Chapter 2

A framework for computations in Algebraic Topology

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

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

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

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

Now, we have undertaken the task of devising a framework, from now on called Kenzo framework, which provides a mediated access to the Kenzo system, constraining the Kenzo functionality, but providing guidance to the user in his navigation on the system.

The rest of this chapter is organized as follows. A brief overview of the Kenzo framework architecture is presented in Section 2.1. Section 2.2 is devoted to provide a detailed description of each one of the Kenzo framework components. Finally, the Kenzo framework execution flow is presented by means of an example in Section 2.3. Two ongoing works devoted to increase the computation capabilities of the Kenzo framework by means of ideas about remote and distributed computations are presented respectively in Section 2.4 and 2.5.

2.1 Framework architecture

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

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

2. **Extensibility of Kenzo.** The system design should be capable of evolving at the same time as the Kenzo system.

3. **Integration with other systems.** In spite of having Kenzo as main computational kernel, the system should be designed in such a way that it could support different connections to other symbolic manipulation systems (GAP [GAP] in computational algebra, for instance, or ACL2 [KM] from the theorem proving side).

4. **Interaction with different clients.** The system should be designed in such a way that it could support several ways of interaction (graphical user interfaces, web services, web applications, Computer Algebra systems, and so on).

5. **Efficiency.** The framework should be roughly equivalent to Kenzo in time and space efficiency.

6. **Error handling.** Our framework should forbid the user some manipulations raising errors, from both structural and semantical points of view.

7. **Representation of mathematical knowledge.** The representation of the data of our framework should be independent of the modules of the framework, that can be programmed in different programming languages.

8. **Communication of the mathematical knowledge.** The encoded data should be easily transferable both between the different modules of the framework and also outside the framework.

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

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

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

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

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

Let us present a brief overview of the framework components, a more detailed description of each one of them will be provided in Section 2.2.
First of all, let us give some flavor about the concept of mediated access in our framework. It is worth noting that the mediated access is not provided just by one of the framework components but by the whole system.

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

- Related to the construction of spaces:
  - mathematical restrictions:
    * type restrictions: a space constructor can only be applied over objects of a concrete type (and, of course, all its subtypes); for instance, the "classifying space" constructor can only be applied over a space which is a simplicial group; and,
    * restrictions of the arguments of space constructors:
      - independent argument restrictions: the value of an argument of the constructor must satisfy some properties which are independent from the rest of the arguments of the space constructor; for instance, the "Moore" constructor takes \( p \) and \( n \) as arguments to construct the Moore space \( M(\mathbb{Z}/p\mathbb{Z}, n) \), both \( p \) and \( n \) are restricted to be natural numbers and \( p \) must be higher than 1; those are examples of independent argument restrictions;
      - functional dependencies: the value of an argument of the space constructor depends on the value of another one; for instance, in the "Moore" constructor the value of \( n \) must be higher or equal than \( 2p - 4 \); that is a functional dependency.
    - Kenzo implementation argument restrictions: some constrains are imposed by the Kenzo implementation of spaces; for instance, the "Sphere" constructor takes as argument a natural number that Kenzo constrains to be lower than 15.

- Related to the computation of (homology and homotopy) groups:
  - restriction of the computation dimension: in the case of computing \( H_n(X) \) the value of \( n \) must be higher or equal than 0; and in the case of computing \( \pi_n(X) \) the value of \( n \) must be higher or equal than 1; and,
  - reduction degree restrictions: the notion of reduction degree was explained in Subsection 1.2.6. In the case of computing \( H_n(X) \) the reduction degree of \( X \) must be higher or equal than 0; and in the case of computing \( \pi_n(X) \) the reduction degree of \( X \) must be higher or equal than 1.

All this knowledge is spread throughout the Kenzo framework in its components. These components are going to be briefly explained in the following paragraphs.
As we have said previously, XML is the chosen technology to encode the data of our framework since it perfectly fulfills requirements 7 and 8, and also let us encode some mathematical knowledge that will be useful to manage error handling. In particular, we have defined an XML language called XML-Kenzo. This language is used for data interchange among the different components of the framework.

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

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

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

Let us present now the rest of the system components.

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

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

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

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

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

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

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

2.2 Framework components

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

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

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

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

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

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

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

- The SS type contains the elements that represent most of the constructors of Kenzo. Some of them construct spaces from scratch such as sphere, moore-space, build-finite-ss, and so on. Others construct spaces from other ones; for instance, crts-prdc which represents the Cartesian product.

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

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

- The CC type contains elements that correspond with the constructors defined at the algebraic level but not at the simplicial one. The element chain-complex constructs a simple chain complex such as the unit chain complex or the circle. The rest of the elements of this type construct spaces from other ones, for instance tnsr-prdc which represents the tensor product.

As we said previously, XML-Kenzo can be used to represent knowledge and restrictions about its elements. We commented in Section 2.1 that there are three kinds of
restrictions related to the construction of spaces that are dealt with in the XML-Kenzo specification; namely, type restrictions, independent argument restrictions of the space constructors and implementation restrictions of the space constructors. Let us present how we handle these restrictions in the specification of the XML-Kenzo language.

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

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

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

This kind of error is controlled in our framework thanks to the XML-Kenzo specification, since the inputs for the operations between spaces can be only selected among the spaces with suitable characteristics (Figure 2.6 shows the specification of the classifying-space element in XML-Kenzo, this element only allows as child an element either from the SG or the ASG group). This enriches Kenzo with a small (semantical) type system.
It is worth noting that the inheritance relations between Kenzo types, see Section 1.2.1, cannot be directly specified using an XML schema, so, we tackle this situation in the following way. If we have the types \( A \) and \( B \), represented in the XML schema as the groups \( A \) and \( B \) respectively, where \( B \) inherits from \( A \) and a function \( f \) that can be applied over the objects of the type \( A \), and of course also over the objects of type \( B \), then the element of the XML schema that represents the function \( f \) has as child an element that belongs either to the \( A \) or the \( B \) group, and this must be included in the schema in an explicit way. For instance, the \texttt{loop-space} Kenzo function is applied over a simplicial set, but, as can be seen in Figure 1.2 of Subsection 1.2.1, simplicial groups and abelian simplicial groups are subtypes of simplicial sets, so the \texttt{loop-space} element of XML-Kenzo has as child an element belonging either to the \( SS \), the \( SG \) or the \( ASG \) group. Moreover, it also has a child that represents the dimension of the loop space, as can be seen in Figure 2.7.

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

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

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

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

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

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

As we commented in Section 2.1, we can handle one of the restrictions related to computations, that is the restriction of the dimension in homology and homotopy computations. This is translated in the XML-Kenzo specification into a constraint of the value of the dim element of both homology and homotopy elements. Nevertheless, the other restriction related to computations (the reduction degree restriction of the space) cannot be managed at this level since its value must be computed from the description of the space.
Once we have presented the way of encoding Kenzo operations in XML-Kenzo, we can construct different XML-Kenzo objects such as the operation $H_5(\Omega^3(S^4))$ which is represented by the following XML-Kenzo object

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 Internal Server

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

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

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

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

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

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

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

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

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

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

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

2.2.3 Microkernel

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

2.2.3.1 Internal Memory

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

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

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

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

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

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

```
(DEFCLASS MK-OBJECT ()
  ;; IDentification NuMber
  (idnm :type fixnum :initform (incf *number-of-objects*) :reader idnm)
  ;; ORiGiN
  (orgn :type string :initarg :orgn :reader orgn)))
```
This class has two slots (which are common for all the microkernel objects):

- \textit{idnm}, an integer being the object identifier for this system. This is generated by the microkernel in a sequential way, each time a new object is created.

- \textit{orgn}, a string containing the XML-Kenzo object that is the \textit{origin} of the space. This comment is unique and is important, because when a module constructs a new \texttt{mk-object} instance, it uses the XML-Kenzo string to search in a specific list, *\texttt{object-list}*, if the object has not been already built. So, one avoids the duplication of instances of the same object.

The *\texttt{object-list}* list stores the state of the microkernel related to the constructed objects.

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

\begin{verbatim}
(DEFCLASS MK-SPACE-KENZO (MK-OBJECT)
  ;; REDuction DEgree
  (rede :type fixnum :initarg :rede :reader rede)
  ;; Kenzo IDentification NUmber
  (kidnm :type fixnum :initarg :kidnm :reader idnm)
)
\end{verbatim}

This class has two additional slots to the ones of the \texttt{MK-OBJECT} class:

- \textit{rede}, an integer giving the reduction degree, see Subsection 1.2.6, of the space.

- \textit{kidnm}, an integer being the object identifier inside the Kenzo system.

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

Let us explain now the memoization strategy implemented in the internal memory to handle computed results. As we have seen in Subsection 1.2.5, computing some groups in Kenzo can require a substantial amount of time. Kenzo is the kernel of our framework, so this situation also happens in our framework. Therefore, it seems desirable to store the computations of groups somewhere once they have been computed, in order to avoid the re-computation in Kenzo. The microkernel stores the groups associated with a concrete space in the internal memory when a computation is executed for the first time, and if this computation is executed again later, the group is simply looked up and returned, without further execution.
This means that the behavior of the functions which compute the groups depends on whether the asked group for an space has been already computed. Otherwise, the group must be stored after it has been calculated. These two extra tasks are done by two pair of functions that are implemented in the computation modules, namely the testers and the setters. The testers take as arguments the identifier of the object, that is stored in the \texttt{idnm} slot of the \texttt{mk-space-kenzo} instance, and the dimension \texttt{n} of the group and return the result or \texttt{nil} according to whether the \texttt{n}-th homology (or homotopy) group of the space whose identifier is \texttt{idnm} had already been stored. The setters take as arguments the identifier \texttt{idnm} of the object, the dimension \texttt{n}, and the result \texttt{result} of computing the \texttt{n}-th homology (or homotopy) group of the object \texttt{idnm} and put the result into the internal memory, where the result can directly look it up. The main procedures, implemented in the computing modules, are called the getters, and from the preceding discussion it is seen that there must really be at least two methods for the getters. One method is used when the tester returns \texttt{nil}; it is the method which first does the real computation by means of the Kenzo functions and then executes the setter with the computed value. A second method is used when the tester does not return \texttt{nil}; it simply returns the stored value.

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

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

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

Therefore, thanks to this memoization strategy implemented in the internal memory the framework efficiency increases.

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

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

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

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

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

### 2.2.3.2 Processing Modules

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

#### 2.2.3.2.1 Reduction degree module

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

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

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

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

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

\subsection{Homotopy algorithm module}

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

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

Let us present the procedure implemented in HAM. That algorithm and its underlying mathematics were presented in \cite{Rea94}.

\textbf{Algorithm 2.1.}

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

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

   (b) If \(r = n\), \(\pi_n(X) = \pi_r(X) = H_r(X)\) (applying the Hurewicz theorem, see \cite{Whi78}).

   (c) If \(r < n\),

      i. If the \(r\)-th homology group is \(\mathbb{Z}\) or \(\mathbb{Z}/2\mathbb{Z}\) go to step 2.

      ii. Otherwise, return a \texttt{warning} XML-Kenzo object informing that the homotopy group is not computable in the current version.
2. Compute the \( r \)-th fundamental cohomology class of \( X \), \( h_r = [\chi_r] \in H^r(X, \pi_r) \), where \( \pi_r = \pi_r(X) = H_r(X) \).

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

\[
K(\pi_r, r - 1) \downarrow \\
T \equiv X' \equiv X^{(r+1)} = X \times_{\tau_r} K(\pi_r, r - 1) \downarrow \\
X
\]

4. Get the total space \( T = X^{(r+1)} \) associated to the above twisting operator.

(a) If \( r + 1 = n \), \( \pi_n(X) = H_n(T) \).

(b) If \( n > r + 1 \) and \( H_{r+1}(T) = \mathbb{Z} \) or \( H_{r+1}(T) = \mathbb{Z}/2\mathbb{Z} \), increase one unit the value of \( r \) and go to step 2 but using the space \( T \) instead of \( X \).

This is the algorithm implemented in HAM.

2.2.3.2.3 Homotopy expert system

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

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

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

- the Working memory (the facts), representing what we know at any time about the problem we are working at,

- the Knowledge base (the rules), containing the domain specific problem-solving knowledge,
2.2 Framework components

**Figure 2.13: Structure of a rule-based expert system**

- the **Inference engine**, a general program that activates the knowledge in the knowledge base. This program depending on the facts applies different rules to obtain a conclusion.

- a **Knowledge acquisition** module, allowing one to acquire and edit the knowledge base,

- an **Explanation facility** module, allowing the user to understand how the expert system obtains the results.

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

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

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

**Fact 1.**

\[ \forall n \in \mathbb{N} : \Delta^n \text{ is a contractible space.} \]

**Fact 2.** If \( X = A \times B \) where \( A \) and \( B \) are contractible spaces, then,

\( X \) is a contractible space.

**Fact 3.**

\[ \forall n \in \mathbb{N} \text{ and } G \text{ group} : K(G,n) \text{ is an Eilenberg MacLane space of type } (G,n) \]

**Fact 4.** If \( X = B(Y) \) where \( Y \) is an Eilenberg MacLane space of type \( (G,n) \), then,

\( X \) is an Eilenberg MacLane space of type \( (G,n + 1) \)

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

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

\begin{verbatim}
(DEFCCLASS MK-SPACE-SPHERE (MK-SPACE-KENZO)
  ;; DIMension
  (dim :type fixnum :initarg :dim :reader dim))
\end{verbatim}

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

To store contractible spaces, we use a new subclass called \texttt{MK-SPACE-CONTRACTIBLE}, whose definition is:

\begin{verbatim}
(DEFCCLASS MK-SPACE-CONTRACTIBLE (MK-SPACE-KENZO) ()
\end{verbatim}

this class does not have any additional slot to the ones of the \texttt{MK-SPACE-KENZO} class.

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

\begin{verbatim}
(DEFCCLASS MK-SPACE-K-G (MK-SPACE-KENZO)
  ;; ITERation
  (iter :type fixnum :initarg :iter :reader iter)
  ;; GROUP
  (group :type fixnum :initarg :group :reader group))
\end{verbatim}

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

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

\begin{verbatim}
(DEFCCLASS MK-SPACE-LS-SPHERE (MK-SPACE-KENZO)
  ;; ITERation
  (iter :type fixnum :initarg :iter :reader iter)
  ;; DIMension Sphere
  (dims :type fixnum :initarg :dims :reader dims))
\end{verbatim}

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

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

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

**Fact 5.**

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

**Fact 6.**

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

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

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

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

**Rule 1.**

IF $X$ is a contractible space
AND $n \geq 1$
THEN $\pi_n(X) = 0$

**Rule 2.**

IF $X$ is an Eilenberg MacLane space of type $(G,n)$
AND $n = r$
THEN $\pi_n(X) = G$

**Rule 3.**

IF $X$ is the sphere $S^2$
AND $\pi_n(S^3)$ has been computed previously
THEN $\pi_n(X) = \pi_n(S^3)$
Chapter 2  A framework for computations in Algebraic Topology

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

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

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

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

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

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

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

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

```lisp
(DEFMETHOD homotopy-HES ((sphere mk-space-sphere) n) ...)
```
2.2 Framework components

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

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

\begin{verbatim}
(defmethod homotopy ((k-g mk-space-k-g) n) ...)
  (cond ((equal (iter k-g) n)
           (explanation-facility-module k-g n
               (format nil "<component>~A</component>" (group k-g))))
        ((not (equal (iter k-g) n))
           (explanation-facility-module k-g n "<component>0</component>")))
\end{verbatim}

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

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

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

This generic function takes as argument an object \( X \), a natural number \( n \), and the value of the \( \pi_n(X) \) group. For each one of the subclasses of the \texttt{mk-space-kenzo} class we have a concrete method which is able to explain the followed reasoning by the HES to obtain the result. With that information, the \texttt{explanation-facility-module} function generates a HES-result XML-Kenzo object which is the returned result by the HES.
Gathering the functionality of the HES and the HAM, our framework is able to compute some homotopy groups. In addition, we can also combine the HES and the HAM to obtain other homotopy groups as we are going to show in the following paragraph.

### 2.2.3.2.4 Integration of the HES and the HAM

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

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

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

### 2.2.3.3 Construction Modules

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

It is worth noting that there are two kinds of functional dependencies over the arguments. On the one hand, if there are at least two arguments and the value of one of them depends on the value of another one, then, we have a functional dependency (for instance, in the "Moore" constructor the value of its argument \( n \) must be higher or equal than two times plus four its argument \( p \)). On the other hand, if there is just one argument, but with the singularity of being compound, and the value of one of the elements of this compound argument depends on the value of another one, then, we also have a functional dependency (for instance, in the "finite-ss" constructor the value of its argument is a list of elements which must determine a simplicial set; then, this com-
pound argument is restricted to the values of its components). Both kinds of functional dependencies are dealt with in the microkernel, namely in the construction modules.

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

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

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

the functionality of the Loop Space module is invoked.

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

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

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

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

2. Search in the *object-list* list if the object was built previously:

   (a) If the object was built previously, its identification number, in an id XML-Kenzo object, is returned.

   (b) Otherwise, go to step 3.

3. Construct an instance of the mk-space-kenzo class where:

   • the value of the rede slot is computed by the reduction degree module; see paragraph 2.2.3.2.1,

   • idnm is automatically generated,

   • kidnm is the value obtained from requesting the internal server the construction of the space, and

   • the XML-Kenzo object received (from the external server) as input is assigned to orgn.

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

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

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

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

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

Both K-Z and K-Z2 modules construct instances of the MK-SPACE-K-G subclass where the value of the iter slot is the number of iterations of the Eilenberg MacLane space (this number is obtained from the XML-Kenzo object) and the value of the group slot is 1 if is K-Z the working module and 2 if is K-Z2.
The **Cartesian Product** module constructs a **MK-SPACE-CONTRACTIBLE** instance if both components of the cartesian product are **MK-SPACE-CONTRACTIBLE** objects; otherwise, it constructs a **MK-SPACE-KENZO** instance with the general procedure.

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

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

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

### 2.2.3.4 Computation Modules

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

#### 2.2.3.4.1 Homology module

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

When the microkernel receives a computation request where the child of the operation element is homology the homology module is invoked. For instance, the request:
activates the homology module. The procedure implemented in the homology module is as follows:

1. Extract the space whose homology wants to be computed.

2. Search in the *object-list* list if the space was built previously:
   - If the space was built previously, go to step 3.
   - Otherwise, the correspondent construction module builds the space and stores it in the *object-list* list.
     - If the construction module returns a warning then return the warning as result in a warning XML-Kenzo object.
     - Otherwise, go to step 4.

3. Search in the internal memory if the computation was performed previously; see Subsubsection 2.2.3.1.
   - If the computation was stored then the result by means of a result XML-Kenzo object is returned.
   - Otherwise, go to step 4.

4. Monitor the value of the reduction degree of the space, that is stored in the rede slot.
   - If the reduction degree is higher or equal than 0, go to step 5.
   - Otherwise, inform with a warning XML-Kenzo object that the system cannot compute the homology group since the reduction degree of the space is lower than 0.

5. Ask to the internal server the homology group of the space.

6. Store the result obtained from the internal server in the internal memory; see Subsubsection 2.2.3.1.

7. Return the result by means of a result XML-Kenzo object.
The result returned by the homology module when receives the above request is

\[
\text{Z}/3\text{Z} \oplus \text{Z}/2\text{Z} \oplus \text{Z}
\]

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

### 2.2.3.4.2 Homotopy module

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

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

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

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

1. Extract the space whose homotopy wants to be computed.

2. Search in the *object-list* list if the space was built previously.
   - If the object was built previously, go to step 3.
   - Otherwise, the correspondent construction module builds the space and stores it in the *object-list* list.
     - If the construction module returns a warning then the warning is returned as result in a warning XML-Kenzo object.
     - Otherwise, go to step 4.

3. Search in the internal memory if the computation was performed previously, see Subsubsection 2.2.3.1.
• If the computation was stored then the result using a result XML-Kenzo object is returned.
• Otherwise, go to step 4.

4. Monitor the value of the reduction degree of the space.
   • If the reduction degree is higher or equal than 1, go to step 5.
   • Otherwise, inform with a warning XML-Kenzo object that the system cannot compute the homotopy group since the reduction degree of the space is lower than 1.

5. If the space whose homotopy wants to be computed is an instance of the classes: mk-space-sphere, mk-space-contractible, mk-space-k-g or mk-space-ls-sphere:
   • Invoke the HES; see Paragraph 2.2.3.2.3.
   • Otherwise, invoke the HAM; see Paragraph 2.2.3.2

6. Store the obtained result in the internal memory; see Subsubsection 2.2.3.1.

7. Return the result using a result XML-Kenzo object if the result was obtained by the HAM or a HES-result XML-Kenzo object if the result was obtained by the HES.

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

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

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

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

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

### 2.2.4 External Server

The external server is a Common Lisp component providing access to all services available in the microkernel interface through an XML-Kenzo interface. The external server receives XML-Kenzo requests, analyzes them, invokes the microkernel and sends the results back to the caller.
The analysis performed in the external server consists in validating the XML-Kenzo requests against the specification given in the XML-Kenzo language. This means that constraints imposed in the XML-Kenzo language (type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, and restriction of the dimension in computations) are checked here.

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

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

### 2.2.5 Adapter

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

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

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

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

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

```xml
<CDDefinition>
then the name of the object

<Name>sphere</Name>

the description

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

an example (namely $S^3$)

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

the commented property in natural language

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

the formal property using the OpenMath standard

<FMP>
  <OMA>
    <OMS cd="logic1" name="and"/>
    <OMA>
      <OMS cd="set1" name="in"/> <OMV name="n"/> <OMS name="N" cd="setname1"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMI>1</OMI> <OMV name="n"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMV name="n"/> <OMI>14</OMI>
    </OMA>
  </OMA>
</FMP>
are given and finally the definition is closed.

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

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

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

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

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

2.3 Execution Flow

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

The OpenMath representation of $\pi_6(S^3)$ is the following one:
The adapter receives the previous OpenMath data from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object:

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

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

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

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

1. The child element of operation is homology or homotopy. ✓

2. The homotopy element has two children. ✓

3. The first child of the homotopy element belongs to one of the groups CC, SS, SG or ASG. ✓

4. The value of the sphere element is a natural number higher than 0 and lower than 15. ✓

5. The second child of the homotopy element is the dim element. ✓

6. The value of the dim element is a natural number. ✓

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

\[
\text{<result>}
\text{<component>12</component>}
\text{</result>}
\]

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

\[
\text{<OMOBJ>}
\text{<OMA>}
\text{<OMS cd="ringname" name="Zm"/>}
\text{<OMI>12</OMI>}
\text{</OMA>}
\text{</OMOBJ>}
\]

and this is the result returned to the client.
2.4 Integration of local and remote Kenzo computations

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

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

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

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

In this new organization we are keeping in our framework the internal sever since both construction and processing modules use it. On the contrary, the computation modules are not longer connected with the internal server to perform calculations but...
with the Kenzo server through the Kenzo server client.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6. **Storage of results.** The results must be stored in a persistent way to avoid recalculations.
7. **Multithreading.** The server framework should be designed in such a way that it could support the process of several requests at the same time.

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

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

The previous requirements led us, inspired by the work of [MIBR07], to choose both the Linda model [Gel85] and the Shared Repository architectural pattern [B+96, B+07] to reorganize the Kenzo framework components in a server framework. The communication between the server framework and its clients will be provided by means of web services [C+07] based on OpenMath. In addition, a mail subscription mechanism will be implemented to avoid the problem of keeping open a connection during too much time.

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

### 2.5.1 The Linda model and the Shared Repository pattern

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

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

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

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

**Example 2.2.** Some examples of tuples are:
In the Linda model, processes access the tuple space using five simple operations:

- **out**: adds a tuple from the process to the tuple space.
- **in**: deletes a tuple from the tuple space and returns it to the process. The process is blocked if the tuple is not available.
- **rd**: returns a copy of one tuple of the tuple space. The process is blocked if the tuple is not available.
- **inp**: a non-blocking version of the **in** operation.
- **rdp**: a non-blocking version of the **rd** operation.

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

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

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

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

### 2.5.2 A server framework based on the Linda model

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

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

#### 2.5.2.1 Components of the server framework

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

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

The main innovation of the server framework with respect to the Kenzo framework is the tuple space of the Linda model. In order to use the Linda model in our context, a running implementation of it must be available. There are different Linda model implementations for Java [FHA99], C++ [Slu07], and even one for Common Lisp [Bra96].
2.5 Towards a distributed framework

Instead of using any of them, we have preferred to develop a fresh Allegro Common Lisp implementation adapted to our very concrete situation since the Common Lisp version presented in [Bra96] is out-of-date and the integration of either Java or C++ versions in our framework could be difficult.

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

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

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

There are two kinds of XML databases: native XML databases and XML enabled databases. Native XML databases has XML documents as its fundamental unit of storage; on the contrary, XML enabled databases work with XML objects as data and use as internal model a tradicional (relational or object oriented) database. We chose XML enabled databases because we are going to work with XML-Kenzo objects. In XML enabled databases, each XML object is mapped to a datum of the internal (relational or object oriented) model of the database giving access to all the features and the performance that can be found in the corresponding database manager. These databases include two kind of processes: map XML data to objects or tables, and convert database elements into XML data.

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

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

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

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

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

writetuple: tuple → Boolean,
taketuple: tuple → tuple,
readtuple: tuple → tuple

taketuplep: tuple → tuple ∨ nil and
readtuplep: tuple → tuple ∨ nil,

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

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

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

Let us explain the three modules.

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

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

5. When the server framework obtains a result for the computation request, the finished tuple which stores the result for that request is read by the persistence layer and an XML-Kenzo object result is returned to the business logic layer.

6. Transform the XML-Kenzo result into an OpenMath result.

7. Return the OpenMath result to the client.

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

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

1. The persistence layer takes the pending tuple.

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

3. The business logic layer checks all the constraints about the XML-Kenzo object:
   (a) the business logic layer indicates to the persistence layer to write a valid tuple in the tuple space if all the constraints are fulfilled.
   (b) Otherwise, indicates to the persistence layer to write a finished tuple in the tuple space with nil as value of the correct slot and the warning produced as value of the result slot.

4. The persistent layer writes the indicated tuple in the tuple space.
2.5 Towards a distributed framework

2.5.2.1.3 The Kenzo module

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

1. The persistence layer takes the valid tuple.
2. The persistence layer transforms the valid tuple into an XML-Kenzo object which is sent to the business logic layer.
3. The business logic computes the value of the request and sends an XML-Kenzo result to the persistence layer.
4. The persistence layer constructs a finished tuple from the XML-Kenzo result and writes it in the tuple space.

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

2.5.2.1.4 Relations among the modules and the tuple space

The relations among the modules and the tuple space are shown in Figure 2.20. It is worth noting that pending and valid tuples are respectively removed from the tuple space by the processing module and the Kenzo module; on the contrary, finished tuples remain in the tuple space to avoid re-computations, achieving the challenge of storing results.
2.5.2.1.5 Web Services  The previous paragraphs have been devoted to present the different components of the framework and their relations with the tuple space. Let us focus now on other innovation with respect to the Kenzo framework: the use of web services to communicate the server framework with different clients.

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

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

On the contrary, in the second web service, the connection is not maintained, and therefore it needs not only the OpenMath computation request but also an e-mail address where the result will be returned. In this case, to send mails, we have chosen to use Java joins with the JavaMail API [Mic99]. The JavaMail API is a set of abstract APIs modeling a mail system and providing the required technology. In order to do this, we need to communicate our framework (developed in Lisp) with a Java application. There exists different solutions to this problem: we could use the middleware CORBA [Gro], or use the jLinker package [Incd] provided by Allegro Common Lisp; nevertheless, these options are too general for our concrete aim. Then, we have used a solution based on one of the jLinker ideas. jLinker requires two open socket connections between Lisp and Java, one for calls from Lisp to Java, and another one for the inverse communication. In our case we only need one socket for communicating Lisp with Java, because our Lisp system does not require information from our Java system. In Java we have a passive socket waiting for the creation of a connection by means of an active socket (it is the same idea presented in Subsubsection 2.5.2.2 to activate the server framework components). In the Lisp system, whenever a new result is created, the web service creates an active socket which connects with the passive socket of Java and sends both the e-mail address of the client and the result of the computation requested. Finally, the Java program sends an e-mail to the client with the result of the computation asked for the client. However, this web service is not fully satisfactory, since even if the computation asked to the server framework only taking a few microseconds, the client always has to check his e-mail to obtain the result.
As we have said, neither of the two previous web services is fully satisfactory, since they implement two extreme interactions: the first one always keeps open the connection until the result is returned; on the contrary, the second one never does it. Then, to solve this problem a third web service has been implemented mixing the good properties of the others. This web service receives an OpenMath computation request and an e-mail address. When the result is obtained the system checks if the connection with the user is still open, in that case the web service directly returns the result to the user; otherwise it sends the result to the e-mail address. Clients of this web service must decide how much time the connection remains open, an issue which depends on each concrete client.

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

2.5.2.2 Communication among modules

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

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

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

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

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

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

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

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

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

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

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

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

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

It is worth noting that in this case the host of all the modules is the same. That is to say, currently, all the modules are located in the same computer. However, we could install each module in a different computer, and devote a dedicated computer for each one of the modules.
In the future, if we want to include a new module as subscriber of the tuple space, we only need to define an instance with its host, port and the interesting tuples. In this way, whenever an interesting tuple for the module is written in the tuple space, a notification is sent to the module by the publisher.

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

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

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

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

Let us stress in the `observer-<module>` definition the use of the `wait-for-input-available` function. This function waits the connection of an active socket with the passive socket `sock`. Therefore, the well-known busy-waiting concurrency problem is solved (a good description of the concurrency issues presented in this memoir can be consulted in [Sch97]). Busy-waiting happens when a process repeatedly checks if a condition is true (for instance, the availability of a shared resource), the problem is that the processor spends all its time waiting for some condition. A sensible solution to this problem consists of using semaphores which is the approach followed with the `wait-for-input-available` function. Note that this function acts as a binary semaphore, to be precise as a P operation (the P operation sleeps the process until the resource controlled by the semaphore becomes available).

Then, the module is not repeatedly checking if a new tuple has been written in the tuple space, but it is slept until someone activates it (this activation is produced when an interesting tuple for the module is written in the tuple space).

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

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

### 2.5.2.3 An example of complete computation

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

The server has received the OpenMath request by means of anyone of the three web services previously presented. The process of the request is the same for all of them, the only difference is the way of returning the result. The OpenMath representation of \(\pi_6(S^3)\) is the following one:

```xml
<OMOBJ>
  <OMA>
    <OMS cd="Computing" name="Homotopy"/>
    <OMA>
      <OMS cd="SS" name="sphere"/> <OMI>3</OMI>
      <OMI>6</OMI>
    </OMA>
  </OMA>
</OMOBJ>
```
The I/O module receives the previous OpenMath data from a client through one of the web services. This module checks that the OpenMath instruction is well-formed and converts the OpenMath object into the following XML-Kenzo object:

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

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

```
> (readtuplep (make-instance 'finished
   :operation (make-instance 'operation :name "homotopy" :dim 6)
   :XML_Object (make-instance 'XML_Object :xml-object "<sphere>3</sphere>")
   :correct '?' :result '?)) *nil*
```
In this case, NIL is returned, the result is not in the database (as we have said, we are working in a fresh session, so, no computation has been performed previously). Then the I/O module writes a pending tuple (corresponding to the request) in the tuple space (from now on, we will not include the attribute values of the instances if they are not different from the previous ones):

```
> (writetuple (make-instance 'pending :operation (...) :XML_Object (...)))
```

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

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

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

```
> (writetuple (make-instance 'valid :operation (...) :XML_Object (...)))
```

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

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

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

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

Then the I/O module is activated and asks again for the result of the request.
2.5 Towards a distributed framework

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

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

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

and this is the result returned to the client.

### 2.5.2.4 Concurrency issues

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

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

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

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

- Absence of deadlocks. A deadlock happens when two processes are waiting for the
other to finish, and thus neither ever does. In our server framework this situation never happens since the execution of each one of the modules is independent from the other ones.

- **Mutual exclusion.** Mutual exclusion occurs when some algorithms avoid the simultaneous modification of a shared resource. In our server, the shared resource is the tuple space; however, each server framework component only access to a kind of tuples. Therefore, in spite of concurrently accessing to the tuple space, the server framework components access to different kinds of tuples; then, inconsistencies are avoided and the good property of mutual exclusion is reached.

- **Absence of starvation.** Starvation describes a situation where a process is unable to access to shared resources and is unable to make progress. This situation is not feasible in our server framework, since the modules are only activated when new resources are included in the tuple space. Moreover, the modules access to the tuple space by means of non blocking operations; then, if a module asks data to the tuple space but there is no datum, then the module backs to sleep. Therefore, our server framework components never starve.

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

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

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

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

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