Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Logical Modelling of a Distributed Mutual Exclusion Algorithm in the Situation Calculus

Alvaro E. Campos, Claudia C. Navarrete, and Javier Pinto

Departamento de Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 306 - Santiago 22 - CHILE
Fax: +56 2 686-4444
[acampos,ccnavarr,jpinto]@ing.puc.cl

Abstract

This article presents the development of a logical model for a distributed mutual exclusion algorithm. This work represents the first step towards the development of a logical language, based on the situation calculus, to describe and analyze the dynamic behavior of distributed algorithms and protocols.

The work shows that the situation calculus may in fact be used. However, it also shows that in its basic form this language has some limitations in its expressiveness. These limitations motivate some extensions that will enrich the language in such a way that some properties of distributed algorithms and protocols may be expressed in a more natural way.

1 Introduction

In recent years, distributed systems have become central to the use of computational systems. They appear as the solution for having increased computational power, even beyond what uniprocessors may ever be able to offer, at reasonable costs [1]. Still, they present great challenges regarding the development of reliable applications. Due to the inherent complexity of programming this type of systems, achieving this goal is not simple.

Although several formal mechanisms have been proposed, in practice, when developing a distributed algorithm, informal language is often used for the specification of the problem it solves, to reason about its properties, and even to "prove" its correctness. Certainly, this hardly ensure that the algorithm really satisfies its specification. One goal in this area is to have mechanical provers that may automatically determine whether an implementation of a given algorithm fulfills its specification.

Many of the above mentioned formal mechanisms are based on logical languages, that is the case, for example, of the assertional methods based in temporal logic [2]; some others are simply ad-hoc procedures, whose general applicability remains to be seen. This work proposes the development of a language based on first order logic to specify the tasks to be performed by distributed algorithms and protocols; it also provides deductive mechanisms for the analysis of their properties. In particular, the situation calculus [3] is used as the basic logic language for modelling.

The situation calculus is a first-order language extended with a second-order induction axiom. This language has been developed for the representation of knowledge about actions and change in dynamic worlds. In particular, it has been used to model transactions in databases, and the semantics of programming languages. Current research focuses on its applicability for the specification of transactions in active and temporal databases, and in the specification of agents in dynamic worlds.

This work aims to extend the situation calculus for modelling agents in a distributed system, and it is based on previous extensions proposed in [4, 5]. The main thesis behind this work is that the situation calculus is suitable as the basic mechanism for specifying distributed-systems behavior and properties, and for the analysis of the dynamics of such systems. In this proposal, algorithms in a distributed system may be modelled as a sequence of actions; each action carrying out changes in the state of the objects participating in the system.

This article shows a first step towards that main objective. It presents the development of a logical model for a distributed mutual exclusion algorithm using the situation calculus. The model allows a formal proof of its specification.

2 Distributed Mutual Exclusion Algorithm

The algorithm modelled in this work was proposed by Ricart and Agrawala in the year 1981 [6]. Their algorithm is distributed and works in a network whose nodes do not share memory, and that can only communicate through sending and receiving messages. In this algorithm, when a node wants to enter its critical section it first sends a message (a *request*) to every other node asking for their permission.

The *request* message contains an identification of the sender and a sequence number. The identification of the sender is an integer that uniquely identifies the sending node. The sequence number is also an integer; it uniquely identifies the message among those sent by that same sender.

When a node B receives a *request* sent by node A, it should send a message to A with its authorization to enter the critical section. This message is called a *reply* and its contents is not relevant. The *reply* is sent immediately if B has no interest in entering its critical section, or if A has priority over B and B has not yet entered its critical section. Node A has priority over B when the sequence number of the *request* sent by A is smaller than that of B; ties in sequence numbers are resolved in favor of the smaller node number. If B has already entered its critical section or if it has priority over A the sending of the *reply* is deferred until after B leaves its critical section.

Next, an implementation of this algorithm is given using pseudo code. Each node in the system runs three independent processes. One process accesses the critical section and, therefore, invokes the mutual exclusion algorithm. A second process receives *request* messages from other nodes and grants permissions. The third process receives and collects *reply* messages. The pseudo code starts with a brief description of the variables and values that are used to implement the algorithm. Those variables and values are local to each node; in general, they are shared among the three processes running in a single node, but are not shared between nodes.

VARIABLES AND VALUES

```
Constants:
  me: this node's identification number
  N:  total number of nodes

Integer:
  my_seq_num: sequence number for a request
  highest_seq_num: highest sequence number
       seen in any request at this node
  outstanding_reply: number of reply messages
       still missing

Boolean:
  requesting_CS: true when node wants access to
       the critical section
  deferred_reply[1..N]: deferred_reply[j] is true
       when a reply to node j has been deferred

Binary semaphore:
  shared_vars: protects access to local
       shared variables
```

PROCESS THAT INVOKES MUTUAL EXCLUSION

```
/* Requests to enter the critical section */
    P(shared_vars); /* choose sequence number */
        requesting_CS = TRUE;
        my_seq_num = highest_seq_num + 1;
    V(shared_vars);
    outstanding_reply = N - 1;
    /* send a request to all the other nodes */
    for j = 1 to N do if (j != me) then
        send_Message(Request(my_seq_num, me), j);

/* Wait for reply messages of all other nodes */
    waitfor(outstanding_reply == 0);

/* Critical section is executed at this point */

/* Leave the critical section */
    requesting_CS = FALSE;
    for j = 1 to N do
        if deferred_reply[j] then
            /* send reply to j */
            send_Message(Reply, j);
            deferred_reply[j] = FALSE;
```

PROCESS THAT RECEIVES MESSAGE `Request(k, j)`

```
/* k is the sequence number - j is the sender */
    highest_seq_num = max(highest_seq_num, k);
    P(shared_vars);
      deferred = requesting_CS and
        ((k>my_seq_num) or
         (k=my_seq_num and j>me));
    V(shared_vars);
    if deferred then
        deferred_reply[j] = TRUE
    else
        send_Message(Reply, j);
```

PROCESS THAT RECEIVES MESSAGE `Reply`

```
    outstanding_reply = outstanding_reply - 1;
```

3 Situation Calculus

The situation calculus [3] is a first-order language extended with a second-order induction axiom. It was designed to allow the representation of worlds that change as the result of agents executing actions. The language has a second-order axiom to characterize the situation's space; it defines the sorts $\mathcal{A}$ and $\mathcal{S}$ that group the possible instances of actions and situations, respectively.

In general, the world is in a situation s, which can only be changed when an agent — for instance, a person, a computer, a process — carry out some action — for instance, move an object, execute an instruction. The situation resulting from carrying out an action a in the situation s is denoted as $do(a, s)$, with $a \in \mathcal{A}$ and $s \in \mathcal{S}$. There is a distinguished initial situation, which is denoted with the constant symbol S_0, from which actions can be executed.

In this language actions are represented as functions that, in general, may have arguments. For example, the action $update(x, var, val)$ may represent the action in which process x updates variable var with the value val. Situations are characterized through relations called *fluents*; the value of fluents may change from one situation to another, as the result of the action. Fluents are denoted with predicate symbols that must have a situation term as one of their arguments. For example, if a process sends a message in the situation s, then in the situation resulting from executing the action there will be a message in transit.

Before an action can be carried out, it is often the case that certain conditions must hold. These are necessary and sufficient conditions, which are called *preconditions*; they are denoted with the predicate that characterizes them. For example, a process can receive a message only if some process sent it a message. On the other hand, to describe the change that the execution of an action produces in the world, *successor-state* axioms are used. These axioms describe the change of each one of the fluents for all possible actions.

It is customary to think that situations are snapshots of the world, and that they describe it at a given time. This work uses the notion of situations as a history, which was introduced in [4]. Under this wiew, each situation has a unique history that produces it, that is, there is only one sequence of actions that brings the world from S_0 to it.

3.1 Function Symbols and Predicates

$do : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{S}$. If a denotes an action and s a situation, then $do(a, s)$ denotes the situation that results from executing a in s.

$\prec \subseteq \mathcal{S} \times \mathcal{S}$. If s_1 and s_2 are different situations, then $s_1 \prec s_2$ is true if some sequence of actions poduces s_2 starting from s_1. Its reflexive closure is defined as $s_1 \preceq s_2 \equiv s_1 \prec s_2 \vee s_1 = s_2$.

$ocurred_s \subseteq \mathcal{A} \times \mathcal{S} \times \mathcal{S}$. If a is an action and s_a a situation such that $s_a \preceq s$, then $ocurred_s(a, s_a, s)$ is true if action a was executed in situation s_a along the history s.

$ocurredOnce \subseteq \mathcal{A} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. If a is an action that has ocurred only once between situations s_i and s_f, then $ocurredOnce(a, s_a, s_i, s_f)$ is true, and the situation in which it happened was s_a.

$Poss \subseteq \mathcal{A} \times \mathcal{S}$. If a is an action and s a situation, then $Poss(a, s)$ is true if it is possible to execute action a in the situation s, that is, if the preconditions for a are satisfied in the situation s.

$< \subseteq \mathcal{S} \times \mathcal{S}$. If s_1 and s_2 denote situations, then $s_1 < s_2$ is true if some $possible$ sequence of actions takes the world from s_1 to s_2. Its reflexive closure is defined as $s_1 \leq s_2 \equiv s_1 < s_2 \vee s_1 = s_2$.

3.2 Occurrence of Actions

It was mentioned above that situations may be seen as histories. When there is a unique sequence of actions that change the world from the initial situation S_0 to a given situation s, this last situation stores all the information about the actions that were executed in the intermediate situations. However, in many cases, it is desirable to be able to refer explicitly to the specific actions that were executed to produce a given history. It is usually desirable as well to be able to refer to the evolution of a fluent's value along the history.

With this purpose in mind, a set of predicates are defined below. These predicates talk about the occurrence of actions in a history. It is also shown a lemma that shows the evolution of the value of a fluent along a history.

The execution of a distributed algorithm has a deterministic and a non deterministic part. The deterministic part is given by the unique order in which each process executes its own instructions. The non deterministic part comes from the concurrence with which different processes execute their corresponding part of the algorithm. At any given moment of the execution, it is possible to know only what instructions have been executed by each process and in which order.

Definition 1 *An action a ocurred in a situation s_a along the history s, if there is another situation s' in the history s that is the result of executing this action. Thus, the predicate $occurred_s$ is defined as:*

$$occurred_s(a, s_a, s) \equiv (\exists s')s' \leq s \wedge s' = do(a, s_a) \quad (1)$$

Definition 2 *An action a has ocurred only once in the situation s_a, starting from s_i, and along a history s_f, if in this path there is only one situation that is the result of executing this action in s_a. Thus, the predicate $occurredOnce$ is defined as:*

$$\begin{aligned} & occurredOnce(a, s_i, s_a, s_f) \equiv \\ & \quad s_i \leq s_a \wedge occurred_s(a, s_a, s_f) \wedge \\ & \quad (occurred_s(a, s'_a, s_f) \supset s'_a < s_i) \end{aligned} \quad (2)$$

Lets consider an action a that occurs in a situation s_a previous to s. Assume that this action makes the value of fluent R true. Further assume that after s_a there has been no action that makes R false. Then, necessarily R should be true in s.

Lemma 1 *Let s_i be a situation previous to s_f. If the value of fluent R in s_i is different to its value in s_f, then this value could have been changed only by the occurrence of an action a in a situation later or equal to s_i but previous to s_f.*

$$\begin{aligned} & R(s_i) \wedge \neg R(s_f) \wedge s_i < s_f \supset \\ & (\exists s_a, a) \\ & \quad occurred_s(a, s_a, s_f) \wedge s_i < s_a \wedge R(s_a) \wedge \\ & \quad \neg R(do(a, s_a)) \end{aligned} \quad (3)$$

For the proof of this lemma, it is necessary to make use of the induction axiom. Informally, the lemma shows that there must be a situation s_a, between s_i and s_f, such that the value of R changes between s_a and its immediate successor in the history s_f.

4 Modelling the System

This section shows the modelling of the mutual exclusion algorithm of Ricart and Agrawala using the situation calculus. First, the language (domains, functions, constants, fluents and actions) is described. Then, the preconditions and successor-state axioms are stated. Next, a predicate stating that a node is in its critical section is defined. Finally, the mutual exclusion condition is defined and its proof is discussed

4.1 Domains and Functions

In addition to domains $\mathcal{A}$ and $\mathcal{S}$, and to the constant S_0 defined in the previous section, it is necessary to include a domain to identify each node in the system.

$\mathcal{N}$: Set of natural numbers.

$Nodes$: Set of nodes in the distributed system.

$id : Nodes \rightarrow \mathcal{N}$. If x is a node in the system, then $id(x)$ returns a unique natural number assigned to that node.

4.2 Fluents

Each process has to remember some variables associated to different events — instructions of the program — that affect the execution of the mutual exclusion algorithm. For example, when a variable is updated it is necessary to remember its new value. Other events that it is necessary to remember are the sending and the reception of a message.

The proposed model represents this type of information through the use of fluents. The next entries define the fluents involved in the development of the model.

$RequestingCS(x, s)$: True if node x wants to enter its critical section in the situation s.

$LargestSeqN(x, n, s)$**:** True if n is the largest sequence number seen in any situation previous to s by node x.

$SeqN(x, n, s)$**:** True if n is the sequence number chosen by node x for its request in the situation s. For simplicity a function defined in terms of this fluent is also provided; the function has the same name of the fluent and is defined as $SeqN(x, s) = n \equiv SeqN(x, n, s)$.

$ReqInTransit(origin, destination, cont, s)$**:** True when node $origin$ has sent a *request* to node $destination$, whose content is $cont$, but this *request* has not been received yet in the situation s.

$ReplyInTransit(origin, destination, s)$**:** It is true when node $origin$ has sent a *reply* to node $destination$, but this *reply* has not been received yet in the situation s.

$ReqPending(x, y, c, s)$**:** True if in a situation previous to s a *request* containing c has arrived to y from x, and in the situation s node y has not sent the *reply* yet.

4.3 Actions

The algorithm executes through the ocurrence of different events that take place. In fact, these events correspond to a group of instructions executed by the nodes in the system. For example, the event *start mutual exclusion protocol* represents the following group of instructions executed by the process that accesses the critical section.

```
P(shared_vars); /* choose sequence number */
   requesting_CS = TRUE;
   my_seq_num = highest_seq_num + 1;
V(shared_vars);
```

These groups of instructions are modelled as actions executed by each one of the nodes. Next, the actions present during the execution of the algorithm are shown. It is necessary to remark that these actions say nothing about the order in which they are executed by the algorithm. The execution order is introduced later as a restriction in the histories in which a process can be when inside its critical section.

$startProtocol(x)$**:** Node x starts the mutual exclusion algorithm.

$chooseSeqN(x)$**:** Node x chooses a sequence number for a *request* to access the critical section.

$SendReq(origin, destination, cont)$**:** Node $origin$ sends a *request* containing $cont$ to node $destination$.

$receiveReq(origin, destination, cont)$**:** A *request* from node $origin$, containing $cont$, has been received by node $destination$.

$SendReply(origin, destination)$**:** Node $origin$ sends a *reply* to node $destination$.

$receiveReply(origin, destination)$**:** A *reply* from node $origin$ is received by node $destination$.

$updateSeqN(x, n)$**:** The highest sequence number is updated by node x to n.

$enterCS(x)$**:** Node x starts to execute in its critical section.

$leaveCS(x)$**:** Node x finishes executing in its critical section.

4.4 Preconditions for the Actions

The following entries define the precondition axioms for the above described actions.

The following actions do not have preconditions, since a given implementation is free to execute them in any order. As it was said above, the appropriate execution order to achieve mutual exclusion is introduced later as restrictions on the histories.

$$Poss(enterCS(x), s) \equiv True$$

$$Poss(chooseSeqN(x), s) \equiv True$$

$$Poss(SendReq(x, y, c), s) \equiv True$$

$$Poss(updateSeqN(x, n), s) \equiv True$$

$$Poss(leaveCS(x), s) \equiv True$$

Next come the control flow instructions. They are modelled as preconditions for actions.

A node x can begin the mutual exclusion protocol if it has not begun it before:

$$Poss(startProtocol(x), s) \equiv \neg RequestingCS(x, s)$$

A node x can send a *reply* to y, if there was an outstanding *request* from y not yet replied by x, and if x does not want to enter its critical section or if y has higher priority than x.

$$\begin{aligned}&Poss(SendReply(x, y), s) \equiv \\ &\quad ReqPending(y, x, (nsecy, idy), s) \wedge \\ &\quad (\neg RequestingCS(x, s) \vee (nsecy \leq SeqN(x, s) \wedge \\ &\quad (nsecy \neq SeqN(x, s) \vee idy \leq id(x)))\end{aligned}$$

Finally, to execute the actions related with the reception of messages, it is necessary that the messages in question have been sent previously.

$$\begin{aligned}&Poss(receiveReq(x, y, c), s) \equiv \\ &\quad ReqInTransit(x, y, c, s)\end{aligned}$$

$$\begin{aligned}&Poss(receiveReply(x, y), s) \equiv \\ &\quad ReplyInTransit(x, y, s)\end{aligned}$$

4.5 Successor-State Axioms

The successor-state axioms model the effects that the actions (instructions) have on the fluents described above.

$$\begin{aligned}&Poss(a, s) \supset RequestingCS(x, do(a, s)) \equiv \\ &\quad (RequestingCS(x, s) \wedge a \neq leaveCS(x)) \vee \\ &\quad a = startProtocol(x)\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset SeqN(x, n, do(a, s)) \equiv \\ &\quad (SeqN(x, n, s) \wedge a \neq chooseSeqN(x)) \vee \\ &\quad (LargestSeqN(x, n, s) \wedge a = chooseSeqN(x))\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset LargestSeqN(x, n, do(a, s)) \equiv \\ &\quad (LargestSeqN(x, n, s) \\ &\quad \wedge (\nexists n')a = updateSeqN(x, n')) \vee \\ &\quad a = updateSeqN(x, n)\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset ReqInTransit(x, y, c, do(a, s)) \equiv \\ &\quad (ReqInTransit(x, y, c, s) \\ &\quad \wedge a \neq receiveReq(x, y, c)) \vee \\ &\quad a = SendReq(x, y, c)\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset ReplyInTransit(x, y, do(a, s)) \equiv \\ &\quad (ReplyInTransit(x, y, s) \\ &\quad \wedge a \neq receiveReply(x, y)) \vee \\ &\quad a = SendReply(x, y)\end{aligned}$$

$$\begin{aligned}&Poss(a, s) \supset ReqPending(x, y, c, do(a, s)) \equiv \\ &\quad (ReqPending(x, y, c, s) \\ &\quad \wedge a \neq SendReply(y, x)) \vee \\ &\quad a = receiveReq(x, y, c)\end{aligned}$$

4.6 The Mutual Exclusion Property

Considering situations as histories [4], a given execution of a program is represented by a situation s. This situation represents the order in which the instructions were executed, because there is only one action sequence that takes the system from its initial situation S_0 to situation s.

Therefore, it is possible to define a predicate that imposes some restrictions on the history s, to represents those executions that satisfy a desired property. In particular, it is interesting to represent all possible histories in which a process can be in its critical section.

Definition 3

$$\begin{aligned}&InCriticalSection(x, s) \equiv \\ &(\exists s_1, s_2, s_8) s_1 < s_2 < s_8 \wedge \\ &\quad occurredOnce(startProtocol(x), s_1, s_1, s) \wedge \\ &\quad occurredOnce(chooseSeqN(x), s_1, s_2, s) \wedge \\ &\quad (\forall y)(x \neq y \supset \\ &\quad\quad (\exists s_3, s_4, s_5, s_6, s_7) \\ &\quad\quad (s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 \wedge \\ &\quad\quad occurredOnce(SendReq(x, y, c), s_1, s_3, s) \wedge \\ &\quad\quad occurredOnce(receiveReq(x, y, c), s_1, s_4, s) \wedge \\ &\quad\quad c = (SeqN(x, s_3), id(x)) \wedge \\ &\quad\quad occurredOnce(updateSeqN(y, SeqN(x, s_3), \\ &\quad\quad\quad\quad s_1, s_5, s) \wedge \\ &\quad\quad occurredOnce(SendReply(y, x), s_1, s_6, s) \wedge \\ &\quad\quad occurredOnce(receiveReply(y, x), s_1, s_7, s))) \\ &\quad occurredOnce(enterCS(x), s_1, s_8, s) \wedge \\ &\quad (\nexists s_9)(s_1 < s_9 \wedge \\ &\quad occurred_s(leaveCS(x), s_9, s))\end{aligned}$$

4.7 Correctness of the Model

The algorithm is correct if indeed it provides mutual exclusion. The objective of the previous axiomatization is to establish the next theorem.

Theorem 4.1 *From the previous axiomatization of the mutual exclusion algorithm, it is inferred that if two nodes are in their critical sections in the same situation s, then they must be the same node.*

$$\begin{aligned}\Sigma \models\ & InCriticalSection(a, s) \wedge \\ & InCriticalSection(b, s) \supset \\ & \quad id(a) = id(b) \end{aligned} \tag{4}$$

Where Σ represents the basic axioms of the situation calculus, together with the definitions and axioms described in sections 3 and 4.

Proof: A complete formal proof of the theorem may be found in [7]. Informally, it starts assuming that two nodes, a and b, are in their critical sections in situation s. Thus, according to definition 3, a series of actions must have happened before s. In particular a should have authorized b to enter its critical section and vice versa; that is, both nodes should have executed the action $SendReply$. Therefore, the preconditions of these actions must have held at the moment in which they were executed. Finally, these preconditions hold simultaneously only if node a is the same as node b.

5 Concluding Remarks

The main objective of this research is to study the feasability of modelling distributed algorithms and protocols using the language of the situation calculus. The work presented in this article is a first step towards that goal. Nonetheless, it shows that it is indeed possible to use the situation calculus to build a model for the mutual exclusion algorithm of Ricart and Agrawala [6]. Not only that, but it also shows the feasability of expressing properties of the algorithm and proving them in a mechanical way. In particular, the article shows how the property of mutual exclusion may be proven; however, it is also possible to model and prove other properties, such as *absence of deadlock* and *eventual entry*.

That is, the situation calculus has enough expressive power to represent the mutual exclusion algorithm. However, it was possible to verify that in many aspects the modelling was rather unnatural. This can be more clearly seen in definition 3, which models a certain execution order of some instructions of a concurrent program. This formulae is too complex, distracting the reader's attention towards understanding its complexity, instead of understanding the ideas behind.

Therefore, it becomes necessary to extend the language of the situation calculus in such a way that some concurrent programming concepts may be expressed in a more natural way. Currently, the definition of such an extension is under development.

Acknowledgments

The authors would like to acknowledge gratefully that this work has been partially funded by Fondecyt grants Nº 196-0381 and Nº 197-1304.

References

[1] Andrew Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[2] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[3] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[4] Javier Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. Submitted to the Journal of Logic and Computation URL = ftp://lyrcc.ing.puc.cl/pub/jpinto/jlc.ps.gz.

[5] Raymond Reiter. The frame problem in the situation calculus: a soundness and completeness result, with an application to database updates. In *Proceedings First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[6] Glenn Ricart and Ashok Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[7] Alvaro E. Campos, Claudia C. Navarrete, and Javier Pinto. Modelación lógica de un algoritmo de exclusión mutua distribuido. Technical Report RT-PUC-DCC-97-03, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, Santiago, Chile, December 1997.