_* be two free resolutions of G. Then

$$H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F_*) \cong H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F'_*), \quad for \;\; all \;\; n \in \mathbb{N}.$$

This theorem leads to the following definition.

Definition 4.3. Given a group G, *the homology groups* $H_n(G)$ are defined as

$$H_n(G) = H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F_*)$$

where F_* is any free resolution for G.

The problem now consists of determining a free resolution F_* for G. For some particular cases, small resolutions can be directly constructed. For instance, let G be the cyclic group of order m with generator t. The resolution F_* for G

$$\ldots \xrightarrow{t-1} \mathbb{Z}G \xrightarrow{N} \mathbb{Z}G \xrightarrow{t-1} \mathbb{Z} \to 0,$$

where N denotes the norm element $1+t+\ldots+t^{m-1}$ of $\mathbb{Z}G$, produces the chain complex of abelian groups

$$\ldots \xrightarrow{0} \mathbb{Z} \xrightarrow{m} \mathbb{Z} \xrightarrow{0} \mathbb{Z} \to 0$$

and therefore

$$H_i(G) = \begin{cases} \mathbb{Z} & \texttt{if } i = 0 \\ \mathbb{Z}/m\mathbb{Z} & \texttt{if } i \texttt{ is odd}, i > 0 \\ 0 & \texttt{if } i \texttt{ is even}, i > 0 \end{cases}$$

In general is not easy to obtain a resolution for a group G. As we will see in Subsection 4.1.2, the GAP package HAP has been designed as a tool for constructing resolutions for a wide variety of groups.

4.1.2 GAP and HAP

GAP [GAP] is a Computer Algebra system, well-known for its contributions, in particular, in the area of Computational Group Theory.

HAP [Ell09] is a homological algebra library (developed by Graham Ellis) for use with GAP still under development. The initial focus of this package is on computations related to cohomology groups. A range of finite and infinite groups are handled, with particular emphasis on integral coefficients. It also contains some functions for the integral (co)homology of: Lie rings, Leibniz rings, cat-1-groups an digital topological spaces.

Let us see an example of the use of the GAP/HAP system. To construct the cyclic group of dimension 5 in the GAP system we proceed in the following way:

```
gap> c5:=CyclicGroup(5); ✠
<pc group of size 5 with 1 generators>
```

A GAP display must be read as follows. The initial `gap>` is the prompt of GAP. The user types out a gap statement, here `c5:=CyclicGroup(5);` and the maltese cross ✠ (in fact not visible on the user screen) marks in this text the end of the GAP statement. The Return key then asks GAP to *evaluate* the GAP statement. Here the cyclic group C_5 is constructed by the GAP `CyclicGroup` function (functions in GAP are case sensitive), taking account of the argument 5, and this cyclic group is *assigned* to the symbol `c5` for later use. Also evaluating a GAP statement *returns* an object, the result of the evaluation, in this case the cyclic group of dimension 5, displayed as `<pc group of size 5 with 1 generators>`.

From GAP we can obtain properties of the group; for instance, if we want to know if our group is abelian we proceed as follows:

```
gap> IsAbelian(c5); ✠
true
```

Therefore, the cyclic group C_5 is an abelian group. It is worth noting that this kind of properties (for instance, being abelian, cyclic, solvable and so on) are assigned to the group based on the mathematical definition of the group when it is constructed, this means that GAP does not perform any computation to know if the group satisfies a property but just checks if the object has associated it.

The homology groups of C_5 are computable by means of the HAP `GroupHomology` function (this function has as input two arguments, a group G and an integer n, and the output is $H_n(G)$) in the following way:

```
gap> GroupHomology(c5,0); ✠
[0]
```

To be interpreted as stating $H_0(C_5) = \mathbb{Z}$. We can compute several homology groups if we introduce the instructions in a loop; for instance, to compute $H_n(C_5)$ with $0 \leq n \leq 6$:

```
gap> for i in [0..6] do Print(GroupHomology(c5,i), " "); od; ✠
[ 0 ] [ 5 ] [  ] [ 5 ] [  ] [ 5 ] [  ]
```

In this case, the above result must be interpreted as $H_0(C_5) = \mathbb{Z}, H_1(C_5) = \mathbb{Z}/5\mathbb{Z}, H_2(C_5) = 0, H_3(C_5) = \mathbb{Z}/5\mathbb{Z}, H_4(C_5) = 0, H_5(C_5) = \mathbb{Z}/5\mathbb{Z}$ and $H_6(C_5) = 0$ (the expected results as we have seen in the previous subsection).

As we have just seen, HAP can be used to make basic calculations in the homology of finite and infinite groups with the command `GroupHomology`. This command performs two steps to compute the homology groups of a group G: (1) construct a free resolution F_* of G and (2) compute the homology from $\mathbb{Z} \otimes_{\mathbb{Z}G} F_*$ using a version of the Smith Normal Form algorithm [Veb31].

4.1.3 Integration of GAP/HAP

As we claimed in Section 2.1 one of the challenges of the *fKenzo* system was the integration of different tools as new internal servers. This is the first example of this integration.

We present here the integration of some of the functionality of the GAP/HAP system in *fKenzo*, namely the functionality devoted to construct cyclic groups, obtain group properties and compute group homology. The procedure followed here to integrate that functionality (a simple example) is general enough to be applied in the integration of other more interesting GAP/HAP functionality without any special hindrance. We have developed a plug-in following the guidelines given in Subsubsection 3.1.2. This new plug-in references the following resources:

```
<code id="gap">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> gap-invoker.lisp </data>
   <data format="Kf/microkernel"> gap-cyclic-m.lisp </data>
   <data format="Kf/microkernel"> gap-homology-m.lisp </data>
   <data format="Kf/adapter"> gap-a.lisp </data>
</code>
```

These resources deserve a detailed explanation that is provided in the following subsubsections.

4.1.3.1 Extending the XML-Kenzo schema

We want to introduce new functionality in our system which allows us to construct cyclic groups, compute their homology groups and obtain some of their properties by means of GAP/HAP. Therefore, we have extended the XML-Kenzo specification to represent

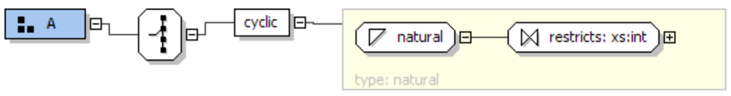


Figure 4.1: `cyclic` element in XML-Kenzo

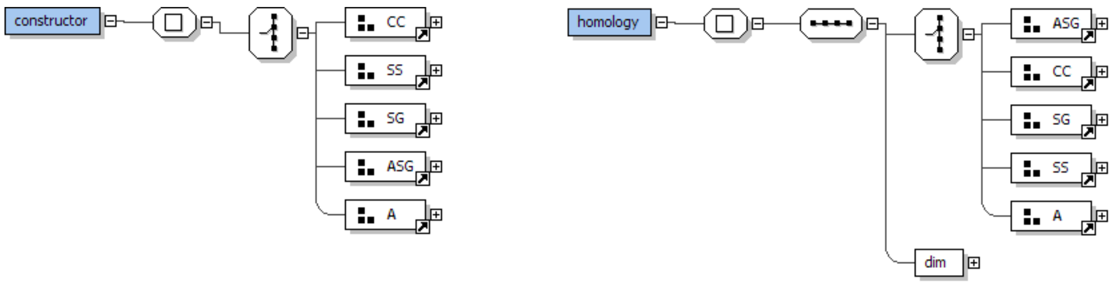


Figure 4.2: `constructor` and `homology` elements

the requests and results related to GAP/HAP. In the XML-Kenzo specification, we have defined a new element: `cyclic`, which has one child of natural number type (see Figure 4.1) and belongs to a new type called `A` (the type of GAP abelian groups).

Moreover, the group `A` has been included as a possible child of both `constructor` and `homology` elements, see Figure 4.2. Therefore, the two following requests are valid XML-Kenzo objects which can be processed.

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

```
<operation>
  <homology>
    <cyclic>7</cyclic>
    <dim>3</dim>
  </homology>
</operation>
```

This means that we can request the construction of cyclic groups and the computation of their homology groups in our system.

In addition, a result type element called `gap-id` which is used to return information about the properties of GAP groups has been defined. In particular this element has as value a natural number and has 9 attributes which represent respectively 9 properties (if the group is abelian, cyclic, elementary abelian, nilpotent, perfect, solvable, polycyclic, supersolvable and monomial) of the group by means of boolean values: `true` if the group satisfies the property and `false` otherwise (see Figure 4.3).

As we explained in Subsection 3.1.2, the external server evolves when the

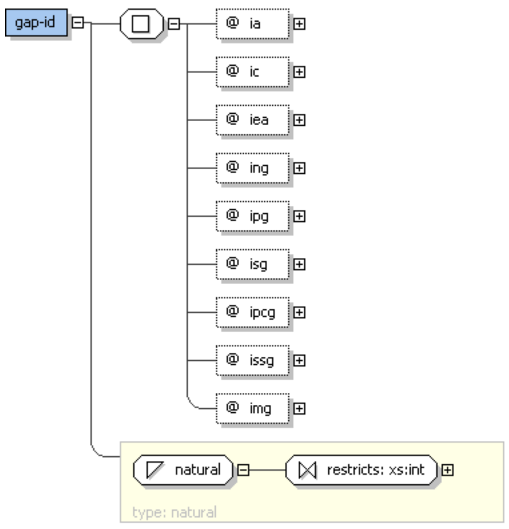


Figure 4.3: `gap-id` element in XML-Kenzo

`XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is upgraded with these new elements, the external server can validate requests such as the above ones.

4.1.3.2 A new internal server: GAP/HAP

Up to now, the Kenzo system was the unique computing kernel of the framework. Now, let us integrate the GAP/HAP system to construct groups and perform group homology computations.

GAP/HAP could be integrated locally as Kenzo was, but the installation of GAP/HAP is not so easy, and will need some interaction of the final user. Since, this option seems to us too uncomfortable, we devised the following organization.

4.1.3.2.1 A new internal server: GAP/HAP + SCSCP The connection with GAP/HAP server is available by means of SCSCP - the Symbolic Computation Software Composability Protocol [FHK+09]. SCSCP is a remote procedure call framework for computational algebra systems in which both data and protocol instructions are encoded in the OpenMath language [Con04]. This protocol has been successfully used to communicate several Computer Algebra systems as can be seen in [F+08]. GAP has a package, called SCSCP [KL09], which implements this protocol. This package has two main components: a server and a client; we are interested in the server part. The server component can be configured to supply the GAP procedures that can be invoked from different clients (which can be a GAP client with the SCSCP package, one of the SCSCP clients developed for Computer Algebra systems or in general a program with both *socket* support to invoke the GAP services and knowledge about the encoding of SCSCP OpenMath requests). In particular, our GAP/HAP server provides procedures to construct cyclic groups, to get some properties of the cyclic groups and to compute homology groups of cyclic groups.

The GAP/HAP server is available thanks to the SCSCP protocol without any additional development from our side; however, it has been necessary to deploy a client, that will be integrated in our framework. This client, from now on called *gap-invoker*, has been implemented as a Common Lisp program and has the functionality included in the `gap-invoker.lisp` file. This program has two parts: a Phrasebook and a *socket* client. The former component is able to transform from the internal XML-Kenzo representation to the SCSCP OpenMath representation and viceversa. It is worth noting that the Phrasebook included in this component is not the same Phrasebook included previously in the adapter (see Subsection 2.2.5), since the OpenMath requests that are sent/received to/from the GAP/HAP server are wrapped with additional information about the location of the server and necessary information for the SCSCP package. An example of this kind of requests is as follows:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Homology"/>
                <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
                <OMI>5</OMI>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

This request has the following parts: the beginning of a SCSCP request (`<?scscp start ?>`), the location of the server (in this case located in the server `esus.unirioja.es` in the port `7500`):

```
<OMATP>
   <OMS cd="scscp1" name="call_id"/>
   <OMSTR>esus.unirioja.es:7500</OMSTR>
</OMATP>
```

the invocation of the GAP/HAP service by means of:

```
<OMS cd="scscp1" name="procedure_call"/>
<OMA>
  <OMS cd="scscp_transient_1" name="Homology"/>
```

In this case the procedure associated with `Homology` is the `GroupHomology` HAP command; the arguments of the procedure `Homology`, in this case the cyclic group and the dimension:

```
<OMA>
  <OMS cd="group1" name="cyclic"/>
  <OMI>5</OMI>
</OMA>
<OMI>5</OMI>
```

and, finally, the end of the SCSCP request (`<?scscp end ?>`).

The *socket* client component of the gap-invoker is a bunch of functions in charge of sending requests and receiving results to/from the GAP/HAP server by means of *sockets* technology.

4.1.3.3 Cyclic groups construction module of the microkernel

The `gap-cyclic-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the microkernel in order to include a new module, called `cyclic-groups`.

When the microkernel receives a construction request where the child of the `constructor` element is `cyclic`, the `cyclic-groups` module of the microkernel is activated. For instance, if the microkernel receives the request:

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

the `cyclic-groups` module is activated.

When the `cyclic-groups` module is activated two situations are feasible: (1) a new group is created in the microkernel or (2) the object was previously built and its identification is simply returned. In the former case, this module constructs an object which represents a cyclic group. It is worth noting that no warnings are produced by the `cyclic-groups` module since all the restrictions (namely, this constructor only has associated the restriction of the type of its argument, which must be a natural number) about this constructor are handled in the XML-Kenzo specification, and, therefore, they are validated in the external server.

In Subsection 2.2.3, we have presented a representation for microkernel objects (`MK-OBJECT`), which was specialized for Kenzo spaces (`MK-SPACE-KENZO`). Now, we need a different specialization for cyclic groups constructed in the microkernel. Instead of defining a specialization just for cyclic groups we have decided to define a general representation for GAP groups, since in the future we could be interested in including support for the construction of other GAP groups. In the same line, in the future we could be interested in including not only groups, but also other GAP objects. As we said in Subsection 2.2.3 to include these new objects (which come from a different internal server) we specialized the `MK-OBJECT` class by means of a subclass, that is the class `MK-GAP`, whose definition is:

```
(DEFCLASS MK-GAP (MK-OBJECT) () )
```

In turn, we specialize this class to represent GAP groups.

```
(DEFCLASS MK-GROUP-GAP (MK-GAP)
     ;; Is Abelian
     (ia :type boolean :initarg :ia :reader ia)
     ;; Is Cyclic
     (ic :type boolean :initarg :ic :reader ic)
     ;; Is Elementary Abelian
     (iea :type boolean :initarg :iea :reader iea)
     ;; Is Nilpotent Group
     (ing :type boolean :initarg :ing :reader ing)
     ;; Is Perfect Group
     (ipg :type boolean :initarg :ipg :reader ipg)
     ;; Is Solvable Group
     (isg :type boolean :initarg :isg :reader isg)
     ;; Is PolyCyclic Group
     (ipcg :type boolean :initarg :ipcg :reader ipcg)
     ;; Is SuperSolvable Group
     (issg :type boolean :initarg :issg :reader issg)
     ;; Is Monomial Group
     (img :type boolean :initarg :img :reader img)
```

This class has nine slots in addition to the `idnm` and `orgn` slots of the `MK-OBJECT` class:

1. `ia`, a boolean, indicates if the group is abelian.
2. `ic`, a boolean, indicates if the group is cyclic.
3. `iea`, a boolean, indicates if the group is elementary abelian.
4. `ing`, a boolean, indicates if the group is nilpotent.
5. `ipg`, a boolean, indicates if the group is perfect.
6. `isg`, a boolean, indicates if the group is solvable.
7. `ipcg`, a boolean, indicates if the group is polycyclic.
8. `issg`, a boolean, indicates if the group is supersolvable.

9. `img`, a boolean, indicates if the group is monomial.

All the information included in the previous nine slots is obtained from GAP/HAP.

The procedure to construct cyclic group instances in the `cyclic-groups` module is very similar to the procedure followed in the construction of spaces presented in Subsubsection 2.2.3.3. However, in this case, this module does not need to check any additional restriction, since all the restrictions are imposed in the XML-Kenzo specification; therefore, all the requests that come from the external server related to cyclic group objects are always safe. The procedure to construct a cyclic group is as follows.

1. Search in the `*object-list*` list if the object was built previously.
   (a) If the object was built previously, return its identification number and the properties about the group in a `gap-id` XML-Kenzo object.
   (b) Otherwise, go to step 2.
2. Construct an instance of the `MK-GROUP-GAP` class where:
   - `idnm` is automatically generated (remember that `idnm` is the object identifier in the microkernel).
   - the XML-Kenzo object received as input is assigned to `orgn`.
   - the information of the rest of the slots is obtained invoking the GAP/HAP internal server.
3. Push the object in the `*object-list*` list of already created objects.
4. Return its identification number and the properties about the group in a `gap-id` XML-Kenzo object.

In this way, cyclic groups are constructed in the microkernel. It is worth noting that in spite of being constructed by different internal servers, all the objects are stored in the `*object-list*` list in the internal memory.

4.1.3.4 Enhancing the homology computation module

The `gap-homology-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the `homology` module of the microkernel in order to include the functionality to compute the homology groups of cyclic groups.

In Paragraph 2.2.3.4.1 the `homology` module of the microkernel was explained. The `homology` module implements a procedure in Common Lisp that allows us to compute homology groups through the microkernel. We have implemented that procedure in such a way that we do not need to overwrite the code of the procedure to allow the microkernel to deal with several internal servers.

Namely, we have used a very powerful tool of Common Lisp: the combination of generic functions and methods [Gra96]. When the class system and the functional organization of Common Lisp are considered, the notions of *generic functions* and *methods* are normally used. A *generic function* is a functional object whose behavior will depend on the class of its arguments; a generic function is defined by a `defgeneric` statement. The code for a generic function corresponding to a particular class of its arguments is a *method* object; each method is defined by a `defmethod` statement. This technique was used in the implementation of the class system of Kenzo [Ser01] and also in the homology computation module of the microkernel.

In particular, we have defined a generic function which corresponds with Step 5 of the procedure explained in Paragraph 2.2.3.4.1:

```
(DEFGENERIC compute-homology (mk-object n))
```

This generic function can have several methods to adapt the generic function to specific cases. In particular, up to now, it had associated just the method:

```
(DEFMETHOD compute-homology ((space mk-space-kenzo) n) ...)
```

This method invokes the Kenzo internal server from an `mk-space-kenzo`.

Now, we have defined a new class of objects; and in this case the system does not use Kenzo as kernel to compute the homology groups, but the GAP/HAP server. Then, the `gap-homology-m.lisp` file defines the method:

```
(DEFMETHOD compute-homology ((group mk-group-gap) n) ...)
```

which allows us to compute homology groups using the GAP/HAP server through the gap-invoker.

It is also worth noting that the instances of the class `mk-group-gap` do not have a `rede` slot which is used in the procedure explained in Paragraph 2.2.3.4.1 to check whether Kenzo could compute the homology groups or not. The way of managing this situation is based on the same idea: use generic functions and methods. Namely, we have defined the generic function which corresponds with the Step 4 of the procedure explained in Paragraph 2.2.3.4.1:

```
(DEFGENERIC check-constraints (mk-object))
```

In the case of dealing with `mk-space-kenzo` instances we have the method:

```
(DEFMETHOD check-constraints ((space mk-space-kenzo))
  (if (>= (rede space) 0) t nil))
```

On the contrary, the method implemented for the case of `mk-group-gap` instances is the following simple method:

```
(DEFMETHOD check-constraints ((group mk-group-gap)) t)
```

In this way, the homology module of the microkernel allows us to use GAP/HAP to perform computations. To sum up, homology groups of spaces are computed with Kenzo and homology groups of groups are computed with GAP. In general, we can use different methods to compute the homology groups of different classes of objects without modifying the main procedure.

4.1.3.5 Increasing the functionality of the adapter

Cyclic groups are objects already defined in the OpenMath language, namely in the `groupname1` Content Dictionary. This Content Dictionary is added to the list of Content Dictionaries which can be processed in the adapter. Moreover, we have defined a `gap-id` object, in the `Aux` Content Dictionary, to return the identification and the information related to a group.

In addition, we have extend the Phrasebook by means of new parsers which are able to convert from XML-Kenzo objects related to GAP/HAP to OpenMath objects and viceversa. For instance, the XML-Kenzo request:

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
     <OMS cd="groupname1" name="cyclic"/>
     <OMI>5</OMI>
   </OMA>
</OMOBJ>
```

Therefore, the `gap-a.lisp` file contains the new parsers and a list with both `groupname1` and `Auxiliar` Content Dictionaries in order to raise the functionality of the adapter to be able to convert from the new OpenMath requests, devoted to GAP

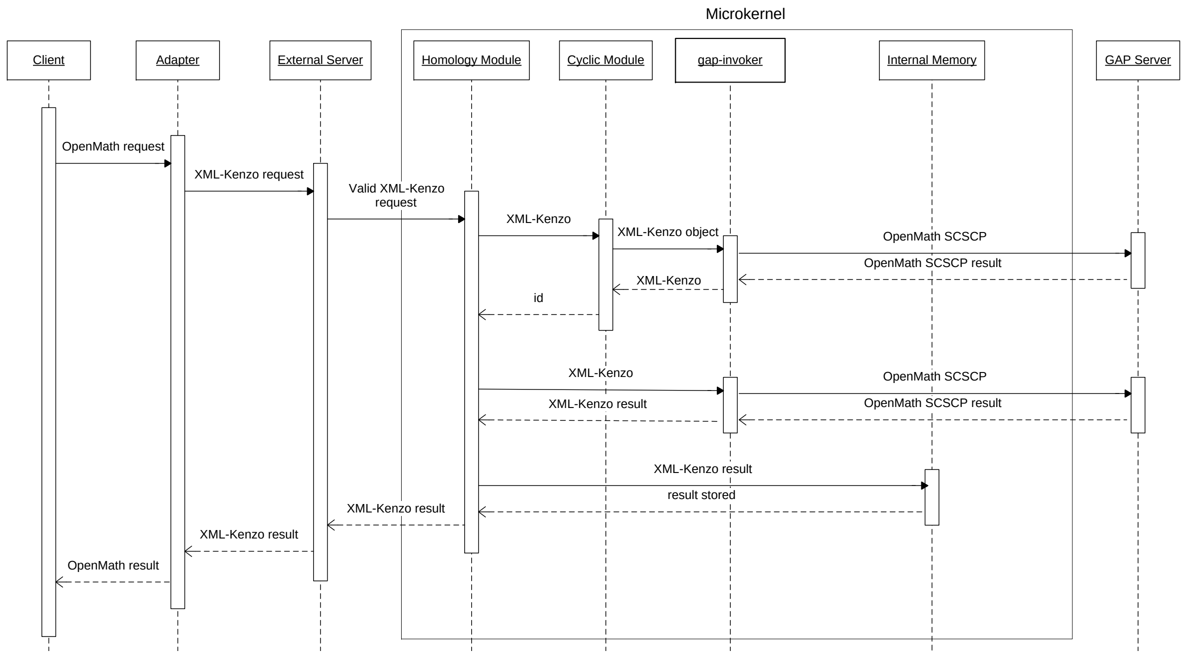


Figure 4.4: UML sequence diagram

operations, to XML-Kenzo requests.

4.1.3.6 Execution flow

To provide a better understanding of the integration of the GAP/HAP system in our framework, let us present an execution scenario where a client asks our framework for computing $H_5(C_5)$ in a fresh session, that is, neither objects were constructed or computations were performed previously. The execution flow of this scenario is depicted in Figure 4.4 with a UML-like sequence diagram.

The OpenMath representation of the request $H_5(C_5)$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homology"/>
        <OMA>
            <OMS cd="groupname1" name="cyclic"/>
            <OMI>5</OMI>
        </OMA>
        <OMI>5</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath request from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the

OpenMath object into the following XML-Kenzo object:

```
<operation>
    <homology>
        <cyclic>5</cyclic>
        <dim>5</dim>
    </homology>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification, in this case as the root element is `operation`, then it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homology` element has two children. ✓
3. The first child of the `homology` element belongs to one of the groups `A`, `CC`, `SS`, `SG` or `ASG`. ✓
4. The value of the `cyclic` element is a natural number. ✓
5. The second child of the `homology` element is the `dim` element. ✓
6. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, we have a valid request that is sent to the microkernel. In the microkernel the `homology` module is activated. When the `homology` module is activated, the procedure explained in Paragraph 2.2.3.4.1 with the extension explained in Subsubsection 4.1.3.4 is executed. First, the `homology` module searches in `*object-list*` list if the cyclic group of dimension 5 was constructed previously; as this object was not constructed, the `cyclic` module is invoked to construct it, this module in turn invokes the GAP/HAP server through the gap-invoker to construct a `mk-group-gap` instance. This instance is stored in the `*object-list*` list to avoid the duplication of elements.

Subsequently, once the object is constructed, the `homology` module sends the XML-Kenzo request to the gap-invoker in order to send a request to the GAP/HAP server to compute the group homology. The request sent to the GAP/HAP server by the gap-invoker is:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Homology"/>
                <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
                <OMI>5</OMI>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

It is worth noting that the above OpenMath request is a bit different from the original one received by the adapter. Namely, it includes SCSCP enhancements.

When the GAP/HAP server receives the above request, it executes the following instruction:

```
gap> GroupHomology(CyclicGroup(5),5); ✠
[5]
```

The result returned by the GAP/HAP server is:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name ="call_id"/>
            <OMSTR> esus.unirioja.es:7500 </OMSTR>
        </OMATP>
        <OMA><OMS cd="scscp1" name ="procedure_completed"/>
            <OMA><OMS cd="list1" name ="list"/><OMI>5</OMI></OMA>
        </OMA>
        </OMATTR>
</OMOBJ>
<?scscp end ?>
```

This result is converted by the gap-invoker into the following XML-Kenzo result.

```
<result>
    <component>5</component>
</result>
```

This result is stored in the internal memory to avoid re-computations by the `homology` module and is sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>5</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client. It is worth noting that the OpenMath representation of the result returned by the GAP/HAP server is different from the representation of the result returned by the adapter. This is due to the fact that the GAP/HAP server uses the SCSCP representation; but we consider that is better to keep a consistent representation for all the results returned by our framework for the computation of homology groups.

It is worth noting that from the client point of view, the internal server employed to perform the computations is transparent.

4.1.4 Properties of objects

As we have explained in the previous subsection, when a (cyclic) group is constructed in our framework not only its identification number is returned but also some properties of that group. This additional information can be helpful, in a client, for instance to allow a student to know some properties of the object.

Then, we realized that in the same way that some properties are associated with groups, we can do the same with spaces. However, there is an important difference, group properties are obtained from the GAP/HAP server, whereas the space properties will be obtained from the knowledge included in the microkernel and not from Kenzo. To include, this improvement in our system, the following small plug-in has been developed.

```
<code id="spaces-properties">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/microkernel"> properties-mk.lisp </data>
   <data format="Kf/adapter"> properties-a.lisp </data>
</code>
```

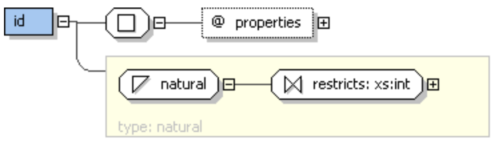


Figure 4.5: `id` element in XML-Kenzo

These resources deserve a detailed explanation that is provided in the following paragraphs.

First of all, we want to modify the `id` XML-Kenzo object to not only store an identification number but also properties about the object which has associated the identifier. Therefore, we have extended the XML-Kenzo specification (XML-Kenzo.xsd file) to admit the new definition of `id` which includes an attribute called `properties`, see Figure 4.5.

The `properties-mk.lisp` file modifies the behavior of the following construction modules: `Sphere`, `Delta`, `K-Z`, `K-Z2`, `Cartesian Product`, `Suspension`, `Classifying Space` and `Loop Space`. That is to say, the modules which are used by the HES, because they provide additional information which can be interesting for a user. Then, the `id` XML-Kenzo objects returned by these modules not only contain the identifiers but also some properties. For instance, when the `Cartesian Product` module is activated and both components of the Cartesian product are contractible spaces, the returned object is:

```
<id properties="The space is contractible because is the cartesian product of two
          contractible spaces"> 1 </id>
```

It is worth noting that the rest of the modules of the microkernel return `id` XML-Kenzo objects whose `properties` attribute is empty.

Finally, the `properties-a.lisp` file includes a new parser in the Phrasebook to be able to handle the new specification of `id` XML-Kenzo objects.

4.1.5 Integration of GAP/HAP in the *fKenzo* GUI

Throughout this section, a plug-in that allows us to include in our system the functionality related to GAP/HAP has been presented. Now, the necessary resources to extend the *fKenzo* GUI to provide support for the new functionality are explained.

In this case we have defined a fresh *fKenzo* module to enhance the GUI with support for the GAP/HAP system. The new OMDoc module references three files: `gap-structure` (that defines the structure of the graphical constituents), `gap-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection.

We have defined three graphical elements, using the XUL specification language, in

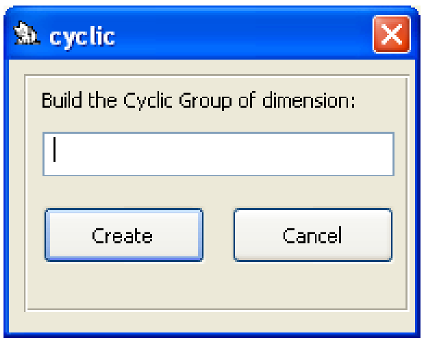


Figure 4.6: Cyclic window

the `gap-structure` file:

- A menu called `GAP` which contains one option: `Cyclic Group`.
- A window called `Cyclic` (see Figure 4.6). It is worth noting that it is not necessary to specify this window from scratch since it has the same structure that, for instance, the `Sphere` window of the Simplicial Set module. So, we use a generic specification with the concrete values of the new window.

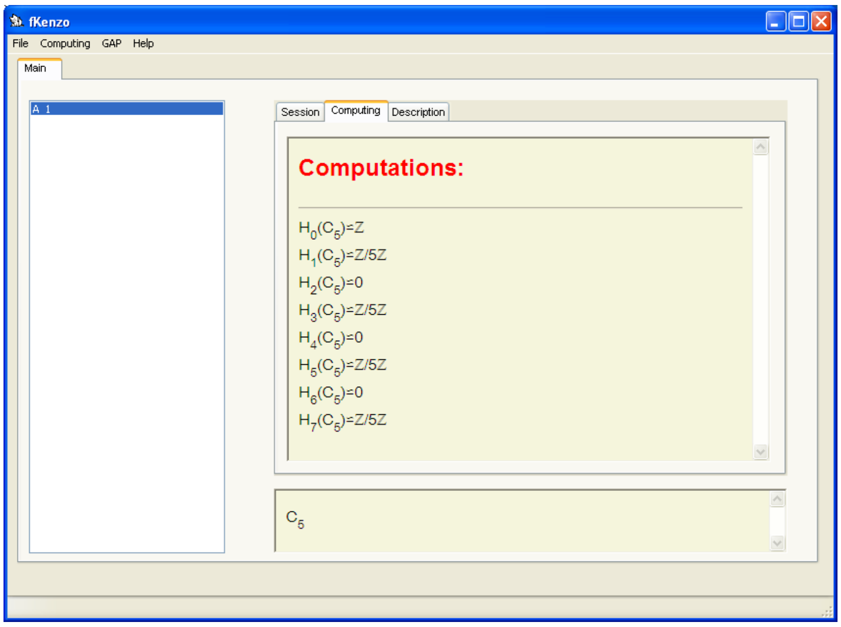
The functionality stored in the `gap-functionality` document related to these components works as follows. A function acting as event handler is associated with the `Cyclic Group` menu option; this function shows a `Cyclic` window (see Figure 4.6). From the `Cyclic` window, the user must introduce the dimension of the cyclic group that must be a natural number. Once the user has introduced the dimension n of the cyclic group, when he presses the `Create` button of the `Cyclic` window, an OpenMath request is generated and sent to the framework.

Then, the cyclic group C_n is built and its identification number in a `gap-id` OpenMath object is returned. Eventually *fKenzo* adds to the constructed objects list (situated in the left side of the main tab of the *fKenzo* GUI) the new group. To identify the (Abelian) groups in *fKenzo* we use the letter A and its identification number.

Finally, if the `Computing` *fKenzo* module is loaded (or has been previously loaded) the user can ask *fKenzo* to compute the homology groups of a group using the `Homology` option of the `Computing` menu, the results are shown, as usual, in the Computing tab. Figure 4.7 shows the computation of the homology groups of C_5.

It is worth noting that from the user point of view the computation of homology groups of both spaces and groups has no differences, since he proceeds in both cases in the same way. Therefore the system used in each moment is transparent to the user, providing access to different software systems in an easy and comfortable way.

In addition to the graphical elements presented about GAP/HAP, a new constituent has been added to the GUI to handle *properties*. Namely, a new tab called `Description`



Computations:

$H_0(C_5)=Z$
$H_1(C_5)=Z/5Z$
$H_2(C_5)=0$
$H_3(C_5)=Z/5Z$
$H_4(C_5)=0$
$H_5(C_5)=Z/5Z$
$H_6(C_5)=0$
$H_7(C_5)=Z/5Z$

C_5

Figure 4.7: Homology groups of C_5

is included in the central panel. This new graphical element is included whenever a construction module or the GAP/HAP module is loaded in *fKenzo*. The functionality associated with this new element is as follows.

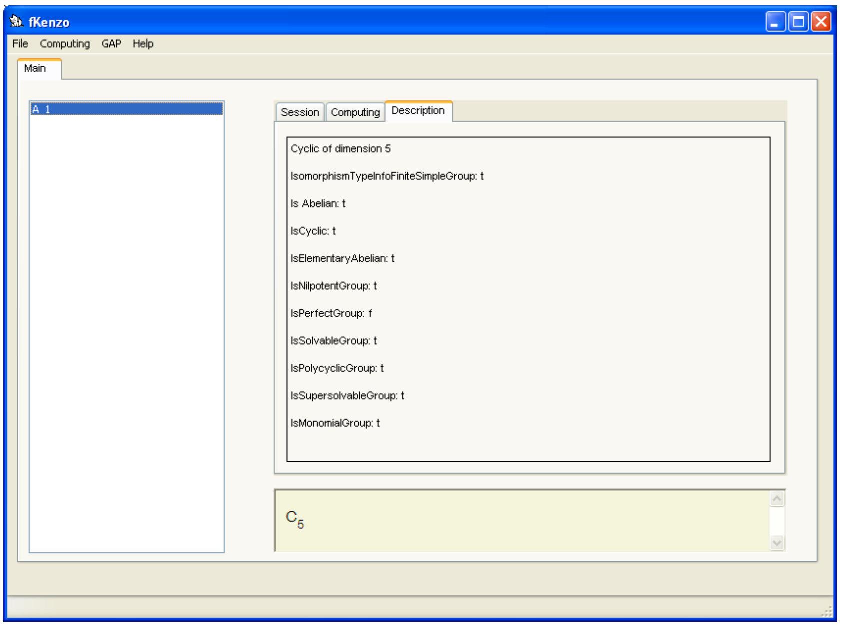
As we explained in Subsection 3.2.1 when an object is selected from the list of constructed spaces, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. This behavior is kept but, in addition, if some additional information is accessible when an object is constructed, then, this information is shown in the `Description` tab. For instance, when the cyclic group of dimension 5 is selected, the `Description` tab shows the properties about the group included in the `gap-id` OpenMath object returned, see Figure 4.8. The same happens when an `id` OpenMath object contains some properties of the object.

The next subsubsection is devoted to present how objects are handled in the *fKenzo* GUI.

4.1.5.1 Management of the objects in the *fKenzo* GUI

Let us present how the objects are managed in the *fKenzo* GUI.

The representation of objects in the *fKenzo* GUI has been inspired by the representation of objects in Kenzo. An object of the *fKenzo* GUI is implemented as an instance of a CLOS class (let us remember that the *fKenzo* GUI is implemented in Common Lisp), the class `FKENZO-OBJECT`, whose definition is:



Cyclic of dimension 5

IsomorphismTypeInfoFiniteSimpleGroup: t

Is Abelian: t

IsCyclic: t

IsElementaryAbelian: t

IsNilpotentGroup: t

IsPerfectGroup: f

IsSolvableGroup: t

IsPolycyclicGroup: t

IsSupersolvableGroup: t

IsMonomialGroup: t

C_5

Figure 4.8: Description tab for GAP groups

```
(DEFCLASS FKENZO-OBJECT ()
     ;; IDentification NuMber
     (idnm :type fixnum :initarg :idnm :reader idnm)
     ;; ORiGiN
     (orgn :type string :initarg :orgn :reader orgn))
     ;; PROPertieS
     (props :type string :initarg :props :reader props)))
```

This class has three slots:

1. `idnm`, an integer, identifier for the object in *fKenzo*. This is the value assigned when a space is constructed.

2. `orgn`, a string containing the OpenMath object that is the *origin* of the object.

3. `props`, a string containing the properties obtained in the construction of the object or the empty string if no property was returned.

The objects are stored in a list called `*FKENZO-OBJECTS*` which is used to generate the shown list to the user in the left side of the *fKenzo* GUI.

Moreover, there is a function, called `fK`, which allows us to get information about the n-th *fKenzo* GUI object. The `fK` function takes as argument a natural number n and returns the n-th *fKenzo* GUI object.

The functionality of the event handler of the list of the left side, when an object is selected from it, works as follows. From the selected object, for instance “SS 3”, extracts its identification number, in this case 3, and obtains the `FKENZO-OBJECT` by means of the `fK` function. Subsequently, it invokes a Common Lisp method [Gra96] associated with that `FKENZO-OBJECT` which shows the mathematical notation of the object in the bottom part of the right side of the *fKenzo* GUI. Moreover, if the value of the `props` slot is not empty, then, the system moves dynamically from the current selected tab to the `Description` tab showing the properties of the object.

It is worth noting that the approach followed to store the properties of an object, just using a string, is likely too simple, and several improvements can be consider, for instance, replace the `props` string with an association with different classes. However, that remains as further work.

Moreover, due to the fact that objects of very different nature can exists in *fKenzo*, we have specialized the `FKENZO-OBJECT` with two different subclasses. On the one hand, we have defined a new subclass of the `FKENZO-OBJECT` class; to store a name given by the user to identify the object. Namely, the new class, `FKENZO-OBJECT-NAME`, whose definition is:

```
(DEFCLASS FKENZO-OBJECT-NAME (FKENZO-OBJECT)
     ;; NAME
     (name :type string :initarg :name :reader name)
```

This is a subclass of the `FKENZO-OBJECT` class with just one additional slot, `name`, which provides a name for the object. The `FKENZO-OBJECT-NAME` instances will be useful to store objects which do not have a concrete mathematical representation but which include a name to identify them in *fKenzo*, for instance a simplicial set build from a list of its elements.

On the other hand, we have defined a new subclass of the `FKENZO-OBJECT` class; to store the path of a file associated with the object. Namely, the new class, `FKENZO-OBJECT-FILE`, whose definition is:

```
(DEFCLASS FKENZO-OBJECT-FILE (FKENZO-OBJECT)
     ;; FILE
     (file :type string :initarg :file :reader idnm)
```

This is a subclass of the `FKENZO-OBJECT` class with just one additional slot, `file`, which provides the path of a file associated with the object. In turn this class can be specialized depending on the type of the file which is associated with the object.

4.1.5.2 Behavior of the objects in the *fKenzo* GUI

It is worth noting that in spite of belonging to different classes, all the instances constructed in *fKenzo* are stored in the `*FKENZO-OBJECTS*`, used to show the list objects in the left side of the *fKenzo* GUI. However, when we select an object in the left list of the GUI, the behavior of the interface depends on the class associated with the selected object. To deal with this question we have used the *Strategy* pattern [GJ94] which is implemented in Common Lisp by means of generic functions and methods.

In particular, we have defined the generic function to show information in the *fKenzo* GUI about the selected object:

```
(DEFGENERIC show-object (object))
```

This generic function have several methods to adapt it to specific cases. The main method is associated with the `FKENZO-OBJECT` class:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT))
   (show-mathematical-notation (orgn object))
   (if (props object) (show-description (props object))))
```

This method shows the mathematical representation of `object` in the bottom part of the right side of the *fKenzo* GUI by means of the `show-mathematical-notation` function which takes as argument the `orgn` slot of the `FKENZO-OBJECT` instance. Moreover, if some additional information is stored in the `props` slot of the object, then, this information is shown in the `Description` tab.

When the object associated with the selection of the left list of the GUI belongs to the class `FKENZO-OBJECT-NAME`, the behavior of the GUI is different since we have defined this new method:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-NAME))
   (show-name (name object))
   (if (props object) (show-description (props object))))
```

which instead of showing the mathematical representation of the object shows the name given by the user. In addition, each specialization of the `FKENZO-OBJECT-FILE` will have a concrete method to specify its behavior. In this way, the behavior of the left list of the GUI is modified without touching the main code.

We foresee the definition of new specializations in the future, but we think that approach presented here is general enough to be extrapolated to other cases.

4.2 Interoperability between Kenzo and GAP/HAP

As we said at the beginning of this chapter, the integration of several tools in *fKenzo* was a means and not the end. The final goal of integrating different tools, as internal servers, in the framework (and in *fKenzo*, too) consists of achieving a composability between them to produce results not reachable if the tools worked in an isolated way.

The first case study is the integration of the two Computer Algebra systems included in our system (Kenzo and GAP/HAP). To achieve the goal of the composability of Kenzo and GAP/HAP we have been inspired by the work presented in [RER09]. In that work, Kenzo and GAP/HAP were manually communicated by means of OpenMath objects in order to compute homology of groups of Eilenberg MacLane spaces of type $K(\pi, 1)$. In addition, this integration allowed the authors to develop new tools to compute more algebraic invariants, such as homology groups of $K(\pi, n)$'s of certain 2-types or of central extensions, as can be seen in [Rom10].

However, the approach followed in that paper to connect Kenzo and GAP/HAP had some drawbacks that will be explained in the next subsection. Then, we have undertaken the task of improving the cooperation between these systems thanks to our framework.

The interoperability between Kenzo and GAP/HAP is translated into the feasibility of constructing Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, thanks to the combination of Kenzo and GAP/HAP; and subsequently use these spaces as any other space of the system (that means, use them for constructing other spaces and computing their homology and homotopy groups).

The rest of this section is organized as follows. Subsection 4.2.1 is devoted to provide an overview of the method to integrate Kenzo and GAP/HAP used in [RER09]; in addition, the necessary mathematical background is also explained there. The composability of Kenzo and GAP/HAP in our framework is explained in Subsection 4.2.2. Finally, the improvements added to the *fKenzo* GUI are presented in Subsection 4.2.3.

4.2.1 Mathematical preliminaries

In Subsection 4.1.1, we explained that is enough to determine a free resolution F_* of a group G to compute its homology groups. One approach consists of considering the bar resolution $B_* = Bar_*(G)$ (explained, for instance, in [Mac63] and which can be always constructed) whose associated chain complex $\mathbb{Z} \otimes_{\mathbb{Z}G} B_*$ can be viewed as the chain complex of the Eilenberg MacLane space $K(G, 1)$. The homology groups of $K(G, 1)$ are those of the group G and this space has a big structural richness. But it has a serious drawback: its size. If $n > 1$, then $K(G, 1)_n = G^n$. In particular, if $G = \mathbb{Z}$, the space $K(G, 1)$ is infinite. This fact is an important obstacle to use $K(G, 1)$ as a means for computing the homology groups of G.

However, the effective homology technique (see Subsection 1.1.3) and the Kenzo program could have a role in the computation of the homology of a group G, since, as we have seen in Section 1.2.2, Kenzo implements Eilenberg MacLane spaces $K(G, n)$ for every n but only for $G = \mathbb{Z}$ and $G = \mathbb{Z}/2\mathbb{Z}$. To our end, we need the Eilenberg MacLane space $K(G, 1)$, for other groups G. The size of this space makes it difficult to calculate the groups in a direct way, but it is possible to operate with this simplicial set making use of the effective homology technique: if we construct the effective homology of $K(G, 1)$ then we would be able to compute the homology groups of $K(G, 1)$, which are those of G. Furthermore, it should be possible to extend many group theoretic constructions to effective homology constructions of Eilenberg MacLane spaces. We thus introduce the following definition.

Definition 4.4. A group G is a group with effective homology if $K(G, 1)$ is a simplicial set with effective homology.

The problem is, given a group G, how can we determine the effective homology of $K(G, 1)$? If the group G is finite, the simplicial set $K(G, 1)$ is effective too, so that it has trivially effective homology. However, the enormous size of this space makes it difficult to obtain real calculations, and therefore we will try to obtain an equivalence $C_*(K(G, 1)) \Longleftarrow\!\!\!\Longrightarrow E_*$ where E_* is an effective and (much) smaller chain complex than the initial chain complex.

In [RER09] the algorithm that computes this equivalence from a resolution of G was explained. Here, we just state the algorithm.

Algorithm 4.5 ([RER09])**.**
Input: a group G and a free resolution F_* of finite type.
Output: the effective homology of $K(G, 1)$, that is, an equivalence $C_*(K(G, 1)) \Longleftarrow\!\!\!\Longrightarrow E_*$ where E_* is an effective chain complex.

This algorithm was implemented, by the authors of [RER09], in Common Lisp enhancing the Kenzo system. The free resolution of the group G is obtained from the GAP/HAP system. Particulary to the case where G is a cyclic group, the process to construct the space $K(G, 1)$, in Kenzo, can be summed up as follows:

1. Load the necessary packages and files in GAP and Kenzo,
2. build the cyclic group G in GAP,
3. build a resolution of the cyclic group G using the HAP package,
4. export from GAP the resolution into a file using the OpenMath format,
5. import the resolution to Kenzo,
6. build the cyclic group G in Kenzo (thanks to a new Kenzo module developed in [RER09]),
7. assign the resolution to the corresponding cyclic group G in Kenzo,
8. build the space $K(G, 1)$ where G is the cyclic group in Kenzo.

This approach has some drawbacks. First of all, the user must install several programs and packages: GAP, its HAP package, the OpenMath package for GAP [SC09], an extension for this OpenMath package developed in [RER09], the Kenzo system and the new module developed in [RER09]. In addition, of course, the user must know how to mix all the ingredients in order to obtain the desired result. Moreover, some of the steps could be performed automatically by a computer program; for instance, the importation/exportation of the resolution from GAP to Kenzo.

Therefore, we undertook the task of integrating this composability of systems in *fKenzo* but overcoming the drawbacks and hiding to the final user the composability details.

4.2.2 Composability of Kenzo and GAP/HAP

We have modified the plug-in used to integrate GAP in our framework to include the functionality related to the construction of Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group. The resources included in that plug-in are:

```
<code id="gap">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> gap-kenzo.lisp </data>
   <data format="Kf/internal-server"> gap-invoker.lisp </data>
   <data format="Kf/microkernel"> gap-cyclic-m.lisp </data>
   <data format="Kf/microkernel"> gap-homology-m.lisp </data>
   <data format="Kf/microkernel"> gap-k-g-1-m.lisp </data>
   <data format="Kf/microkernel"> homotopy-extension-m.lisp </data>
   <data format="Kf/adapter"> gap-a.lisp </data>
</code>
```

It is worth noting that some new resources have been included to the original GAP plug-in (in particular the files `gap-kenzo.lisp`, `gap-k-g-1-m.lisp` and `homotopy-extension-m.lisp`), others have been modified to allow the integration of the new tools (`XML-Kenzo.xsd`, `gap-invoker.lisp` and `gap-a.lisp`) and other files remain untouched.

The modification of the resources and the new resources are going to be explained in the following subsubsections.

4.2.2.1 New elements of the XML-Kenzo schema

We want to introduce a new kind of objects in our system (Eilenberg MacLane spaces of type $K(G, 1)$ where G is a group); then, it is necessary to provide a representation for those objects in our framework. Therefore, we have extended the XML-Kenzo specification to admit the new objects related to Eilenberg MacLane spaces of type $K(G, 1)$, where G is a group. In the specification, we have defined a new element: `k-g-1`, which

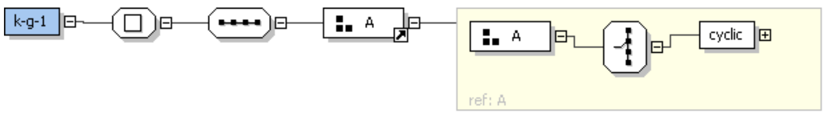


Figure 4.9: `k-g-1` element in XML-Kenzo

has one child, that is, an element, of the group `A` (the type of the GAP abelian groups in our specification, at this moment this group just has the `cyclic` element but we foresee the inclusion of new groups), see Figure 4.9. In addition, the element `k-g-1` belongs to the `ASG` group (the type of Abelian Simplicial Groups); and therefore, this element can be used as any other element of the `ASG` group, that is to say, we can construct requests to compute its homology and homotopy groups and use it for constructing other spaces.

Therefore, the following request is a valid XML-Kenzo object.

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

As we explained in Subsection 3.1.2, the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified the external server can validate requests such as the above one.

4.2.2.2 Enhancements of Kenzo internal server and GAP/HAP server

The Kenzo internal server, the GAP/HAP server and the gap-invoker have been modified in order to integrate the composability of Kenzo and GAP/HAP.

The GAP/HAP server presented in Subsubsection 4.1.3.2.1 supplied services to construct cyclic groups, to obtain properties of them and to compute their homology groups using the GAP Computer Algebra system and its HAP package. Now, we have included a new service which allows the construction of a resolution of a (cyclic) group using the HAP package. Then, a client can invoke this new service. For instance, when the GAP/HAP server receives the following request:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
              <OMS cd="scscp_transient_1" name="Resolution"/>
              <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

which asks the construction of a resolution for the cyclic group of dimension 5, the GAP server executes the instruction:

```
gap> ResolutionFiniteGroup(CyclicGroup(5)); ✠
Resolution in characteristic 0 for <pc group of size 5 with 1 generators>
```

Subsequently, the GAP server transforms the result to its OpenMath format.

```
<OMOBJ>
 <OMA>
  <OMS cd="resolutions" name="resolution"/>
  <!-- GROUP -->
  <OMA>
   <OMS cd="group1" name="cyclic_group"/>
   <OMI>5</OMI>
  </OMA>
  <!-- More than 1000 lines skipped -->
  ...
</OMOBJ>
```

The OpenMath format of resolutions was explained in [RER09].

The gap-invoker (presented in Subsubsection 4.1.3.2.1), which was able to invoke the GAP/HAP server in order to construct cyclic groups, obtain their properties and compute their homology groups has been upgraded in the `gap-invoker.lisp` file in order to be able to request resolutions to the GAP/HAP server. When this program invokes the GAP/HAP server asking for a resolution of a group, the result returned by this program is the resolution obtained from the GAP/HAP server keeping its OpenMath format. We decided to keep resolutions with their OpenMath format instead of transforming them to an XML-Kenzo representation. The main reason was due to the fact that we have re-used the programs implemented in [RER09] that work with resolutions based on the OpenMath format. Therefore, we considered that converting from an OpenMath resolution to an XML-Kenzo resolution and subsequently performing the inverse transformation to obtain the original OpenMath resolution was unnecessary. Besides, the idea of using XML-Kenzo to check the correctness of the resolutions against some rules (as we have done with the rest of the objects of the system) was rejected because we completely trust in the resolutions returned by the gap-invoker. This means (if connection problems with the GAP/HAP server do not appear in which case the system manages the situation) that the returned resolutions are always safe since they are not manually produced, but automatically generated by a program following the rules which ensure their correctness.

Finally, the functionality of the Kenzo kernel is increased by means of the `gap-kenzo.lisp` file which loads in the Kenzo system the functionality implemented in [RER09] to construct Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group. The functionality implemented in [RER09] includes the functions to construct cyclic groups, to transform an OpenMath resolution into a Common Lisp functional object and, eventually, to construct Eilenberg MacLane spaces of type $K(G, 1)$ in the Kenzo system.

Besides, the functionality of the Kenzo internal server is increased allowing the invocation of the service which allows the construction of new Eilenberg MacLane spaces. Namely, this service takes as argument an XML-Kenzo object which represents the Eilenberg MacLane space of a group G and an OpenMath resolution of G obtained from GAP. When this new service is activated, it extracts the cyclic group encoded in the XML-Kenzo object and constructs a Kenzo object which represents that cyclic group. Subsequently, the OpenMath resolution is codified as a Common Lisp functional object and assigned to the cyclic group. Afterwards, the space $K(G, 1)$, where G is the cyclic group previously constructed in Kenzo, is built. As a result, an instance of the `Abelian-Simplicial-Group` Kenzo class is obtained and its identification number is returned as a result, in an `id` XML-Kenzo object, by the Kenzo internal server.

4.2.2.3 Increasing the functionality of the microkernel

The `gap-k-g-1-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the microkernel in order to include a new construction module, called `k-g-1`, to construct Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group.

When the microkernel receives a construction request where the child of the `constructor` element is `k-g-1`, the `k-g-1` module of the microkernel is activated. For instance, if the microkernel receives the request:

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

the `k-g-1` module is activated.

When the `k-g-1` module is activated two situations are feasible: (1) a new space is created in the microkernel or (2) the object was previously built and its identification number is simply returned.

The procedure to construct Eilenberg MacLane spaces of type $K(G,1)$, where G is a cyclic group, in the `k-g-1` module is very similar to the procedure followed in the construction of spaces presented in Subsubsection 2.2.3.3. However, in this case, some additional steps are needed.

1. Search in the `*object-list*` list if the object was built previously.
   (a) If the object was built previously its identification number, in an `id` XML-Kenzo object, is returned.
   (b) Otherwise, go to step 2.
2. Extract the XML-Kenzo object which represents the cyclic group of the `k-g-1` XML-Kenzo object.
3. Activate the gap-invoker to obtain a resolution associated with the XML-Kenzo object which represents the cyclic group obtained in the previous step.
4. Invoke the service of the Kenzo internal server which allow the construction of Eilenberg MacLane space of a group G with the `k-g-1` XML-Kenzo object and the resolution obtained in the previous step as arguments.
5. Construct an instance of the `mk-space-k-g` class (see Subsubsection 2.2.3.1) where:
   - the value of the slot `rede` is 0,
   - `idnm` is automatically generated,
   - `kidnm` is the value returned by the Kenzo internal server in the previous step,
   - the XML-Kenzo object received as input is assigned to `orgn`,
   - the value of the slot `iter` is 1, and
   - the value of the slot `group` is the dimension of the cyclic group.
6. Push the object in the `*object-list*` list of already created spaces.
7. Return the `idnm` of the object in an `id` XML-Kenzo object.

In this way, Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, are constructed in the microkernel. It is worth noting that the above procedure does not need to check any extra constraint for these spaces, since the restriction (namely, the constraint is the correct type of the argument of this constructor) about these Eilenberg MacLane spaces is validated in the external server thanks to the XML-Kenzo specification.

As we have explained previously, Eilenberg MacLane spaces play a key role in the computation of homotopy groups of spaces in our system (see Paragraph 2.2.3.2.2). Then, the `homotopy-extension-m.lisp` file extends the procedure implemented in the HAM (see Paragraph 2.2.3.2.2). The procedure implemented in that module only allowed the computation of homotopy groups if the first non null homology group of the space was $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$; using the new functions implemented in [RER09] and included in the Kenzo internal server, we can compute homotopy groups of spaces whose first non null homology group is a cyclic group using the algorithm implemented in the HAM and the new functionality related to Eilenberg MacLane spaces.

4.2.2.4 Increasing the functionality of the adapter

Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, have been defined in the `ASG` OpenMath Content Dictionary. Then, we have upgraded the functionality of `gap-a.lisp` (which was presented in Subsubsection 4.1.3.5) to raise the functionality of the adapter in order to be able to convert from the new OpenMath requests, devoted to the construction of Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, to XML-Kenzo requests. Namely, we have extend the Phrasebook by means of a new parser in charge of that task, then, the following XML-Kenzo request:

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
     <OMS cd="ASG" name="k-g-1"/>
       <OMA>
         <OMS cd="groupname1" name="cyclic"/>
         <OMI>5</OMI>
       </OMA>
   </OMA>
</OMOBJ>
```

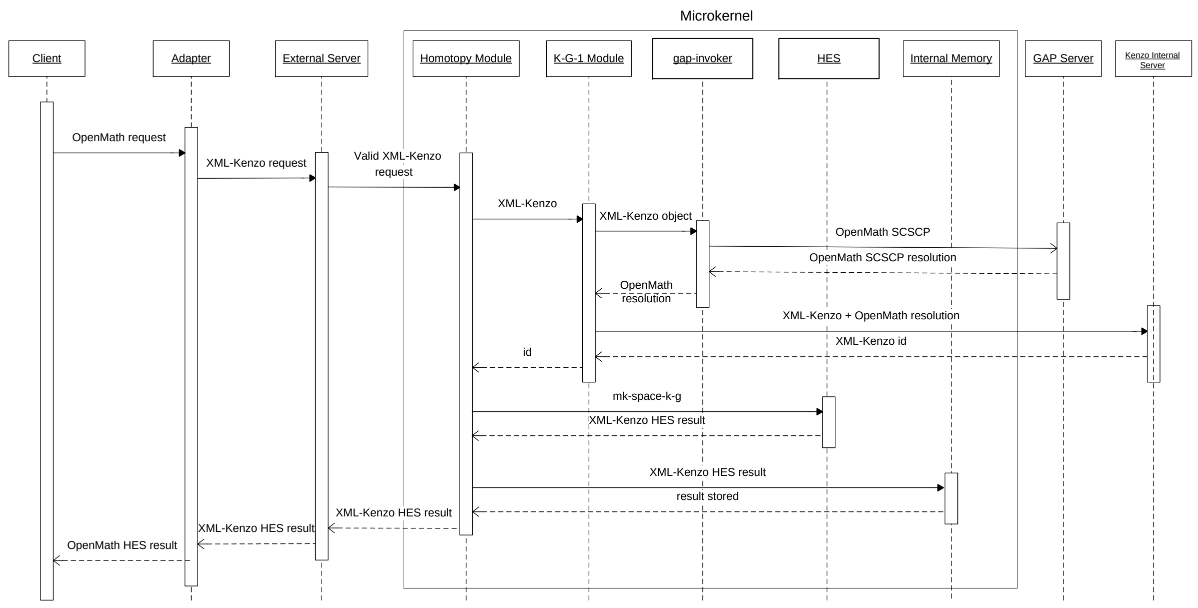


Figure 4.10: UML sequence diagram

4.2.2.5 Execution flow

To provide a better understanding of the composability of Kenzo and GAP/HAP, let us present an execution scenario where a client wants to compute $\pi_1(K(C_5,1))$ in a fresh session, that is to say, neither objects were constructed nor computations were performed previously. The execution flow of this scenario is depicted in Figure 4.10 with a UML-like sequence diagram.

The OpenMath representation of the request to compute $\pi_1(K(C_5,1))$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
           <OMS cd="ASG" name="k-g-1"/>
           <OMA> <OMS cd="groupname1" name="cyclic"/> <OMI>5</OMI> </OMA>
        </OMA>
        <OMI>1</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath request from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object

```
<operation>
    <homotopy>
       <k-g-1> <cyclic>5</cyclic> </k-g-1>
       <dim>1</dim>
    </homotopy>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification. In this case as the root element is `operation` it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homotopy` element has two children. ✓
3. The first child of the `homotopy` element belongs to one of the groups `CC`, `SS`, `SG` or `ASG`. ✓
4. The `k-g-1` element has one child which belongs to the group `A`. ✓
5. The value of the `cyclic` element of the `k-g-1` element is a natural number. ✓
6. The second child of the `homotopy` element is the `dim` element. ✓
7. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, we have a valid request that is sent to the microkernel. In the microkernel the `homotopy` module is activated. When the `homotopy` module is activated, the procedure explained in Paragraph 2.2.3.4.2 is executed. First, the `homotopy` module searches in `*object-list*` if the space $K(C_5, 1)$ was constructed previously; as this space was not constructed, the `k-g-1` module is invoked to construct it. When the `k-g-1` module is activated, the procedure explained in Subsubsection 4.2.2.3 is executed.

The gap-invoker is activated with the aim of obtaining a resolution from GAP/HAP of the cyclic group C_5 represented with the following XML-Kenzo object.

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

From the previous XML-Kenzo object, the gap-invoker constructs the following request, which is sent to the GAP/HAP server.

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Resolution"/>
                <OMA>
                    <OMS cd="group1" name="cyclic"/>
                    <OMI>5</OMI>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

When the GAP/HAP server receives the above request the following instruction is executed in the GAP/HAP server:

```
gap> ResolutionFiniteGroup(CyclicGroup(5)); ✠
```

The result returned by the GAP/HAP server to the gap-invoker (once we have removed the SCSCP wrapper) is:

```
<OMOBJ>
 <OMA>
  <OMS cd="resolutions" name="resolution"/>
  <!-- GROUP -->
  <OMA>
   <OMS cd="group1" name="cyclic_group"/>
   <OMI>5</OMI>
  </OMA>
  <!--More than 1000 lines skipped-->
  ...
</OMOBJ>
```

This resolution is used by the `k-g-1` module to invoke the Kenzo internal server and construct both in the Kenzo kernel and in the microkernel the space $K(C_5, 1)$. The identifier of the new object is returned to the `homotopy` module.

Afterwards, a `mk-space-k-g` is built in the microkernel as was explained in Subsubsection 4.2.2.3. Subsequently, once the space is constructed in the microkernel, as we are working with a `mk-space-k-g`, the `HES` is activated to compute the homotopy group of the space. The HES uses its rules and returns the result:

```
<result-HES>
    <component>5</component>
    <explanation>
       The space was the Eilenberg MacLane space $K(C_5, 1)$. The homotopy groups
       of an Eilenberg MacLane space $K(G, m)$ are: $\pi_m(K(G, m)) = G$ and
      $\pi_r(K(G, m)) = 0$ if $m \neq r$.
    </explanation>
</result-HES>
```

This result is stored in the internal memory to avoid re-computations by the `homotopy` module and sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="computing" name="result-HES"/>
        <OMI>5</OMI>
        <OMSTR>
          The space was the Eilenberg MacLane space $K(C_5, 1)$. The homotopy groups
          of an Eilenberg MacLane space $K(G, m)$ are: $\pi_m(K(G, m)) = G$ and
         $\pi_r(K(G, m)) = 0$ if $m \neq r$.
        </OMSTR>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

4.2.3 Composability of Kenzo and GAP/HAP in the *fKenzo* GUI

This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to provide support for the new functionality presented in the previous subsection.

In this case we have modified the GAP *fKenzo* module, presented in Subsection 4.1.5, to enhance the GUI with support for Eilenberg MacLane spaces of type $K(G, 1)$. Now, the GAP/HAP module references three files: `gap-structure` (that defines the structure of the graphical constituents), `gap-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection (this plug-in is an extension of the presented in Subsection 4.1.3).

We have defined additional graphical elements for the GAP module, using the XUL specification language, in the `gap-structure` file:

- A menu option called `k-g-1` into the menu `Abelian Simplicial Groups`.

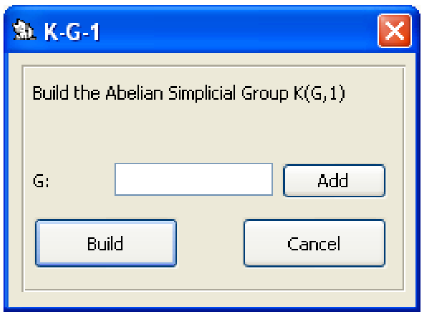


Figure 4.11: K-G-1 window

- A window called `k-g-1` (see Figure 4.11). As we have explained previously, we do not specify from scratch this window, but we use a generic specification with the desired structure whose attributes take the concrete values of the new window.

In addition to the functionality explained in Subsection 4.1.5, the `gap-functionality` document includes the functionality related to these new components. Namely, a function acting as event handler is associated with the `k-g-1` menu option; this function shows the `k-g-1` window (see Figure 4.11) if a group was constructed previously in the session; otherwise, it shows a message which indicates that a group must be constructed before using this menu option.

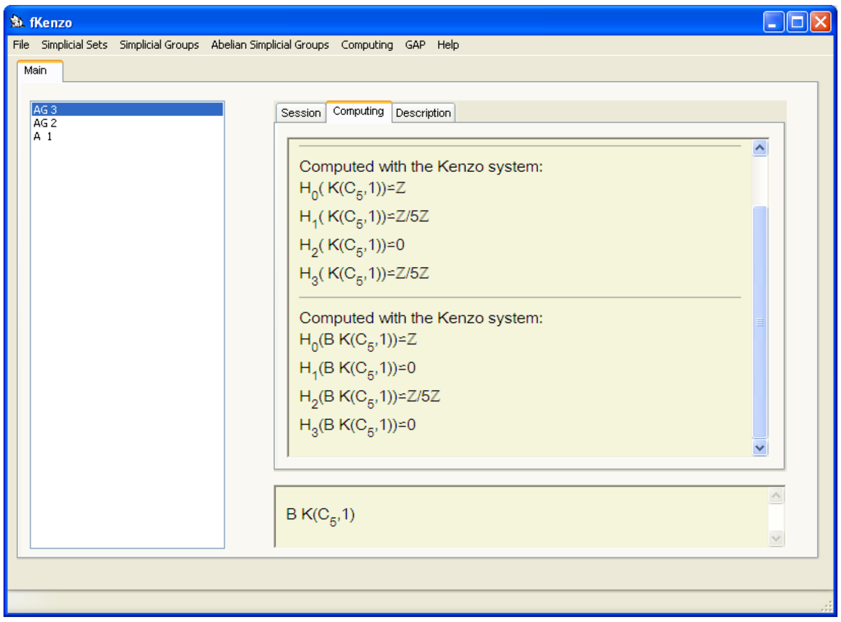
From the `k-g-1` window, the user must select a group from a list with the `Add` button. Once the user has selected a group G, when he presses the `Build` button of the `k-g-1` window, an OpenMath request is generated and sent it to our framework. The Eilenberg MacLane space $K(G, 1)$ and its identification number is returned. Eventually, *fKenzo* adds to the list of constructed objects (situated in the left side of the main tab of the *fKenzo* GUI) the new object.

Then, a *fKenzo* user can use an Eilenberg MacLane space of type $K(G, 1)$ with G a cyclic group as any other space. In particular, he can employ them to construct other spaces (for instance the classifying space of these Eilenberg MacLane spaces, see Figure 4.12) or to compute its homology and homotopy groups, see Figures 4.12 and 4.13.

It is worth noting that from the user point of view he can construct and use the space $K(G, 1)$ as any other space, so he does not know that internally the process to construct this kind of spaces involves the composability of two Computer Algebra systems. Therefore, we are providing an interoperability tool to the user without disconcerting him by the technicalities needed to perform this composability.

In particular, the procedure that the user must follow to achieve the same behavior presented in [RER09] is:

1. Load the GAP *fKenzo* module,
2. build the cyclic group G,

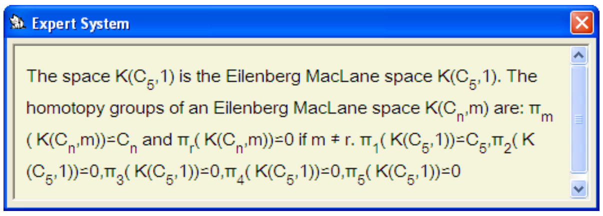

fKenzo
File Simplicial Sets Simplicial Groups Abelian Simplicial Groups Computing GAP Help
Main
AG 3
AG 2
A 1
Session Computing Description

Computed with the Kenzo system:
H_0(K(C_5,1))=Z
H_1(K(C_5,1))=Z/5Z
H_2(K(C_5,1))=0
H_3(K(C_5,1))=Z/5Z

Computed with the Kenzo system:
H_0(B K(C_5,1))=Z
H_1(B K(C_5,1))=0
H_2(B K(C_5,1))=Z/5Z
H_3(B K(C_5,1))=0

B K(C_5,1)

Figure 4.12: Homology groups of $K(C_5, 1)$ and $B(K(C_5, 1))$



The space K(C_5,1) is the Eilenberg MacLane space K(C_5,1). The homotopy groups of an Eilenberg MacLane space K(C_n,m) are: π_m(K(C_n,m))=C_n and π_r(K(C_n,m))=0 if m ≠ r. π_1(K(C_5,1))=C_5,π_2(K(C_5,1))=0,π_3(K(C_5,1))=0,π_4(K(C_5,1))=0,π_5(K(C_5,1))=0

Figure 4.13: Explanation facility window for homotopy groups of $K(C_5, 1)$

3. build the space $K(G,1)$.

As can be seen, this is a much more simpler approach than the one presented in [RER09] from the user point of view. However, no reward comes without its corresponding price and some of the constructions developed in [RER09, Rom10] cannot be made available in *fKenzo* (for instance 2-types, since their construction involves a study of its internal structure and a knowledge of the definition of Lisp functions, and such a meticulous study is difficult to integrate in a GUI, at least in an easy and usable way, that is, without giving access to the internal Common Lisp code).

4.3 Integration of the ACL2 Theorem Prover

As we claimed in Section 2.1 one of the challenges of our system was the integration of different kinds of tools; in particular, we were not only interested in integrating tools which allow us to perform computations (such as Computer Algebra systems) but also to certify the results (by means of Theorem Proving tools).

To this aim, as a first step, we have taken advantage of the semantical possibilities of OpenMath. Concretely, we have added, in our Content Dictionaries, the properties which the mathematical structures must satisfy. This opens the chance of interfacing OpenMath with different theorem provers. A similar approach (in the sense that it involved both OpenMath and a Theorem Prover), using the proof checkers Lego and Coq, was proposed by Caprotti and Cohen in [CC99]. The approach followed in that paper consisted of checking whether OpenMath expressions were well-typed with Lego and Coq Theorem Provers or not. Our approach is a bit different, on the one hand, we have used the ACL2 theorem prover instead of Coq and, on the other hand, we want to define mathematical structures in Content Dictionaries; then, an interpreter will transform those Content Dictionaries into ACL2 encapsulates (see Subsection 1.3.2) which can be used later on in ACL2. The importance of this case study comes from the fact that we are giving the first steps to store mathematical theories developed in Theorem Provers in OpenMath/OMDoc documents, increasing the portability of those theories to different theorem provers.

The rest of this section is organized as follows. Subsection 4.3.1 is devoted to present how axiomatic information is added to our Content Dictionaries. The transformation from Content Dictionaries to ACL2 encapsulates is explained in Subsection 4.3.2. The integration of ACL2 in our framework and in the *fKenzo* GUI are presented respectively in subsections 4.3.3 and 4.3.4.

4.3.1 Adding axiomatic information to Content Dictionaries

Up to now, we have defined four Content Dictionaries (see Subsection 2.2.5) related to the different kind of objects that can be built in our system (Chain Complexes, Simplicial

Sets, Simplicial Groups and Abelian Simplicial Groups). These Content Dictionaries, which are www-available at [Her11], defined several objects (such as spheres, loop spaces and so on) which instantiate a mathematical structure. However, they did not formally define the mathematical structure. To deal with this question we have proceeded as follows.

The Kenzo mathematical structures (see Figure 1.2 of Subsection 1.2.1) are algebraic structures which properties are axiomatically given and which have associated a signature with the arities of the functions which define an object of that structure. For each one of the Kenzo mathematical structures we have defined an OpenMath Content Dictionary (or extended the ones already defined) to include the formal definition of these mathematical structures.

To this aim we have based on the Small Type System formalism, see [Dav99] for details, which has been designed to give semi-formal signatures to OpenMath symbols. By using this mechanism we have included signatures in the OpenMath objects definition. In addition, we have specified their properties in two different ways (by means of `<FMP>` and `<CMP>` tags) and we have associated an instance example with them.

We are going to focus on the `SS` Content Dictionary which defines the notion of simplicial sets introduced in Definition 1.17; the rest of Content Dictionaries are based on the same ideas.

To define the *simplicial set* structure, we must provide the disjoint sets $\{K^q\}_{q\geq 0}$ and both face and degeneracy operators. The sets $\{K^q\}_{q\geq 0}$ can be seen as a graded set; so, it is possible to consider its characteristic function which, from an element x and a degree g, determines if the element x belongs to the set K^g. To be precise, an *invariant* function can be used in order to encode the characteristic function of the graded set $\{K^q\}_{q\geq 0}$.

Based on the previous way of representation, the following signature, let us called it `SS`, has been defined for simplicial sets.

```
inv  : u   nat       -> bool
face : u   nat   nat -> u
deg  : u   nat   nat -> u
```

where `inv`, `face` and `deg` represent the characteristic function of the underlying set and the face and degeneracy operators respectively, and `u` denotes the Universe, of Lisp objects in this case.

The `SS` signature can be codified using the OpenMath `Signature` element as follows:

```
<Signature name="simplicial-set">
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
            <OMS name="mapsto" cd="sts"/>
            <OMA id="inv">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMS cd="setname2" name="boolean"/>
            </OMA>
            <OMA id="face">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMV name="PositiveInteger"/>
                <OMV name="Element"/>
            </OMA>
            <OMA id="degeneracy">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMV name="PositiveInteger"/>
                <OMV name="Element"/>
            </OMA>
            <OMV name="Simplicial-Set"/>
        </OMA>
    </OMOBJ>
</Signature>
```

The above OpenMath `Signature` must be read as follows. Each application `OMA` inside the main `mapsto` of the `simplicial-set` signature represents each one of the functions of the `SS` signature. The value of the `id` of the application tag (`<OMA id=" ">`) is the name of the function. The `mapsto` symbol, inside of the application tag, is applied to n variables and/or symbols, the first $n-1$ will be the inputs and the last one the output of the function. The "type" of the inputs and outputs is also included. In this way, we can provide the signature of each Kenzo mathematical structure.

The formal mathematical properties of the simplicial sets are given in the `<FMP>` tags of the *simplicial set* definition. In this case `<FMP>` elements state the properties of invariance of face and degeneracy operators and the relations between them (the five properties included in the definition of simplicial sets). All of them have also been included in natural language by using `<CMP>` elements. For instance, the face operator invariance ($x \in K^q \Rightarrow \partial_i x \in K^{q-1}$) is represented as follows:

```
<CMP> Face operator invariance: x \in K^q => \partial_i x \in K^{q-1} </CMP>
<FMP>
 <OMA>
  <OMS cd="logic1" name="implies"/>
  <OMA> <OMS name="inv"/> <OMV name="x"/> <OMV name="q"/> </OMA>
  <OMA>
    <OMS name="inv"/>
    <OMA> <OMS name="face"/> <OMV name="x"/> <OMV name="i"/> <OMV name="q"/> </OMA>
    <OMA> <OMS cd="arith1" name="minus"/> <OMV name="q"/> <OMI>1</OMI> </OMA>
  </OMA>
 </OMA>
</FMP>
```

Finally, an example of a concrete simplicial set has been included. Namely, the simplicial set with one element belonging to each set K^q and with each face and degeneracy operation of degree q returning the element of degree $q-1$ and $q+1$ respectively has been considered.

```
<Example>
    ...
    <OMBIND>
        <OMS name="face"/>
        <OMBVAR>
            <OMV name="x"/>  <OMV name="i"/> <OMV name="q"/>
        </OMBVAR>
        <OMS cd="list" name="nil"/>
    </OMBIND>
    ....
</Example>
```

In this way, all the Kenzo mathematical structures can be defined by means of OpenMath Content Dictionaries.

4.3.2 From Content Dictionaries to ACL2 encapsulates

Content Dictionaries, defined in the way presented in the previous subsection, open the chance of interfacing OpenMath with theorem provers; namely, in our case, with the ACL2 Theorem Prover (presented in Section 1.3). The main reason to choose ACL2 was the fact that, as Kenzo, it is a Common Lisp program, then, we can use ACL2 to verify real Kenzo code, as we will see from the next chapter, improving in this way the reliability of our system.

It is worth noting that our Content Dictionaries include all the necessary information to generate ACL2 encapsulates. Let us remember that ACL2 supports the constrained introduction of new function symbols by means of the encapsulate notion, a detailed description of this ACL2 functionality was presented in Subsection 1.3.2. Briefly, an *encapsulate* allows the introduction of function symbols in ACL2, without a completely specification of them, but just assuming some properties which define them partially. An ACL2 encapsulate consists of a set of function signatures, a set of properties of these functions and a "witness" for each one of the functions, where a witness is an existing function that can be proved to have the required properties (witnesses are provided to avoid the introduction of inconsistencies in ACL2).

From each Content Dictionary specified as the one presented in the previous subsubsection, an ACL2 encapsulate can be generated. From now on, we are going to explain the transformation from one of our Content Dictionaries (in particular the `SS` Content Dictionary) to an ACL2 encapsulate.

First of all, the OpenMath signatures must be transformed to ACL2 signatures. Each application `OMA` inside the main `mapsto` of the simplicial set signature is translated into a function of the encapsulate in the following way. The value of the `id` of the application tag (`<OMA id=" ">`) will be the name of the function, the `mapsto` symbol inside the application tag is converted to `=>` in ACL2. The `mapsto` symbol is applied to n variables and/or symbols, the first $n-1$ will be the inputs and the last one the output. Note that ACL2 is a system without explicit typing, so, although the "type" of the objects has been included in the Content Dictionary they will be translated into asterisks in ACL2. Adding the necessary brackets, the following ACL2 signature is obtained.

```
((inv * *) => *))
(((face * * *) => *)
((degeneracy * * *) => *)
```

The following step consists of transforming the mathematical properties, specified in the Content Dictionary, into ACL2 lemmas. In order to do this, we proceed as follows. First of all, the `<CMP>` tags will be translated into ACL2 comments, expressed with "`;`". To each `<FMP>` tag, a new lemma, `defthm` in ACL2 syntax, with the name `prop-n` where `n` is a variable that indicates the number of the property, must be defined. The `implies`, `and`, `eq` and `minus` OpenMath symbols are translated respectively into the `implies`, `and`, `equal` and "`-`" ACL2 functions (these are some examples of the equivalences established between OpenMath symbols and ACL2 functions). For instance, the following ACL2 lemma about the face operator invariance is obtained from the respective property in the Content Dictionary.

```
; Face operator invariance
(defthm prop-1 (implies (inv x q) (inv (face x i q) (- q 1))))
```

And, finally, from the `<Example>` tags, the witnesses are obtained. Each example in a Content Dictionary will be a local definition in an ACL2 encapsulate. The `OMBIND` symbol indicates the beginning of the definition, `defun` in ACL2, its first argument is a symbol which indicates the name of the function, the second one is an `OMBVAR` element which specifies the name of the parameters and the third one is the body of the function. If some of the arguments of the function does not appear in its body, they will be ignored to obtain a correct ACL2 function, as we show in the translation of the previous subsubsection.

```
(local (defun face (x i q) (declare (ignore x i q)) nil))
```

Therefore, we have an interpreter which is able to construct ACL2 encapsulates from some concrete Content Dictionaries. This interpreter is a Common Lisp program which takes as input an OMDoc file specified in the way presented in Subsubsection 4.3.1 and constructs an ACL2 encapsulate. All the generated encapsulates can be evaluated in ACL2.

In the same way that we have developed an interpreter which generates ACL2 encapsulates from Content Dictionaries, we can also implement other interpreters which supply suitable code for other Theorem Provers such as Isabelle or Coq.

4.3.3 Integration of ACL2

To integrate the ACL2 Theorem Prover in our framework we have developed a plug-in following the guidelines given in Subsubsection 3.1.2. This new plug-in will allow us to verify ACL2 scripts (an ACL2 script is a file of ACL2 forms, such as definitions or theorems, which is processed in a sequential way) obtaining as result a file with the ACL2 output obtained from the evaluation of the script in ACL2. Moreover, we have included the interpreter presented in the previous subsection as a module of the microkernel to generate ACL2 encapsulates from Content Dictionaries. This plug-in references the following resources:

```
<code id="ACL2">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> ACL2-is.lisp </data>
   <data format="Kf/microkernel"> ACL2-m.lisp </data>
   <data format="Kf/microkernel"> CD-to-ACL2.lisp </data>
   <data format="Kf/adapter"> ACL2-a.lisp </data>
</code>
```

These resources deserve a detailed explanation that is provided in the following subsubsections.

4.3.3.1 Extending the XML-Kenzo schema

As we have just explained, the new plug-in will allow us to execute ACL2 scripts and store the output produced by ACL2 in a file. Then, we need to represent the way

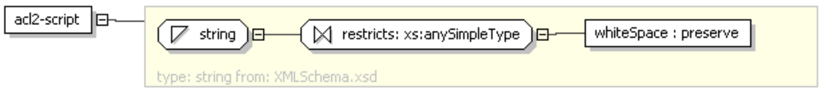


Figure 4.14: acl2-script XML-Kenzo specification

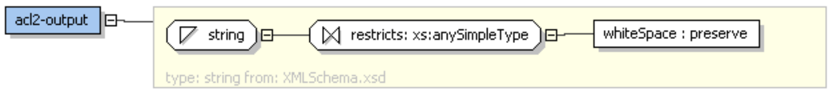


Figure 4.15: acl2-output XML-Kenzo specification

of providing the path of both the ACL2 script and the ACL2 output. Therefore, we have extended the XML-Kenzo specification (`XML-Kenzo.xsd` file) to this aim. In this specification, we have defined two new elements: a new element of the `requests` group called `acl2-script` (see Figure 4.14), whose value is a string which indicates the path of the ACL2 script, and a new element of the `results` group called `acl2-ouput` (see Figure 4.15) whose value is a string which indicates the path of an ACL2 output file.

Moreover, we want to introduce new functionality in our system which allows us to generate an ACL2 encapsulate from a Content Dictionary. Therefore, we have extended the XML-Kenzo specification to admit this new functionality. In the XML-Kenzo specification, we have defined a new element of the `requests` group called `CD-to-ACL2` (see Figure 4.16), whose value is a string which indicates the path of an OMDoc document; and a new element of the `results` group called `ACL2-encapsulate` (see Figure 4.17) whose value is a string, namely an ACL2 encapsulate.

As we explained in Subsubsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified the external server is able to receive XML-Kenzo objects such as:

```
<acl2-script> acl2-script-path </acl2-script>
```

```
<OMDoc-to-ACL2> OMDoc-path-file </OMDoc-to-ACL2>
```

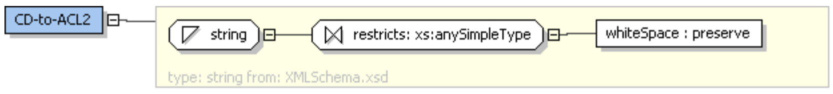


Figure 4.16: `CD-to-ACL2` XML-Kenzo specification

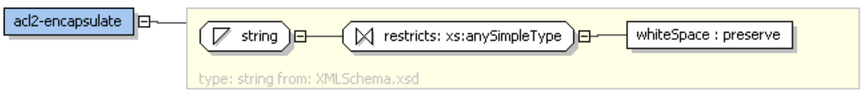


Figure 4.17: `acl2-encapsulate` XML-Kenzo specification

4.3.3.2 A new internal server: ACL2

To integrate the ACL2 Theorem Prover, we have followed the same methodology explained in the Kenzo (see Subsection 2.2.2) case: we have ACL2 wrapped with an XML-Kenzo interface and the communication between the microkernel and ACL2 is performed by means of XML-Kenzo requests through an external interface which offers the available services of our ACL2 internal server (at this moment the only available service allow the execution of ACL2 scripts). As in the case of the GAP/HAP system we must install ACL2 (the ACL2 installer can be downloaded from [KM]).

The `ACL2-is.lisp` provides all the necessary functionality to integrate the ACL2 internal server; that is to say, ACL2, the wrapper and the interface. The ACL2 interface provides just one service called `execute-acl2-script`. The function associated with this service, also called `execute-acl2-script`, takes as input an `acl2-script` XML-Kenzo object. From the file indicated in the `acl2-script` XML-Kenzo object, the `execute-acl2-script` function extracts the ACL2 code and executes it in ACL2. The `execute-acl2-script` function returns the path of a file where the output generated by ACL2, when executing the script, is stored. This path is returned in an `acl2-output` XML-Kenzo object.

4.3.3.3 New modules of the microkernel

The `ACL2-m.lisp` file defines a new module for the microkernel called `ACL2`. The procedure implemented in this module is just in charge of checking that the path indicated by the `acl2-script` XML-Kenzo object exists in which case the module invokes the ACL2 internal server; if the path does not exist this module informs the user about this situation by means of a `warning` XML-Kenzo object. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the `ACL2` module.

This new module can be considered neither a computation module nor a construction module. This module belongs to a new category of modules related to theorem proving tools called *verification modules.*

Moreover, the `CD-to-ACL2.lisp` defines a new verification module for the microkernel called `CD-to-ACL2`. The procedure implemented in this module is the interpreter which transforms an OMDoc document indicated in an `OMDoc-to-ACL2` XML-Kenzo object to an ACL2 encapsulate. In addition, this file enhances the interface of the microkernel in order to be able to invoke the new module.

4.3.3.4 Increasing the functionality of the adapter

Finally, we have extended the `Aux` Content Dictionary by means of the definition of four new objects: `acl2-script`, `acl2-ouput`, `CD-to-ACL2` and `ACL2-encapsulate`. Therefore, the `ACL2-a.lisp` file contains the functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects, devoted to indicate the path of a file, to XML-Kenzo requests. Namely, we have extend the Phrasebook by means of a new parser in charge of this task. For instance, the XML-Kenzo request:

```
<acl2-script> acl2-script-path </acl2-script>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
      <OMS cd="Aux" name="acl2-script"/>
      <OMSTR> acl2-script-path </OMSTR>
   </OMA>
</OMOBJ>
```

4.3.4 Integration of ACL2 in the *fKenzo* GUI

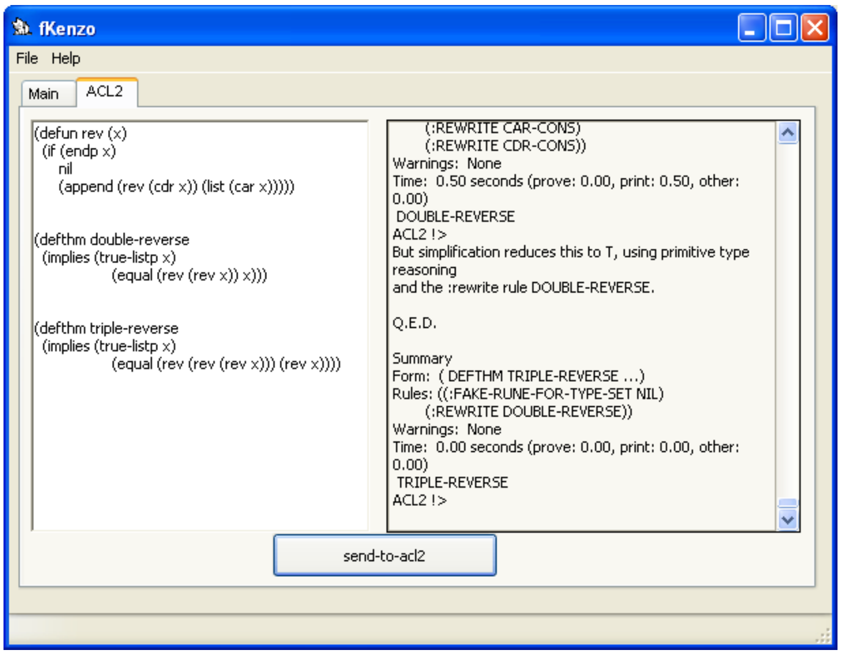
This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to provide access to the new functionality presented in the previous subsection.

In this case we have defined a fresh *fKenzo* module to enhance the GUI with support for the ACL2 system. The new module references three files: `acl2-structure` (that defines the structure of the graphical constituents), `acl2-functionality` (which provides the functionality related to the graphical constituents) and the last plug-in introduced.

We have defined three graphical elements, using the XUL specification language, in the `acl2-structure` file:

- A new tab called `ACL2` which is included in the main panel (see Figure 4.18).
- A menu called `ACL2` which contains two options: `CD to ACL2` and `CD to ACL2 in file`.
- A window called `CD-to-ACL2` (see Figure 4.19).

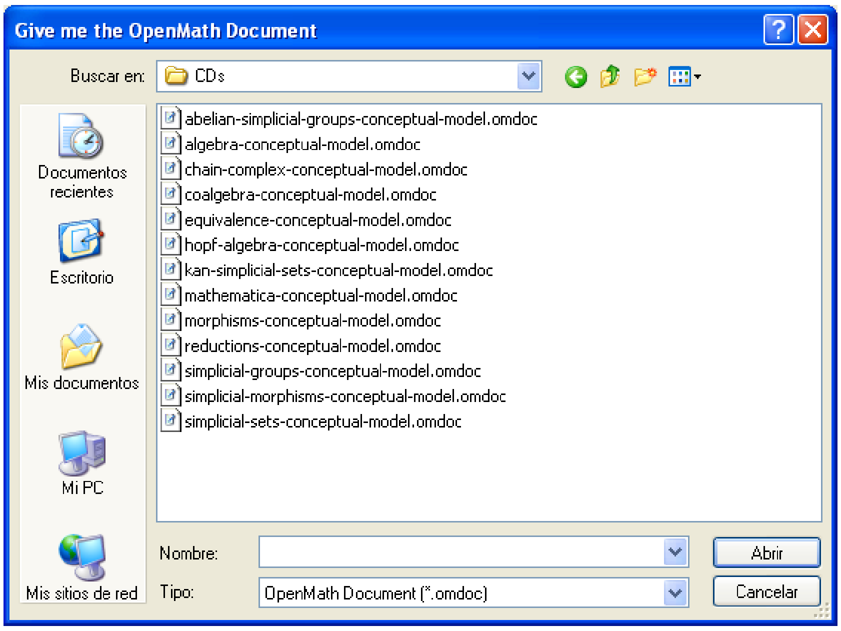
The new tab page contains two areas and one button: the left area will be used to display ACL2 instructions, the button will send the instructions of the left side to ACL2, and finally, the right area will show the ACL2 result obtained from the evaluation of the



```
(defun rev (x)
 (if (endp x)
     nil
     (append (rev (cdr x)) (list (car x)))))


(defthm double-reverse
 (implies (true-listp x)
          (equal (rev (rev x)) x)))


(defthm triple-reverse
 (implies (true-listp x)
          (equal (rev (rev (rev x))) (rev x))))
```

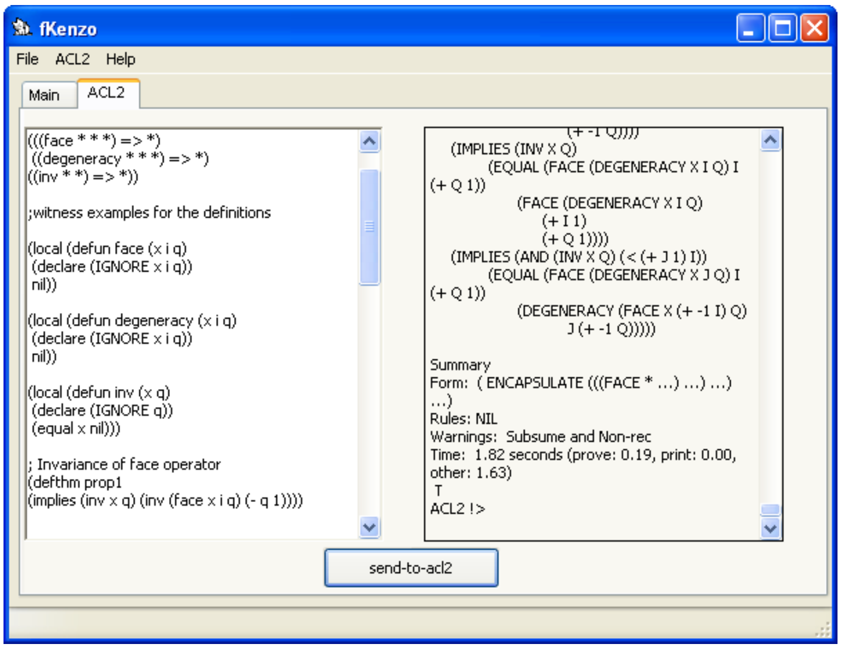
```
       (:REWRITE CAR-CONS)
       (:REWRITE CDR-CONS))
Warnings:  None
Time:  0.50 seconds (prove: 0.00, print: 0.50, other:
0.00)
 DOUBLE-REVERSE
ACL2 !>
But simplification reduces this to T, using primitive type
reasoning
and the :rewrite rule DOUBLE-REVERSE.

Q.E.D.

Summary
Form:  ( DEFTHM TRIPLE-REVERSE ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
       (:REWRITE DOUBLE-REVERSE))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other:
0.00)
 TRIPLE-REVERSE
ACL2 !>
```

Figure 4.18: The *ACL2* tab with an example



abelian-simplicial-groups-conceptual-model.omdoc
algebra-conceptual-model.omdoc
chain-complex-conceptual-model.omdoc
coalgebra-conceptual-model.omdoc
equivalence-conceptual-model.omdoc
hopf-algebra-conceptual-model.omdoc
kan-simplicial-sets-conceptual-model.omdoc
mathematica-conceptual-model.omdoc
morphisms-conceptual-model.omdoc
reductions-conceptual-model.omdoc
simplicial-groups-conceptual-model.omdoc
simplicial-morphisms-conceptual-model.omdoc
simplicial-sets-conceptual-model.omdoc

Figure 4.19: The CD-to-ACL2 window



```
(((face * * *) => *)
 ((degeneracy * * *) => *)
((inv * *) => *))

;witness examples for the definitions

(local (defun face (x i q)
 (declare (IGNORE x i q))
 nil))

(local (defun degeneracy (x i q)
 (declare (IGNORE x i q))
 nil))

(local (defun inv (x q)
 (declare (IGNORE q))
 (equal x nil)))

; Invariance of face operator
(defthm prop1
(implies (inv x q) (inv (face x i q) (- q 1))))
```

```
                         (+ -1 Q))))
    (IMPLIES (INV X Q)
          (EQUAL (FACE (DEGENERACY X I Q) I
(+ Q 1))
               (FACE (DEGENERACY X I Q)
                   (+ I 1)
                   (+ Q 1))))
    (IMPLIES (AND (INV X Q) (< (+ J 1) I))
          (EQUAL (FACE (DEGENERACY X J Q) I
(+ Q 1))
               (DEGENERACY (FACE X (+ -1 I) Q)
                         J (+ -1 Q)))))

Summary
Form:  ( ENCAPSULATE (((FACE * ...) ...) ...)
...)
Rules: NIL
Warnings:  Subsume and Non-rec
Time:  1.82 seconds (prove: 0.19, print: 0.00,
other: 1.63)
 T
ACL2 !>
```

Figure 4.20: The *ACL2* tab with the simplicial sets encapsulate

instructions of the left area. A *fKenzo* user can manually write ACL2 definitions and theorems in the left area and then execute them, but the idea is that the ACL2 scripts are going to be automatically generated by the system and displayed in the left area of the ACL2 tab.

The functionality stored in the `acl2-functionality` includes the new graphical constituents. On the one hand, the event handler associated with the `send-to-acl2` button is defined in this file. This event handler obtains the ACL2 instructions which have been written in the left side of the ACL2 tab, afterwards it stores them in a temporal file, subsequently an `acl2-script` OpenMath request is created with the path of the temporal file and sent to the framework. The returned result is shown in the right side of the ACL2 tab.

On the other hand, the event handlers associated with the `CD to ACL2` and `CD to ACL2 in file` menu options are also included in the `acl2-functionality` file. The former event handler shows the `CD-to-ACL2` window (see Figure 4.19) which allows the user to choose a Content Dictionary with the description of one of the Kenzo mathematical structures. Once the user has selected a Content Dictionary; the system generates an ACL2 encapsulate which is displayed in the left part of the `ACL2` tab (see Figure 4.20). Then, the user only has to use the functionality associated with the `send-to-acl2` button and the ACL2 output produced when the encapsulate is executed in ACL2 is shown in the right side of the `ACL2` tab (see Figure 4.20). The latter event handler, the one associated with the `CD to ACL2 in file` menu option, instead of writing the encapsulate generated in the left part of the `ACL2` tab, generates a file with the encapsulate and asks a path to the user to save the file.

As can be noticed, our interface to interact with ACL2 is a plain text editor which is obviously less usable than typical ACL2 interfaces (Emacs [Sta81] or ACL2 sedan [DMMV07]). It is worth noting that our goal was not the creation of an ACL2 interface which could compete with regular ACL2 editors. The idea is that the ACL2 scripts are automatically generated by the system and written in the left area of the ACL2 tab (as we have seen for the case of the encapsulates generated from Content Dictionaries) and the user only has to press the `send-to-acl2` button, without typing any additional ACL2 command, to obtain certificates.

4.4 Interoperability between Kenzo, GAP/HAP and ACL2

As we have said several times throughout this chapter, we are interested not only in integrating several tools in our framework and use them individually, but also in making them work together.

The integration of Kenzo and GAP to construct the effective homology version of Eilenberg MacLane spaces of type $K(G,n)$ where G is a cyclic group was presented in Section 4.2. Now, we want to increase the reliability of those programs by means of the ACL2 Theorem Prover. The importance of this verification lies in the fact that Eilenberg MacLane spaces of type $K(G,n)$ where G is a cyclic group are instrumental in the computation of homotopy groups.

We have integrated the already available tools in *fKenzo* to increase the reliability of some Kenzo programs. In particular, we want to verify the correctness of the following Kenzo statement using the ACL2 Theorem Prover.

```
> (cyclicgroup 5) ✠
[K1 Abelian-Group]
```

This means that we want to prove in ACL2 that the returned object by the `cyclicgroup` Kenzo function really satisfies the axioms of an abelian group. This is important, for instance, to ensure the following function,

```
> (DEFMETHOD K-G-1 ((group AB-GROUP)) ...) ✠
```

which constructs the Eilenberg MacLane space of its argument, is really applied over a meaningful input (that is to say, an actual abelian group). Let us recall that the construction of the effective homology of that kind of Eilenberg MacLane spaces involves the use of GAP/HAP to obtain a resolution (see Algorithm 4.5). To sum up, we use Kenzo to construct Eilenberg MacLane spaces of cyclic groups; in addition by means of GAP/HAP we are able to construct the effective homology of these Eilenberg

MacLane spaces; and, ACL2 increases the reliability of the construction of these Eilenberg MacLane spaces verifying the correctness of their input argument. Then, these three systems are combined producing a powerful and reliable tool.

The rest of this section is organized as follows. Subsection 4.4.1 explains the implementation of cyclic groups in the Kenzo system. Subsection 4.4.2 deals with the proof in ACL2 of the correctness of the implementation of cyclic groups introduced in Subsection 4.4.1. The integration of Kenzo, GAP/HAP and ACL2 in our framework and in the *fKenzo* GUI is presented in Subsections 4.4.3 and 4.4.4 respectively.

4.4.1 Implementation of cyclic groups in Kenzo

The abelian Group structure is a mathematical structure which was not included in the original version of Kenzo; but it was included in the development of [RER09] to construct Eilenberg MacLane spaces of abelian groups. This structure was defined, following the same schema that the one used to define mathematical structures in Kenzo (see Subsection 1.2.1), using the two following Common Lisp class definitions:

```
(DEFCLASS GROUP ()
   ((elements :type group-basis :initarg :elements :reader elements)
   (cmpr :type cmprf :initarg :cmpr :reader cmpr1)
   (mult :type function :initarg :mult :reader mult1)
   (inv :type function :initarg :inv :reader inv1)
   (nullel :type gnrt :initarg :nullel :reader nullel)
   (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
   (orgn :type list :initarg :orgn :reader orgn)
   (resolution :type reduction :initarg :resolution :reader resolution)))
```

```
(DEFCLASS AB-GROUP (GROUP) ())
```

It is worth noting that the `AB-GROUP` class, which represents abelian groups, is a subclass, without any additional slot, of the `GROUP` class. The relevant slots (relevant in the sense of being important for our verification process) of this class are `elements`, a list of the elements of the group; `mult`, the function defining the binary operation of two elements of the group; `inv`, the function defining the inverse of the elements of the group and `nullel`, which is the null element of the group.

The `cyclicgroup` function constructs instances of the `AB-GROUP`. In particular, it takes as argument a natural number n and constructs an instance of the `AB-GROUP` which represents the cyclic group C_n. The concrete definition of the `cyclicgroup` function is:

```
(defun cyclicgroup (n)
  (build-ab-group
   :elements (<a-b> 0 (1- n))
   :mult #'(lambda (g1 g2) (mod (+ g1 g2) n))
   :inv #'(lambda (g) (mod (- n g) n))
   :nullel 0
   :orgn `(Cyclic-group of order ,n)))
```

This piece of code must be read as follows. From a natural number n the `cyclicgroup` function constructs an `AB-GROUP` instance where the slot `elements` has as value the ordered list of natural numbers from 0 to $n-1$ (generated from the `<a-b>` function), the slot `mult` is a functional slot which from two elements $g1$ and $g2$ returns the value of $(g1+g2)$ mod n, the slot `inv` is a functional slot which from an element g returns the value of $(n-g)$ mod n and 0 is the null element stored in the slot `nullel`. Finally, the `orgn` slot is a comment which provides the "origin" of the object.

The rest of the slots of the `AB-GROUP` instance are either automatically generated, in the case of `idnm`, or a value is assigned after the construction of the object using a GAP resolution, in the case of the `resolution` slot.

4.4.2 The proof in ACL2 about cyclic groups

A group is a mathematical structure which can be defined by means of four functions based on the following signature, called `GRP`:

```
invariant  : u    -> bool
mult       : g  g -> g
inv        : g    -> g
nullel     :      -> g
```

The four functions of this signature are: the `invariant` unary operation (which represents if an element belongs to the group, that is, the characteristic function of the underlying set), the `mult` binary operation (the product), the `inv` unary operation (the inverse) and the `nullel` constant operation (the null element). As concrete axiomatization of a group we have chosen that of [Lip81].

Therefore, if we want to prove in ACL2 that four concrete functions, with the correct arities given by the `GRP` signature, determine an (abelian) group, we need to prove that these functions satisfy the properties of (abelian) groups.

Then, for instance if we want to verify in ACL2 that the Kenzo implementation of the cyclic group C_5 is an abelian group we must proceed as follows. First of all, we need to define the four functions which determine the group (we add the suffix `cn`, where `n` is the dimension of the group, to the function names):

```
(defun invariant-c5 (g) (member g (<0-n> 5))

(defun mult-c5 (g1 g2) (mod (+ g1 g2) 5))

(defun inv-c5 (g) (mod (- 5 g) 5))

(defun nullel-c5 () 0)
```

It is worth noting that the definitions of `mult-c5`, `inv-c5` and `nullel-c5` are exactly the same as the ones used in the `cyclicgroup` function for the slots `mult`, `inv` and `nullel` of the `AB-GROUP` instance in the case of $n = 5$. The `invariant-c5` function is the characteristic function of the group and is directly obtained from the `elements` slot of the `AB-GROUP` instance; the transformation from this slot to our function can be considered safe (we have replaced the `<a-b>` function with the `<0-n>` function which constructs the ordered list of natural numbers from 0 to $n-1$; that is, the `<0-n>` function with argument n has the same behavior that the `<a-b>` function with arguments 0 and n). Then, if we prove that these functions (which have the correct arities specified in the `GRP` signature) determine an abelian group (that is, they satisfy the axioms of abelian groups) we can claim that the object constructed in the Kenzo system determines an abelian group. Then, we need to prove theorems in ACL2 such as the following one:

```
(defthm abelian-c5
    (implies (and (invariant-c5 a) (invariant-c5 b))
             (equal (mult-c5 a b) (mult-c5 b a))))
```

The ACL2 Theorem Prover is able to generate a proof of this kind of theorems without any external help. Therefore, we have a proof of the correctness of the `cyclicgroup` Kenzo function in the case of $n = 5$; and we could proceed in the same way for every concrete case of n. Then, we can say that the `cyclicgroup` Kenzo function taking as input the value 5 produces an Abelian Group. However, it would be more interesting to have a proof saying that for every natural number n the `cyclicgroup` Kenzo function produces an Abelian Group.

The `GRP` signature represents a group. In order to handle different groups, in [LPR99] an operation between signatures, called $()_{imp}$ operation was introduced. From a signature for an algebraic structure, a new signature for a family of the above algebraic structures can be defined. In our case, the signature $\texttt{GRP}_{imp}$ is defined as follows:

`imp-invariant`	:	imp_{GRP} `g`	`-> bool`
`imp-mult`	:	imp_{GRP} `g  g`	`-> g`
`imp-inv`	:	imp_{GRP} `g`	`-> g`
`imp-nullel`	:	imp_{GRP}	`-> g`

where the new sort imp_{GRP} is the sort for groups. This signature, really signatures

$()_{imp}$, is the one really used in the Computer Algebra systems because it allows the manipulation of algebraic structures as data.

Therefore, if we want to prove in ACL2 that four concrete functions, with the correct arities given by the $\texttt{GRP}_{imp}$ signature, determine an (abelian) group, we need to prove that these functions satisfy the axioms of (abelian) groups for every element of the imp_{GRP} sort.

In particular, we have the following four definitions for the case of cyclic groups defined in Kenzo:

```
(defun imp-invariant (n g) (member g (<0-n> n))

(defun imp-mult (n g1 g2) (mod (+ g1 g2) n))

(defun imp-inv (n g) (mod (- n g) n))

(defun imp-nullel (n) (declare (ignore n)) 0)
```

which can be considered as the definitions used by the Kenzo systems for the construction of objects which represent cyclic groups. Then, if we prove that these functions determine an abelian group (that is, they satisfy the properties of abelian groups) for every natural number, we can claim that every object constructed in the Kenzo system with the `cyclicgroup` function, taking as argument a natural number, determines an abelian group. Then, we need to prove theorems such as the following one:

```
(defthm imp-abelian
    (implies (and (natp n) (imp-invariant n a) (imp-invariant n b))
             (equal (imp-mult n a b) (imp-mult n b a))))
```

The ACL2 Theorem Prover is able again to generate a proof of those theorems without any external help if we load in ACL2 an arithmetic library. Therefore, we have a proof of the correctness of the `cyclicgroup` Kenzo function for every natural number. That is to say, for every natural number n the `cyclicgroup` Kenzo function constructs an Abelian Group.

4.4.3 Composability of Kenzo, GAP/HAP and ACL2

In the previous subsubsection we have presented the ACL2 code to verify that the implementation of cyclic groups in the Kenzo system really constructs abelian groups. We have included a new verification module in the microkernel to generate the ACL2 functions and theorems necessary to certify that a concrete group of the Kenzo system is really an (abelian) group. That is to say, given a group G, our framework will generate the four functions (`invariant`, `mult`, `inv` and `nullel`) which define the group and the theorems which ensure that those four functions determine a group.

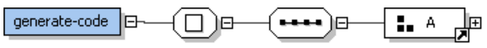


Figure 4.21: generate-code XML-Kenzo specification

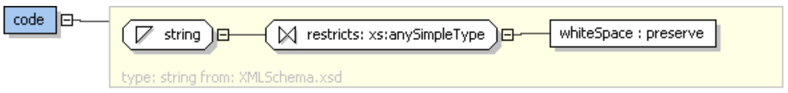


Figure 4.22: code XML-Kenzo specification

We have developed a new plug-in which allows us to integrate a code generator of the functions and theorems explained in the previous subsection.

```
<code id="certificate">
   <data format="Kf"> GAP </data>
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/microkernel"> code-generator.lisp </data>
   <data format="Kf/adapter"> certificate-a.lisp </data>
</code>
```

It is worth noting that this plug-in needs the GAP plug-in presented in Subsection 4.2.2 to work; then the GAP plug-in is referenced from the `certificate` plug-in. Let us explain now the rest of resources.

We want to introduce new functionality in our system which allows us to obtain the necessary ACL2 functions and theorems to verify that a concrete group of the Kenzo system is really an (abelian) group. Therefore, we have extended the XML-Kenzo specification to admit this new functionality. In the XML-Kenzo specification, we have defined a new element of the `requests` group called `generate-code` (see Figure 4.21), whose value is an element of the `A` group, and a new element of the `results` group called `code` (see Figure 4.22), whose value is a string, namely the functions and theorems associated with a concrete group.

As we explained in Subsection 2.2.4 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the `XML-Kenzo.xsd` file is modified the external server is able to receive XML-Kenzo objects such as:

```
<generate-code> <cyclic> 5 </cyclic> </generate-code>
```

The `code-generator.lisp` defines a new module for the microkernel called `code-generator`. The procedure implemented in this module is a Common Lisp program which generates the functions and theorems associated with the group indicated in the `generate-code` XML-Kenzo object which invokes this module. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the new module.

Finally, we have extended the `Aux` Content Dictionary by means of the definition of two new objects: `generate-code` and `code`. Therefore, the `certificate-a.lisp` file contains the functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects to XML-Kenzo objects. For instance, the above XML-Kenzo request is generated by the adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="Aux" name="generate-code"/>
      <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
   </OMA>
</OMOBJ>
```

It is worth noting that the generation of certificable code is only available for cyclic groups; but we are willing to try the certification of all the objects that can be constructed in our system.

4.4.4 Composability of Kenzo, GAP/HAP and ACL2 in the *fKenzo* GUI

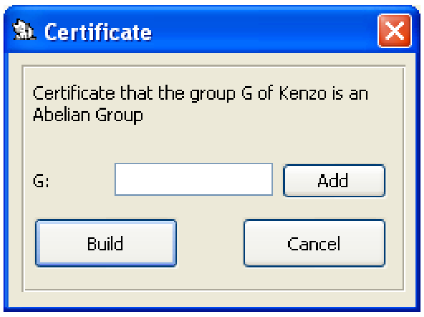
The composability of Kenzo, GAP/HAP and ACL2 in *fKenzo* to provide certificates of the correctness of the implementation of Kenzo cyclic groups does not involve a great development, since most of the ingredients were available from the modules related to GAP/HAP and ACL2.

The new module which provides the tools to compose Kenzo, GAP/HAP and ACL2 in *fKenzo* is called `GAP-Kenzo-ACL2`. This module references the GAP/HAP (Subsection 4.2.3) and ACL2 (Subsection 4.3.4) modules and also three additional files: `gap-kenzo-acl2-structure` (that defines the structure of the new graphical constituents), `gap-kenzo-acl2-functionality` (which provides the functionality related to the graphical constituents), and the plug-in explained in the previous subsection.

When this new module is loaded in *fKenzo*, the graphical elements of the GAP/HAP and ACL2 modules and their functionality are loaded in the system. Besides, two new graphical elements have been defined, using the XUL specification language, in the `gap-kenzo-acl2-structure` file:

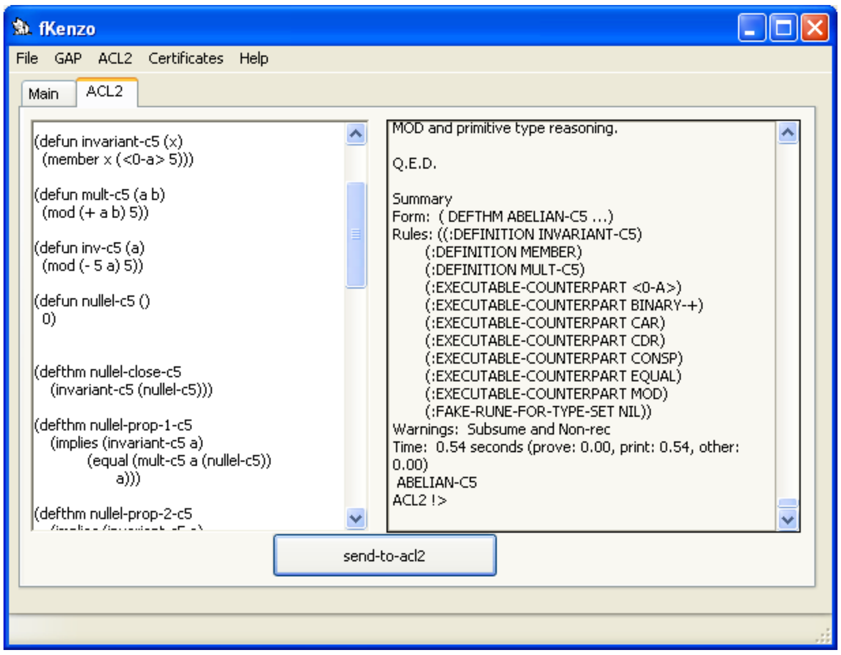
- A menu called `certificates` which contains one option called: `certificate`.
- A window called `certificate` (see Figure 4.23).

The functionality stored in the `gap-kenzo-acl2-functionality` document related to these components works as follows. A function acting as event handler is associated with the `certificate` menu option; this function shows the `certificate` window (see



Certificate that the group G of Kenzo is an Abelian Group

Figure 4.23: Certificate window



```
(defun invariant-c5 (x)
  (member x (<0-a> 5)))

(defun mult-c5 (a b)
  (mod (+ a b) 5))

(defun inv-c5 (a)
  (mod (- 5 a) 5))

(defun nullel-c5 ()
  0)


(defthm nullel-close-c5
   (invariant-c5 (nullel-c5)))

(defthm nullel-prop-1-c5
   (implies (invariant-c5 a)
         (equal (mult-c5 a (nullel-c5))
             a)))

(defthm nullel-prop-2-c5
```

```
MOD and primitive type reasoning.

Q.E.D.

Summary
Form:  ( DEFTHM ABELIAN-C5 ...)
Rules: ((:DEFINITION INVARIANT-C5)
        (:DEFINITION MEMBER)
        (:DEFINITION MULT-C5)
        (:EXECUTABLE-COUNTERPART <0-A>)
        (:EXECUTABLE-COUNTERPART BINARY-+)
        (:EXECUTABLE-COUNTERPART CAR)
        (:EXECUTABLE-COUNTERPART CDR)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART MOD)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  Subsume and Non-rec
Time:  0.54 seconds (prove: 0.00, print: 0.54, other:
0.00)
 ABELIAN-C5
ACL2 !>
```

Figure 4.24: Verification of the correctness of the cyclic group C_5

Figure 4.23) if a cyclic group was constructed previously in the session, otherwise the system informs the user about the need of constructing a group.

When, the user selects a cyclic group with the `Add` button and subsequently presses the `Build` button the system invokes our framework with a `generate-code` OpenMath object which has as argument the cyclic group. From the request, a `code` object with the definition of the four functions which determine the group (that is the functions `invariant`, `mult`, `inv` and `nullel` described in Subsection 4.4.2 with the corresponding suffix) and the ACL2 theorems which state that these functions satisfy the abelian group properties are generated. The *fKenzo* GUI extracts the code and writes it in the left side of the `ACL2` tab as can be seen in Figure 4.24.

Finally, if the user sends the script to ACL2 with the button `send-to-acl2`, a proof trial is automatically generated and finally the proof of the correctness of the implementation of the cyclic group is obtained in the right side of the `ACL2` tab (see Figure 4.24).

It is worth noting that in this case ACL2 obtains a proof without the help of the user, a situation that does not usually happens. More interesting interactions between Kenzo and ACL2 will be explained in the next chapters.

This is a simple example of the integration of Kenzo, GAP/HAP and ACL2 where each system has a concrete aim. If we gather the composability of systems presented in this subsection with the one presented in Section 4.2 we can claim that Kenzo, GAP/HAP and ACL2 work together to provide a powerful and reliable tool thanks to the *fKenzo* system.

4.5 Methodology to integrate a new internal server in our system

Throughout this chapter we have presented how to integrate both GAP/HAP (see Section 4.1) and ACL2 (see Section 4.3) in our system. The reader can notice that the same process is followed in both cases, so we can extrapolate a methodology to integrate any Computer Algebra system or Theorem Prover tool in our system.

Let us suppose that we are interested in integrating some functionality of a system called *CAS-or-TP*, it does not really matters if it is a Computer Algebra system or a Theorem Prover since the way of integrating them is analogous.

First of all, we must include the *CAS-or-TP* system as a new internal server of our framework. To that aim, we define a new plug-in which references, at least, the following resources.

```
<code id="CAS-or-TP">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> cas-or-tp-is.lisp </data>
   <data format="Kf/microkernel"> cas-or-tp-m.lisp </data>
   <data format="Kf/adapter"> cas-or-tp-a.lisp </data>
</code>
```

We want to introduce some functionality in our framework which allows us to interact with some of the procedures of the *CAS-or-TP* system. Therefore, we must extend the XML-Kenzo specification to represent the requests and results related to the *CAS-or-TP* system. Then, we define in the XML-Kenzo specification the necessary new elements.

Subsequently, we include the *CAS-or-TP* system as a new internal server following the same method explained in both the Kenzo (see Subsection 2.2.2) and ACL2 (see Subsubsection 4.3.3.2) cases. To be more concrete, we have the *CAS-or-TP* system wrapped with an XML-Kenzo interface and the communication between the microkernel and the *CAS-or-TP* system is performed by means of XML-Kenzo requests through an external interface which offers the available services of our *CAS-or-TP* internal server.

This component is included in the file `cas-or-tp-is.lisp`.

The `cas-or-tp-m.lisp` file defines new modules for the microkernel related to the *CAS-or-TP* system. The procedures implemented in these modules depend on the functionality which is provided by the *CAS-or-TP* internal server. If several new modules are included in the microkernel, it is better to devote a concrete file per each one of them.

Finally, the `cas-or-tp-a.lisp` file contains the functions (new parsers) to raise the functionality of the adapter to be able to convert from OpenMath objects, devoted to ask requests and return results related to the *CAS-or-TP* internal server, to XML-Kenzo requests.

In addition, if we want to be able to interact with the new functionality by means of the *fKenzo* GUI, we must define a new module which references three files: `cas-or-tp-structure` (that defines the structure of the graphical constituents), `cas-or-tp-functionality` (which provides the functionality related to the new graphical constituents) and the plug-in introduced previously.

In this way, different tools can be integrated in our system as new internal servers.

Glossary

ACID A set of properties (Atomicity, Consistency, Isolation and Durability) that guarantee that transactions over a shared space are processed reliably. The atomicity rule said that a modification over the shared space is fully completed; otherwise the shared space state is unchanged. The consistency rule said that a modification of the shared state will take the shared space from one consistent state to another. The isolation rule refers to the requirement that other process cannot access data that has been modified during a modification that has not yet completed. Finally, durability rule ensures that a datum inserted in the shared space remains until a process explicitly deletes it. 85

active socket An active socket is connected to a remote socket via an open data connection. 93

architectural pattern An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. 42

binary semaphore A semaphore which has two operations (`P` and `V`) for controlling access by multiple processes to a common resource. The `P` operation sleeps the process until the resource controlled by the semaphore becomes available, at which time the resource is immediately claimed. The `V` operation is the inverse, it makes a resource available again after the process has finished using it. 94

bmp Image format used to store bitmap digital images. 210

byu Classic Brigham Young University file format for surfaces in 3D. 210

CLOS Common Lisp Object System (CLOS) [Gra96] is the facility for object-oriented programming which is part of ANSI Common Lisp. 20, 21, 48

component A part of a software system. A component has an interface that provides access to its services. On a programming language level components may be represented as modules, classes, objects or a set of related functions. 42, 53, 72

concurrency Property of systems in which several computations are executing simultaneously. 97

CORBA The Common Object Request Broker Architecture, a distributed object computing middleware standard defined by the Object Management Group (OMG), see [Gro]. 40

Curry Curry [Han06] is a logic *functional programming* language, based on the *Haskell* language. It merges elements of functional and logic programming. 113

event An action initiated either by the user or the computer. An example of a user event is any mouse movement or a keystroke. An example of an internally generated event is a notification based on the time of day. 327

event handler A software routine that provides the processing for various *event*s such as mouse movement, a mouse click, a keystroke or a spoken word. 114

expert system An expert system is an interactive computer-based decision tool that uses both facts and heuristics to solve difficult decision problems based on knowledge acquired from an expert. 60

framework A software system intended to be instantiated. A framework defines the architecture for a system. 42

functional programming A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. 20, 325

hash table Hash tables are data structures which use a hash function to map certain identifiers (keys) to associated values. 57

Haskell Haskell [Hut07] is a *functional programming* language. 325

IDE Integrated Development Environment (IDE). An editor that lets the programmer edit, compile and debug all from within the same program. 108, 114

interface A publicly accessible portion of a component, subsystem, or application. 53, 55

jpeg Image format used by digital cameras and other photographic image capture devices. 210

jvx Geometry file format for curves, surfaces and volumes in n-dimensional space. 210

Maple Maple is a general-purpose commercial computer algebra system. 39

Mathematica Mathematica is a computational software program used in scientific, engineering, and mathematical fields and other areas of technical computing. 39

MathML Mathematical Markup Language (MathML) [A+08] is a standard XML language adopted by the World Wide Web Consortium (W3C) as the approved way of expressing math on the web. 46

Maxima Maxima [Sch09] is a Lisp system for the manipulation of symbolic and numerical expressions, including differentiation, integration, Taylor series, Laplace transforms, ordinary differential equations, systems of linear equations, polynomials, and sets, lists, vectors, matrices, and tensors. 40

memoization An optimization technique used to speed up computer programs by storing the results of function calls for later reuse, rather than recomputing them at each invocation of the function. 30, 44

module A portion of a program that carries out a specific function and may be used alone or combined with other modules of the same program. 118

Native XML databases A data persistence software system that allows data to be stored in XML format; the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files. 85

obj Popular geometry format 3D geometry file format for surfaces in 3D originally used in Wavefront's Advanced Visualizer and now by Sun's Java3D. 210, 212, 216, 218

OMDoc OMDoc [Koh06] is an open markup language for mathematical documents, and the knowledge encapsulate in them. This format extends OpenMath and hence provides some features not available in OpenMath, for example a theory level and a way of incorporating executable code. 103

OpenMath OpenMath [Con04] is an XML standard for representing mathematical objects with their semantics, allowing them to be exchanged between computer programs, stored in databases, or published on the worldwide web. 46, 73

passive socket A passive socket is not connected, but rather awaits an incoming connection, which will spawn a new active socket. 93

pattern A pattern describes a problem that happens over and over, and also the solution to that problem. Patterns cover various ranges of scale and abstraction. Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components, or of the relationships between them. 41

pbm Portable Bit Map (PBM). The PBM format is a monochrome file format for images. 210–212, 215, 218

Phrasebook A piece of software which transforms from the internal representation of a data structure of an application to its OpenMath representation and viceversa. 45, 75

platform The software that a system uses for its implementations. Software platforms include operating systems, libraries, and frameworks. A platform implements a virtual machine with applications running on top of it. 42

plug-in A set of software components that adds specific capabilities to a larger software application. Plug-in based applications can be executed with no plug-ins. 101–104, 118

png Image format, that does not requiring a patent license, used to store bitmap digital images. 210

prefix notation Mathematical notation where the function is noted before the arguments it operates on. 35

process The actual running of a program module. 98

prompt A message on the computer screen indicating that the computer is ready to accept user input. In a command-line interface, the prompt may be a simple ">" symbol or "READY >" message, after which the user may type a command for the computer to process. 107

raster A raster is a rectangular grid of picture elements representing graphical data for display. 211

request An event sent by a client to a service provider asking it to perform some processing on the client's behalf. 45

semaphore A mechanism for controlling access by multiple processes to a common resource in a concurrent programming environment. 94

service A set of functionality offered by a service provider or server to its clients. 53, 72, 73

SOAP Simple Object Access Protocol is a standard protocol for exchanging structured information in the implementation of Web Services. 90

socket A mechanism for interprocess communication. A socket is an end-point of communication that identifies a particular network address and port number. 93, 134–136

stack overflow An error which occurs when too much memory is used on the data structure that stores information about the active computer program. 267

stylesheet A program used to render an XML document into another format. 115, 124

thread One subprocess in a multithreaded system. See multithreading. 98

UI User Interface. 113

web service A web service is traditionally defined by the World Wide Web Consortium as a software system designed to support interoperable machine-to-machine interaction over a network. 82, 90

well-formed XML XML that follows the XML tag rules listed in the W3C Recommendation for XML 1.0. A well-formed XML document contains one or more elements; it has a single document element, with any other elements properly nested under it; and each of the parsed entities referenced directly or indirectly within the document is well formed. 73, 75

XML enabled databases A data persistence software system that allows data to be stored in XML format; the internal model of such databases is a tradicional database (relational or object oriented). This databases do the conversion between XML and its internal representation. 85

XML schema A formal specification of element names that indicates which elements are allowed in an XML document, and in what combinations. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It defines whether an element is empty or can include text. The schema can also define default values for attributes. 46

XPath XML Path Language, is a query language for selecting nodes from an XML document. 85

XQuery XQuery is a query and functional programming language that is designed to query collections of XML data. 85

XSLT eXtensible Stylesheet Language Transformations (XSLT) [K$^+$07] are procedures, defined in XML, for converting one kind of XML into another. For viewing on the Web, an XSLT can be written for conversion to XHTML. 124

XUL XML User Interface [H$^+$00], it is Mozilla's XML-based user interface language which lets us build feature rich cross-platform applications defining all the elements of a User Interface. 113

Bibliography

[A$^+$08] R. Ausbrooks et al. Mathematical Markup Language (MathML). version 3.0 (third edition), 2008. `http://www.w3.org/TR/2008/WD-MathML3-20080409/`.

[Aas05] J. Aasman. Allegrocache: A high-performance object database for large complex problems. *In 5th International Lisp Conference. Standford University*, 2005.

[ABR08] J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4), pp. 271–292, 2008.

[ABR10] J. Aransay, C. Ballarin, and J. Rubio. Generating certified code from formal proofs: a case study in homological algebra. *Formal Aspects of Computing*, 2(22), pp. 193–213, 2010.

[AD09] J. Aransay and C. Domínguez. Modelling Differential Structures in Proof Assistants: The Graded Case. In *Proceedings 12th International Conference on Computer Aided Systems Theory (EUROCAST'2009)*, volume 5717 of *Lecture Notes in Computer Science*, pp. 203–210, 2009.

[ADFQ03] R. Ayala, E. Domínguez, A.R. Francés, and A. Quintero. Homotopy in digital spaces. *Discrete Applied Mathematics*, 125, pp. 3–24, 2003.

[ALR07] M. Andrés, L. Lambán, and J. Rubio. Executing in Common Lisp, Proving in ACL2. In *Proceedings 14th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2007)*, volume 4573 of *Lectures Notes in Artificial Intelligence*, pp. 1–12. Springer-Verlag, 2007.

[ALRRR07] M. Andrés, L. Lambán, J. Rubio, and J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. *Proceedings of ACL2 Workshop 2007*, pp. 34–39, 2007.

[APRR05] M. Andrés, V. Pascual, A. Romero, and J. Rubio. Remote Access to a Symbolic Computation System for Algebraic Topology: A Client-Server

Approach. In *Proceedings 5th International Conference on Computational Science (ICCS'2005)*, volume 3516 of *Lecture Notes in Computer Science*, pp. 635–642, 2005.

[B$^+$96] F. Buschmann et al. *Pattern-Oriented Software Architecture. A system of Patterns*, volume 1 of *Software Design Patterns*. Wiley, 1996.

[B$^+$07] F. Buschmann et al. *Pattern-Oriented Software Architecture. A Pattern Language for Distributed Computing*, volume 4 of *Software Design Patterns*. Wiley, 2007.

[B$^+$08] T. Bray et al. Extensible markup language (XML) 1.0 (fifth edition), 2008. `http://www.w3.org/TR/REC-xml/`.

[Bai99] A. Bailey. *The machine-checked literate formalisation of algebra in type theory*. PhD thesis, Manchester University, 1999.

[BGKM91] R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In V. Lifschiz, editor, *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honor of John McCarthy*, pp. 7–26. Academic Press, 1991.

[BH06] R. S. Boyer and W. A. Hunt. Function Memoization and Unique Object Representation for ACL2 Functions. In *Proceedings 6th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2006.

[Bil] A. Billingsley. DragMath. `http://www.dragmath.bham.ac.uk/`.

[BKM96] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'1996)*, volume 1166 of *Lecture Notes in Computer Science*, pp. 275–293, 1996.

[BM84] R. Boyer and J S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3), pp. 441–458, 1984.

[BM97] R. Boyer and J S. Moore. Nqthm, the Boyer-Moore theorem prover, 1997. `ftp.cs.utexas.edu/pub/boyer/nqthm/index.html`.

[Bra96] R. Bradford. An implementation of telos in common lisp. *Object Oriented Systems*, 3, pp. 31–49, 1996.

[Bro57] E. H. Brown, Jr. Finite computability of Postnikov complexes. *Annals of Mathematics*, 65(1), pp. 1–20, 1957.

[Bro67] R. Brown. The twisted Eilenberg-Zilber theorem. *Celebrazioni Archimedi de Secolo XX, Simposio di Topologia*, pp. 34–37, 1967.

[Bro82] K. S. Brown. *Cohomology of Groups.* Springer-Verlag, 1982.

[Bro97] B. Brock. defstructure for ACL2 Version 2.0. Technical report, Computational Logic, Inc., 1997.

[BS84] B. G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems The MYCIN Experiments of the Standford Heuristic Programming Project.* Addison-Wesley, 1984.

[C$^+$07] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0, 2007. `http://www.w3.org/standards/techs/wsdl#w3c_all`.

[Cal] The Calculemus Project. `http://www.calculemus.net/`.

[CC99] O. Caprotti and A. Cohen. Connecting proof checkers and computer algebra using OpenMath. In *Proceedings 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLS'1999)*, volume 1690 of *Lecture Notes in Computer Science*, pp. 109–112, 1999.

[CDDP04] O. Caprotti, J. Davenport, M. Dewar, and J. Padget. Mathematics on the (semantic) net. In *Proceedings 1st European Semantic Web Symposium (ESWS'2004)*, volume 3053 of *Lecture Notes in Computer Science*, pp. 213–224, 2004.

[CH97] J. Calmet and K. Homann. Towards the Mathematics Software Bus. *Theoretical Computer Science*, 187, pp. 221–230, 1997.

[Chr03] J. Chrzaszcz. Implementing Modules in the Coq System. In *Proceedings 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *Lecture Notes in Computer Science*, pp. 270–286, 2003.

[CM95] G. Carlsson and R. J. Milgram. *Handbook of Algebraic Topology*, chapter Stable homotopy and iterated loop spaces, pp. 505–583. North-Holland, 1995.

[CMDZ06] C. Cordero, A. De Miguel, E. Domínguez, and M. A. Zapata. Modelling interactive systems: an architecture guided by communication objects. *HCI related papers of Interaction 2004*, pp. 345–357, 2006.

[CoC] CoCoATeam. CoCoA: a system for doing Computations in Commutative Algebra. `http://cocoa.dima.unige.it`.

[Con04] The OpenMath Consortium. Openmath standard 2.0, 2004. `http://www.openmath.org/standard/om20-2004-06-30/omstd20.pdf`.

[CRZ03] A. B. Chaundry, A. Rashid, and R. Zicari. *XML data management: native XML and XML-enabled database systems.* Addison-Wesley, 2003.

[Dav99] J. Davenport. A small openmath type system. University of Bath, 1999. http://www.openmath.org/standard/sts.pdf.

[Dav00] J. Davenport. On writing openmath content dictionaries. Technical report, University of Bath, 2000. www.openmath.org/documents/writingCDs.pdf.

[DLR07] C. Domínguez, L. Lambán, and J. Rubio. Object oriented institutions to specify symbolic computation systems. *Rairo- Theoretical Informatics and Applications*, 41, pp. 191–214, 2007.

[DMMV07] P. Dillinger, P. Manolios, J S. Moore, and D. Vroon. ACL2s: "The ACL2 Sedan". 2007.

[Doe98] J. P. Doeraene. Homotopy pull backs, homotopy push outs and joins. *Bulletin of the Belgian Mathematical Society*, 5, pp. 15–37, 1998.

[DR11] C. Domínguez and J. Rubio. Effective Homology of Bicomplexes, formalized in Coq. *Theoretical Computer Science*, 412, pp. 962–970, 2011.

[DRSS98] X. Dousson, J. Rubio, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/.

[DST93] J. Davenport, Y. Siret, and E. Tournier. *Computer Algebra.* Academic Press, 1993.

[DSW05] M. Dewar, E. Smirnova, and S. M. Watt. XML in Mathematical Web Services. In *Proceedings of XML 2005 Conference Syntax to Semantics (XML'2005)*, 2005.

[Dus06] A. Duscher. Interaction patterns of Mathematical Services. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler Univeristy, Linz, 2006.

[DZ07] E. Domínguez and M. A. Zapata. Noesis: Towards a situational method engineering technique. *Information Systems*, 32(2), pp. 181–222, 2007.

[E$^+$07] B. Evjen et al. *Professional XML.* Wiley Publishing Inc., 2007.

[Ell09] G. Ellis. HAP package for GAP, 2009. http://www.gap-system.org/Packages/hap.html.

[EZ50] S. Eilenberg and J. A. Zilber. Semi-simplicial complexes and singular homology. *Annals of Mathematics*, 51(3), pp. 499–513, 1950.

[F$^+$08] S. Freundt et al. Symbolic Computation Sofware Composability. In *Proceedings 7th International Conference on Mathematical Knowledge Management (MKM'2008)*, volume 5144 of *Lectures Notes in Computer Science*, pp. 285–295. Springer-Verlag, 2008.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *Javaspaces. Principles, Patterns, and Practice.* Addison Wesley, 1999.

[FHK$^+$09] S. Freundt, P. Horn, A. Konovalov, S. Lindon, and D. Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specification, version 1.3, 2009. `http://www.symbolic-computation.org/scscp`.

[G$^+$] M. Gudgin et al. Soap version 1.2 specification. `http://www.w3.org/TR/soap12-part1/`.

[G$^+$08] D. A. Greve et al. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(01), pp. 15–46, 2008.

[GAP] GAP - Groups, Algorithms, Programming - System for Computational Discrete algebra. `http://www.gap-system.org`.

[GDMRSP05] R. González-Díaz, B. Medrano, P. Real, and J. Sánchez-Peláez. Algebraic Topological Analysis of Time-Sequence of Digital Images. In *Proceedings 8th International Conference on Computer Algebra in Scientific Computing (CASC'2005)*, volume 3718, 2005.

[GDR05] R. González-Díaz and P. Real. On the Cohomology of 3D Digital Images. *Discrete Applied Math*, 147(2-3), pp. 245–263, 2005.

[Gel85] D. Gelenter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7, pp. 80–112, 1985.

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2009)*, volume 5674 of *Lecture Notes in Computer Science*, pp. 327–342, 2009.

[GJ94] P. G. Goerss and J. F. Jardine. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GL$^+$09] M. J. González-López et al. TutorMates, 2009. `http://www.tutormates.es`.

[GPWZ02] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4), pp. 271–286, 2002.

[GR05] J. C. Giarratano and G. D. Riley. *Expert Systems: Principles and Programming.* PWS Publishing Company, 2005.

[Gra96] P. Graham. *ANSI Common Lisp.* Prentice Hall, 1996.

[Gro] Object Management Group. Common object request broker architecture (corba). `http://www.omg.org`.

[Gro09] The Object Management Group. UML specification version 2.2, 2009. http://www.omg.org.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/.

[H$^+$00] D. Hyatt et al. XML User Interface Language (xul) 1.0, 2000. http://www.mozilla.org/projects/xul/.

[Hac01] M. Hachimori. Simplicial complex library, 2001. http://infoshako.sk.tsukuba.ac.jp/~hachi/math/library/index_eng.html.

[Han06] M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. http://www.curry-language.org.

[Hat02] A. Hatcher. *Algebraic Topology.* Cambridge University Press, 2002. http://www.math.cornell.edu/~hatcher/AT/ATpage.html.

[Her] L. J. Hernández. Orografía homológica cúbica para la minería de datos. Technical report, University of La Rioja. http://www.unirioja.es/cu/luhernan/datamining.

[Her09] J. Heras. The *fkenzo* program. University of La Rioja, 2009. http://www.unirioja.es/cu/joheras.

[Her11] J. Heras. PhD thesis, University of La Rioja, 2011. http://www.unirioja.es/cu/joheras/Thesis.

[HK09] M. Hanus and C. Kluß. Declarative Programming of User Interfaces. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL'2009)*, volume 5418 of *Lectures Notes in Computer Science*, pp. 16–30. Springer-Verlag, 2009.

[HT98] J. Harrison and L. Théry. A skeptic approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21, pp. 279–294, 1998.

[Hur35] W. Hurewicz. Beiträge zur Topologie der Deformationen I-II. *Proceedings of the Akademie van Wetenschappen*, 38, pp. 112–119, 521–528, 1935.

[Hur36] W. Hurewicz. Beiträge zur Topologie der Deformationen III-IV. *Proceedings of the Akademie van Wetenschappen*, 39, pp. 117–126, 215–224, 1936.

[Hut07] G. Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[HW67] P. J. Hilton and S. Wylie. *Homology Theory.* Cambridge University Press, 1967.

[IAM] Internet Accessible Mathematical Computation (iamc). `http://icm.mcs.kent.edu/research/iamc.html`.

[Inca] Franz Inc. Allegro CL Socket Library. `http://www.franz.com/support/documentation/current/doc/socket.htm`.

[Incb] Franz Inc. Allegro Common Lisp. `http://www.franz.com/`.

[Incc] Franz Inc. jLinker - A Dynamic Link between Lisp and Java. `http://www.franz.com/support/documentation/current/doc/jlinker.htm`.

[Incd] Franz Inc. A soap 1.1 api for allegro cl. `http://www.franz.com/support/documentation/current/doc/soap.htm`.

[Inc05] Franz Inc. Common Lisp for Service Oriented Architecture Programs. Technical report, Franz Inc., 2005. `http://www.franz.com/resources/educational_resources/white_papers/CL_for_SOA.pdf`.

[Ini03] The Open Services Gateway Initiative. Osgi service platform, release 3, March 2003. `http://www.osgi.org`.

[Jac95] P. Jackson. *Enhancing the Nuprl proof-development system and applying it to computational abstract algebra.* PhD thesis, Cornell University, 1995.

[K$^+$07] M. Kay et al. XSL transformations (xslt) version 2.0, 2007. `http://www.w3.org/TR/xslt20/`.

[Kau00] M. Kaufmann. *Computer-Aided Reasoning: ACL2 Case Studies*, chapter Modular proof: the fundamental theorem of calculus. Kluwer Academic Publishers, 2000.

[KBN04] S. J. Koyani, R. W. Bailey, and J. R. Nall. *Research-Based Web Design and Usability Guidelines.* U.S. Dept. of Health and Human Services, 2004.

[KKL] V. Komendantsky, A. Konovalov, and S. Linton. Interfacing Coq + SS-Reflect with GAP. `http://www.cs.st-andrews.ac.uk/vk/Coq+GAP/`.

[KL09] A. Konovalov and S. Lindon. GAP package SCSCP, 2009. `http://www.gap-system.org/Packages/scscp.html`.

[KM] M. Kaufmann and J S. Moore. ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`.

[KM01a] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2), pp. 161–203, 2001.

[KM01b] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2), pp. 161–203, 2001.

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An approach.* Kluwer Academic Publishers, 2000.

[Koh06] M. Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2].* Springer Verlag, 2006.

[KSN10] K. Kofler, P. Schodl, and A. Neumaier. Limitations in Content MathML. Technical report, University of Vienna, Austria, 2010. `http://www.mat.univie.ac.at/~neum/FMathL/content-mathml-limitations.pdf`.

[LBFL80] R. K. Lindsay, B. G. Buchanan, E. A. Feigenbaum, and J. Lederberg. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill Companies, Inc, 1980.

[Lip81] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Benjamin/Cummings Publishing Company, Inc., 1981.

[LMMRRR10] L. Lambán, F. J. Martín-Mateos, J. Rubio, and J. L. Ruíz-Reina. When first order is enough: the case of Simplicial Topology. Preprint. `http://wiki.portal.chalmers.se/cse/uploads/ForMath/wfoe`, 2010.

[LPR99] L. Lambán, V. Pascual, and J. Rubio. Specifying Implementations. In *Proceedings 12th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (ISSAC'1999)*, pp. 245–251, 1999.

[LPR03] L. Lambán, V. Pascual, and J. Rubio. An object-oriented interpretation of the EAT system. *Applicable Algebra in Engineering, Communication and Computing*, 14, pp. 187–215, 2003.

[Mac63] S. MacLane. *Homology*, volume 114 of *Grundleren der Mathematischen Wissenschaften.* Springer, 1963.

[Mac03] D. Mackenzie. Topologists and Roboticists Explore and Inchoate World. *Science*, 8, pp. 756, 2003.

[Mah67] M. E. Mahowald. *The Metastable Homotopy of* S^n, volume 72 of *Memoirs of the American Mathematical Society.* 1967.

[MAP] MAP network. `http://map.disi.unige.it/`.

[Mata] MathBroker: A Framework for Brokering Distributed Mathematical Services. `http://www.risc.uni-linz.ac.at/projects/basic/mathbroker/`.

[Matb] MathBroker II: Brokering Distributed Mathematical Services. `http://www.risc.uni-linz.ac.at/projects/mathbroker2/`.

[Matc] MathServe Framework. `http://www.ags.uni-sb.de/~jzimmer/mathserve.html`.

[MATd] MATHWEB-SB: A Software Bus for MathWeb. `http://www.ags.uni-sb.de/~jzimmer/mathweb-sb/`.

[Mat76] M. Mather. Pull-Backs in Homotopy Theory. *Canadian Journal of Mathematics*, 28(2), pp. 225–263, 1976.

[Mau96] C.R.F. Maunder. *Algebraic Topology.* Dover, 1996.

[May67] J. P. May. *Simplicial objects in Algebraic Topology*, volume 11 of *Van Nostrand Mathematical Studies.* 1967.

[McD82] J. McDermott. R1: A rule based configurer of computer systems. *Artificial Intelligence*, 19(1), 1982.

[Mic99] Sun Microsystems. JavaMail Guide for Service Providers, 1999. `http://www.oracle.com/technetwork/java/index-138643.html`.

[MKM] The Mathematical Knowledge Management Interest Group. `http://www.mkm-ig.org/`.

[MlBR07] E. Mata, P. Álvarez, J. A. Bañares, and J. Rubio. Formal Modelling of a Coordination System: from Practice to Theory and back again. In *Proceedings 7th Annual International Workshop Engineering Societies in the Agents World (ESAW'2006)*, volume 4457 of *Lectures Notes in Artificial Intelligence*, pp. 229–244. Springer-Verlag, 2007.

[MMAHRR02] F. J. Martín-Mateos, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz-Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. *Proceedings of the Third ACL2 workshop. Grenoble, Francia*, pp. 188–203, 2002.

[MMRRR09] F. J. Martín-Mateos, J. Rubio, and J. L. Ruiz-Reina. ACL2 verification of simplicial degeneracy programs in the kenzo system. In *Proceedings 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2009)*, volume 5625 of *Lectures Notes in Computer Science*, pp. 106–121. Springer-Verlag, 2009.

[MMS03] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development. In *Proceedings International Conference NetObject-Days (NODe'2002)*, volume 2591 of *Lecture Notes in Computer Science*, pp. 87–102. Springer-Verlag, 2003.

[Mon] MoNET: Mathematics on the Net. http://monet.nag.co.uk/cocoon/monet/index.html.

[Moz] mozilla.org: An introduction to hacking mozilla. http://www.mozilla.org/hacking/coding-introduction.html.

[MPRR10] I. Medina, F. Palomo, and J. L. Ruiz-Reina. A verified Common Lisp implementation of Buchberger's algorithm in ACL2. *Journal of Symbolic Computation*, 45(1), pp. 96–123, 2010.

[Nie94] J. Nielsen. *Usability Inspection Methods*, chapter Heuristic evaluation. John Wiley & Sons, New York, NY, 1994.

[NSS59] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pp. 256–264, 1959.

[OS03] D. Orden and F. Santos. Asymptotically efficient triangulations of the d-cube. *Discrete and Computational Geometry*, 30(4), pp. 509–528, 2003.

[OTI03] Inc Object Technology International. Eclipse platform technical overview., February 2003. http://www.eclipse.org.

[P$^+$02] K. Polthier et al. JavaView - Interactive 3D Geometry and Visualization. Multimedia Tools for Communicating Mathematics, 2002. http://www.javaview.de/index.html.

[Pau96] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[Pec08] A. Peck. *Beginning GIMP: From Novice to Professional.* Apress, 2008. http://gimpbook.com/.

[Poi95] H. Poincaré. Analysis Situs. *Journal de l'École Polytechnique*, 1, pp. 1–121, 1895.

[Rav86] D. C. Ravenel. *Complex cobordism and stable homotopy groups of spheres.* Academic Press, 1986.

[Rea94] P. Real. Sur le calcul des groupes d'homotopie. *C. R. Acad. Sci. Paris Sér. I Math.*, 319(5), pp. 475–478, 1994.

[RER09] A. Romero, G. Ellis, and J. Rubio. Interoperating between computer algebra systems: computing homology of groups with kenzo and gap. In *Proceedings 34th International Symposium on Symbolic and Algebraic Computation (ISSAC'2009)*, pp. 303–310, 2009.

[Rey98] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4), pp. 363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[Rom10] A. Romero. Effective homology and discrete morse theory for the computation of homology of groups. Mathematics Algorithms and Proofs, 2010. `http://www.unirioja.es/dptos/dmc/MAP2010/Slides/Slides/talkRomeroMAP2010.pdf`.

[RRS06] A. Romero, J. Rubio, and F. Sergeraert. Computing spectral sequences. *Journal of Symbolic Computation*, 41(10), pp. 1059–1079, 2006.

[RS97] J. Rubio and F. Sergeraert. Constructive Algebraic Topology, Lecture Notes Summer School on Fundamental Algebraic Topology. Institut Fourier, Grenoble, 1997. `http://www-fourier.ujf-grenoble.fr/~sergerar/Summer-School/`.

[RS02] J. Rubio and F. Sergeraert. Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques*, 126(5), pp. 389–412, 2002.

[RS06] J. Rubio and F. Sergeraert. Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs. University of Genova, 2006. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Genova-Lecture-Notes.pdf`.

[RSS90] J. Rubio, F. Sergeraert, and Y. Siret. EAT: Symbolic Software for Effective Homology Computation. Institut Fourier, Grenoble, 1990. `ftp://fourier.ujf-grenoble.fr/pub/EAT`.

[Rus92] D. Russinoff. A mechanical prof of quadratic reciprocity. *Journal of of Automated Reasoning*, 8(1), pp. 3–21, 1992.

[Rus98] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1, pp. 148–200, 1998.

[SC09] A. Solomon and M. Costantini. GAP package OpenMath, 2009. `http://www.gap-system.org/Packages/openmath.html`.

[Sch97] F. B. Schneider. *On Concurrent Programming*. Springer, 1997.

[Sch09] W. Schelter. Maxima v. 5.18.1, 2009. `http://www.maxima.sourceforge.net/index.shtml`.

[Ser80] J. P. Serre. *Trees*. Springer, 1980.

[Ser87] F. Sergeraert. Homologie effective. *Comptes Rendus des Séances de l'Academie des Sciences de Paris*, 304(11 and 12), pp. 279–282 and 319–321, 1987.

[Ser90] F. Sergeraert. Functional coding and effective homology. *Astérisque*, 192, pp. 57–67, 1990.

[Ser92] F. Sergeraert. Effective homology, a survey. Technical report, Institut Fourier, 1992. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Survey.pdf`.

[Ser94] F. Sergeraert. The computability problem in Algebraic Topology. *Advances in Mathematics*, 104(1), pp. 1–29, 1994.

[Ser01] F. Sergeraert. Common Lisp, Typing and Mathematics. Technical report, University of La Rioja, 2001. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Ezcaray.pdf`.

[Ser10] F. Sergeraert. *Contribuciones científicas en honor de Mirian Andrés Gómez*, chapter Triangulations of complex projective spaces, pp. 507–519. 2010.

[SGF03] F. Ségonne, E. Grimson, and B. Fischl. Topological Correction of Subcortical Segmentation. In *Proceedings 6th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI'2003)*, volume 2879 of *Lecture Notes in Computer Science*, pp. 695–702, 2003.

[Sha88] N. Shankar. A mechanical proof of the church-rosser theorem. *Journal of Association for Computing Machinery*, 35(3), pp. 475–522, 1988.

[Sha94] N. Shankar. *Metamathematics, Machines, and Godel's Proof.* Cambridge University Press, 1994.

[Shi62] W. Shih. Homologie des espaces fibrés. *Publications mathématiques de l'Institut des Hautes Études Scientifiques*, 13, pp. 1–88, 1962.

[Slu07] T. A. Sluga. *Modern C++ Implementation of the LINDA coordination language.* PhD thesis, University of Hannover, 2007.

[SSW04] E. Smirnova, C. M. So, and S. M. Watt. An architecture for distributed mathematical web services. In *Proceedings 3rd International Conference on Mathematical Knowledge Management (MKM'2004)*, volume 3119 of *Lecture Notes in Computer Science*, pp. 363–377, 2004.

[Sta81] R. Stallman. Emacs: The extensible, customizable display editor. In *ACM Conference on Text Processing*, 1981.

[Ste] W. Stein. SAGE mathematical software system. `http://www.sagemath.org`.

[SvdW10] B. Spitters and E. van der Weegen. Developing the Algebraic Hierarchy with Type Classes in Coq. In *Proceedings International Conference on Interactive Theorem Proving (ITP'2010)*, volume 6172 of *Lecture Notes in Computer Science*, pp. 490–493, 2010.

[Tec] Wavefront Technologies. Object files (.obj). `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/`.

[Tod62] H. Toda. *Compositional methods in homotopy groups of spheres.* Princeton University Press, 1962.

[Tur36] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings London Mathematical Society*, 42, pp. 230–265, 1936.

[Veb31] O. Veblen. *Analysis Situs.* AMS Coll. Publ., 1931.

[Wey80] R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence Journal*, 13(1), pp. 133–170, 1980.

[Whi78] G. W. Whitehead. *Elements of Homotopy Theory*, volume 61 of *Graduate texts in Mathematics.* Springer-Verlag, 1978.

[Woo89] J. Wood. Spinor groups and algebraic coding theory. *Journal of Combinatorial Theory*, 50, pp. 277–313, 1989.