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 \textit{fKenzo}. Since it would be too pretentious to try to fully solve 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 \textit{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 ones 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 has consisted not only in having several Computer Algebra systems and Theorem Prover tools, and use them individually by means of a common GUI, but also in 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 \textit{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 \textit{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 \textit{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 $F_\ast$ 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 $F_\ast$ is said to be a free resolution.

Given a free resolution $F_\ast$, one can consider the chain complex of $\mathbb{Z}$-modules (that is to say, abelian groups) $C_\ast = (C_n, d_{C_n})_{n \in \mathbb{N}}$ defined by

$$C_n = (\mathbb{Z} \otimes_{\mathbb{Z}G} F_\ast)_n, \quad n \geq 0$$
(where \( \mathbb{Z} \equiv \mathcal{C}_*(\mathbb{Z}, 0) \) is the chain complex with only one non-null \( \mathbb{Z} \mathbb{G} \)-module in dimension 0) with differential maps \( d_{n} : C_{n} \rightarrow C_{n-1} \) induced by \( d_{n} : F_{n} \rightarrow F_{n-1} \).

Although the chain complex of \( \mathbb{Z} \mathbb{G} \)-modules \( F_{*} \) is acyclic, \( C_{*} = \mathbb{Z} \otimes_{\mathbb{Z} \mathbb{G}} F_{*} \) 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

\[
H_{n}(\mathbb{Z} \otimes_{\mathbb{Z} \mathbb{G}} F_{*}) \cong H_{n}(\mathbb{Z} \otimes_{\mathbb{Z} \mathbb{G}} F_{*}'), \text{ for all } n \in \mathbb{N}.
\]

This theorem leads to the following definition.

**Definition 4.3.** Given a group \( G \), the homology groups \( H_{n}(G) \) are defined as

\[
H_{n}(G) = H_{n}(\mathbb{Z} \otimes_{\mathbb{Z} \mathbb{G}} F_{*})
\]

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 \( G \)

\[
\cdots \xrightarrow{t} \mathbb{Z} \mathbb{G} \xrightarrow{N} \mathbb{Z} \mathbb{G} \xrightarrow{t} \mathbb{Z} \rightarrow 0,
\]

where \( N \) denotes the norm element \( 1 + t + \cdots + t^{m-1} \) of \( \mathbb{Z} \mathbb{G} \), produces the chain complex of abelian groups

\[
\cdots \xrightarrow{0} \mathbb{Z} \xrightarrow{m} \mathbb{Z} \xrightarrow{0} \mathbb{Z} \rightarrow 0
\]

and therefore

\[
H_{i}(G) = \begin{cases} 
\mathbb{Z} & \text{if } i = 0 \\
\mathbb{Z}/m\mathbb{Z} & \text{if } i \text{ is odd, } i > 0 \\
0 & \text{if } i \text{ is even, } i > 0
\end{cases}
\]

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 and 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]
```

The result must 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 cases 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:

```xml
<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 sub-subsections.

#### 4.1.3.1 Extending the XML-Kenzo schema

We want to introduce new functionality in our framework 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 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
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 based on the use of a GAP/HAP remote server.

#### 4.1.3.2.1 A remote server: GAP/HAP + SCSCP

Our framework is connected with a GAP/HAP server. The connection with this 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 addi-
tional 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 XML-Kenzo representation to the
SCSCP OpenMath one and viceversa. It is worth noting that the Phrasebook included
in this component is not the same Phrasebook implemented 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:

```xml
<?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):

```xml
<OMATP>
  <OMS cd="scscp1" name="call_id"/>
  <OMSTR>esus.unirioja.es:7500</OMSTR>
</OMATP>
```

the invocation of the GAP/HAP service by means of:
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:

```xml
<OMA>
  <OMS cd="group1" name="cyclic"/>
  <OMI>5</OMI>
</OMA>
```

and, finally, the end of the SCSCP request (&lt;?scscp end ?&gt;).

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:

```xml
<constructor>
  <cyclic>5</cyclic>
</constructor>
```

the `cyclic-groups` module is triggered.

When the `cyclic-groups` module is activated two situations are feasible: (1) a new object 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 constraint 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 class), which was specialized for Kenzo spaces (MK-SPACE-KENZO class). 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:

\[
\text{(DEFCLASS MK-GAP (MK-OBJECT) ())}
\]

In turn, we specialize this class to represent GAP groups.

\[
\text{(DEFCLASS MK-GROUP-GAP (MK-GAP))}
\]

This class has nine slots in addition to the idnm and orgn slots of the MK-OBJECT class:

1. \textit{ia}, a boolean, indicates if the group is abelian.
2. \textit{ic}, a boolean, indicates if the group is cyclic.
3. \textit{iea}, a boolean, indicates if the group is elementary abelian.
4. \textit{ing}, a boolean, indicates if the group is nilpotent.
5. \textit{ipg}, a boolean, indicates if the group is perfect.
6. \textit{isg}, a boolean, indicates if the group is solvable.
7. \texttt{ipcg}, a boolean, indicates if the group is polycyclic.

8. \texttt{issg}, a boolean, indicates if the group is supersolvable.

9. \texttt{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 \texttt{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 \texttt{*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 \texttt{gap-id} XML-Kenzo object.
   
   (b) Otherwise, go to step 2.

2. Construct an instance of the \texttt{MK-GROUP-GAP} class where:
   
   • \texttt{idnm} is automatically generated (remember that \texttt{idnm} is the object identifier in the microkernel).
   
   • the XML-Kenzo object received as input is assigned to \texttt{orgn}.
   
   • the information of the rest of the slots is obtained invoking the GAP/HAP internal server.

3. Push the object in the \texttt{*object-list*} list of already created objects.

4. Return its identification number and the properties about the group in a \texttt{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 \texttt{*object-list*} list in the internal memory.

4.1.3.4 Enhancing the homology computation module

The \texttt{gap-homology-m.lisp} file contains the Common Lisp functions which allow the plug-in framework to extend the \texttt{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 \texttt{homology} module of the microkernel was explained. The \texttt{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:

\begin{verbatim}
(DEFGENERIC compute-homology (mk-object n))
\end{verbatim}

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:

\begin{verbatim}
(DEFMETHOD compute-homology ((space mk-space-kenzo) n) ...)
\end{verbatim}

This method invokes the Kenzo internal server from an \texttt{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 \texttt{gap-homology-m.lisp} file defines the method:

\begin{verbatim}
(DEFMETHOD compute-homology ((group mk-group-gap) n) ...)
\end{verbatim}

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 \texttt{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:

\begin{verbatim}
(DEFGENERIC check-constraints (mk-object))
\end{verbatim}
In the case of dealing with \text{mk-space-kenzo} instances we have the method:

\begin{verbatim}
(DEFMETHOD check-constraints ((space mk-space-kenzo))
  (if (>= (rede space) 0) t nil))
\end{verbatim}

On the contrary, the method implemented for the case of \text{mk-group-gap} instances is the following simple method:

\begin{verbatim}
(DEFMETHOD check-constraints ((group mk-group-gap)) t)
\end{verbatim}

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 group homology is 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 \text{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 \text{gap-id} object, in the \text{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:

\begin{verbatim}
<constructor>
  <cyclic>5</cyclic>
</constructor>
\end{verbatim}

is generated by the adapter when the following OpenMath request is received:

\begin{verbatim}
<OMOBJ>
  <OMA>
    <OMS cd="groupname1" name="cyclic"/>
    <OMI>5</OMI>
  </OMA>
</OMOBJ>
\end{verbatim}

Therefore, the \text{gap-a.lisp} file contains the new parsers and a list with both \text{groupname1} and \text{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 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 previously performed. 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:

```xml
<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 triggered. 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:
4.1 Integration of the GAP Computer Algebra system

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 information.

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:

This result is converted by the gap-invoker into the following XML-Kenzo 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:

\[
\text{<OMOBJ>}
\text{  <OMA>}
\text{   \text{<OMS cd="ringname" name="Zm"/>}}
\text{   <OMI>5</OMI>}
\text{  </OMA>}
\text{ </OMOBJ>}
\]

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.

### 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.

\[
\text{<code id="spaces-properties">}
\text{  \text{<data format="Kf/external-server"> XML-Kenzo.xsd </data>}}
\text{  \text{<data format="Kf/microkernel"> properties-mk.lisp </data>}}
\text{  \text{<data format="Kf/adapter"> properties-a.lisp </data>}}
\text{ </code>}
\]

These resources deserve a detailed explanation that is provided in the following paragraphs.
First of all, we want to modify the \textit{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 \textit{id} which includes an attribute called \texttt{properties}, see Figure 4.5.

The \texttt{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 \textit{id} XML-Kenzo objects returned by these modules not only contain the identifiers but also some properties. For instance, when the \textbf{Cartesian Product} module is activated and both components of the Cartesian product are contractible spaces, the returned object is:

\begin{verbatim}
<id properties="The space is contractible because is the cartesian product of two contractible spaces"> 1 </id>
\end{verbatim}

It is worth noting that the rest of the modules of the microkernel return \textit{id} XML-Kenzo objects whose \texttt{properties} attribute is empty.

Finally, the \texttt{properties-a.lisp} file includes a new parser in the Phrasebook to be able to handle the new specification of \textit{id} XML-Kenzo objects.

\subsection{Integration of GAP/HAP in the \textit{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 \textit{fKenzo} GUI to provide support for the new functionality are explained.

In this case we have defined a fresh \textit{fKenzo} module to enhance the GUI with support for the GAP/HAP system. The new OMDoc module references three files: \texttt{gap-structure} (that defines the structure of the graphical constituents), \texttt{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 the \texttt{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. In general, when we define a new window, we do it by means of an instance of a generic specification.

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 the 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 is included in the central panel. This new graphical element is included whenever a
4.1 Integration of the GAP Computer Algebra system

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:
This class has three slots:

1. \texttt{idnm}, an integer, identifier for the object in \textit{fKenzo}. This is the value assigned when a space is constructed.

2. \texttt{orgn}, a string containing the OpenMath object that is the origin of the object.

3. \texttt{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 \texttt{*FKENZO-OBJECTS*} which is used to generate the shown list to the user in the left side of the \textit{fKenzo} GUI.

Moreover, there is a function, called \texttt{fK}, which allows us to get information about the \textit{n}-th \textit{fKenzo} GUI object. The \texttt{fK} function takes as argument a natural number \textit{n} and returns the \textit{n}-th \textit{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 \texttt{FKENZO-OBJECT} by means of the \texttt{fk} function. Subsequently, it invokes a Common Lisp method \cite{Gra96} associated with that \texttt{FKENZO-OBJECT} which shows the mathematical notation of the object in the bottom part of the right side of the \texttt{fKenzo} GUI. Moreover, if the value of the \texttt{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 \texttt{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 \texttt{fKenzo}, we have specialized the \texttt{FKENZO-OBJECT} with two different subclasses. On the one hand, we have defined a new subclass of the \texttt{FKENZO-OBJECT} class; to store a name given by the user to identify the object. Namely, the new class, \texttt{FKENZO-OBJECT-NAME}, whose definition is:

\begin{verbatim}
(DEFCCLASS FKENZO-OBJECT-NAME (FKENZO-OBJECT)
  ;; NAME
  (name :type string :initarg :name :reader name)
)
\end{verbatim}

This is a subclass of the \texttt{FKENZO-OBJECT} class with just one additional slot, \texttt{name}, which provides a name for the object. The \texttt{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 \texttt{fKenzo}, for instance a simplicial set built from a list of its elements.

On the other hand, we have defined a new subclass of the \texttt{FKENZO-OBJECT} class; to store the path of a file associated with the object. Namely, the new class, \texttt{FKENZO-OBJECT-FILE}, whose definition is:

\begin{verbatim}
(DEFCCLASS FKENZO-OBJECT-FILE (FKENZO-OBJECT)
  ;; FILE
  (file :type string :initarg :file :reader idnm)
)
\end{verbatim}

This is a subclass of the \texttt{FKENZO-OBJECT} class with just one additional slot, \texttt{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.

\subsection{Behavior of the objects in the \texttt{fKenzo} GUI}

It is worth noting that in spite of belonging to different classes, all the instances constructed in \texttt{fKenzo} are stored in the \texttt{*FKENZO-OBJECTS*} list, which is 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:

\[
\text{(DEFGENERIC show-object (object))}
\]

This generic function have several methods to adapt it to specific cases. The main
method is associated with the FKENZO-OBJECT class:

\[
\text{(DEFMETHOD show-object ((object FKENZO-OBJECT))}
\text{\hspace{1em} (show-mathematical-notation (orgn object))}
\text{\hspace{1em} (if (props object) (show-description (props object))))}
\]

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:

\[
\text{(DEFMETHOD show-object ((object FKENZO-OBJECT-NAME))}
\text{\hspace{1em} (show-name (name object))}
\text{\hspace{1em} (if (props object) (show-description (props object))))}
\]

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 \textit{fKenzo}, too) consists in 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 \cite{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, certain 2-types and central extensions, as can be seen in \cite{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 \cite{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 \textit{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 in considering the bar resolution $B_* = \text{Bar}_*(G)$ (explained, for instance, in \cite{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 $C_* (K(G, 1)) \Leftrightarrow E_*$ 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 $C_* (K(G, 1)) \Leftrightarrow E_*$ 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. Particularly 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 $f\text{Kenzo}$ 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:

```xml
<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/adapter"> 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
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.

```xml
<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:
4.2 Interoperability between Kenzo and GAP/HAP

which asks the construction of a resolution for the cyclic group of dimension 5, the GAP server executes the instruction:

\[
gap> \text{ResolutionFiniteGroup(CyclicGroup(5));}
\]

Resolution in characteristic 0 for <pc group of size 5 with 1 generators>

Subsequently, the GAP server transforms the result to its OpenMath format.

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 \texttt{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] which 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:
the \texttt{k-g-1} module is activated.

When the \texttt{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 \texttt{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 \texttt{*object-list*} list if the object was built previously.
   (a) If the object was built previously its identification number, in an \texttt{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 \texttt{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 \texttt{k-g-1} XML-Kenzo object and the resolution obtained in the previous step as arguments.
5. Construct an instance of the \texttt{mk-space-k-g} class (see Subsubsection 2.2.3.1) where:
   \begin{itemize}
   \item the value of the slot \texttt{rede} is 0,
   \item \texttt{idnm} is automatically generated,
   \item \texttt{kidnm} is the value returned by the Kenzo internal server in the previous step,
   \item the XML-Kenzo object received as input is assigned to \texttt{orgn},
   \item the value of the slot \texttt{iter} is 1, and
   \item the value of the slot \texttt{group} is the dimension of the cyclic group.
   \end{itemize}
6. Push the object in the \texttt{*object-list*} list of already created spaces.
7. Return the \texttt{idnm} of the object in an \texttt{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:

```xml
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

is generated by the adapter when the following OpenMath request is received:

```xml
<OMOBJ>
  <OMA>
    <OMS cd="ASG" name="k-g-1"/>
    <OMA>
      <OMS cd="groupname1" name="cyclic"/>
      <OMI>5</OMI>
    </OMA>
  </OMA>
</OMOBJ>
```
4.2 Interoperability between Kenzo and GAP/HAP

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>
  </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:
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.

```xml
<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.
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"/>
<OMATP>
</OMOBJ>
```

This resolution is used by the \textit{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 \textit{homotopy} module.

Afterwards, a \textit{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 \textit{mk-space-k-g}, the \textit{HES} is activated to compute the homotopy group.
of the space. The HES uses its rules and returns the result:

\[
\text{The space was the Eilenberg MacLane space } K(C_5,1). \text{ The homotopy groups of an Eilenberg MacLane space } K(G,m) \text{ are: } \pi_m(K(G,m)) = G \text{ and } \pi_r(K(G,m)) = 0 \text{ if } m \neq r.
\]

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:

\[
\text{The space was the Eilenberg MacLane space } K(C_5,1). \text{ The homotopy groups of an Eilenberg MacLane space } K(G,m) \text{ are: } \pi_m(K(G,m)) = G \text{ and } \pi_r(K(G,m)) = 0 \text{ if } m \neq r.
\]

and this is the result returned to the client.

### 4.2.3 Composability of Kenzo and GAP/HAP in the \textit{fKenzo} GUI

This subsection is devoted to present the necessary resources to extend the \textit{fKenzo} GUI to provide support for the new functionality presented in the previous subsection.

In this case we have modified the GAP \textit{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: \textit{gap-structure} (that defines the structure of the graphical constituents), \textit{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 \textit{gap-structure} file:

- A menu option called \textit{K-G-1} into the menu \textit{Abelian Simplicial Groups}. 

A window called $K\times G\times 1$ (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 $K\times G\times 1$ menu option; this function shows the $K\times G\times 1$ 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 $K\times G\times 1$ 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 $K\times G\times 1$ 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 $G$,
Figure 4.12: Homology groups of $K(C_5, 1)$ and $B(K(C_5, 1))$

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 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 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 in 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.

\[
\begin{align*}
\text{inv} &: \text{u nat} \rightarrow \text{bool} \\
\text{face} &: \text{u nat nat} \rightarrow \text{u} \\
\text{deg} &: \text{u nat nat} \rightarrow \text{u} 
\end{align*}
\]

where \( \text{inv} \), \( \text{face} \) and \( \text{deg} \) represent the characteristic function of the underlying set and the face and degeneracy operators respectively, and \( \text{u} \) denotes the Universe, of Lisp objects in this case.

The SS signature can be codified using the OpenMath Signature element as follows:
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:
Face operator invariance: \( x \in K^q \Rightarrow \partial_i x \in K^{q-1} \)  

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.

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.

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 $\text{OMA}$ inside the main $\text{mapsto}$ 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 ($$\text{OMA id="">}$$) will be the name of the function, the $\text{mapsto}$ symbol inside the application tag is converted to $\Rightarrow$ in ACL2. The $\text{mapsto}$ 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.

\[
((\text{inv} \; * \; *) \Rightarrow *) \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\###
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.

\[
\text{(local (defun face (x i q) (declare (ignore x i q)) nil))}
\]

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 sub-subsections.

#### 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 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-output (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:

```xml
<acl2-script> acl2-script-path </acl2-script>

<OMDoc-to-ACL2> OMDoc-path-file </OMDoc-to-ACL2>
```

### 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 case (see Subsection 2.2.2): 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-output, 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 extended the Phrasebook by means of a new parser in charge of this task. For instance, the XML-Kenzo request:

\[
\text{<acl2-script> acl2-script-path </acl2-script>}
\]

is generated by the adapter when the following OpenMath request is received:

\[
\text{<OMOBJ>}
\begin{align*}
\text{<OMA>}
& \text{<OMS cd="Aux" name="acl2-script"/>} \\
& \text{<OMSTR> acl2-script-path </OMSTR>}
\end{align*}
\text{</OMA>}
\text{</OMOBJ>}
\]

4.3.4 Integration of ACL2 in the \textit{fKenzo} GUI

This subsection is devoted to present the necessary resources to extend the \textit{fKenzo} GUI to provide access to the new functionality presented in the previous subsection.

In this case we have defined a fresh \textit{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 instructions of the left area. A \textit{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.
Figure 4.18: The ACL2 tab with an example

Figure 4.19: The CD-to-ACL2 window
4.3 Integration of the ACL2 Theorem Prover

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 prove the correctness 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 prove the correctness of some Kenzo programs. In particular, we want to verify the correctness of Kenzo statement like the following one using the ACL2 Theorem Prover.

```lisp
> (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,

```lisp
> (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 $f$Kenzo 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:

```lisp
(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)))
```

It is worth noting that the \texttt{AB-GROUP} class, which represents abelian groups, is a subclass, without any additional slot, of the \texttt{GROUP} class. The relevant slots (relevant in the sense of being important for our verification process) of this class are \texttt{elements}, a list of the elements of the group; \texttt{mult}, the function defining the binary operation of two elements of the group; \texttt{inv}, the function defining the inverse of the elements of the group and \texttt{nullel}, which is the null element of the group.

The \texttt{cyclicgroup} function constructs instances of the \texttt{AB-GROUP}. In particular, it takes as argument a natural number \texttt{n} and constructs an instance of the \texttt{AB-GROUP} which represents the cyclic group $C_n$. The concrete definition of the \texttt{cyclicgroup} function is:
This piece of code must be read as follows. From a natural number \( n \) the \texttt{cyclicgroup} function constructs an \texttt{AB-GROUP} instance where the slot \texttt{elements} has as value the ordered list of natural numbers from 0 to \( n - 1 \) (generated from the \texttt{<a-b>} function), the slot \texttt{mult} is a functional slot which from two elements \( g_1 \) and \( g_2 \) returns the value of \((g_1 + g_2) \mod n\), the slot \texttt{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 \texttt{nullel}. Finally, the \texttt{orgn} slot is a comment which provides the “origin” of the object.

The rest of the slots of the \texttt{AB-GROUP} instance are either automatically generated, in the case of \texttt{idnm}, or a value is assigned after the construction of the object using a GAP resolution, in the case of the \texttt{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 \texttt{GRP}:

<table>
<thead>
<tr>
<th>\texttt{signature}</th>
<th>\texttt{arity}</th>
<th>\texttt{domain}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{invariant}</td>
<td>(u)</td>
<td>(\to) \texttt{bool})</td>
</tr>
<tr>
<td>\texttt{mult}</td>
<td>(g), (g)</td>
<td>(\to) \texttt{g})</td>
</tr>
<tr>
<td>\texttt{inv}</td>
<td>(g)</td>
<td>(\to) \texttt{g})</td>
</tr>
<tr>
<td>\texttt{nullel}</td>
<td></td>
<td>(\to) \texttt{g})</td>
</tr>
</tbody>
</table>

The four functions of this signature are: the \texttt{invariant} unary operation (which represents if an element belongs to the group, that is, the characteristic function of the underlying set), the \texttt{mult} binary operation (the product), the \texttt{inv} unary operation (the inverse) and the \texttt{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 \texttt{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 \texttt{cn}, where \(n\) is the dimension of the group, to the function names):

```lisp
(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 \texttt{mult-c5}, \texttt{inv-c5} and \texttt{nullel-c5} are exactly the same as the ones used in the \texttt{cyclicgroup} function for the slots \texttt{mult}, \texttt{inv} and \texttt{nullel} of the \texttt{AB-\text{\textsc{GROUP}}} instance in the case of \( n = 5 \). The \texttt{invariant-c5} function is the characteristic function of the group and is directly obtained from the \texttt{elements} slot of the \texttt{AB-\text{\textsc{GROUP}}} instance; the transformation from this slot to our function can be considered safe (we have replaced the \texttt{\langle a-b \rangle} function with the \texttt{\langle 0-n \rangle} function which constructs the ordered list of natural numbers from 0 to \( n - 1 \); that is, the \texttt{\langle 0-n \rangle} function with argument \( n \) has the same behavior that the \texttt{\langle a-b \rangle} function with arguments 0 and \( n \)). Then, if we prove that these functions (which have the correct arities specified in the \texttt{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:

\begin{verbatim}
(defthm abelian-c5
  (implies (and (invariant-c5 a) (invariant-c5 b))
    (equal (mult-c5 a b) (mult-c5 b a))))
\end{verbatim}

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 \texttt{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 \texttt{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 \texttt{cyclicgroup} Kenzo function produces an abelian group.

The \texttt{GRP} signature represents a group. In order to handle different groups, in [LPR99] an operation between signatures, called \( \langle \text{imp} \rangle \) 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}\_\texttt{imp} is defined as follows:

\begin{verbatim}
imp-invariant : impGRP g -> bool
imp-mult : impGRP g g -> g
imp-inv : impGRP g -> g
imp-nullel : impGRP -> g
\end{verbatim}

where the new sort \texttt{impGRP} is the sort for groups. This signature, really signatures \( \langle \text{imp} \rangle \), 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}_\texttt{imp} signature, determine an (abelian) group, we need to prove that these functions satisfy the axioms of (abelian) groups for every element of the \texttt{impGRP} sort.

In particular, we have the following four definitions for the case of cyclic groups
defined in Kenzo:

```lisp
(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:

```lisp
(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.

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.
4.4 Interoperability between Kenzo, GAP/HAP and ACL2

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:

\[
<\text{generate-code}> \quad <\text{cyclic}> 5 \quad </\text{cyclic}> \quad </\text{generate-code}>
\]

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.

\[
<\text{OMOBJ}>
  <\text{OMA}>
    <\text{OMS cd="Aux" name="generate-code"}/>
    <\text{OMA}>
      <\text{OMS cd="group1" name="cyclic"}/>
      <\text{OMI}>5</\text{OMI}>
    </\text{OMA}>
  </\text{OMA}>
</\text{OMOBJ}>
\]

It is worth noting that the generation of certifiable 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 \textit{fKenzo} GUI

The composability of Kenzo, GAP/HAP and ACL2 in \textit{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 \textit{fKenzo} is called \textit{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: \textit{gap-kenzo-acl2-structure} (that defines the structure of the new graphical constituents), \textit{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 \textit{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 \textit{gap-kenzo-acl2-structure} file:

- A menu called \texttt{certificates} which contains one option called: \texttt{certificate}.

- A window called \texttt{certificate} (see Figure 4.23).

The functionality stored in the \textit{gap-kenzo-acl2-functionality} document related to these components works as follows. A function acting as event handler is associated with the \texttt{certificate} menu option; this function shows the \texttt{certificate} window (see 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.
4.4 Interoperability between Kenzo, GAP/HAP and ACL2

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 $f\text{Kenzo}$ 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.

```xml
<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
4.5 Methodology to integrate a new internal server in our system

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.