World Academy of Science, Engineering and Technology
International Journal of Electronics and Communication Engineering
Vol:1, No:11, 2007

Formal Verification of a Multicast Protocol in Mobile Networks

M. Matash Borujerdi, S.M. Mirzababaei

***Abstract*—**As computer network technology becomes increasingly complex, it becomes necessary to place greater requirements on the validity of developing standards and the resulting technology. Communication networks are based on large amounts of protocols. The validity of these protocols have to be proved either individually or in an integral fashion. One strategy for achieving this is to apply the growing field of formal methods. Formal methods research defines systems in high order logic so that automated reasoning can be applied for verification. In this research we represent and implement a formerly announced multicast protocol in Prolog language so that certain properties of the protocol can be verified. It is shown that by using this approach some minor faults in the protocol were found and repaired. Describing the protocol as facts and rules also have other benefits i.e. leads to a process-able knowledge. This knowledge can be transferred as ontology between systems in KQML format. Since the Prolog language can increase its knowledge base every time, this method can also be used to learn an intelligent network.

***Keywords*—** Formal methods, MobiCast, Mobile Network, Multicast.

I. Introduction

NOWADAYS as the computer systems become increasingly complex, the difficulty of developing highly reliable technology increases as well. Numerous design flaws have been chronicled such as the Intel Pentium processor floating-point bug and the Denver Airport baggage handling system problem. System developers have to test and verify the products before releasing them to the market to avoid later problems. One of the most complex system categories that may have flaws is the computer network. Two major types of methods are used to validate network protocols, manual methods and formal methods. M. Goda [6,7] categorized these methods in 94, 95 and divided the manual methods to static and well-formed formulas. J. Holzman [1] categorized the automatic methods into four groups: full search, controlled partial search, random simulation, and formal verification. There are other similar works in formal verification of IPv6 [2] and also the session layer [5]. The potentials of formal methods to provide solutions in these areas are also found in [8, 9, and 10].

This paper demonstrates how a formerly announced network protocol such as MobiCast can be specified for theorem proving in Prolog language. Then semi-automated reasoning can be applied to the specification to verify certain properties. In the following sections we will first discuss the MobiCast protocol. Then we will show the relationships between definitions of graphs and communications and the implementation of the MobiCast. After that we will discuss our verification method before our final conclusions.

Manuscript received January 9, 2004.

M. Matash Borujerdi, is with the Department of IT and Computer Engineering, Amirkabir University of Technology, Tehran, Iran, (e-mail: borujerm@aut.ac.ir).

S.M. Mirzababaei, also is with the Department of IT and Computer Engineering, Amirkabir University of Technology, Tehran, Iran, (e-mail: mirzababaei@morva.net).

II. MobiCast

MobiCast [11] introduced by Lin Tan is an appropriate protocol for a network with micro-cells whose base-stations stand on a high speed wired network capable of transmission of multicasting [3] packets to several recipients [12, 13]. This method isolates the movement of mobile hosts from multicast tree to minimize disconnections during sessions. This mechanism uses the hierarchical mobility management to isolate the movement of MHs(Mobile Host) from the main multicast transfer tree. Every external domain must use an agent.

To send a multicast packet the MH has to encapsulate it in a regular packet and send it to domain agent. The agent opens the packet and sends it to receivers, like a multicast packet. Whenever a station wants to subscribe in a multicast group it must send its subscription request to DA (Domain Agent) via its BS (Base Station). The DA subscribes to that multicast group instead of the MH and receives the requested packets and then sends them to the MH. There is another multicast address that denoted as translated multicast address. This translated multicast address has to be unique in the domain and must be related to main multicast group. The BS subscribes for reception of these packets in the translated multicast group (whose address is the translated one), and sends the received packets to the MH. The roaming of the mobile hosts is transparent from other group members while it exists in the domain of the mentioned DFA (Domain Foreign Agent) and there is no need for new calculation of the multicast main tree resulting in minimizing the breaks in multicast sessions.

The Lin Tan's protocol organizes and builds bigger cells with composition of micro-cells and names it DVM (Synamic Macro-Cell). When an MH subscribes to a multicast group via DFA the current base station will inform the other base

stations that are in its DVM to subscribe in that translated multicast group. While only the serving base station actively forwards multicast data packets to the mobile host, the other base stations in the same DVM buffer recent packets and quickly forwards those to the mobile host whenever a handoff occurs. This provides short handoff latency, and the use of buffers at the base stations reduces packet loss due to handoff. It also eliminates multicast group join and graft latencies since the new base station has already subscribed to the multicast group prior to the handoff. Hence, the disruptions to the multicast session due to handoffs of mobile host group members are minimized.

Each multicast group is associated with a translated multicast group address, and serving BSs of interested MHs only need to subscribe to this translated multicast group to receive the desired multicast data. Besides delivering multicast data in an efficient manner (as compared to multiple unicasts to interested MHs within its domain), the use of multicast as the forwarding mechanism from the DFA to interested MHs in its domain also alleviates the DFA from the task of keeping track of the exact location of the MH to ensure correct multicast data delivery. Since physically adjacent cells are most likely to reside on the same network segment, the extra network load generated due to the other member BSs in the same DVM subscribing to the same multicast group is negligible. This is especially so for the case of shared medium networks such as Ethernet.

MobiCast is developed to work with IP and is compatible with existing multicast routing algorithms such as DVMRP, CBT, MOSPF, PIM-DM and PIM-SM. The base stations are network-layer routers with buffers, and are capable of subscribing to multicast groups. Compared to a link-layer solution adopted by most wireless LAN, a network-layer base station is capable of forwarding only those multicast packets with interested mobile receivers in its cell, thus achieving efficient utilization of wireless bandwidth. Furthermore, a network-layer base station is able to differentiate packets with different service types for IPv6 so as to support QoS (Quality of Service) for mobile hosts in its cell. This scheme aims to support best effort IP multicast efficiently for mobile hosts in an environment with Micro-cells, while maintaining the quality of the multicast session during handoffs.

As appears in figure 1 when each MH moves, it leaves the wireless coverage of one cell and enters into another, resulting in a handoff between the base stations. In small wireless cells at the fringes of the Internet, such handoffs during a multicast session will be frequent as wireless cells may be of the size of a few meters. When an MH arrives at a foreign network and obtains an in-care-of address, the MH sends a location update message to inform it's HA (Home Agent) of its care-of address. The care-of address identifies an FA (Foreign Agent) in the foreign subnet where the MH is. The FA can be a separate node or reside in the MH. The MH has to subscribe to the desired multicast group via FA1 when it is at subnet A and via FA2 when it moves to subnet B.

MobiCast has the following advantages. First, the use of hierarchical mobility management architecture separates the mobility of the MH from the main multicast delivery tree. As long as the MH remains within the domain of the DFA, the mobility of the MH is shielded from the rest of the multicast group. No re-computation of the main multicast delivery tree is needed. Second, this scheme requires MHs, which are interested in receiving multicast data to re-subscribe again via the DFA when they are in a foreign domain. This approach is somewhat similar to the remote subscription method proposed by Mobile IP, and network routes taken by the multicast packets to the MHs are efficient and the inefficiencies and scalability problems associated with the Mobile IP are totally avoided. Third, this scheme uses multicast to forward the multicast packets from the DFA to the interested MHs within its domain.

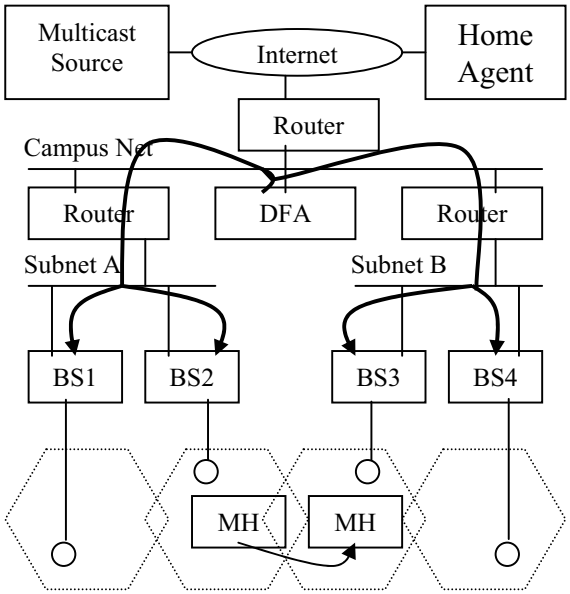

Figure 1 : Before the handoff of MH from BS 2 to 3

III. Communication Definitions

Graph structures are for representation in many applications, such as representing relations, situations or problems. A set of nodes and a set of edges define a graph, where each edge is a pair of nodes. When the edges are directed they are called arcs. Ordered pairs represent arcs. Such a graph is a directed graph. The edges can be attached with costs, names, or any kind of labels, depending on the application. Graphs can be represented in prolog in several ways. One method is to represent each edge or arc separately as one clause. The graph can be thus represented by sets of clauses, for examples:

connected (a, b).
connected (b, c).
arc (s,t,3).
arc (t,v,1).
arc (u,t,,2).

. . .

Another method is to represent the whole graph as one data object. A graph can be thus represented as a pair of two sets: nodes and edges. Each set can be represented as a list; each edge is a pair of nodes. Let us choose the functor graph to combine both sets into a pair, and the functor e for edges. Then one way to represent the (undirected) graph in the following code:

```
G1=graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)]).
```

To represent a directed graph we can choose the functor digraph and a (for arcs). The directed graph of the following code is then:

```
G2=digraph([s,t,u,v],[a(s,t,3),a(t,v,1),a(t,u,5)
,a(u,t,2),a(v,u,2)]).
```

If each node is connected to at least one other node then we can omit the list of nodes from the representation as the set of nodes is then implicitly specified by the list of edges. Yet another method is to associate with each node a list of nodes that are adjacent to that node. Then a graph is a list of pairs consisting of a node plus its adjacency list. Our example graphs can then, for example, be represented by:

```
G3=[p(a,[arcto(b,3),arcto(c,1)]),p(b,[arcto(e,2),arcto(f,1)])].
```

What will be the most suitable representation will depend on the task to be performed on graphs. Two typical operations are finding a path between two given nodes and finding a sub-graph with some specified properties. Finding a spanning tree of a graph is an example of the latter operation. In the following we will look at some simple definitions, which will be used in finding paths and spanning trees. For path finding we can define a path in the following manner where G is a graph, A and Z are two nodes in G, and P is an acyclic path between A and Z in G. P is represented as a list of nodes on the path.

```
path(A,Z,G,P,C):-path1(A,[Z],0,G,P,C).
path1(A,[A|P1],C1,G,[A|P1],C1).
path1(A,[Y|P1],C1,G,P,C):adjacent(X,Y,CXY,G),not(member(
X,P1)),C2=C1+CXY,path1(A,[X,Y|P1],C2,G,P,C).
adjacent(X,Y,graph(N,E)):-
member(e(X,Y),E);member(e(Y,X),E).
```

And accordingly to define a Hamiltonian path:

```
hamiltonian(G,P):-path(_,_,G,P),covers(P,G).
covers(P,G):-not(node(N,G),not(member(N,P)).
```

Similarly to define the minimum cost path

```
path(n1,n2,G,MinPath,MinCost),not(path(n1,n2,G,_,C)
,C<MinCost)
```

And to define the maximum cost path in the graph

```
path(_,_,G,MaxPath,MaxCost),not(path(_,_,G,_,C)
,C>MaxCost)
```

It should be noted that this is a very inefficient way for finding minimal or maximal paths. This method unselectively investigates possible paths and is completely unsuitable for large graphs because of its high running time complexity.

The most important part in building a multicast network in the Internet is the building of a Steiner tree. The Steiner tree is a sub-tree of the minimum spanning tree. The Steiner tree does not contain all the nodes in the graph but includes all the specified nodes and some other nodes to connect the specified nodes in a minimum pathway. To find a Steiner tree we must define these rules: The spanning tree is a connected sub-graph of the main graph that has no cycles. To find the minimum spanning tree we start from an empty set of edges, and add more edges in a way that no cycles be generated and continue to add edges until no more edges can be added. This set will be the spanning tree. And the following is also a useful rule in accomplishing the task: We can add those edges that have only one terminal in the set. Following prolog rules is the implementation of the above rules:

```
stree(G,T):-member(E,G),spread([E],T,G).
spread(T1,T,G):-addedge(T1,T2,G),spread(T2,T,G).
spread(T,T,G):-not(addedge(T,_,G)).
addedge(T,[e(A,B)|T],G):-
adjacent(A,B,G),node(A,T),not(node(B,T)).
adjacent(N1,N2,G):-
member(e(N1,N2),G);member(e(N2,N1),G).
node(N,G):-adjacent(N,_,G).
```

All three arguments of the spread (T1,T,G) are sets of the edges. G is fully connected and T1 and T are sub-sets from G that represents the tree. T is a spanning tree that built form adding zero or one edge from G to T1 in every step. We define the above rules in declarative manner as follows.

```
stree(G,T):-subset(G,T),tree(T),covers(T,G).
tree(T):-connected(T),not(hasacycle(T)).
connected(G):not(node(A,G),node(B,G),not(path(A,B,G,_))).
hasacycle(G):adjacent(N1,N2,G),path(N1,N2,G,[N1,X,Y|_]).
covers(T,G):-not(node(N,G),not(node(N,T))).
subset([],[]).
subset([X|S],Ss):-subset(S,Ss).
subset([X|S],[X|Ss]):-subset(S,Ss).
```

Now we must choose some special nodes and use them to find a Steiner tree, so the covers rule has to use from G1 instead of G. To define the MobiCast protocol we must also define some high level definitions and represent them in prolog.

1- Building a DVM: we must build a DVM with the cell and its adjacent cells, when a mobile host enters a cell then destroy the DVM when the mobile host leaves the cell. This work will be done via subscribing to a new multicasting group.

2- Subscribing to the multicasting group: the mobile host instructs the nearest DFA to subscribe to the interested multicast group.

3- Unsubscribing from the multicasting group: the mobile host instructs the nearest DFA that this node is not interested to be subscribed to the multicasting group any more and the DFA can leave the multicasting group according to its policies.

4- Subscription of the DFA in the multicast group: DFA subscribes to the multicast group, so the tree has to be calculated again.

5- DFA Unsubscribing from multicast group: DFA leaves the multicast group; this event can lead to a new tree calculation.

6- Sending a packet to the multicast group: The mobile host sends a packet to the DFA so it can send the packet to the multicast group instead of the sender.

7- Receiving a packet by DFA: When the DFA receives a packet from a translated multicast address; it converts the translated multicast address to the main multicast address and resends it. Otherwise if the received packet has the main multicast address, the translated address has to be found to forward the packet to the DVM. These steps have been show in the figure 2.

As we only need to prove the correctness of the protocol and do not need to give good performance, we can simplify the search for the multicast addresses or translated multicast addresses by defining the multicast addresses as those above 1024 and the translated multicast addresses as those that are below 1024. This way the MobiCast is represented and implemented in Prolog as illustrated below.

database

single receivedpacket(node,integer)
groupNo(graph,integer)
groupNo(tree,integer)
mhDFA(mhid,node)
mhGroup(mhid,integer)
cell(mhid,node)

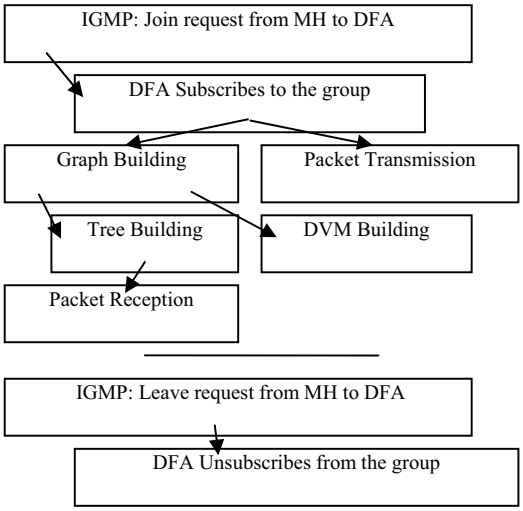

Figure 2: The protocol stages dependencies

clauses

carryInTree(CurNode,[],X):-!.
carryInTree(CurNode,T,X):-
not(receivedpacket(CurNode,X)),assert(receivedpacket(CurNode,X)),nextNode(CurNode,T,N),carryInTree(N,T,X),fail.
carryInTree(CurNode,T,X).
nextNode(CurNode,[e(CurNode,Node)|_],Node).
nextNode(CurNode,[e(Node,CurNode)|_],Node).
nextNode(CurNode,[_|T],Node):-nextNode(CurNode,T,Node).
carryInDFA(CurNode,X,I):-I>1024,II=I-1024,
groupNo(T,II),carryInTree(CurNode,T,X).
carryInDFA(CurNode,X,I):-
I<=1024,II=I+1024,groupNo(T,II),
carryInTree(CurNode,T,X).
sendPacket(MHId,X):-
mhDFA(MHId,DFA),mhGroup(MHId,I),
carryInDFA(DFA,X,I).
enMemberVAddr(MHId,I):-
mhDFA(MHId,DFA),cell(MH,BTS),
constractDVM(BTS,I),enMemberAddr(DFA,I),
II=I-1024, assert (mhGroup(MHId,II)).
enMemberAddr(DFA,I):-groupNo(T,I),gourpNo(G,I),
addToSubGraph(DFA,G,G1),stree(G1,T1),
retract(groupNo(T,I)),retract(groupNo(G,I)),
assert(groupNo(T1,I)),assert(groupNo(G1,I)).
deMemberVAddr(MHId,I):-mhDFA(MHId,DFA),
II=I-1024,
deMemberAddr(DFA,I),cell(MHId,BTS),
destructDVM(BTS,I),retract(mhGroup(MHId,II)).
deMemberAddr(DFA,I):-mhDFA(MHId,DFA),II=I+1024,
mhGroup(MHId,II),!.
deMemberAddr(DFA,I):-groupNo(T,I),groupNo(G,I),
OmitFromGraph(DFA,G,G1),stree(G1,T1),
retract(groupNo(T,I)),retract(groupNo(G,I)),
assert(groupNo(T1,I)),assert(groupNo(G1,I)).
makeDVM(NBts,OBts,I):-
destructDVM(OBts,I),constructDVM(NBts,I).

```
destructDVM(Bts,I):-neighborBts(Bts,NBts,graph),
deMemberAddr(NBts,I),destructDVM(Bts,I),fail.
destructDVM(_,_).
neighborBts(Bts,NBts,[e(Bts,NBts)|_]).
neighborBts(Bts,NBts,[e(NBts,Bts)|_]).
neighborBts(Bts,NBts,[_|G]):-neighborBts(Bts,NBts,G).
```

IV. Verification Method

The program analysis studies the relation between the input and the output of a program. Formally, a program consists of an input (variable) vector $\bar{x} = (x_1, \cdots, x_L)$, a program (variable) vector $\bar{y} = (y_1, \cdots, y_M)$, an output (variable) vector $\bar{z} = (z_1, \cdots, z_L)$, and a finite direct graph (V,A) such that the following conditions are satisfied.

1- In the graph (V,A), there is exactly one vertex, called the start vertex $S \in V$, That is not a terminal vertex of any arc; there is exactly one vertex called the halt vertex $H \in V$, that is not an initial vertex of any arc; and every vertex v is on some path from S to H.

2- In (V,A), each arc a not entering H is associated with a qualifier-free formula $P_a(\bar{x}, \bar{y})$ and an assignment $\bar{y} \leftarrow f_a(\bar{x}, \bar{y})$; each arc entering the halt vertex H is associated with a quantifier-free formula $P_a(\bar{x}, \bar{y})$ and an assignment $\bar{z} \leftarrow f_a(\bar{x}, \bar{y})$. (The P_a is called the testing predicate associated with arc a and $P_a(\bar{x}, \bar{y})$ is called the testing formula associated with arc a.)

3- For each vertex $(v \neq H)$, let $a_1, a_2, \cdots, a_r$ be all the arcs leaving v and let $P_{a_1}, P_{a_2}, \cdots, P_{a_r}$ be the testing predicates associated with arcs $a_1, a_2, \cdots, a_r$, respectively. Then for all $\bar{x}, \bar{y}$, one and only one of $P_{a_1}(\bar{x}, \bar{y})$, $P_{a_2}(\bar{x}, \bar{y})$, … , $P_{a_r}(\bar{x}, \bar{y})$ is True.

The formal representation is used to provide answers to following problems:

1- Termination problem: Given a certain input, will this program terminate?

2- Response problem: Given a certain input, if the program terminates, what is the output of the program?

3- Correctness problem: Given a certain input, will the output of this program satisfy the specification (input-output relationship) of the program?

4- Equivalence problem: Given two programs, will the programs yield the same results if the inputs are the same?

5- Specialization problem: Given a program P that is written to accept a set S of inputs, if we are only interested in a nonempty subset S* of S, how can we simplify P to another P* such that P* runs faster on S* than P does?

Evidently, we need some information to answer any of the above questions. In general, we need the following information:

1- Axioms describing the execution of the program. We describe this by A_p.

2- Axioms concerning testing predicates and assignment functions. For example, we might need the axioms concerning equality or some appropriate induction schema. We describe this by A_s.

3- Axioms concerning the input. For instance, we might require the input to be positive integers. We describe this by A_t.

We assume that $A_s \wedge A_t$ is consistent. But, we have to prove that $A_p \wedge A_s \wedge A_t$ is consistent with the description of the output of the program. There is a theorem that says: Given a program P, let S denote the set of clauses representing $A_p \wedge A_s \wedge A_t$, Then S is satisfiable[14].

So the declarative information of the MobiCast protocol prepared using Prolog language. Of course the non-declarative features in the Prolog language such as the depth-first search rule, a non-logical negation (by failure), and a number of non-logical operations such as the test predicate (e.g., var) and the cut are nevertheless necessary to make Prolog reasonably efficient. Prolog language has some first order logic predicates that formed in clause forms (in CNF composition and without quantifiers in form of prenex like). We used the SLD resolution for automatic theorem proving that showed below.

```
domains
expr=s(symbol);if_(expr,expr);
and_(expr,expr);or_(expr,expr);not_(expr); true
database
nondeterm clause (expr)
done (expr,expr,expr)
modified
predicates
nondeterm contradiction
nondeterm remove_true_clause
nondeterm simplify_clause
nondeterm resolution_step
nondeterm delete_(expr,expr,expr)
nondeterm in(expr,expr)
translate(expr)
translatenot(expr)
nondeterm transform(expr,expr)
nondeterm run

clauses
clause(true).
contradiction:-clause(X),clause(not_(X)), write("contradiction
found(Formula is true).").
remove_true_clause:-clause(C),in(P,C),in(not_(P),C),
retract(clause(C)).
simplify_clause:-clause(C),delete_(P,C,C1),in(P,C1),
retract(clause(C)),assert(clause(C1)).
resolution_step:-
clause(P),clause(C),delete_(not_(P),C,C1),not(done(P,C,P)),
assert(clause(C1)),assert(done(P,C,P)).
resolution_step:-
```

```
clause(not_(P)),clause(C),delete_(P,C,C1),not(done(not_(P),
C,P)), assert(clause(C1)),assert(done(not_(P),C,P)).
resolution_step:-
clause(C1),delete_(P,C1,CA),clause(C2),delete_(not_(P),C2,
CB),not(done(C1,C2,P)),
assert(clause(or_(CA,CB))),assert(done(C1,C2,P)).
% delete(P,E,E1) if deleting a disjunctive subexpression P
from E gives E1
delete_(X,or_(X,Y),Y).
delete_(X,or_(Y,X),Y).
delete_(X,or_(Y,Z),or_(Y,Z1)):-delete_(X,Z,Z1).
delete_(X,or_(Y,Z),or_(Y1,Z)):-delete_(X,Y,Y1).
% in(P,E) if P is a disjunctive subexpression in
in(X,X).
in(X,Y):-delete_(X,Y,_).
% translate(Formula) : translate propositional Formula into
clauses and assert each resulting clause C as clause(C)
translate(and_(F,G)):-!,translate(F),translate(G).
translate(F):-transform(F,NewF),!,translate(NewF).
translate(F):-assert(clause(F)).
% transform(F1,F2) F2 is equal to F1 but closer to clause
form
transform(not_(not_(X)),X). % double negation
transform(if_(X,Y),or_(not_(X),Y)).
% the Implication
transform(not_(and_(X,Y)),or_(not_(X),not_(Y))).
   % De Morgan's law
 transform(not_(or_(X,Y)),and_(not_(X),not_(Y))).
transform(or_(and_(X,Y),Z),and_(or_(X,Z),or_(Y,Z))).
% Distribution
transform(or_(X,and_(Y,Z)),and_(or_(X,Y)
,or_(X,Z))).
transform(or_(X,Y),or_(X1,Y)):-transform(X,X1).
% transform subexpression
transform(or_(X,Y),or_(X,Y1)):-transform(Y,Y1).
transform(not_(X),not_(X1)):-transform(X,X1).

 % Test Stub
run:-contradiction.
run:-remove_true_clause,assert(modified),fail.
run:-simplify_clause,assert(modified),fail.
run:-resolution_step,assert(modified),fail.
run:-modified,retract(modified),run.
translatenot(X):-translate(not_(X)).
```

For example, we now show the abstract level verification of two parts of the program. First we verify the tree building in the program and then we will verify packet delivery mechanism. At the first step we must have a description of the program execution, statements, inputs and outputs of tree building. There is no need to isolate these parts from each other so we extracted all of these descriptions manually form the prolog code. We need an automatic converter if we want a full automatic verifier

The database

SrcNodes(node)
DestNodes(node)
Arc(node,node)
TreeArcs(node,node)

The clauses

$Tree \rightarrow SrcNodes(S) \wedge DestNodes(D) \wedge Arc(S,D) \wedge TreeArcs(S,D) \wedge SrcNodes(D).$
$TreeArcsDesc(N1,N2) \rightarrow \sim Path(N2,N1).$
$Path(N2,N1) \rightarrow TreeArcs(N2,N3) \wedge Path(N3,N1).$

Now we feed the resolution program with these facts and the negation of the result. These clauses will be found in memory in case of program trace.

Path(b,a) ~Path(b,a) Arc(a,b)
TreeArcs(a,b) ~TreeArcs(N,b)

And then the program concludes the contradiction that means the consistency of the program. Again when we want to verify the packet delivery mechanism, have to do as the above, so we prepare the description of the program execution, statements, inputs and outputs.

The database

MhDFA(mhid,node)
MhGroup(mhid,grpid)
GroupNo(tree,grpid)
ReceivedPacket(node,mhid)
NextNode(node,tree,node)
TransG(grpid,grpid)

The clauses

$SendPacket(S,M) \rightarrow MhDFA(S,D) \wedge MhGroup(S,I) \wedge CarryInDFA(D,M,I).$
$CarryInDFA(D,M,I) \rightarrow TransG(I,II) \wedge GroupNo(T,II) \wedge CarryInTree(D,T,M).$
$CarryInTree(D,T,M) \rightarrow ReceivedPacket(D,M) \wedge NextNode(D,T,Dx) \wedge CarryInTree(Dx,T,M).$

Then we present these sentences to the SLD resolution program, with the negation of the result "not(ReceivedPacket(N,M)". These clauses will be found in memory in case of program trace.

SendPacket(a,b) ~ReceivedPacket(c,b)
MhDFA(a,c) MhGroup(a,d)
CarryInDFA(c,b,d) TransG(d,e)
CarryInTree(c,f,b) ReceivedPacket(c,b)
NextNode(c,f,b)

So then the program concludes the contradiction between the second (negation of the result) and 8th fact that means the consistency of the program.

V. Conclusions

The specification of the MobiCast protocol partially presented here demonstrates how the designers of a protocol

can work out inconsistencies prior to release. Several inconsistencies were found in the protocol and resolved with minor variation, though mostly with contradictions between different sections in the protocol. The major benefit of this work is that the result is a well-defined set of criteria that implementers must meet. As such, it is advocated that formal methods should become a critical component of the standardization process. In fact, if an implementation is first defined in formal languages and then in the final language, the implementation can be verified against the specification, increasing the probability that different implementations would have no inconsistencies. Future work will also include verifying that two different protocols can co-exist together such as TCP-IP [4].

REFERENCES

[1] G.J.Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall International Editions, AT&T Bell Laboratories 1991.
[2] J.F.Leathrum, J.Rasha, M.B.E.Morsi, T.E.Leathrum, "Formal Verification of Communication Protocols", 1996.
[3] Host Extensions for IP multicasting, IETF RFC 1112 specification.
[4] IETF TCP/IP Specifications : The TCP protocol, IETF RFC 793, IETF RFC 791, IETF RFC 1883.
[5] M.Barjaktarovic, "Formal specification and verification of the OSI Session Layer using the calculus of Communicating Systems (CCS)", Ph.D. thesis, Dept. of Electrical and Computer Engineering, Syracuse University, USA , 1995.
[6] M.G.Gouda , J.Y.Han, "Protocol Validation by fair progress state exploration", Computer Networks and ISDN Systems , Vol. 9, 1985.
[7] M.G.Gouda , Y.T.Yu, "Protocol Validation by maximal progress state exploration", IEEE Trans. on Communications, Vol. COM-32, No. 1, 1984.
[8] M.Fahimi, "Artificial Intelligence", Jelveh Publications, 2000.
[9] W.F.Clocksin, C.S.Mellish, "Programming in Prolog", Springer-Verlag, 1987.
[10] I.Bratko, "Prolog Programming for Artificial Intelligence", Addison-Wesley Publishing Company, 1994.
[11] C.L.Tan, S.Pink , “MobiCast : A Multicast scheme for wireless networks” , Baltzer Science Publishers BV, Mobile Networks and Applications 5, 2000.
[12] G.Xylomenos and G.C.Polyzos, “IP Multicasting for wireless mobile hosts”, Proc. of the IEEE MILCOM Conf. on Military Communications, Vol. 3, 1996.
[13] T.G.Harrison, C.L.Williamsom, W.L.Mackrell and R.B.Bunt, “Mobile Multicast (MoM) protocol: Multicast support for mobile hosts”, Proc. of ACM/IEEE MobiCom, 1997.
[14] C.Chang and R.C.Lee, "Symbolic Logic and Theorem Proving", Academic Press, San Diego, CA, 1973.