An Enablement Detection Algorithm for Open Multiparty Interactions

J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro
Dept. Lenguajes y Sistemas Informáticos
Universidad de Sevilla
{jperez,corchu,druiz,mtoro}@lsi.us.es

ABSTRACT

Coordination amongst an arbitrary number of entities has become an important issue in recent years in fields such as e–commerce, web–based applications and so on. Traditionally, classical client/server primitives have been used to implement synchronisation and communication. But, when more than two entities need to coordinate by means of those primitives, the coordination must be decomposed into a number of client/server biparty interactions, leading the programmer to the need of thinking in terms of the protocols needed to achieve properties like livenes, atomicity and so on. In this paper, we present an algorithm to perform enablement detection to implement open multiparty interactions. This primitive provides a high level of abstraction since the programmer can implement multiparty coordination without the need of thinking in terms of protocols.

Keywords: Multiparty interactions, coordination algorithms.

1. INTRODUCTION

In recent years, the development of distributed applications has been paid much attention, mainly due to the Internet boom. Traditionally, most services provided by means of the network involved only two entities, a provider entity and a client entity. For example, a purchase trough the web involved just two entities: a seller and a purchaser. Recently, more complicated scenarios have emerged, since frequently several entities collaborate to provide a service. For example, in a purchase through the web, the purchaser may order his or her bank to transfer the sale amount to the seller bank account. And both seller and purchaser may require each other to certificate their identity, involving then certification entities in the scenario.

When more than two entities need to collaborate to achieve a common goal, coordination becomes an important issue. Traditionally, coordination has been achieved by means of client/server primitives (remote procedure call, message passing, and so on). Those primitives are biparty because they only involve two entities that need to synchronise before exchanging data, but this concept can be easily extended to an arbitrary number of entities that need to agree and cooperate to achieve a common goal. These interactions are usually said to be multiparty, and they provide a higher level of abstraction because they allow to express complex cooperations as atomic units. A taxonomy of languages offering linguistic support for multiparty interactions can be found in [6]. We think that those interaction models are interesting because they allow to express coordination regardless of the protocols needed to achieve it.

Most interaction models proposed in the literature are aimed at coordinating a set of entities that must be known in advance, i.e., they are *static* models. Those models are not adequate for open scenarios such as e–commerce where frequently entities need to collaborate without knowing one another. For example, in the purchase through the web, nor the purchaser knows the account of the seller, neither vice versa. Furthermore, the seller do not need to know the client in advance.

In [2], the CAL language is presented. The CAL language is aimed at increasing the level of abstraction of a programme by considering the concurrent behaviour of components as an aspect where multiparty interactions are the sole means for synchronisation and communication. CAL relies on an open interaction model that allows to express coordination amongst entities that do not know one another in advance.

Although several authors have proposed algorithms to implement multiparty interactions as a coordination primitive [9, 1, 5, 4, 6], they have focused on finding solutions to achieve exclusion in the scope of static interaction models. To implement an open, dynamic interaction model we must deal not only with the exclusion problem, but also with the enablement detection problem. The enablement detection consists on finding sets of entities that agree to coordinate through a given interaction. In an open context, this is an important problem since in general, it has a high computational cost.

In this paper we present the algorithms we have devised to implement enablement detection in the CAL interaction model. The problem of finding enablements in this model has a computational combinatorial complexity. But, by means of the adequate data structures, the algorithms we propose behave quite efficiently in most practical situations. Furthermore, they can be customized with a selection algo-

SAC 2002, Madrid, SPAIN

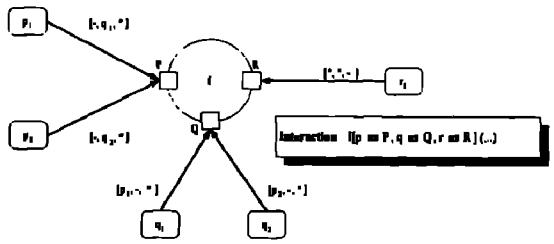

Figure 1: Example of CAL interaction model.

rithm to solve the exclusion problem.

The rest of this paper is organised as follows: section 2 sketches the CAL interaction model; section 3 outlines our strategy to implement this model; section 4 focuses on the algorithms we have developed to perform enablement detection, and section 5 glances at the problem of enablement selection. Next, we show some performance results from our implementation in section 6, and finally, section 7 shows our conclusions.

2. AN OPEN MULTIPARTY INTERACTION MODEL

This section presents the CAL [2] interaction model. To the best of our knowledge, this is the only open, dynamic interaction model proposed so far. It improves static models in that it can coordinate entities that do not need to know one another. This is very important because it makes it feasible as a coordination primitive for open systems. As we describe in this section, interactions in CAL work as templates of static interactions.

In CAL, each interaction is given a *name*, a number of *roles* and a number of *slots* associated with it. The name of the interaction is a string which unambiguously identifies an interaction in the system. When an object is ready to coordinate with other objects, it offers to participate in one or more interactions by means of their names.

So, every object can offer participation in one or more interactions simultaneously. In every offer, a participant states which role it plays in the interaction, and may establish constraints on what objects should play the other roles. An interaction may be executed as long as a set of objects satisfying the following constraints is found: (i) there is an object per role willing to participate in that interaction and play that role; (ii) those objects agree in interacting with each other, i.e., the constraints they establish are satisfied. A set of objects which can execute an interaction is what we call an *enablement*.

Figure 1 shows an example with an interaction called I amongst three objects that must play roles P, Q and R. Objects p_1 and p_2 make offers to play role P, objects q_1 and q_2 make offers to play role Q, and object r_1 makes an offer to play role R. The objects p_1 and p_2 require that role Q must be played by q_1 and q_2 respectively, and vice versa. Neither p_1, q_1, p_2 nor q_2 establish constraints on what object should play role R. On the other hand, the object r_1 accepts that roles P and Q can be played by any object.

Since exclusion must be guaranteed, an object cannot commit to more than one interaction at a time. But, since an object can offer participation simultaneously in more than one interaction, it can be in more than one enablement. So, when two or more enablements share objects, they cannot be executed simultaneously. The set of enablements that cannot be executed are *refused*.

When an enablement of an interaction is executed, the objects in it can communicate by means of the interaction slots. A slot is a shared variable amongst the objects in the enablement which is created when the enablement is executed. These slots make up a local state that simulates the temporary global combined state in IP [3], being the most important difference that an object does not need to have access to the local state of other objects in order to get the information it needs. Obviously, a multiparty interaction delays an object that tries to read a slot that has not been initialized yet by another object.

3. STRATEGY TO IMPLEMENT THE CAL INTERACTION MODEL

In this section, we describe the strategy we have used to implement the CAL multiparty interaction model. We follow a "divide–and–conquer" strategy, since we split the execution of an interaction into two steps:

Synchronisation: The execution of an interaction begins when a set of entities get synchronised and commit to the interaction. This synchronisation can also be divided into two steps:

- **Enablement detection:** The participation offers are analysed to find sets of objects that agree in participating in an interaction, i.e, enablements.
- **Enablement selection:** When one or more enablements have been detected, as many as possible of them should be executed simultaneously, ensuring exclusion. Thus, an election under conflicting enablements needs to be held.

Communication: Once an enablement of an interaction having slots has been selected to execute, a set of slots must be instatiated for it. The details concerning communication fall beyond the scope of this paper.

The algorithm we have developed to implement synchronisation is called α. This algorithm uses a special resource per interaction, called *interaction coordinator*. This is an object which receives the offers of participation in a given interaction issued by entities in the system.

Enablement detection may be performed in a local manner in the interaction coordinator. In other words, a coordinator can compute the enablements originated by a set of offers made to it disregarding the offers made to other coordinators of potentially conflicting interactions. When the enablement detection algorithm finds out one enablement, this enablement is dealt with as if it was an entity which must compete to achieve exclusion over every participant in it. This task is performed by an *enablement selection* algorithm.

4. ENABLEMENT DETECTION

In this section we describe *α–solver*, which is the algorithm we have devised to implement enablement detection in the scope of α. *α–solver* builds a data structure which holds the information about every offer being processed. This data

structure is updated (i) every time an offer is received by the coordinator and (ii) every time that an offer is given up because the participant that made it is executing an interaction. The latest may happen either because an enablement of the interaction has been selected for execution, or because an enablement of another conflicting interaction has been selected for execution.

4.1 Data structures

Consider for example a system like the one in Figure 1 with an interaction called I amongst three objects that must play roles P, Q and R. Assume that objects p_1 and p_2 make offers to play role P, objects q_1 and q_2 make offers to play role Q and that object r_1 makes an offer to play role R.

α-solver uses an *acyclic directed graph* to store the information about the offers being processed. Every node in the graph holds a data structure that we call *tuple*, such as $[p_1, (q_1), ()]$. This tuple represents the offer made by p_1 and it means that it wants to play role P in interaction I, requires q_1 to play role Q, and does not care about which object should play role R. We say that role P is consolidated in this tuple, whereas role Q requires object q_1 and role R accepts any object. So, when an offer is received, a tuple with its information is created for it, and the graph is updated with this tuple and probably with other tuples calculated from it. Some of this calculated tuples may stand for an enablement. Furthermore, when the participant that made the offer commits to an interaction, the tuple which holds the information related to such offers is removed from the graph, and the tuples calculated from it are removed or updated to take this change into account.

Figure 2 shows the graph built by our algorithm as the offers made by the objects in our example arrive at the coordinator responsible for interaction I. Assume that the offer made by p_1 arrives first so that *α-solver* constructs a graph with only one node $[p_1, (q_1), ()]$. If the second offer is made by object p_2, a new node of the form $[p_2, (q_2), ()]$ is added to the graph, and no connecting node is constructed because the tuples so far processed are not *compatible*, i.e., objects p_1 and p_2 cannot interact together. If the offer made by q_1 is then received, a node of the form $[(p_1), q_1, ()]$ is added. Since it is compatible with $[p_1, (q_1), ()]$, a connecting node of the form $[p_1, q_1, ()]$ is added. It indicates that both p_1 and q_1 want to participate in interaction I and agree in committing to it together with any object playing role R. Notice that no enablement is found until object r_1 makes its offer. When this happens, two enablements are found simultaneously, but they are conflicting because they share r_1.

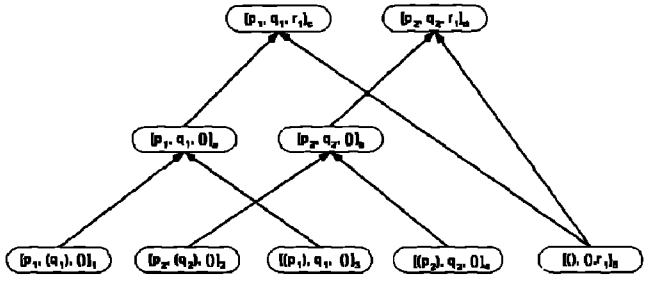

Figure 2: Consolidation graph for the system in Figure 1.

In order to formalise the concept of *compatibility* amongst tuples, we define a *consolidation* operator that is defined on both the tuples of the graph and its elements. We refer to this operator as $\odot$ and it is defined on tuples as $[e_1, e_2, \ldots, e_n] \odot [e'_1, e'_2, \ldots, e'_n] = [e_1 \odot e'_1, e_2 \odot e'_2, \ldots, e_n \odot e'_n]$. It is defined on the elements of a tuple by means of the following axioms:

1. $p_1 \odot (p_2) = (p_2) \odot p_1 = p_1$, as long as $p_1 = p_2$
2. $(p_1) \odot (p_2) = (p_2) \odot (p_1) = (p_1)$, as long as $p_1 = p_2$
3. $p \odot () = () \odot p = p$
4. $(p) \odot () = () \odot (p) = (p)$
5. $() \odot () = ()$

Note that this operation is defined on two tuples if and only if both tuples represent offers that can lead to an enablement. Furthermore, if the $\odot$ operation is defined between two tuples, the resulting tuple holds the combined information of the *consolidated* tuples. For example, this is the case of $[(p_1), q_1, ()]$ and $[p_1, (q_1), ()]$ in Figure 2. The consolidation of those tuples is a tuple $[p_1, q_1, ()]$, which means that p_1 wants to play role P in the interaction, that q_1 wants to play role Q, and that both of them accept that role R can be played by any object. Furthermore, the consolidation operation is not defined on tuples $[p_1, (q_1), ()]$ and $[p_2, (q_2), ()]$, because they are incompatible since both p_1 and q_1 are willing to play role P, and p_1 requires that role Q be played by q_1 and p_2 requires the same role to be played by q_2. Since the graph is built with consolidation amongst the tuples which represent the offers, we usually refer to it as the *consolidation graph*. In this graph, we refer to the top–most tuples (having no outgoing edge) as *roots*, and we refer to the bottom–most tuples (with no incoming edge) as *leaves*.

4.2 Processing offers

Figure 3 shows a routine called $ProcessOffer(T, G)$. This routine is the entry–point to *α-solver*. Parameters T and G represent the offer being processed and the current consolidation graph, respectively. It simply iterates over the set of roots of graph G and calls routine $Search(T, R)$ (presented in the same figure) on each one. Its parameters T and R represent the current offer and the root where search begins, respectively. This routine first tries to consolidate tuples T and R, and if it is possible, the consolidated tuple is returned and inserted in the graph as a parent of both T and R. Otherwise, a recursive search is performed in the subgraph whose root is the left child of R. If a consolidation $left$ is found there, it recursively tries to find out a new consolidation of $left$ with a tuple in the subgraph whose root is the right child of R. If such a consolidation if found, then it is returned because it is the most consolidated tuple that has been found; else, $left$ is returned. If no consolidation is found while examining the left subgraph of R, then the right subgraph is also explored. If no consolidation is found, then *null* is returned.

4.3 Offer cancellation processing

As we stated before, when an object commits to an interaction, the offers it made give up being valid. So, the information related to them must be removed from the consolidation graph.

For example, assume that participant p_1 in Figure 1 commits to another interaction. Then, the tuple $[p_1, (q_1), ()]$ must be removed from the consolidation graph in Figure 2. If this tuple was just removed from the graph, it would result in an inconsistent state since the tuple $[p_1, q_1, ()]$ would have one only descendant. Every tuple in the consolidation graph has either two or zero descendants, since the $\odot$ operator is a binary operator.

Figure 4 shows a routine called $ProcessCancel(P, G)$. This routine is the entry–point to the *α–solver* offer cancellation algorithm. Parameters P and G are the participant whose offers are canceled and the current consolidation graph, respectively. The $ProcessCancel$ routine iterates over the set of leaves of graph G having object P in consolidated state (in other words, the offers made by P), and calls routine $Delete(T, R)$ on each one. Its parameter T represents the offer being deleted. The deletion algorithm consists of (i) replacing every parent of the tuple being deleted by the other descendant of the parent (we assume that $Brother(T, p)$ returns the other descendant that p of tuple T) and (ii) calling a recursive routine $Rebuild(T)$ (Figure 5) which recursively re–consolidates the ancestors of tuple T.

The full cancellation process for the offers of p_1 is sketched in Figure 6. The only leaf having p_1 in consolidated state is $[p_1, (q_1), ()]$, so this tuple is passed to routine *Delete* as argument T. This algorithm first (1) replaces tuple $[p_1, q_1, ()]$ by its other descendant, the tuple $[(p_1), q_1, ()]$. Since this would left the graph into an inconsistent state, the *Rebuild* routine reconsolidates (2) the tuple $[(), (), r_1]$ with $[(p_1), q_1, ()]$, giving a tuple $[(p_1), q_1, r_1]$ which replaces $[p_1, q_1, r_1]$. Finally (3), the tuple $[p_1, (q_1), ()]$ is deleted.

It is worth noting that the new root $[(p_1), q_1, r_1]$ which replaces $[p_1, q_1, r_1]$ it is not an enablement. In other words, the consolidation graph has lost an enablement as a consequence of the cancellation. This is what we expected to happen, because p_1 has cancelled its offers because it committed to another interaction. When one enablement of an interaction executes, every conflicting enablement of other or even the same interaction must be refused. In our example, the enablement $[p_1, q_1, r_1]$ has been refused.

The $ProcessOffer$ and the $CancelOffer$ algorithms are formally proven to be correct in [8]. The correctness proof for the $ProcessOffer$ algorithm relies on proving that the *Search* algorithm always finds the most consolidated tuples for the input tuple T. We can easily prove that if the input tuple T can consolidate with a root of the graph, the result is the most consolidated tuple for that root. And if the consolidation with the root is not possible, it can be recursively proven that the *Search* algorithm finds the most consolidated tuple under that root. Then, since a tuple having every role consolidated is an enablement, and since every root in the graph is processed, we can prove that the algorithm finds out every enablement originated by an offer. The correctness proof for the $CancelOffer$ algorithm relies on proving that it only affects the tuples containing information about the cancelled offer.

4.4 Optimizing $ProcessOffer$

Figure 3 shows the basic enablement detection algorithm. The basic *Search* algorithm shown in this figure can be optimized in three ways, without loss of correctness.

It is worth noting the costs that those optimizations imply. The first optimization is just an algorithmic criterion,

```
ProcessOffer (T: Tuple; G: Graph): Set of Tuple
  enablements: Set of Tuple
  roots: Set of Tuple
  C: Tuple

  enablements ← ∅
  roots ← Roots(G)
  add T to G as an unconnected leaf

  for every R in roots do
    C ← Search (T, R)
    if (C is not null) and
       (every role in C is consolidated then
      enablements ← enablements ∪ {C}
    end if
  end for

  return enablements
end ProcessOffer

Search (T: Tuple; R: Tuple): Tuple
  result: Tuple;
  left, right: Tuple;

  if T and R can consolidate then
    result ← T ⊙ R
    let result be the parent tuple of both T and R
  else
    if T is not a leaf then
      left ← Search (T, leftChild(R))
      if left is not null then
        right ← Search (left, rightChild(R))
        result ← (right is not null ? right : left)
      else
        right ← Search (T, rightChild(R))
        result ← (right is not null ? right : null)
      end if
    else
      result ← null
    end if
  end if

  return result
end Search
```

Figure 3: *α–solver* enablement detection algorithm.

```
ProcessCancel (P: Object, G:Graph)

  for every leaf p in graph G having P consolidated do
    Delete (p)
  end for

end ProcessCancel

Delete(T: Tuple)
  brother: Tuple

  for every p in Parents(T) do
     brother ← Brother (T, p)
     Replace (p, brother)
     for every grandparent in Parents(p) do
       Rebuild (grandparent)
     end for
  end for
  delete T from G

end Delete
```

Figure 4: *α–solver* cancellation function.

```
Rebuild (T: Tuple)
  parents: Set of Tuple
  left, right: Tuple

  left ← leftChild (T)
  right ← rightChild (T)
  Replace (T, left ⊙ right)
  for every p in Parents(T) do
    Rebuild (p)
  end for

end Rebuild
```

Figure 5: Recursive rebuild after deletion.

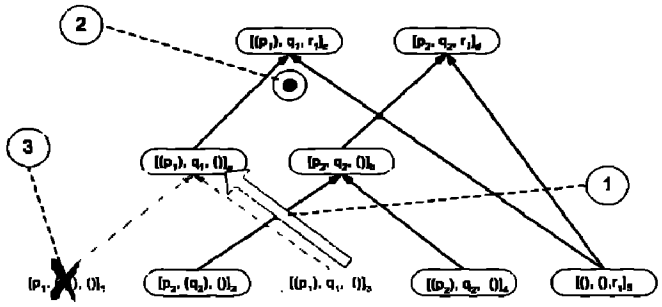

Figure 6: Cancellation of offers from p_1 in the graph in Figure 2.

having no cost on memory usage. The second optimizations require two boolean and one integer attributes for every tuple, and one offer counter. The third optimization requires the use of indexing functions, that are usually implemented by means of hash tables or similar data structures. Since in some theoretical scenarios this can amount to an important memory consumption, we have implemented it as an optional feature, allowing the operator to decide when it should be performed or when it should not.

4.4.1 Search Stop

The first optimisation we can apply relies on the fact that in the *Search* algorithm, when it tries to consolidate the input tuple T with a R tuple in the graph, and every role in *consolidated* state in R fails to consolidate with its partner in T, there is no tuple R' descendant of R that could be consolidated with T. Then, no more recursion is needed to process the descendants of T.

For example, let us assume that the input tuple $T = [p_1, (q_1), (r_1), (s_1)]$ is being checked for consolidation with a tuple $R = [(), q_2, r_2, s_2]$. Tuple R has consolidated q_2, r_2 and s_2 in roles Q, R and S respectively. But those three fail to consolidate with (q_1), (r_1) and (s_1). So, it can be proven that there is no tuple among the descendants of R that can be consolidated with T.

4.4.2 Avoiding re–processing of nodes

The second optimization we propose relies on the fact that the consolidation graph consists of a number of binary trees sharing nodes. Since a node can be reached from more than one root, it may be processed more than once. But, note that in algorithm *Search* a tuple in the graph can be checked for consolidation (i) with the input tuple R or (ii) with a consolidation of the input tuple T with another tuple in the graph. This happens when a consolidation is found in the graph while processing the left subtree of a tuple, and then a consolidation for it is searched in the right subtree of the tuple. So, we can label every tuple in the graph with a boolean *T−checked* flag that is set to true when the tuple is checked with the T input tuple. The *Search* algorithm, when is searching a consolidation for the T input tuple, checks the *T−checked* flag of every tuple about to be processed, ignoring the tuples (and its descendants) having this flag set to *true*. Furthermore, it can be proven that if no consolidation was found under a tuple when processing an offer, regardless of tuple being processed (T or a consolidation of T with other tuple), no consolidation can be found in a subsequent processing. So, we can label every tuple with a *empty* boolean flag that is set to true when the tuple is processed and no consolidation is found under it. When a tuple with *empty* = *true* is about to be processed, it can be ignored since no consolidation can be found.

Labeling every tuple with attributes has an important drawback, because they must be initialized before every offer is processed. This requires that the whole graph must be traversed, and that is precisely what we need to avoid. A solution to this problem is to label every tuple with an *age* integer attribute, and using an offer counter that is updated with every offer processed. When a tuple is going to be processed by the *Search* algorithm, its *age* attribute is compared with the offer counter. If its *age* value is smaller than the offer counter, that is the first time that the tuple is being processed for the current offer, so its attributes *T−checked* and *empty* must initially be set to *false*. If its *age* attribute equals the offer counter, that means that the tuple has already been visited along the current offer processing, so its *T−checked* and *empty* attributes must be taken into account. Note that when the offer counter is about to overflow, it must be reset to zero, and the full graph must be traversed to reset every tuple *age* attribute. But, if we use 32 or 64 bits for the counters, this is not a problem in practice.

4.4.3 Starting Search on advantageous nodes

Finally, the *Search* algorithm can be optimized making that the search process begin in nodes as close as possible to tuples in the graph that consolidates with the input tuple T, instead of beginning the search from the roots of the graph. This optimization relies on the principle that if a T input tuple is like $[p_1, \ldots]$ any tuple that consolidates with it must be like $[(p_1), \ldots]$ or $[(), \ldots]$. We can prove that when a process of a p participant under role P is being processed, if the search begins (i) in the tuples that *require* the participant p under role P and (ii) in the tuples that *accept* any participant under role P, such that there is no other tuple above it in the graph that requires or accepts the p participant under role P, the correctness of the *Search* algorithm is preserved.

5. SELECTION ALGORITHMS

Once an enablement has been found by the enablement detection algorithm, it must have a chance to be executed. A selection algorithm must ensure exclusion, deciding thus whether each enablement must be selected or refused. An enablement that does not conflict with others should be selected immediately. On the other hand, if an enablement is

rejected, that is because it conflicts with another one that has already been selected.

Note that since an enablement is determined by a fixed set of participants, the enablement selection problem is analogous to the problem of interaction selection in a static interaction model. since *α–solver* is well–encapsulated into both routines *ProcessCancel* and *ProcessOffer*, it could work together with any selection algorithm that fulfills the following requirements:

- Coordinators of interactions do not need to be aware one another.
- Participants in an interactions do not need to be aware one another

If coordinators of interactions do not need to be aware one another, there is no problem in that new interactions (enablements) do appear at run–time, since there is no need nor possibility of communication among them. On the other hand, since participants do not necessarily know at compile–time the other participants that are going to participate with them, it is very important that the algorithm do not need the participants to be aware one another.

We have developed an algorithm with fulfills those requirements, an that have been successfully integrated with *α–solver*. This algorithm is called *α–core*, and constitutes together with *α–solver* the implementation of α that we developed for the framework that provides CAL run–time support. A sketch of *α–core* can be found at [7], and the full description and proofs of correctness can be found at [8].

6. PERFOMANCE

We have implemented the algorithms *ProcessOffer* and *ProcessCancel* described at section 4, and have perform some tests in order to measure their performance. The scenario we have used in our tests is depicted in Figure 7.

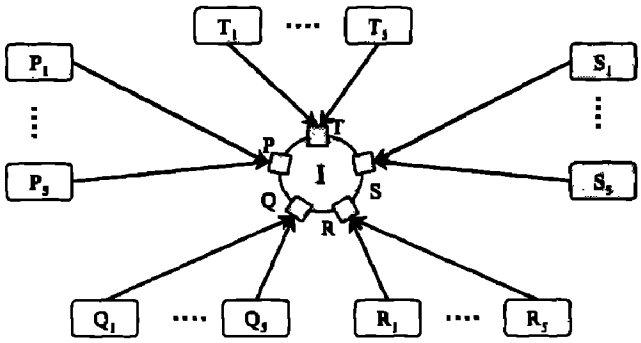

Figure 7: Scenario for testing *α–solver*.

This scenario consists of an interaction I with five roles P, Q, R, S and T, and 25 participants, offering five of them to participate under each role in the interaction: participants $P_1, \ldots, P_5$ offer to participate under rol P, participants $Q_1, \ldots, Q_5$ offer to participate under rol Q, and so on. We have used a five–party interaction because is accepted that greater cardinality interactions have little practical applications [3].

As a measure of our algorithms performance, we have measured the average number of times that the consolidation operation $\odot$ is computed amongst two tuples, every time that an offer or a cancellation is processed.

6.1 Perfomance of *ProcessOffer*

To measure how the algorithm *ProcessOffer* performs, we have run five tests $T_1, \ldots T_5$. Test T_1 is the worst case for the algorithms, since every participant accepts that every role in the interaction can be played by any other participant. So, $5^5 = 3,125$ enablements are found. In tests $T_2, T_3, \ldots$ participants make their offers more and more restrictive, imposing a restriction on the participant that must play role $P, Q, \ldots$ and so on, being T_5 the best case since every participant restricts that every role in the interaction must be played by another concrete participant. So, the number of enablements found in test $T_2 \ldots T_5$ are respectively $5^4, 5^3, 5^2$ and 5^1.

Since the third optimization proposed on *ProcessOffer* proposed in section 4.4 algorithm has been implemented as optional, we have run twice every test $T_1 \ldots T_5$: once with the optimization enabled, and once with this optimization disabled. This permits to compare how good the optimization is. The increase of memory usage due to the optimization has never been greater than 15%.

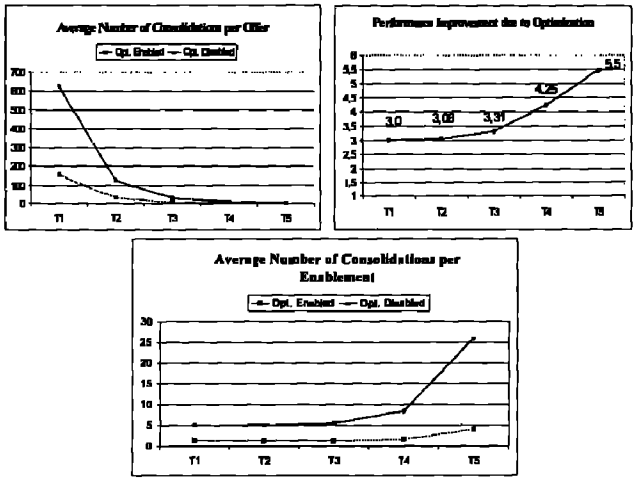

Figure 8: *ProcessOffer* algorithm performance.

The first plot in Figure 8 shows the average number of consolidation operations computed per offer in each run. We can appreciate that the number of operations computed decreases dramatically when the test is run with the optimization enabled. The second plot shows the relationship amongst the number of operations computed with and without the optional optimization. The improvement due to the optimization increases as the restrictions imposed by the participants are more restrictive, since they provide more information useful to determine where the search process should begin.

Nevertheless, the results of test T_1 in first plot may seem poor. Is executing an average of 470 or 156 consolidation operations per offer a good performance? Yes, indeed it is, because we should also take into account the average number of enablements found per offer. The third plot in Figure 8 shows the average number of consolidation operations computed per enablement found, as much in the test with and

without the optional optimization. This gives us the cost of finding an enablement. So, in test T_1 the average cost of finding an enablement is less than four consolidation operations if the optional optimization is not enabled, and less than two otherwise. It is very important noting that the cost of finding an enablement decreases as the number of enablements that can be found increases.

6.2 Perfomance of *CancelOffer*

To measure how the algorithm *CancelOffer* performs, we have run again the same five tests $T_1, \ldots T_5$ from previous section, cancelling every offer after each run.

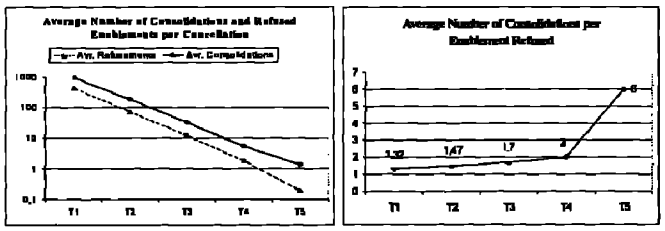

Figure 9: *CancelOffer* algorithm performance.

The first plot in Figure 9 shows the average number of consolidation operations computed per cancellation, and the average number of enablements refused each time. As in the *ProcessOffer* algorithm, the numbers in test T_1 may seem excessively high. But once again, we should take into account the number of enablements refused by every cancellation. The second plot in the Figure shows the average number of consolidation operations computed per enablement refused. This is the cost of refusing an enablement. As for *ProcessOffer* algorithm, we can see how the cost of refusing an enablement decreases as the number of enablements increases. We can find the explanation for this effect in the consolidation graph topology. When there are many enablements in the graph, frequently many of those enablements share leaves nodes. When an offer having many enablements as ancestors in the graph is canceled, the cost of refusing those enablements is smaller that if the tuple had one or none enablements as ancestors.

7. CONCLUSIONS

In this paper, we have described the algorithms we have developed to implement the CAL interaction model, focusing on the problem of enablement detection. This is an open multiparty model useful for applications that require coordination amongst entities that are not fixed beforehand. Although the problem of finding all the sets of entities that agree to coordinate through an interaction has a high computational cost, our algorithms performs quite well since it behaves more efficiently as the complexity increases.

Our enablement detection algorithm can cooperate with any selection algorithm that fulfills some conditions. We think that this is an important feature, because it leaves an open door to deal with other problems related to selection, such as fairness.

APPENDIX

A. REFERENCES

[1] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, Sept. 1989.

[2] R. Corchuelo, J. Pérez, and M. Toro. A Multiparty Coordination Aspect Language. *ACM SIGPLAN*, 35(12):24–32, Dec. 2000.

[3] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[4] Y. Joung. A comprehensive study of the complexity of multiparty interaction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages POPL'92*, pages 142–153. ACM Press, Jan. 1992.

[5] Y. Joung and S. Smolka. A completely distributed and message-efficient implementation of synchronous multiprocess communication. In P.-C. Yew, editor, *Proceedings of the 19th International Conference on Parallel Processing. Volume 3: Algorithms and Architectures*, pages 311–318, Urbana-Champaign, Illinois, Aug. 1990. Pennsylvania State University Press.

[6] Y. J. Joung and S. A. Smolka. Strong interaction fairness via randomization. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 475–483, Hong Kong, May 1996. IEEE Computer Society Press.

[7] J. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. A framework for aspect–oriented multiparty coordination. In *New Developments in Distributed Applications and Interoperable Systems*, pages 161–174. Kluwer Academic Publishers, 2001.

[8] J. A. Pérez. *Un Framework Orientado a Aspectos para la Descripción del Comportamiento Coordinado. Aplicación a los Sistemas Multiorganizacionales*. PhD thesis, Facultad de Informática y Estadística. Dpto. de Lenguajes y Sistemas Informáticos. Universidad de Sevilla, 2001.

[9] Y. Tsay and R. Bagrodia. A real-time algorithm for fair interprocess synchronization. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 716–723, Washington, D.C., USA, June 1992. IEEE Computer Society Press.