Applying Generative Communication to Symbolic Computation in Common Lisp

Jónathan Heras, Vico Pascual, and Julio Rubio
(University of La Rioja, Spain
{jonathan.heras, vico.pascual, julio.rubio}@unirioja.es)

Abstract: In this paper, an architecture to interact with a system called Kenzo is described. Kenzo is a Common Lisp system devoted to mathematical symbolic computation, namely in the area of Algebraic Topology. The architecture presented in this paper is an evolution of a previous proposal, where several aspects have been improved. In particular, we have now uncoupled the different components, and implemented mechanisms to reuse intermediary computations. The new technology allowing us these improvements is based on the Linda Model (a model where processes communicate by means of a shared tuple space), implemented through AllegroCache (an object oriented database for Allegro Common Lisp). The developed framework acts as a middleware between some Kenzo clients and Kenzo itself.

Key Words: Linda Model implementation, AllegroCache, Symbolic computation systems.
Category: I.1, I.2.4

1 Introduction

Kenzo [Dousson et al. 1999] is a Common Lisp system devoted to Symbolic Computation in Algebraic Topology. It was developed in 1997 under the direction of F. Sergeraert and has been successful, in the sense that it has been capable of computing homology groups unreachable by any other means. Having detected accessibility and usability as two weak points in Kenzo (implying difficulties in increasing the number of users of the system), several proposals have been studied to interoperate with Kenzo (since the original user interface is Common Lisp itself, the search for other ways of interaction seems mandatory to extend the use of the system). In [Heras et al. 2008] we presented a proposal to interact with Kenzo by means of an *intermediary layer* which was designed to avoid some of the drawbacks of the Kenzo system. In particular, a standard way was defined to communicate through MathML [Ausbrooks et al. 2008], an XML standard for mathematics, wrapping Kenzo with a small *type system*. This mediated access gave us the possibility of including some *intelligent processing*, avoiding some typical flawed interactions with the Kenzo system.

Even if the system reported in [Heras et al. 2008] was a great improvement with respect to the previous Kenzo user interface, it had several drawbacks. The first one was its message passing style, which prevented us from designing an uncoupled access to the different components of the system. The second

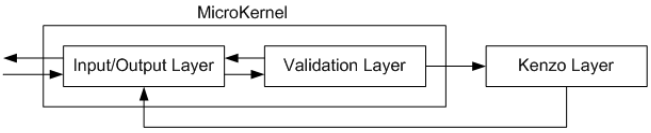

Figure 1: Simplified diagram of the previous architecture.

drawback was related to the communication pattern, too. Since it was based on a *stateless* protocol, intermediary computations were produced on-the-fly, and then recalculated if needed. Considering both aspects together, we obtained a system with severe difficulties with respect to further scalability and enhancements from the distributed computing point of view.

These difficulties are overcome in this paper. To this aim, inspired by the work of [Mata et al. 2007], we have implemented the well-known generative communication paradigm (and, more concretely the Linda Model [Gelenter 1985]), by means of AllegroCache [Aasman 2005], an object-oriented database for Allegro Common Lisp [Franz Inc.].

The organization of the paper is as follows. In [Section 2] we introduce the new architecture. Firstly, we explain the model which inspired us and subsequently we give an implementation of it. We finish [Section 2] by presenting an example. How the communication among the different components of the system is carried out is tackled in [Section 3]. In addition, some concurrency issues are also explained in this Section. To prove the adaptation of this architecture, in [Section 4] two kinds of clients are explained. This work finishes with Conclusions and Further Work.

2 Architecture of the system

The architecture presented in [Heras et al. 2008] is based on the *MicroKernel* pattern [Gamma et al. 1994]. The MicroKernel design consists of two layers: the first one manages the input of requests and the output of results, and the second one deals with the validation of requests. The internal server, a layer wrapping *Kenzo*, executes the computations and returns the results to the input/output layer. A simplified diagram of this architecture appears in [Figure 1].

As was explained in the Introduction, our previous implementation was rigid from the point of view of the execution flow, giving an essentially sequential way of interaction. In particular, if the system receives two requests, the second one must wait until the first one is finished. In addition, Kenzo computations used to be very time and space consuming (requiring, typically several days of CPU time on powerful dedicated computing servers); therefore to store these results

in a persistent way would be useful to avoid recalculations. This possibility was not considered in our previous proposal.

To improve these aspects our new software system has been inspired by both the *Linda Model* [Gelenter 1985] and the *Shared Repository* architectural pattern [Buschmann et al. 1996].

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 a *tuple space*. Thus the shared memory contains *tuples* produced by the processes. As soon as a tuple is inserted in the tuple space, it has an independent existence.

Tuples can contain both *actual* and *formal* items. An actual item contains a specific value, like 3 or 4.3. A formal item acts as a typed placeholder (a character `?` denotes a variable which has to be treated as formal, rather than using the value stored in the variable as actual).

In the Linda Model, the 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 operation *in*.

rdp- A non blocking version of the operation *rd*.

By means of the last four operations the 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] shows a feasible framework which brings together both Linda Model and the shared repository pattern.

This kind of architecture solves the problems that the previous one presented, stated at the beginning of this section. On the one hand, the communication is not based on a message passing model but the modules are communicated using the tuple space (for example while a component is computing a request, another component can validate another request). On the other hand, the persistency

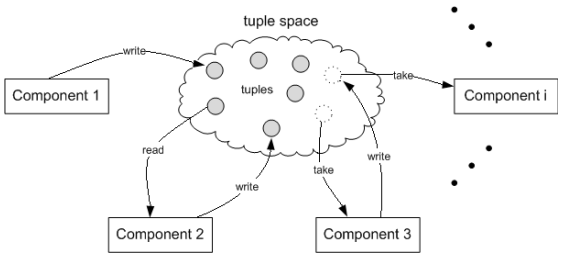

Figure 2: Feasible framework.

problem is solved because all the previous computations have been stored in the tuple space.

2.1 An adapted implementation of the Linda Model

As can be seen in [Figure 1], three layers (Input/Output, Validation and Kenzo layers) were developed in our previous architecture. In the new one, these layers are uncoupled so play the role of components (modules). More specifically, the *I/O module* deals with the input and output of the external requests; the *Intelligent System* validates requests (the functionality of this module was described in detail in [Heras et al. 2008]); and finally, the requests are computed by the *Kenzo module*.

Besides, we defined an XML schema called XML-Kenzo, diagrammatically described in [Figure 3] (XML-Kenzo provides something like a "type system" for Kenzo), to communicate the different layers of the system. With the aim of using it in our new architecture, our tuples rely on valid XML data for the XML-Kenzo schema (from now on, will be called XML-Kenzo data).

The tuple space, the three modules and the XML-Kenzo data are the main ingredients of the new architecture as is shown in [Figure 4].

In order to use the Linda Model in our framework, a running implementation of it must be available. There are different implementations for Java [Freeman et al. 1999], C++ [Sluga 2007], and even one for Common Lisp [Bradford 1996]. Instead of using any of them, we have preferred to develop a fresh Allegro Common Lisp implementation adapted to our very concrete context.

We have developed our implementation of a tuple space to hold 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 in implementing the Linda Model from scratch,

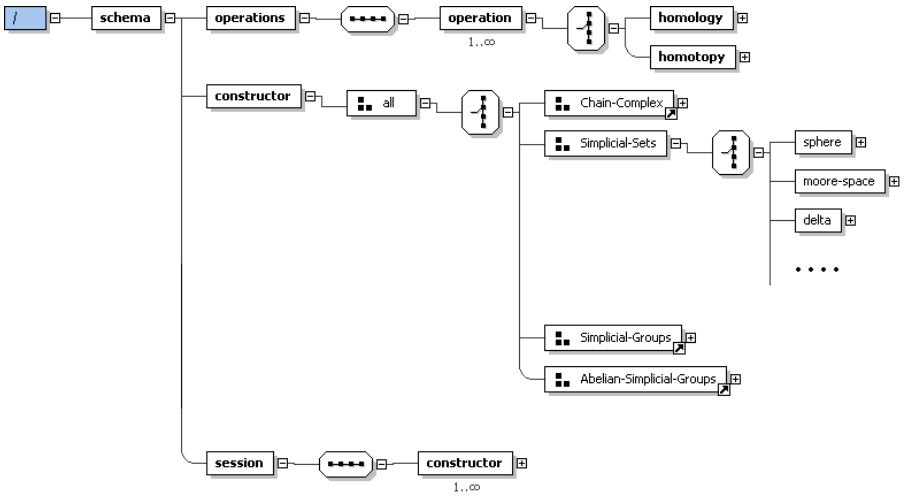

Figure 3: Description of the XML-Kenzo Schema.

without using any kind of external tool (for instance, the tuple space could be installed in computer memory and we could store the information as a list). This option would be very time consuming as 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 [Evjen et al. 2007] should be implemented in our case, too.

These problems oriented us towards the second possibility: using already existing technology. Our previous remark gave us a clue where to look: XML databases could be a good support in our case.

There are two kinds of XML databases [Chaudhri et al. 2003]: *Native XML databases* and *XML enabled databases*. We have used an XML enabled database because this kind of databases is built on top of relational or object oriented databases. Each XML data is mapped to a data of relational or object oriented database giving access to all the features and the performance found in the corresponding database manager. These databases include two kind of processes: from an XML data they map it to objects or tables, and from the database an element is converted into an XML data.

When choosing a specific database, we took into account that the rest of the framework is based on Allegro Common Lisp [Franz Inc.]. Even once Allegro was chosen, several options were still open (among others, we thought of connecting the relational database MySQL thanks to AllegroMySQL). Finally we decided to use AllegroCache [Aasman 2005].

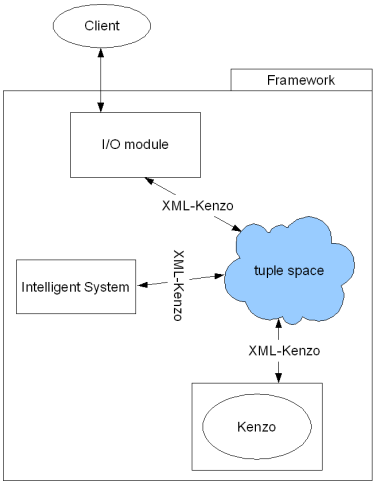

Figure 4: The architecture presented.

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 referential 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 the 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 to objects and viceversa. We have devised a classes system represented in UML in [Figure 5]. A tuple is implemented as an object belonging to any of the subclasses of the `tuple` class, that is, an object of type `pending, valid` or `finished`.

Each Kenzo request is made up of both a topological space and an operation over it, so two classes, `XML_Object` and `operation`, have been defined to model the two components of a request. There are different kinds of operations in Kenzo that we can group and identify by different subclasses of the `operation` class. For instance, in the case of an homology operation (that has two attributes, the name of the operation and the dimension), an `operation-with-dimension` subclass has been defined. Just to give another example, the face operation has three attributes (the name, a dimension and an index) so another class must

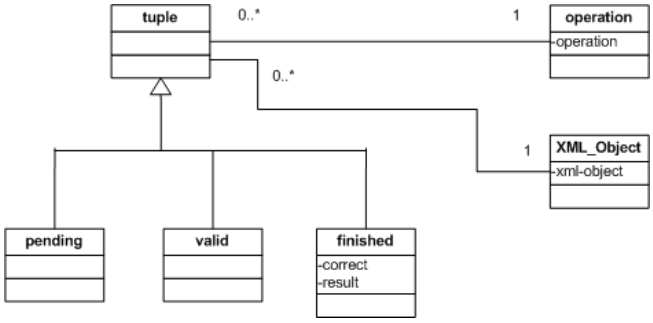

Figure 5: Class diagram.

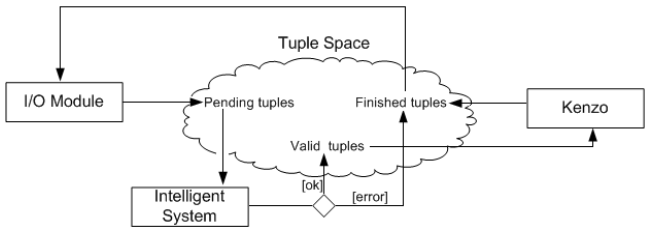

Figure 6: Relations of the modules with the tuple space.

be defined. The operations are applied over a space being an instance of the `XML_Object` class, whose objects are made up only of an XML-Kenzo data.

The `tuple` class binds these two components of a request. When an XML-Kenzo data representing a Kenzo request arrives, our program instantiates one of the three specialized classes of the tuple class. The object type determines what modules of the framework can access it. The relations among the modules and the tuple space are shown in [Figure 6].

With respect to the operations, they are implemented in a natural way from the `select`, `insert` and `delete-instance` of AllegroCache. For instance, the *inp* operation of the Linda Model can be programmed by combining a `select` operation and a `delete-instance` one. Thus, we have built five methods with the following signatures:

$$\begin{aligned}
&\texttt{writetuple:}\ \texttt{tuple} \rightarrow \mathit{Boolean},\\
&\texttt{taketuple:}\ \ \texttt{tuple} \rightarrow \texttt{tuple},\\
&\texttt{readtuple:}\ \ \texttt{tuple} \rightarrow \texttt{tuple-persistent},\\
&\texttt{taketuplep:}\ \texttt{tuple} \rightarrow \texttt{tuple} \vee \texttt{nil} \text{ and}\\
&\texttt{readtuplep:}\ \texttt{tuple} \rightarrow \texttt{tuple-persistent} \vee \texttt{nil},
\end{aligned}$$

Those are, respectively, our versions for *out*, *in*, *rd*, *inp*, *rdp*. Let us remark that

the structure of objects must be duplicated: one standard version, and another one *persistent*. This implies that the classes hierarchy in [Figure 5] is duplicated too, with each class having a persistent couple (with the `-persistent` suffix in its name). We have employed this fact in the previous specification to stress that the `readtuple` and `readtuplep` methods are not erasing the corresponding tuple from the database.

The different modules of the system interact with the tuple space through those methods by means of a subscription mechanism (which will be technically explained in [Section 3]). The interaction with the tuple space of the different software modules is as follows. The I/O module writes `pending` tuples and it is subscribed to `finished` tuples. The Intelligent System is subscribed to `pending` tuples and writes `valid` tuples and `finished` non valid tuples. And finally, the Kenzo module which is subscribed to `valid` objects, writes `finished` objects in the tuple space.

And now, we present an execution scenario which is illustrated with a UML-like sequence diagram [Figure 7]. An I/O module asks the tuple space if it has the result of a request. In this case the result is not stored in the tuple space, so the I/O module writes a new request in the tuple space. This request is taken by the Intelligent System from the tuple space and this module validates it. The Intelligent System writes a valid request in the tuple space because the tuple is valid. Then Kenzo takes the valid request from the tuple space and computes the result. When Kenzo finishes, the result is written in the tuple space. Finally, the I/O module reads this result and sends it to the client. Note that there is no direct communication among components. It is also clear that the issue of activating software components at the right time and in the adequate order is essential (this aspect will be treated in [Section 3]).

2.2 An example of complete computation

We have just presented a scenario; now we detail it in a particular instance: the computation of the sixth homotopy group of the sphere of dimension 3, $\pi_6(S^3)$.

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

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

The I/O Module receives the previous XML-Kenzo data. This module, which has access 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

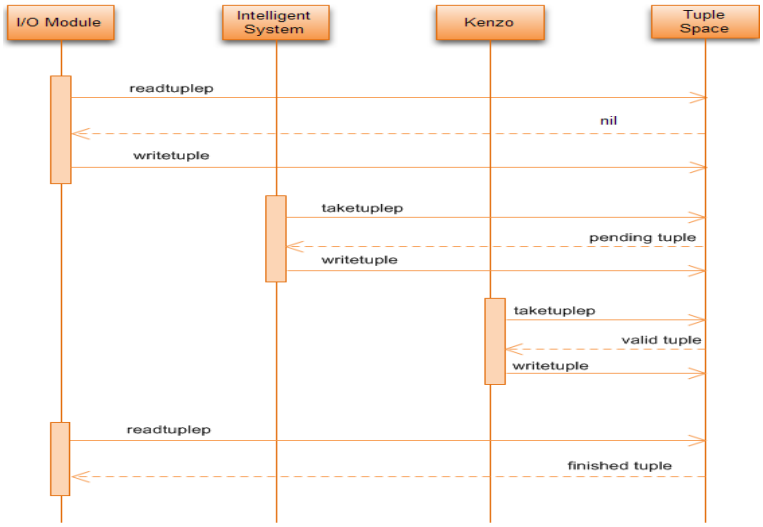

Figure 7: UML sequence diagram.

created from the XML-Kenzo data 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-with-dimension
                  :operation "homotopy"
                  :dimension 6)
     :XML_Object (make-instance 'XML_Object
                   :xml-object "<sphere>3</sphere>")
     :correct '? :result '?))
NIL
```

In this case, `NIL` is returned, meaning the result is not in the database (that calculation has not been requested and computed previously). Then the I/O module writes a `pending` tuple (corresponding to the request) in the tuple space (from now on, we will not include the attribute values of the instances if they are not different from the previous ones):

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

Due to the existence of a new pending tuple the Intelligent System 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 Intelligent System asks for a general pending tuple. The `taketuplep` method deletes the element of the database and it creates a non persistent object used by the Intelligent System to validate the request. In this case, the request is considered sensible so the Intelligent System writes a valid tuple in the tuple space:

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

Kenzo is activated due to the new valid tuple. Then it asks for the new tuple:

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

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

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

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

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

Note that the result is not deleted from the tuple space because the `readtuplep` operation is used. In this way, the result can be used again, without recomputing it.

3 Communication among modules

Up to now, how the different modules communicate has not been explained. This is one of the most important issues, and has been tackled by means of a reactive mechanism. This is a kind of asynchronous communication.

The message passing protocol used in the architecture presented in [Heras et al. 2008] has been replaced with a *publish-subscribe* machinery. This mechanism is based on the existence of both *subjects*, which can be modified, and

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

Figure 8: Subscription class.

observers subscribed to any possible subjects modification. When a modification subscribed for any agent is produced, this is notified all the subscribed ones which respond to it. This design has been implemented by means of the *Observer pattern* [Gamma et al. 1994].

To store the subscriptions, the tuple space has a database associated, again an AllegroCache database. To be notified when any events occurs in the tuple space, each module has to insert an instance of the `subscription` persistent class (defined in [Figure 8]) in the database.

An instance of the persistent class `subscription` has a `host`, a `port` and a `type-tuples` indicating the type of tuples subscribed.

For instance, the Intelligent System is located at the `host` "cosmos.unirioja.es", in the `port` 8002, and it is subscribed to all the new pending tuples, so when it starts, it must write:

```
(make-instance 'subscription
  :host "cosmos.unirioja.es" :port 8002 :type-tuples 'pending)
```

To notify each one of the different subscribers, a passive socket in the port indicated in the subscription has been created. Besides, the Allegro Common Lisp `wait-for-input-available` function, that waits for the next available input stream, in this case a socket, has been used. Then when a new tuple is written in the tuple space, the database observer consults the database of subscriptions and sends the subscribers of this type of tuple a notification. To be precise, it opens an active socket for each one of them.

And now let us show an example. The definition of the Intelligent System observer appears in [Figure 9], the `wait-for-input-available` function waits for any input stream, in this case a socket, avoiding the *busy-waiting* problem (see [Schneider 1997] for any one concurrency issues presented in this paper). Note that this function acts as a *semaphore*, to be precise as the `P` operation.

When the tuple space receives a new pending instance, it notifies the Intelligent System of it, being the only module subscribed to the pending instances. The tuple space needs to know the modules subscribed to each one of the types of tuples. To this aim it is connected to the database and reads the subscriptions

```
(defun observer ()
  (loop
    (let ((sock
           (make-socket :connect :passive :local-port 8002)))
      (wait-for-input-available sock)
      (close sock) (validate))))
```

Figure 9: The Intelligent System observer.

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

Figure 10: The notify function.

of a concrete type. [Figure 10] shows the `notify` function code, which can be seen as the `V` operation in semaphores.

Up to now, we have presented the case where in each moment there is only one request in the system. In general, as several requests could be in the system at the same moment, some concurrency problems could appear which must be dealt with. One of the main keys in order to solve them is that our system is based on the Linda Model which founds on sharing a data repository instead of the data themselves. As our system 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 (*inp*) and the insertion (*out*) operations are atomic). Therefore different processes keep independent among themselves, avoiding the occurrence of a number of problems related to concurrency. In the next two paragraphs we comment on two situations in which problems could appear, but where the Linda model avoids them.

On the one hand, our system allows different modules work at the same time keeping *safety properties* (because as well as the properties of the Linda Model, each module works with a concrete type of tuples). In this way properties like *mutual exclusion* or *absence of deadlocks* are satisfied.

On the other hand, different processes of the same module can work at the same time. For that, every time that a module is activated a new process (in charge of executing the instruction of the correspondent module) is built from the main process. In order to do this, the Multiprocessing package of Allegro Common Lisp which provides all the modules of our framework with the main tools for working in a concurrent way (management of *threads*, *locks*, *queues* and so on) has been used. Note that if two processes read (*rdp*) the same tuple when they want to write their results in the tuple space two different tuples will be created; avoiding in this way problems like *race conditions*.

Once the concurrency issues have been commented on, we introduce in the following Section the distributed computing context on which the application runs.

4 Clients of our framework

As we commented previously, increasing the usability of Kenzo is one of our objectives. To test the quality of our proposals, two kinds of client have been constructed. On the one hand, using the Integrated Development Environment of Allegro Common Lisp, a GUI (Graphical User Interface) has been developed. The GUI uses OpenMath Content Dictionaries [Buswell et al. 2004] to organize the interaction, and the computed results can be rendered in standard displays (with the usual mathematical notations). Details on this client can be found in [Heras et al. 2009].

In addition, two *Web Services*, which connect with the framework, have been developed. These web services allow the use of the framework only from the XML-Kenzo data (no knowledge of Lisp or Kenzo is needed). So, a developer can build any other application in Java, .NET and so on, knowing only the XML-Kenzo syntax and, of course, the web services technology.

We have implemented these web services using the SOAP 1.1 API for Allegro Common Lisp, that uses both the SAX parser and the AllegroServe modules.

The difference between both clients based on web services lies on the way results are returned. The first web service makes a request and waits until the result is available. On the contrary, in the second web service, the connection is not maintained, and therefore it needs not only the request but also an e-mail address where the result of the request will be returned. In this case, to send mails, we have chosen to use *Java joins* with the *JavaMail API* (the Java Mail 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 [CORBA], or use the package *jLinker* provided by Allegro Common Lisp. But we have used a different 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 other way. 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 server waiting for a client socket. In the Lisp system, the client socket must send two lines, the first one indicating the e-mail address of the receiver and the second one will be the message, encoding the result of the computation requested.

5 Conclusions and Further Work

In this paper we have reported on a program to interact with the Kenzo Computer Algebra system. We have built it on a previous proposal presented in [Heras et al. 2008], trying to obtain a system which can process queries concurrently and where persistent results are available. We can conclude that both objectives have been satisfied, thanks to an adapted implementation in Common Lisp of the Linda Model.

But even more important than these objectives is the remark that our new architecture will allow us in the near future, to move to a distributed computing context. We are thinking of a federated architecture, where several rich clients (each one with its own tuple space) communicate with a central powerful computing server. The central server acts as a general repository and also provides computing power to deal with difficult calculations. In a possible scenario a request is obtained in a rich client from an external user. The first action is to check if this request has already been computed, locally or in the central server. If this is not the case, the local Intelligent System should decide if the computation is easy enough to be solved locally or whether it must be sent to the central server. In any case, the result should be stored persistently, both in the local client and in the central tuple space. Our architecture scales well to this new context. The most difficult problem is to devise good heuristics to decide what is the meaning of the fuzzy predicate "to be an easy computation" (for a discussion of this topic we refer to [Heras et al. 2008]).

Another related issue could be tackled: the processing of several tasks in a parallel way. In each request, the data are independent of the operation, so our problem is classified inside the MIMD (Multiple Instruction, Multiple Data) paradigm according to Flynn's taxonomy [Flynn 1972]. Here it will be important again to know what kind of computations are low or high consuming, to get a good load balance. From the point of view of parallel computing, our setting can be considered as an *embarrassingly parallel problem* [Foster 1995] because the tasks can be executed more or less independently, without communication.

Putting together these two research lines (distributed and parallel computing), we hope that new challenging computations can be carried out, beyond the

possibilities showed by Kenzo, executed as a stand-alone program.

References

[Aasman 2005] Aasman J. *AllegroCache: A high-performance object database for large complex problems*, In 5th International Lisp Conference, Stanford University, June 2005.

[Ausbrooks et al. 2008] Ausbrooks R. et al., *Mathematical Markup Language (MathML)* Version 3.0 (second edition), 2008. `http://www.w3.org/TR/2008/WD-MathML3-20080409/`.

[Bradford 1996] Bradford R., *An Implementation of Telos in Common Lisp*, Object Oriented Systems, 3 (1996) 31–49.

[Buschmann et al. 1996] Buschmann F., Meunier R., Rohnert H., Sommerland P., Stal M., *Pattern-oriented software architecture. A system of patterns, Volume 1*, Wiley, 1996.

[Buswell et al. 2004] Buswell S., Caprotti O., Carlisle D.P., Dewar M.C., Gaëtano M., Kohlhase M. *OpenMath* Version 2.0, 2004. `http://www.openmath.org/`.

[Chaudhri et al. 2003] Chaudhri A. B., Rashid A., Zicari R. *XML data management: native XML and XML-enabled database systems*, Addison-Wesley, 2003.

[CORBA] Object Management Group. *Common Object Request Broker Architecture (CORBA)*. `http://www.omg.org`.

[Dousson et al. 1999] Dousson X., Sergeraert F., Siret Y., *The Kenzo program*, Institut Fourier, Grenoble, 1999. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.

[Evjen et al. 2007] Evjen B., Sharkey K., Thangarathinam T., Kay M., Vernet A., Ferguson S., *Professional XML*, Wiley Publishing, Inc., 2007.

[Flynn 1972] Flynn M. J., *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Computers C-21 (1972) 948 – 960.

[Foster 1995] Foster I. *Designing and Building Parallel Programs*, Addison-Wesley (1995).

[Franz Inc.] Franz Inc. *Allegro Common Lisp*. `http://www.franz.com/`.

[Freeman et al. 1999] Freeman E., Hupfer S., Arnold K., *Javaspaces. Principles, Patterns, and Practice*, Addison Wesley, 1999.

[Gamma et al. 1994] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gelenter 1985] Gelernter D. *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, 7 (1985) 80–112.

[Heras et al. 2008] Heras J., Pascual V., Rubio J., Sergeraert F., *Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology*, Proceedings of 1st European Lisp Symposium, University of Bourdeaux, France (2008) 155–176.

[Heras et al. 2009] Heras J., Pascual V., Rubio J., *Mediated Access to Symbolic Computation Systems: An OpenMath Approach.*, Preprint.

[Mata et al. 2007] Mata E., Alvarez P., Bañares J. A., Rubio J., *Formal Modelling of a Coordination System: from Practice to Theory and back again*, ESAW2006, Lecture Notes in Artificial Intelligence 4457 (2007) 229-244.

[Schneider 1997] Schneider F. B., *On Concurrent Programming*, Springer (1997).

[Sluga 2007] Sluga T. A. *Modern C++ Implementation of the LINDA coordination language*, PHD Thesis, University of Hannover, 2007.

Acknowledgments

Partially supported by Universidad de La Rioja, project API08/08, and Ministerio de Educación y Ciencia, project MTM2006-06513.