A P2P Groupware Framework based on Operational Transformations

Leandro Quiroga, Alejandro Fernandez
LIFIA, Facultad de Informática, UNLP
leandro.quiroga, alejandro.fernandez @lifia.info.unlp.edu.ar

***Abstract* - Groupware applications deal with propagation of changes to networked users, consistency maintenance on concurrent access, and provision of awareness. We have built a P2P groupware framework for the sub-domain of synchronous groupware. The framework is based on the concept of operational transformations to provide transparent change propagation and consistency maintenance. This paper provides an overview of the framework design and illustrates how to use it to build an application.**

I. Introduction

Groupware applications help groups of people to collaborate, for example, to achieve a common goal. Using synchronous groupware, such as a shared whiteboard, group members collaborate in real time [11].

Building effective groupware is challenging. Correctly specifying the functionality of a groupware system is difficult because the very nature of working together continually changes as a consequence of changing work needs, but also as a consequence of how the systems themselves tend to change work relationships and processes. This phenomenon is known as co-adaptation [24].

Besides providing functionality for the task at hand (e.g., creating whiteboard documents), groupware applications must deal with other complex concerns such as propagation of changes to networked users, consistency maintenance on concurrent access and provision of awareness. Several applications may share requirements for change propagation, consistency maintenance and awareness. Object-oriented groupware frameworks [27, 28, 29] hide complexity and foster reuse (both of design and of implementation) thus reducing the effort of developing groupware.

Most groupware systems (and therefore, most groupware frameworks) rely on a centralized, client-server architecture. In a centralized architecture there is a distinguished entity (the server) in charge of coordination. This distinguished entity provides the advantage of simplifying concurrency control algorithms. However, having a central server implies having a single point of failure. Any server failure will disable the complete system. Moreover, the system's scalability is limited by the server's performance.

Lately, attention has moved toward peer to peer applications. Peer to peer groupware applications have a decentralized architecture. That is to say, they do not require singular components for its operation as a client-server architecture would. On the contrary, they enhance the availability of services by distributing them among a great number of nodes, which are not assumed to fail all at the same time. A peer to peer system is a self organizing system of equal, autonomous entities (peers) which aims for the shared usage of distributed resources in a networked environment avoiding central services [25].

Although building P2P groupware is becoming more common, support for building P2P applications is rare. There are few libraries or components that facilitate P2P groupware development. One of these is JXTA [9]. It is a set of open protocols that allow any connected device on the network (ranging from cell phones and wireless PDAs to PCs and servers) to communicate and collaborate in a P2P manner.

We have built a P2P groupware framework for the sub-domain of synchronous groupware. The framework is based on the concept of operational transformations to solve the problems of change propagation and consistency maintenance. This paper provides an overview of the framework with insight to the design and implementation of the operational transformation engine.

The paper is organized as follows. The next section explores the issues of repeatedly having to implement a concurrency control algorithm for facing peer to peer groupware applications consistency troubles. Section 3 presents frameworks as a mechanism for reuse of design and implementation. The proposed P2P framework is presented in Section 4. An usage example is described in Section 5. Section 6 presents conclusions and future work.

II. Maintaining consistency in P2P groupware

Many groupware applications rely on the shared data being replicated in all participating clients. Maintaining the replicated data consistent is a problem that has been extensively covered by the literature [1, 8, 22]. The fundamental cause of the emergence of inconsistency is that mapping changes at one location onto changes at all other locations consumes an unpredictable amount of time, due to network latency, processing time and queuing time [8]. Problems arise when conflicting changes are propagated from different nodes at the same time (see figure 1.B). This may result in different ordering of changes and therefore divergent results at different locations [8].

Consider, as an example, a collaborative puzzle application where a group of people try to solve a puzzle moving and fitting pieces concurrently. All nodes are aware of all movements and fits generated by other nodes. Maintaining consistency implies that after changes have been made, all nodes must have the same puzzle data. There are several conflicting situations in this scenario. For example, at the same time two players may decide to move the same piece to different locations.

One option to obtain the same result in all nodes is to establish a global total ordering among operations. It is based in the fact that if all nodes have the same initial state and execute operations in the same order, the final state at all nodes would be identical. In a client-server architecture the server can establish it considering the arriving order. Thus, it can attach a serial number to operations and forward them to all nodes, which will execute them in order (see figure 1.A).

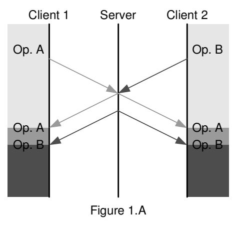


Figure 1.A

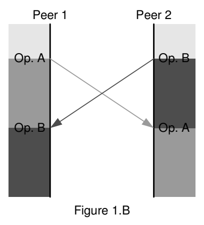


Figure 1.B

In a pure peer to peer architecture there is no distinguished entity which can provide a global and total ordering of operations (having a distinguished entity would break its pureness). Therefore, consistency must be ensured in a distributed manner.

Another strategy is not to accept actions that may cause inconsistency, i.e., to deny some actions by some users. This may be done by locking, a form of coordination that gives only one user at a time the (temporary) privilege to initiate actions that may cause inconsistency, while restricting others to do such actions as long as the user holds the lock [8]. Establishing locks on entities decrease interactivity which is not desirable in synchronous groupware applications.

III. OT ALGORITHMS TO MAINTAIN CONSISTENCY

Operational Transformations (OT) [1] is an alternative to maintain consistency in totally distributed environments. With this method it is not necessary to execute operations in a total order, nor establishing locks on entities.

In OT algorithms, changes are encapsulated in operations. Peers generate operations which are executed locally and then propagated to other peers. Operations received from other peers are executed following strict rules.

Conceptually, each operation O has a context. This context is composed by the operations needed to be executed to bring the model from its initial state to the state on which O was defined (definition context). The effect of an operation can be correctly interpreted only in its definition context. When executing O, if the current context (named execution context) is different from O's definition context, O has to be delayed or transformed so that it can be executed in the current context.

OT algorithms transform operations to include/exclude the effects of other operations. Intuitively, transformations shift operation's attributes before execution, to incorporate the effects of previously executed operations that it was not aware of, at the time of generation [23]. This method was originally designed for collaborative text editors and to our knowledge it is not been adapted to others domains.

Basically OT algorithms consist of three parts:

- **Operations**: application domain specific commands that model actions (e.g., moves and fits in the puzzle game).
- **Transformers**, entities responsible of modifying operations if they are in conflict with previously executed operations. As we said before, they adapt an operation if its definition context differs from the execution context, to make it executable.
- **An integration algorithm**, in charge of receiving operations, deciding when to execute them, deciding if they must be transformed before execution, executing them and propagating operations to all other peers.

Several OT algorithms exist [1, 2, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. They differ in their architecture (centralized or replicated, unicast or multicast), their abilities (intention preservation, causality preservation and copies convergence), undo support, operations execution order, when operations are propagated, etc.

IV. IMPLEMENTING OT ALGORITHMS

OT algorithms are hard to understand, implement and debug. An OT algorithms complexity indication is that papers that have been published with errors in their algorithms [1, 2] were discovered long after publication.

When building P2P groupware applications, in which inconsistencies between peers can arise, selecting some consistency preservation mechanism is a must. Generally, business logic code and consistency preservation code are intertwined (e.g. information such as details of preceding operations is embedded in domain specific abstractions or locking code is dispersed all over the business logic code). In such scenario, consistency preservation code can hardly be reused and must be designed, implemented and tested from scratch. As we said before, these are not trivial tasks and it is not desirable to do them over and over again for each new application.

Separating concurrency control from business logic facilitates respective development stages. Doing a slight analysis (see next section), it becomes clear that when developing applications from different application domain, although business logic differs, considerable parts of consistency maintenance functionality are identical. This generates an ideal environment to define a consistency management tier, which provides all common behavior and allows configuring domain specific aspects.

V. REUSING OT ALGORITHMS

It is possible to provide reusable implementations of OT algorithms that can be specialized to implement applications from different domains. It implies encapsulating the parts that are common across domains while providing mechanisms for extension and specialization for those parts that differ.

Analyzing OT algorithms in accordance to its composition and possible reutilization we conclude that:

- Operations are domain specific. For example in a collaborative puzzle there will be *move piece* and *fit piece* operations, whereas in a simple collaborative text editor the operations would be to insert or delete characters. Such specific behavior cannot be reused. However, the general design principle of operations and a base implementation could be provided, for example, following the Command pattern [26].
- Transformers base in operation's attributes to return transformed operations. As we said before, operations are domain specific. Consequently, transformers depend on application domain. Like operations, its specific behavior could not be reused, but its design can.
- Last but not least, the integration algorithm does not depend on application domain, since:
    - Operation's receiving and sending could be treated as an independent layer abstracted of application domain.
    - To decide the operation's execution time and if it must be transformed for execution, the OT algorithm only needs to compare its definition context with the execution context, so it does not depend on the application's domain.
    - Executing operations could be generalized using the Command pattern.

So, a big portion of OT algorithms behavior is common.

A. *Frameworks as a mechanism for reuse*

Frameworks are a design reuse technique in the object-oriented paradigm. A framework constitutes the inner structure of applications and gives the possibility to tune the specific details. The places where these specific details are specified are called framework's hot spots.

Hot spots are defined by abstract classes. Creating subclasses of these abstract classes is how they are specified. These subclasses have the option of redefine existing methods and the obligation to define the abstract ones. Software developers do not have to know how the entire framework is implemented. They only have to know about how hot spots are specified [6, 7].

In the previous section we argued that a considerable part of the consistency management code is repeated in different applications, even if they come from different domains. In the next section we show how these common parts can be encapsulated in a framework. The solution we propose is the result of combining these previous ideas and it is presented in next section.

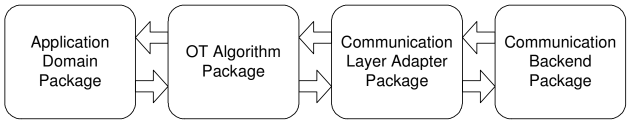
VI. A P2P FRAMEWORK BASED ON OPERATIONAL TRANSFORMATIONS

The frameworks theory provides tools needed to encapsulate common behavior and specify points of variability. Our contribution consists of a framework to help in the development of P2P synchronic groupware with heavy consistence management.

A. *Framework architecture*

As show in the packages schema below, the framework is divided in four different but related packages:

- Application domain package.
- OT algorithm package.


Application Domain Package
OT Algorithm Package
Communication Layer Adapter Package
Communication Backend Package

- Communication layer adapter package.
- Communication backend package.

Application domain package

The application domain package is basically composed by operations and transformers. Operations are subclasses of the Operation abstract class. They redefine the execute() method and two more methods (see Framework hotspots section for details). The execute() method receives an object as a parameter and performs some actions over this object. Groupware application peers will communicate using operations. Any action that must be propagated must be encapsulated in an operation.

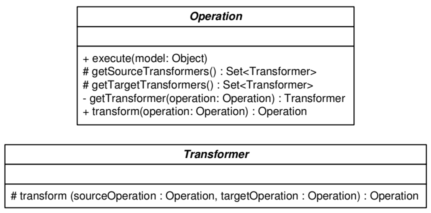
In the puzzle application example, fit piece and move piece actions need to be encapsulated by operations. Thus there will be one Operation subclass for each action type (FitPieceOperation and MovePieceOperation, see section VI.*C* for details).

Transformers are subclasses of the Transformer abstract class. As said before, transformers adapt operations which definition context differs from execution context, to make them executables. Each transformer is a Singleton [26]. It redefines the transform() method, which receives two operations and returns a transformed one.

There will be one transformer for each Operation subclass and two transformers for any binary combination of these subclasses. So if we have two operations classes, A and B, there will be one transformer for transforming A given another A, other for B given another B, other for A given a B and other for B given an A.

Going back to the puzzle application example, it will have four transformers. One for transforming fit piece operations with itself, another transformer for transforming move piece operations with itself, another transformer for transforming fit piece operations with move piece operations and another transformer for transforming move piece operations with fit piece operations.

These two groups of classes (operations and transformers) are the only application specific abstractions needed to guarantee consistency.


Operation
+ execute(model: Object)
# getSourceTransformers() : Set<Transformer>
# getTargetTransformers() : Set<Transformer>
- getTransformer(operation: Operation) : Transformer
+ transform(operation: Operation) : Operation
Transformer
# transform (sourceOperation : Operation, targetOperation : Operation) : Operation

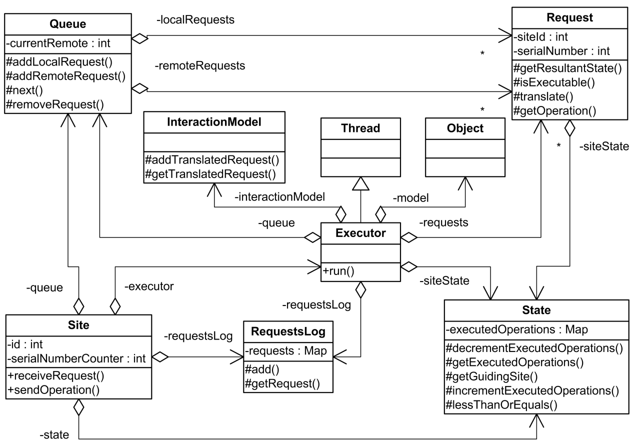
OT algorithm package

Among all available integration algorithms, we choose to implement adOPTed [2], because:

- It is proved that the integration algorithm is correct [4].
- There is enough documentation to implement it [2].
- It can be extended to support undo operations [5].
- It allows a replicated and multicast architecture. With adOPTed, the model is in each peer and peers can directly communicate with each other.
- Although it is proved that its transformations are wrong [3], it is not necessary to use them. But in case we need them, others [10] can be used.

The framework implements all concepts involved in the adOPTed algorithm. There is a Site class in charge of sending and receiving operations. Before executing, operations are pushed into a Queue that prioritizes local operations over remote ones. An Executor, which runs in an independent thread, pops executable operations from Queue, transforms them (if it is necessary) and finally executes them. Operations are encapsulated into Requests to be sent between peers. Requests also contain the source peer id, the operation serial number and the definition context. In adOPTed, both definition context and execution context are represented by State vectors. These State vectors have a hash table that keeps a counter of operations executed from each site.

There is also a RequestsLog and an InteractionModel. The former stores all departing local Requests and all arriving remote Requests. It is used when operations need to be transformed. The latter stores operations resulting from transformations. There are some cases in which it must be done the same transformation over the same operations. The InteractionModel is used to prevent doing the same transformations over and over again.



Communication layer adapter package

This package purpose is to relate the OT algorithm implementation with the communication software. It consists in three classes, called SiteConnection, GroupManager (abstract) and Receiver (abstract), and four interfaces, named Sender, Marshaller, RequestsReceiver and Unmarshaller.

The SiteConnection class is the most important one because it acts as the communication software facade for the Site. It has methods for creating, joining, leaving and searching for groups, closing the connection, returning peer id and sending and receiving requests. For doing these activities it interacts with instances of GroupManager, Receiver, Sender and Marshaller.

GroupManager is an abstract class which objective is to do group activities like creating, joining, leaving and searching for groups of peers. After joining a group, it also creates the Receiver, Sender, Marshaller and Unmarshaller and passes them to the SiteConnection.

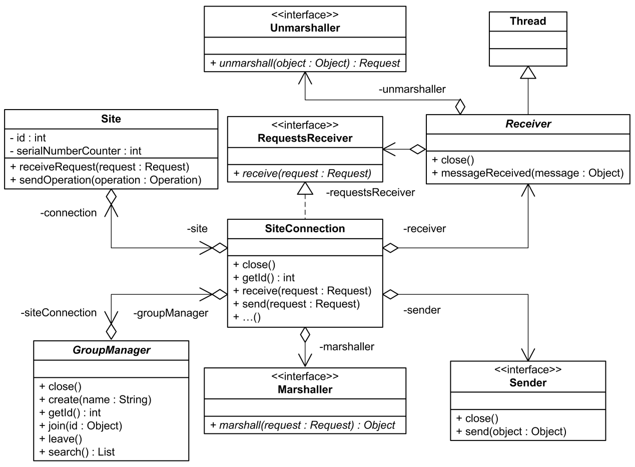
The way of doing all these activities clearly depends on the communication software to use. Thus, they are all represented by abstract methods. For each different communication software and each different sending/receiving mechanism there will be one GroupManager subclass. The current framework implementation has a concrete subclass (JxtaGroupManager) for using JXTA pipes.

The Receiver is the object in charge of receiving messages, transforming them into Requests using an Unmarshaller and passing them to the RequestsReceiver. For each way of receiving messages to use, there must be one Receiver subclass. The framework provides a PipeReceiver for receiving messages through JXTA pipes.

The Unmarshaller interface only declares one method, unmarshall(), that takes objects and returns requests. An implementation, XmlUnmarshaller, for use with JXTA is provided.

Sender implementations need to implement two methods, close() and send(). The former interrupts connections with other peers. The latter sends a message, previously created by a Marshaller implementation, to all other peers. The current framework implementation provides a PipeSender and an XmlMarshaller.

If you decide to use another communication software, you just have to implement the appropriate subclasses of GroupManager and Receiver and the appropriate implementations of Sender, Marshaller and Unmarshaller.



Communication backend package

Although it could be replaced by other frameworks (like Conference XP), we used JXTA [9] for communication. Its most important services are, peer discovering and identifying, group creation and management and security management.

B. *Framework hotspots*

The instantiation of the framework consists of several steps, involving from selecting some communication software to implementing operations and transformers. The framework hotspots are divided into three subsections.

Communication layer package

The first step is to create an instance of the SiteConnection class. In the constructor we will need to specify a GroupManager subclass. Currently the framework only provides one subclass for connecting with JXTA, so create a JxtaGroupManager instance. This is a ready-to-use class that employs XML over JXTA pipes to communicate. Using this class you would not need to configure a receiver, sender, marshaller and unmarshaller. If you want to employ some different JXTA communication mechanism or even some different communication software, you may define new GroupManager and Receiver subclasses and new Sender, Marshaller and Unmarshaller implementations.

OT algorithm package, operations

The next step is to create a Site instance, specifying the SiteConnection previously created and the model. The model is the object over which operations will be executed.

Just to avoid casting the model to its specific class in each operation, we create an Operation abstract subclass which implements the execute method as showed below:

```
public void execute(Object model) {
  this.execute((SpecificModel) model);
}
```

We replace SpecificModel with our model's class name. We must also declare the following abstract method:

```
public abstract void execute(SpecificModel
        specificModel);
```

For example, at the puzzle game, we have defined a PuzzleOperation abstract class as showed below.

```
public abstract class PuzzleOperation extends Operation {

    public void execute(Object model) {
        this.execute((PuzzleModel) model);
    }

    public abstract void execute(PuzzleModel
puzzleModel);
}
```

Then, we create the representative classes for those operations which may be sent between peers. We make them subclasses of the previously created Operation abstract subclass, and implement the execute method in each one. The execute method should only contain a call to a model method.

OT algorithm package, transformers

Next step is to create a Transformer subclass for each binary combination of conflictive operations. Each of which may implement the Singleton pattern. Each Transformer must implement the transform method. In it, it must check certain properties or attributes of parameterized operations and return the correct transformed operation.

When different transformers are implemented, it may be taken in account diverse criteria to find out the resultant operation. Depending on interferences between transforming operations, three cases are distinguished:

- Operations do not interfere with each other. In this case, the sourceOperation must be returned without any transformation.
- Operations are equals. This is the case in which two participants concurrently create the same operation. In this case, as the targetOperation was already executed, the surceOperation may be ignored. So a NoOperation must be returned.
- Operations interfere with each other. So, a transformation that resolves the conflict must be applied.

There are some test cases to evaluate a set of transformers correctness [10].

Each non abstract operation subclass created above may implement the following methods:

```
Set<Transformer> getSourceTransformers()

Set<Transformer> getTargetTransformers()
```

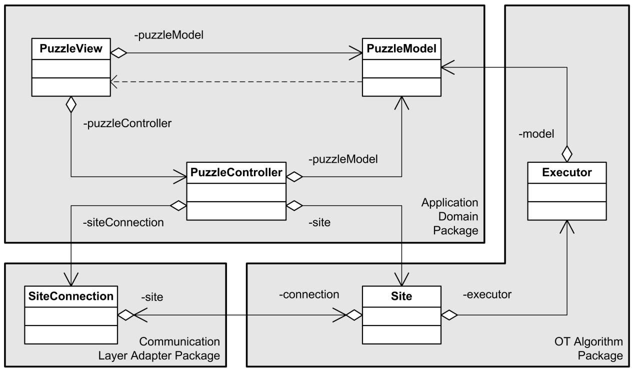
The former returns a set of transformers which take as source operation, instances of the class that contains the method. The latter returns a set of transformers which take as target operation, instances of the class that contains the method.

C. *Usage example*

Collaborative puzzle implementation

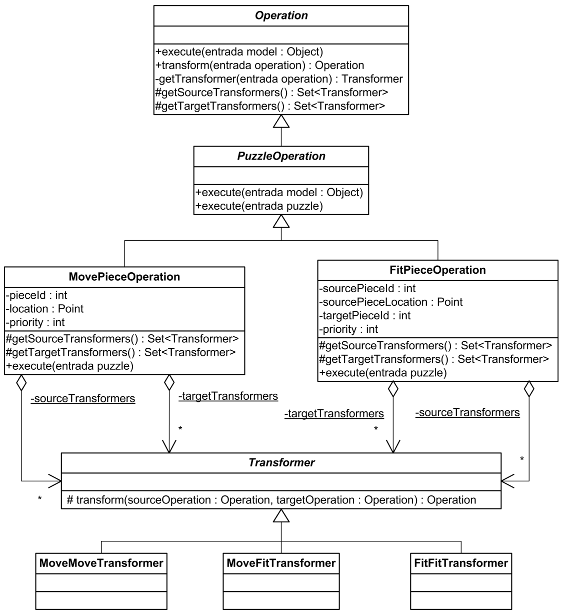
This section will show how the framework is used to implement a collaborative puzzle. The picture below shows an schematic class diagram of the collaborative puzzle. In it you can find three different packages and how they are related. At the Application Domain Package, the PuzzleController is in charge of communicating with the Site for sending operations. It also communicates with the SiteConnection for creating, joining leaving and searching for groups. As explained before, at the OT Algorithm Package, the Executor will execute operations on the PuzzleModel. The Site and the SiteConnection collaborate with each other for sending and receiving requests.

The instantiation of the whole application will happen in the following way. The PuzzleController instantiates the SiteConnection class passing as a parameter a GroupManager instance. Then, the PuzzleController instantiates the Site class, passing as parameters the SiteConnection previously created and a PuzzleModel instance. Thus, operations will be executed on this PuzzleModel instance.



Operations can arrive to the Site in two ways, from PuzzleController (if they are locally generated) or from SiteConnection (if they are remotely generated). In each peer the PuzzleController will generate operations with information passed by the PuzzleView and required to the PuzzleModel.

There are two types of operations, MovePieceOperation and FitPieceOperation. These classes are PuzzleOperation subclasses, which is abstract and subclass of Operation. PuzzleOperation class only implements the abstract method execute(Object) (defined in Operation class) in which invokes the execute(PuzzleModel) abstract method.


Operation
+execute(entrada model : Object)
+transform(entrada operation) : Operation
-getTransformer(entrada operation) : Transformer
#getSourceTransformers() : Set<Transformer>
#getTargetTransformers() : Set<Transformer>
PuzzleOperation
+execute(entrada model : Object)
+execute(entrada puzzle)
MovePieceOperation
-pieceId : int
-location : Point
-priority : int
#getSourceTransformers() : Set<Transformer>
#getTargetTransformers() : Set<Transformer>
+execute(entrada puzzle)
FitPieceOperation
-sourcePieceId : int
-sourcePieceLocation : Point
-targetPieceId : int
-priority : int
#getSourceTransformers() : Set<Transformer>
#getTargetTransformers() : Set<Transformer>
+execute(entrada puzzle)
-sourceTransformers
-targetTransformers
-targetTransformers
-sourceTransformers
*
*
*
*
Transformer
# transform(sourceOperation : Operation, targetOperation : Operation) : Operation
MoveMoveTransformer
MoveFitTransformer
FitFitTransformer

Each PuzzleOperation subclass must implement the getSourceTransformers(), getTargetTransformers() and execute(PuzzleModel) methods. In MovePieceOperation case, the getSourceTransformers() method returns a set with one instance of MoveMoveTransformer class and another of MoveFitTransformer class. The getTargetTransformers() method returns a set with one instance of MoveMoveTransformer class. Finally, the execute(PuzzleModel) method invokes the movePiece() method on the puzzleModel.

In FitPieceOperation case, the getSourceTransformers() method returns a set with one instance of FitFitTransformer class. The getTargetTransformers() method returns a set with one instance of FitFitTransformer class and another of MoveFitTransformer class. Finally the execute(PuzzleModel) method invokes the fitPiece() method on the puzzleModel.

Previously named transformers implement the Singleton pattern and are subclasses of Transformer abstract class.

There are some cases in which the transformer always returns the source operation without any transformation. In our puzzle, fit operations have precedence over move operations. That is to say, when a move and a fit operation over the same piece are concurrently created, peer's resultant states after both operations execution will be the same as not have executed the move operation. So transforming a fit operation with a move operation always returns the fit operation without any changes. In these cases it is not necessary to implement the transformer. That is why the FitMoveTransformer class does not exist.

Each Transformer subclass implements the transform(Operation, Operation): Operation method in the following way:

MoveFitTransformer: it checks if operations are over the same piece or if pieces involved are fitted, in this case it returns a NoOperation. Else it returns the source operation without any changes.

```
class MoveFitTransformer extends Transformer {
  ...

  Operation transform(Operation sourceOperation,
Operation targetOperation) {
    if (operations are over the same piece or pieces
are fitted) {
      return new NoOperation(sourceOperation);
    } else {
      return sourceOperation;
    }
  }
}
```

MoveMoveTransformer: if movements are over the same piece and to the same location it returns a NoOperation. If movements are over the same piece but to different locations it checks operation's priorities to select which operation will remain. If operations are over different pieces it returns the source operation without any transformation.

```
class MoveMoveTransformer extends Transformer {
  ...

  Operation transform(Operation sourceOperation,
Operation targetOperation) {
    if (movements are over the same piece) {
      if (to the same location) {
        return new NoOperation(sourceOperation);
      } else {//to different location
        if (source operation has higher priority) {
          return sourceOperation;
        } else {
          return new NoOperation(sourceOperation);
        }
      }
    } else {//movements are over different pieces
      return sourceOperation;
    }
  }
}
```

FitFitTransformer: if fits are over the same piece and with the same piece it returns a NoOperation. If fits are over the same piece but with different pieces it checks operation's priorities to select which operation will remain. On the other hand, if fits are over different pieces but are symmetrical, it checks operation's priorities to select which operation will remain. Finally if fits are not symmetrical it returns the source operation without any transformation.

VII. Conclusions, Open Issues and Future Work

We have presented a new framework to help in the development of synchronic groupware. It is based in Operation Transformations assuring highly dynamic interaction between participants.

Some open issues are supporting late comers and provide some way to test transformations. The former could be implemented passing the current state to new users and when a late comer requires

```
class FitFitTransformer extends Transformer {
  ...

  Operation transform(Operation sourceOperation,
Operation targetOperation) {
    if (fits are over the same piece) {
      if (fits are with the same piece) {
        return new
NoOperation(sourceFitPieceOperation);
      } else {//fits are with different pieces
        if (source operation has higher priority) {
          return sourceOperation;
        } else {
          return new FitPieceOperation(
sourceFitPieceOperation.getTargetPieceId(), null,
sourceFitPieceOperation.getPriority(),
sourceFitPieceOperation.getSourcePieceId());
        }
      }
    } else {//fits are over different pieces
      if (fits are symmetrical) {
        if (source operation has higher priority) {
          return new MovePieceOperation(
targetFitPieceOperation.getSourcePieceId(),
targetFitPieceOperation.getSourcePieceLocation(),
sourceFitPieceOperation.getPriority());
        } else {
          return new NoOperation(sourceOperation);
        }
      } else {//fits are not symmetrical
        return sourceOperation;
      }
    }
  }
}
```

an old operation it asks the group for. The latter is important because developing transformers is not an easy task. See [3] for details.

The future work will be focus on previous open issues, some technique to factorize transformers (to reduce its number) and testing the framework in various and more complicated domains.

References

[1] S. J. Gibbs C. A. Ellis. Concurrency control in groupware systems. Proceedings of the 1989 ACM SIGMOD international conference on Management of data, pages 399–407, June 1989.
[2] D. Nitsche-Ruhland, M. Ressel and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. Proceedings of the 1996 ACM conference on Computer supported cooperative work, pages 288–297, 1996.
[3] G. Oster A. Imine, P. Molli and M. Rusinowitch. Development of transformation functions assisted by a theorem prover. 9/2003.
[4] Lushman, Cormack. Proof of correctness of Ressel's adopted algorithm. Information Processing Letters, 86:303–310, 2003.
[5] Rul Gunzenhäuser Matthias Ressel. Reducing the problems of group undo. Proceedingsof the international ACM SIGGROUP conference on Supporting group work, pages 131–139, 1999.
[6] Ralph E. Johnson. Components, Frameworks, Patterns. 1997
[7] Fayad, Mohamed; Schmidt, Douglas; Johnson, Ralph. "Building application frameworks: object-oriented foundations of framework design". New York: John Wiley & Sons, 1999.
[8] Ter Hofte, H., Working Apart Together – Foundations for Component Groupware, Telematica Instituut, Enschede, 1998
[9] JXTA, http://www.jxta.org
[10] A. Imine, P. Molli, G. Oster, M. Rusinowitch, Achieving Convergence with Operational Transformation in Distributed Groupware Systems.
[11] Baecker, R. M., Grudin, J., Buxton, W. A. S., Greenberg, S. 1995 "Readings in Human-Computer Interaction: Towards the Year 2000" (Second Edition) Morgan Kaufmann Publishers, Inc.
[12] D. A. Nichols, P. Curtis, M. Dixon and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. Proceedings of the 8th annual ACM symposium on User interface and software technology, pages 111–120, 1995.
[13] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction (TOCHI), pages 63–108, 1998.
[14] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors:issues, algorithms, and achievements. Proceedings of the 1998 ACM conference on Computer supported cooperative work, pages 59–68, 1998.
[15] M. Suleiman, M. Cart, J. Ferrie. Serialization of concurrent operations in a distributed collaborative environment. Proceedings of the international ACM SIG-GROUP conference on Supporting group work: the integration challenge, pages 435–445, 1997.
[16] Maher Suleiman, Michele Cart, and Jean Ferrie. Concurrent operations in a distributed and mobile collaborative environment. Proceedings of the Fourteenth International Conference on Data Engineering, pages 36–45, 1998.
[17] Jean Ferrie Nicolas Vidot, Michelle Cart and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. Proceedings of the 2000 ACM conference on Computer supported cooperative work, pages 171–180, 2000.
[18] Rui Li, Du Li, and Chengzheng Sun. A time interval based consistency control algorithm for interactive groupware applications. In ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference on (ICPADS'04), page 429, Washington, DC, USA, 2004. IEEE Computer Society.
[19] Haifeng Shen and Chengzheng Sun. Flexible notification for collaborative systems. Proceedings of the 2002 ACM conference on Computer supported cooperative work, pages 77–86, 2002.
[20] Du Li and Rui Li. Ensuring content and intention consistency in real-time group editors. 2004.
[21] Du Li and Rui Li. Preserving operation effects relation in group editors. Proceedings of the 2004 ACM conference on Computer supported cooperative work, pages 457–466, 2004.
[22] Munson, J.P. and P. Dewan, 'A concurrency control framework for collaborative systems'. In [CSCW96], p. 278-287.
[23] ACE a collaborative editor, Report evaluation algorithms.
[24] Mackay, W. (1990) Users and Customizable software: A Co-Adaptive Phenomenon. Doctoral Dissertation, MIT.
[25] A. Oram, Peer-toPeer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Inc., 2001.
[26] Gamma, E., Helm, R., Johnson, J., Vlissides, J.: Design Patterns. Elements of reusable object-oriented software, Addison Wesley 1995.
[27] C. Schuckmann, L. Kirchner, J. Schummer, J. Haake. Designing object-oriented synchronous groupware with COAST. Proceedings of Computer Supported Collaborative Work CSCW, 1996.
[28] D.A. Tietze: A Framework for Developing Component-based Co-Operative Applications. GMD Research Series Nr.7/2001.
[29] S. Lukosch and C. Unger: Flexible Management of Shared Groupware Objects, in Proceedings of the Second International Network Conference (INC 2000), pp. 209-219.