Case Study - VoIP Model Graph Transformation Rules

Mayur Bapodra, Reiko Heckel

Department of Computer Science, University of Leicester, UK
mb294@mcs.le.ac.uk, reiko@mcs.le.ac.uk

1 Introduction

This report details a case study based on a very simple overview of peer-to-peer (P2P) connections over a voice over IP (VoIP) network such as that used for Skype. We present the concrete version of the model (with aggregating attributes already added) followed by the resulting abstract model.

Rule name abbreviations that have been used in accompanying research are given after each rule name.

2 Concrete Model

The VoIP network is constructed from Client and Super nodes, each servicing a single user, administrated by a central registry. Figure 1 shows the concrete type graph. *User* represents a physical user. *Client* represents a node that users attach to in order to make a call. Users can also attach to a *Super* node, but since Supers form the overall communication network via *ovl* edges, linking Clients to each other through this network, the user in question must meet a bandwidth threshold. In our model, we simplify this threshold by not explicitly measuring bandwidth, but instead controlling the likelihood that a user has the minimum bandwidth through stochastic parameters.

Our smallest start graph is shown in Figure 2. It consists of the registry node, one super node (with associated user) and six disconnected users. In the abstract model we aim to simplify the type graph by hiding User and Client types. There are therefore three aggregating attributes in this model to retain information. The Registry stores the number of clients not yet connected to the overlay network via a link edge to a Super (as *freeClients*). It also stores the number of users that have not yet turned on their VoIP program and therefore do not have an associated Client or Super node (as *offlineUsers*). Additionally, Supers store the number of clients linked to them. The OCL constraints giving their values in the concrete model are as follows:

```
context: Registry
inv: self.offlineUsers = User.allInstances -> count (u:User |
     Node.allInstances -> forall(n:Node | n.usr ≠ u))
```

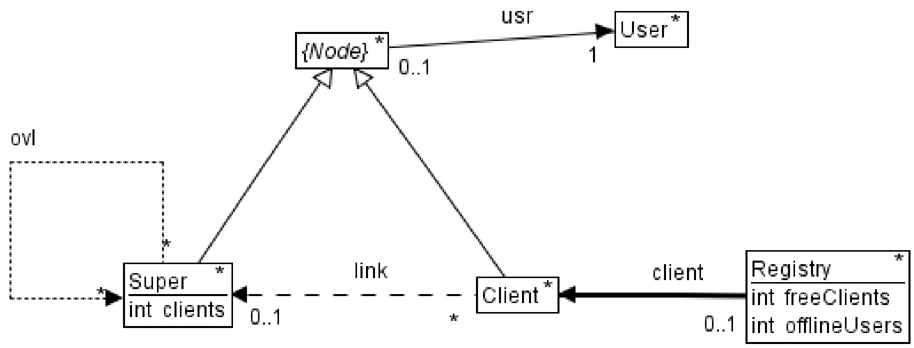

Fig. 1. Concrete type graph

```
context: Registry
inv: self.freeClients = Client.allInstances -> count (c:Client |
     c.link -> isEmpty())
```

```
context: Super
inv: self.clients = Client.allInstances ->
     count (c : Client | c.link = self)
```

The behavioural semantics of the model are explained through the GT rules. Figure 3 shows the creation of a new client when a user connects to the VoIP program. The NAC shows that the user must not already be connected. The new client is connected at the registry. The condition on the *offlineUsers* aggregating attribute is naturally redundant since we have the user node and the NAC at the concrete level.

Figure 4 illustrates a user wishing to communicate with another via the overlay network set up between Super nodes. The client must not have an existing link to a super node, and the super it connects to must have fewer than five clients. This is considered the maximum shareable bandwidth before a super node is overloaded. Note that ideally, although redundant, a NAC should also be present to represent this condition, but we omit it and leave the condition on the aggregating attribute only for simplicity.

Figure 5 shows the promotion of a client to a super when it is connected to an overloaded super node. This reduces some of the pressure on the overused super node.

Figure 6 depicts the rule that is analogous to the creation of a new client, except that the user has sufficient bandwidth to support a super node. The super node is immediately connected to the overlay network via an existing super node.

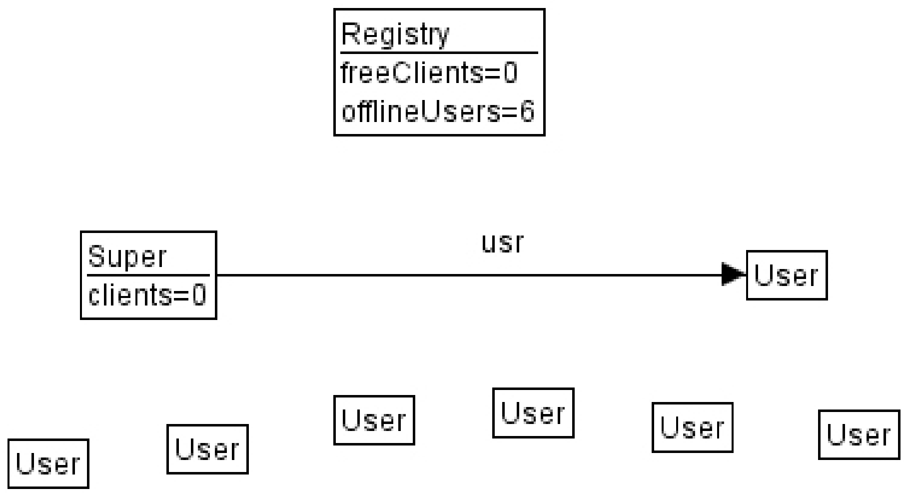

Fig. 2. Concrete start graph

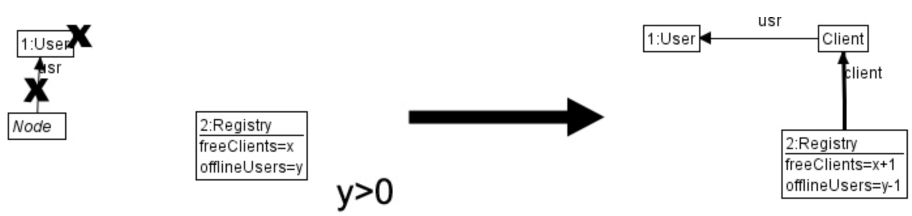

Fig. 3. New client, concrete GT rule (NC)

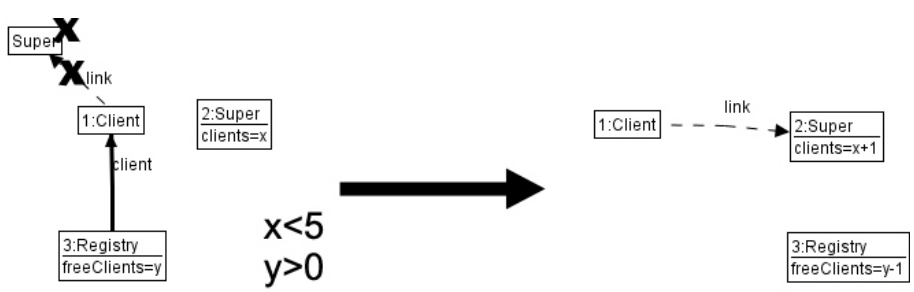

Fig. 4. Link client, concrete GT rule (LC)

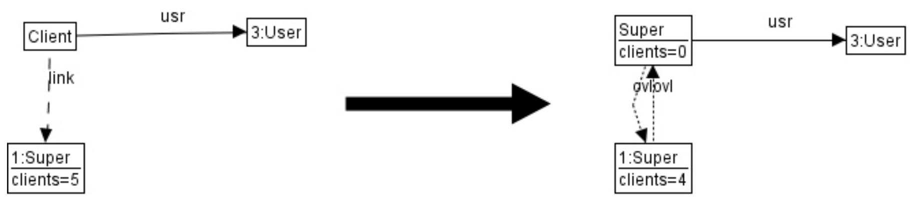

Fig. 5. Promote client, concrete GT rule (*PC*)

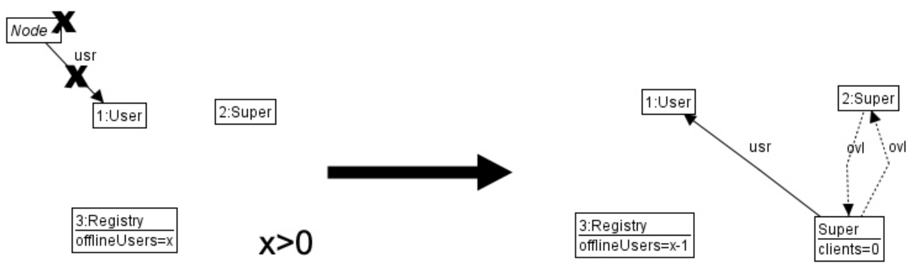

Fig. 6. New super, concrete GT rule (*NS*)

Figure 7 shows the termination of a client while it is connected to the overlay network (i.e., during a call). The rule represents the shutting down of the VoIP program by a user.

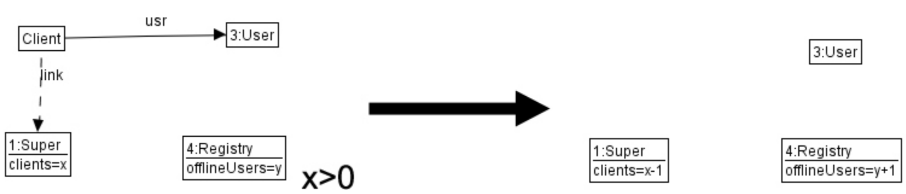

Fig. 7. Terminate linked client, concrete GT rule (*TL*)

The rule in Figure 8 represents the shutting down of the VoIP program by a user that is connected directly through a super node. This is only allowed if there is at least one other super node present. To prevent violation of the dangling condition, we must explicitly represent all other graph vertices the deleted node is connected to. Furthermore, all clients that were connected to the deleted super node are returned to the registry. Through the deletion of a super node by this rule, the overlay network may become disconnected so that some clients may become unreachable from others.

Figure 9 depicts the loss of an unlinked client, i.e., a user switching off the VoIP program without being in a call.

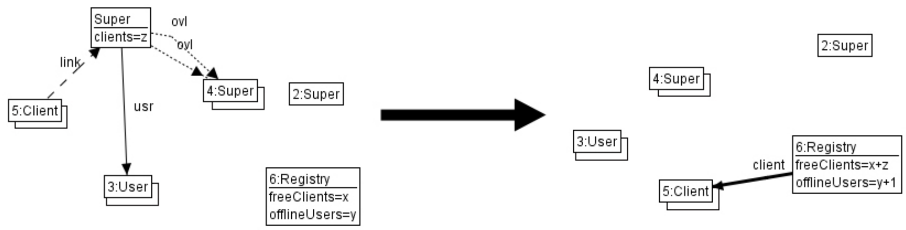

Fig. 8. Terminate super, concrete GT rule (*TS*)

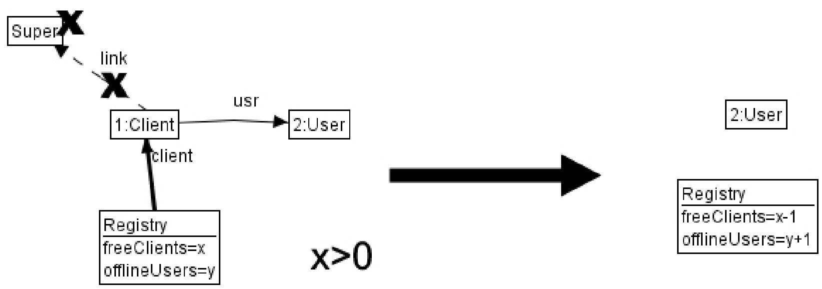

Fig. 9. Terminate unlinked client, concrete GT rule (*TU*)

Finally, Figure 10 describes the creation of redundant links between supers in order to make the overlay network more robust. Client pairs are less susceptible to becoming disconnected when a super is terminated if there is more than one path between the supers to which they are connected. Note that the rule has two NACs: one stating that there should be no existing overlay connection between the supers that are to be newly linked, and secondly, that there is no alternate pathway between these supers via a fourth super node.

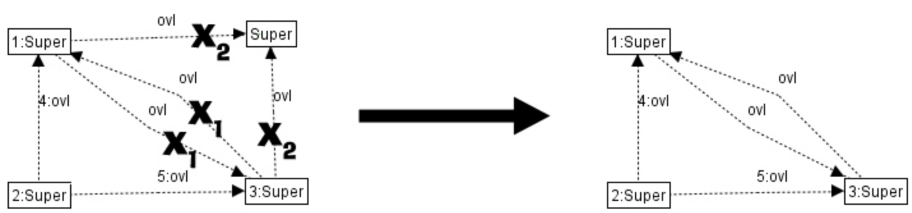

Fig. 10. Create shortcut, concrete GT rule (*CS*)

The global behaviour of interest in this model is the proportion of total clients at any time that are connected to the overlay network, and the proportion of total super node pairs that are unreachable from each other (i.e., there is no transitive closure of the *ovl* edge between them).

For the former, we create a probe rule that searches matches for Client nodes, and another that searches matches for Client nodes linked to Super nodes, dividing the second by the first to find the measure we require. This measurement is taken after every step in the simulation and we use the average value over each of these steps as our result.

Transitive closure measurements require modifications to the standard VIATRA2 installation [1]. A description of the necessary steps are outlined by the authors in an online addendum to [1] at http://viatra.inf.mit.bme.hu/grats. We specify the pattern between just two super nodes with a pair of oppositely directed *ovl* edges between them. The transitive closure version of the probe then looks for all pairs of super nodes between which there is a chain of *ovl* edges. A negation of this pattern then finds all such pairs between which there is no such transitive connection. We divide this number of pairs by the total number of super node pairs to find our measure. Just as with the connected client measure, we take the average of this value over all simulation steps.

Code Fragment 1.1 shows the VTCL specification implementing the entire behaviour of the concrete model, complete with probes and auxiliary patterns.

Code Fragment 1.1. VTCL Specification of Concrete Model

```

namespace p2p;

import DSM.metamodels.p2p_TG;
import datatypes;

@incremental
machine rules_and_constraints {

//////////////////////////////////////////
// PATTERNS
/////////////////////////////////////////

    pattern UserConnected(U) = {
        User(U);
        P2PNode(N);
        P2PNode.usr(U1, N, U);
    }

    pattern ClientLinked(C) = {
        Client(C);
        Super(S);
        Client.link(L, C, S);
    }

    pattern Clients(C) = {
        Client(C);
    }

    pattern SuperPairs(S1, S2) = {
        Super(S1);
        Super(S2);
    }

    shareable pattern SuperLinked(S1, S2) = {
        Super(S1);
        Super(S2);
        Super.ovl(Ov1, S1, S2);
        Super.ovl(Ov2, S2, S1);
    }
```

```


    @Incremental(reinterpret=transitiveClosure, ofPattern=SuperLinked)
    pattern transitiveClosureOfSuperLinked(S1, S2) = {}



//////////////////////////////////////////
// CONCRETE RULES
//////////////////////////////////////////

    gtrule NewClient() = {

        precondition pattern lhs(U, OU, FC, R) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
            User(U);
            neg find UserConnected(U);
        }

        action {
            let C=undef,
            C1=undef,
            U1=undef,
            Model=DSM.models.model
            in seq {
                new (Client(C) in Model);
                rename(C, "cl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(C)), "_", ""), "un", ""));
                new(P2PNode.usr(U1, C, U));
                rename(U1, "usr");
                new (Registry.client(C1, R, C));
                rename(C1, "client_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(C1)), "_", ""), "un", ""));
                setValue(OU, toString(toInteger(value(OU)) - 1));
                setValue(FC, toString(toInteger(value(FC)) + 1));
                println("GT RULE... NewClient applied to create Client "
                     + name(C));
            }
        }
    }


    gtrule LinkClient() = {

        precondition pattern lhs(FC, Cl1, C, S, S_C) = {
            Super(S);
            Super.Clients(S_C) in S;
            Super.clients(S_Cx, S, S_C);
            check (toInteger(value(S_C)) < 5);
            Registry(R);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
            Client(C);
            Registry.client(Cl1, R, C);
            neg find ClientLinked(C);
        }

        action {
            let L=undef
            in seq {
                delete(Cl1);
                new(Client.link(L, C, S));
                rename(L, "link");
                setValue(S_C, toString(toInteger(value(S_C)) + 1));
```

```
                setValue(FC, toString(toInteger(value(FC)) - 1));
                println("GT RULE... LinkClient applied to link Client "
                    + name(C));
            }
        }
    }


    gtrule PromoteClient() = {

        precondition pattern lhs(S, C, U, L, U1, S_C) = {
            Super(S);
            Super.Clients(S_C) in S;
            check (toInteger(value(S_C)) >= 5);
            Super.clients(S_Cx, S, S_C);

            Client(C);
            Client.link(L, C, S);

            User(U);
            P2PNode.usr(U1, C, U);
        }

        action {
            let S2=undef,
            S_CNew=undef,
            S_CNew_X=undef,
            Ovl1=undef,
            Ovl2=undef,
            U2=undef,
            Model=DSM.models.model
            in seq {
                print("GT RULE... PromoteClient applied to promote
                    Client " + name(C));
                delete(L);
                delete(U1);
                delete(C);

                new(Super(S2) in Model);
                rename(S2,"super_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(S2)), "_", ""), "un", ""));

                new(entity(S_CNew) in S2);
                rename(S_CNew, "Clients");
                setValue(S_CNew, 0);
                new (instanceOf(S_CNew, ref("DSM.metamodels.p2p_TG.Super
                    .Clients")));
                new(Super.clients(S_CNew_X, S2, S_CNew));
                rename(S_CNew_X, "clients");

                new(Super.ovl(Ovl1, S2, S));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S, S2));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl2)), "_", ""), "un", ""));
                setValue(S_C, toString(toInteger(value(S_C)) - 1));

                new(P2PNode.usr(U2, S2, U));
                rename(U2, "usr");

                println(" to Super " + name(S2));
            }
        }
    }


    gtrule NewSuper() = {
```

```

        precondition pattern lhs(U, S, OU) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            User(U);
            Super(S);
            neg find UserConnected(U);
        }

        action {
            let S2=undef,
            S_CNew=undef,
            S_CNew_X=undef,
            Ovl1=undef,
            Ovl2=undef,
            Us1=undef,
            Model=DSM.models.model
            in seq {

                new(Super(S2) in Model);
                rename(S2,"super_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(S2)), "_", ""), "un", ""));

                new(entity(S_CNew) in S2);
                rename(S_CNew, "Clients");
                setValue(S_CNew, 0);
                new (instanceOf(S_CNew, ref("DSM.metamodels.p2p_TG.Super
                    .Clients")));
                new(Super.clients(S_CNew_X, S2, S_CNew));
                rename(S_CNew_X, "clients");

                new(P2PNode.usr(Us1, S2, U));
                rename(Us1, "usr");
                new(Super.ovl(Ovl1, S2, S));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S, S2));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl2)), "_", ""), "un", ""));
                setValue(OU, toString(toInteger(value(OU)) - 1));
                println("GT RULE... NewSuper applied to create Super " +
                     name(S2) +" with Clients attribute value " + value(
                    S_CNew));
            }
        }
    }


    gtrule TerminateLinkedClient() = {

        precondition pattern lhs(C, S, S_C, OU, U1, L) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Super(S);
            Super.Clients(S_C) in S;
            Super.clients(S_Cx, S, S_C);
            User(U);
            Client(C);
            P2PNode.usr(U1, C, U);
            Client.link(L, C, S);
        }

        action {
            delete(L);
            delete(U1);
            setValue(S_C, toString(toInteger(value(S_C)) - 1));
```

```
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateLinkedClient applied to terminate Client " + name(C));
            delete(C);
        }
    }


    gtrule TerminateUnlinkedClient() = {

        precondition pattern lhs(C, OU, FC, U1, C1) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);

            User(U);
            Client(C);
            P2PNode.usr(U1, C, U);
            Registry.client(C1, R, C);
        }

        action {
            delete(C1);
            delete(U1);
            setValue(FC, toString(toInteger(value(FC)) - 1));
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateUnlinkedClient applied to terminate Client " + name(C));
            delete(C);
        }
    }


    gtrule TerminateSuper() = {

        precondition pattern lhs(OU, FC, S2, R, S_C) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);

            Super(S1);
            Super(S2);
            Super.Clients(S_C);
            Super.clients(S_CX, S2, S_C);
        }

        action {
            setValue(FC, toString(toInteger(value(FC)) + toInteger(value(S_C))));
            iterate choose with apply relinkClients(S2, R);
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateSuper applied to terminate Super " + name(S2));
            delete(S2);
        }
    }


    gtrule CreateShortcut() = {

        precondition pattern lhs(S1, S2) = {
            Super(S1);
            Super(S2);
            Super(S3);
```

```
            Super.ovl(O1, S3, S1);
            Super.ovl(O2, S3, S2);
            neg find SupersConnected(S1, S2);
            neg find SupersBypass(S1, S2, S3);
        }

        action {
            let Ovl1=undef,
            Ovl2=undef
            in seq {
                new(Super.ovl(Ovl1, S1, S2));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S2, S1));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl2)), "_", ""), "un", ""));
            }
        }
    }


    gtrule countClients(in S, in R, in Count) = {

        precondition pattern lhs(S, L, C) = {

            Super(S);
            Client(C);
            Client.link(L, C, S);
            Registry(R);
        }
        action {
            let C1=undef
            in seq{
                setValue(Count, toString(toInteger(value(Count)) + 1));
                delete(L);
                new(Registry.client(C1, R, C));
                rename(C1, "client_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(C1)), "_", ""), "un", ""));
            }
        }
    }


    gtrule relinkClients(in S, in R) = {

        precondition pattern lhs(S, L, C, R) = {

            Super(S);
            Client(C);
            Client.link(L, C, S);
            Registry(R);
        }
        action {
            let C1=undef
            in seq{
                delete(L);
                new(Registry.client(C1, R, C));
                rename(C1, "client_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(C1)), "_", ""), "un", ""));
            }
        }
    }



//////////////////////////////////////////
// PROBE RULES
//////////////////////////////
```

```
    gtrule Probe_ConnectedClients(inout C) = {
        precondition find ClientLinked(C)
    }

    gtrule Probe_AllClients(inout C) = {
        precondition find Clients(C)
    }

    gtrule Probe_Disconnected(inout S1, inout S2) = {
        precondition pattern lhs(S1, S2) = {
            Super(S1);
            Super(S2);
            neg find transitiveClosureOfSuperLinked(S1, S2);
        }
    }

    gtrule Probe_AllSuperPairs(inout S1, inout S2) = {
        precondition find SuperPairs(S1, S2)
    }

    gtrule Probe_Supers(inout S1) = {
        precondition pattern lhs(S1) = {
            Super(S1);
        }
    }

}
```

3 Abstract Model

In the abstract model, we retain only the *Super* and *Registry* types, with the inheritance of *Super* from *Node* as an inconsequential artefact. The abstract type graph is shown in Figure 11. The abstraction in this case does not reduce the number of rules, but with fewer graph elements in each rule and in each instance graph, there are fewer and smaller matches for each rule. Therefore, an increase in performance during stochastic simulation is still expected. The start graph is reduced to just two elements: the registry and the first super node, as shown in Figure 12.

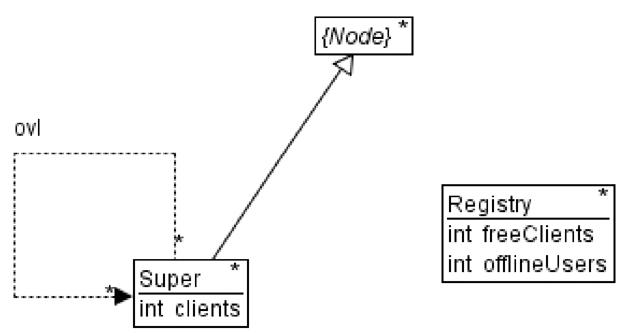

Fig. 11. Abstract type graph

Fig. 12. Abstract start graph

Figures 13 to 19 show the projection of the concrete rules to the abstract type graph. Note that except for those of create shortcut, all NACs are lost but conditions on aggregating attributes still remain.

Fig. 13. New client, abstract GT rule (*A_NC*)

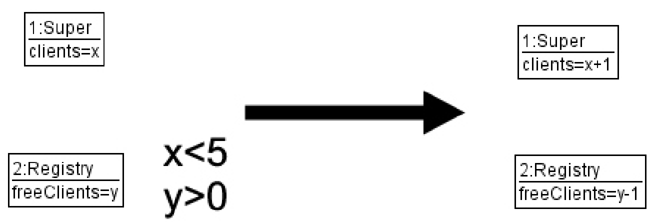

Fig. 14. Link client, abstract GT rule (*A_LC*)

The probes responsible for finding disconnected pairs of super nodes can still be used in the abstract model since the *Super* type is still present. However, to calculate the proportion of clients that are connected to the overlay network, the *clients* attribute of *Super* must be accessible along with the *freeClients* attribute of *Registry*. The sum of the *clients* attribute for all super nodes divided by this value plus the *freeClients* value gives us the measure we require.

Since the reading of attribute values is not currently supported by VIATRA2, a hard coded, model specific solution was implemented as a temporary measure. Probe rules were created to pass required attribute entities in the modelspace to the simulation engine. The hard coded solution recognized the probe by name and returns/sums the required attribute value for all matches of the probe rather than simply counting matches themselves. The solution will be incorporated into the stochastic simulation package of VIATRA2 once a naming convention/methodology for attribute value probes is decided upon.

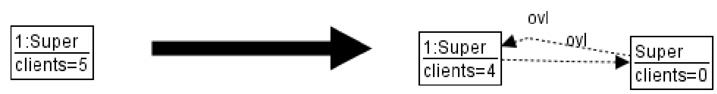

Fig. 15. Promote client, abstract GT rule (*A_PC*)

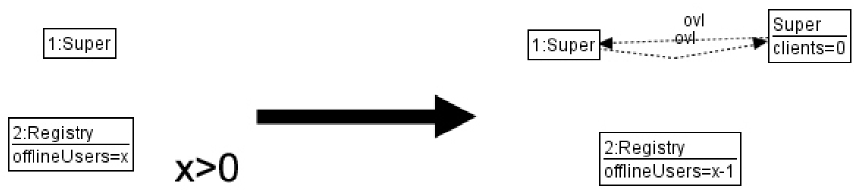

Fig. 16. New super, abstract GT rule (*A_NS*)

Fig. 17. Terminate linked client, abstract GT rule (*A_TL*)

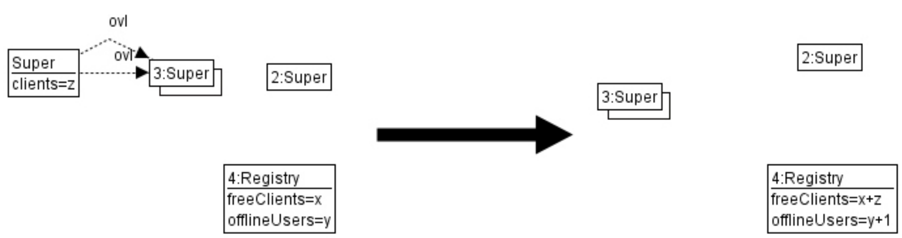

Fig. 18. Terminate super, abstract GT rule (*A_TS*)

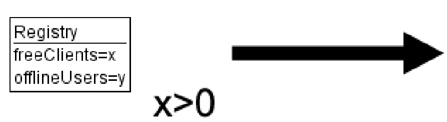

Fig. 19. Terminate unlinked client, abstract GT rule (*A_TU*)

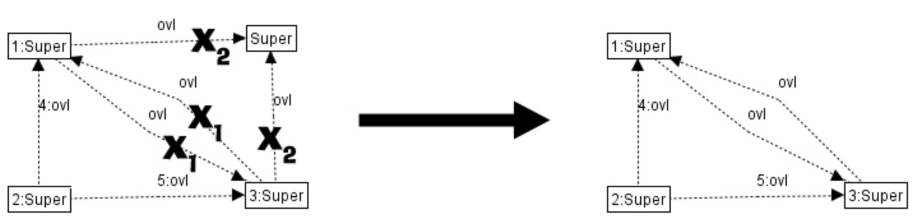

Fig. 20. Create shortcut, abstract GT rule (*A_CS*)

The portion of the *rules.vtcl* file that defines these abstract rules is given in Code Fragment 1.2.

Code Fragment 1.2. VTCL Specification of Abstract Model

```

///////////////////////////////////////////
// ABSTRACT RULES
///////////////////////////////////////////

    gtrule Abstract_NewClient() = {

        precondition pattern lhs(OU, FC) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
            check (toInteger(value(OU)) > 0);
        }

        action {
            setValue(OU, toString(toInteger(value(OU)) - 1));
            setValue(FC, toString(toInteger(value(FC)) + 1));
            println("GT RULE... NewClient applied to create Client");
        }
    }


    gtrule Abstract_LinkClient() = {

        precondition pattern lhs(FC, S, S_C) = {
            Registry(R);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
            check (toInteger(value(FC)) > 0);
            Super(S);
            Super.Clients(S_C) in S;
            Super.clients(S_Cx, S, S_C);
            check (toInteger(value(S_C)) < 5);
        }

        action {
            setValue(S_C, toString(toInteger(value(S_C)) + 1));
            setValue(FC, toString(toInteger(value(FC)) - 1));
            println("GT RULE... LinkClient applied to link Super " +
                name(S));
        }
    }


    gtrule Abstract_PromoteClient() = {

        precondition pattern lhs(S, S_C) = {
            Super(S);
            Super.Clients(S_C) in S;
            Super.clients(S_Cx, S, S_C);
            check (toInteger(value(S_C)) >= 5);
        }

        action {
            let S2=undef,
            S_CNew=undef,
            S_CNew_X=undef,
            Ovl1=undef,
            Ovl2=undef,
            Model=DSM.models.model
            in seq {
```

```
                new(Super(S2) in Model);
                rename(S2,"super_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(S2)), "_", ""), "un", ""));

                new(entity(S_CNew) in S2);
                rename(S_CNew, "Clients");
                setValue(S_CNew, 0);
                new (instanceOf(S_CNew, ref("DSM.metamodels.p2p_TG.Super.Clients")));
                new(Super.clients(S_CNew_X, S2, S_CNew));
                rename(S_CNew_X, "clients");

                new(Super.ovl(Ovl1, S2, S));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S, S2));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(Ovl2)), "_", ""), "un", ""));

                setValue(S_C, toString(toInteger(value(S_C)) - 1));

                println("GT RULE... PromoteClient applied to new Super " + name(S2));
            }
        }
    }


    gtrule Abstract_NewSuper() = {

        precondition pattern lhs(S, OU) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            check (toInteger(value(OU)) >0);
            Super(S);
        }

        action {
            let S2=undef,
            S_CNew=undef,
            S_CNew_X=undef,
            Ovl1=undef,
            Ovl2=undef,
            Model=DSM.models.model
            in seq {
                new(Super(S2) in Model);
                rename(S2,"super_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(S2)), "_", ""), "un", ""));

                new(entity(S_CNew) in S2);
                rename(S_CNew, "Clients");
                setValue(S_CNew, 0);
                new (instanceOf(S_CNew, ref("DSM.metamodels.p2p_TG.Super.Clients")));
                new(Super.clients(S_CNew_X, S2, S_CNew));
                rename(S_CNew_X, "clients");

                new(Super.ovl(Ovl1, S2, S));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S, S2));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.toLowerCase(name(Ovl2)), "_", ""), "un", ""));
                setValue(OU, toString(toInteger(value(OU)) - 1));
                println("GT RULE... NewSuper applied to create Super " + name(S2));

```

```
            }
        }
    }


    gtrule Abstract_TerminateLinkedClient() = {

        precondition pattern lhs(S, S_C, OU) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Super(S);
            Super.Clients(S_C) in S;
            Super.clients(S_Cx, S, S_C);
            check (toInteger(value(S_C)) > 0);
        }

        action {
            setValue(S_C, toString(toInteger(value(S_C)) - 1));
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateLinkedClient applied to
                terminate Client on Super" + name(S));
        }
    }


    gtrule Abstract_TerminateUnlinkedClient() = {

        precondition pattern lhs(OU, FC) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
            check (toInteger(value(FC)) >0);
        }

        action {
            setValue(FC, toString(toInteger(value(FC)) - 1));
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateUnlinkedClient applied");
        }
    }


    gtrule Abstract_TerminateSuper() = {

        precondition pattern lhs(OU, FC, S2, R, S_C) = {
            Registry(R);
            Registry.OfflineUsers(OU) in R;
            Registry.offlineUsers(Ou, R, OU);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);

            Super(S1);
            Super(S2); // to delete
            Super.Clients(S_C);
            Super.clients(S_CX, S2, S_C);

        }

        action {
            setValue(FC, toString(toInteger(value(FC)) + toInteger(value
                (S_C))));
            setValue(OU, toString(toInteger(value(OU)) + 1));
            println("GT RULE... TerminateSuper applied to terminate
                Super " + name(S2));
            delete(S2);
```

```
        }
    }


    gtrule Abstract_CreateShortcut() = {

        precondition pattern lhs(S1, S2) = {
            Super(S1);
            Super(S2);
            Super(S3);
            Super.ovl(O1, S3, S1);
            Super.ovl(O2, S3, S2);
            neg find SupersConnected(S1, S2);
            neg find SupersBypass(S1, S2, S3);
        }

        action {
            let Ovl1=undef,
            Ovl2=undef
            in seq {
                new(Super.ovl(Ovl1, S1, S2));
                rename(Ovl1,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl1)), "_", ""), "un", ""));
                new(Super.ovl(Ovl2, S2, S1));
                rename(Ovl2,"ovl_"+str.replaceAll(str.replaceAll(str.
                    toLowerCase(name(Ovl2)), "_", ""), "un", ""));
            }
        }
    }

//////////////////////////////////////////
// PROBE RULES
//////////////////////////////

    gtrule Probe_Disconnected(inout S1, inout S2) = {
        precondition pattern lhs(S1, S2) = {
            Super(S1);
            Super(S2);
            neg find transitiveClosureOfSuperLinked(S1, S2);
        }
    }

    gtrule Probe_AllSuperPairs(inout S1, inout S2) = {
        precondition find SuperPairs(S1, S2)
    }

    gtrule Probe_Supers(inout S1) = {
        precondition pattern lhs(S1) = {
            Super(S1);
        }
    }

    gtrule Probe_AttributeFreeClients(inout FC) = {
        precondition pattern lhs(FC) = {
            Registry(R);
            Registry.FreeClients(FC) in R;
            Registry.freeClients(Fc, R, FC);
        }
    }

    gtrule Probe_AttributeConnectedClients(inout S_C) = {
        precondition pattern lhs(S_C) = {
            Super(S);
            Super.Clients(S_C);
            Super.clients(S_CX, S, S_C);
        }
    }
```

4 Bayesian Network

The dynamic incidence matrix produced for abstract rules over the concrete model (i.e., diagonal analysis) is given in Table 1. The arbitrary partial order on rules names was decided as $LC > NS > NC > TU > PC > TS > TL > CS$. The resulting Bayesian network is shown in Figure 21.

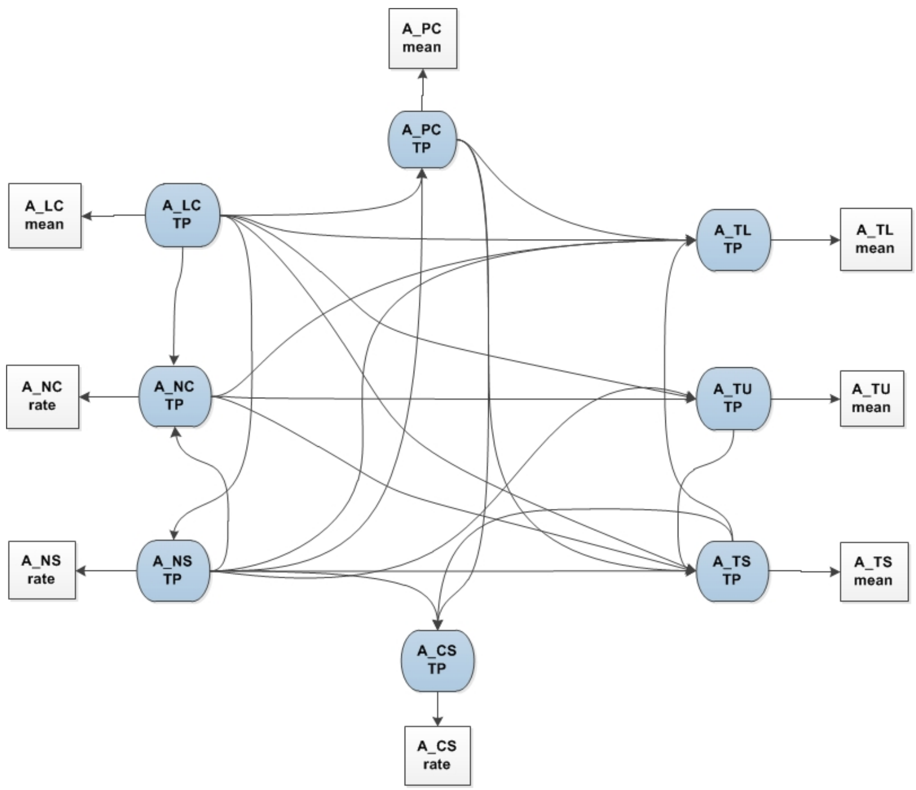

Fig. 21. BN generated for VoIP case study (TP abbreviates throughput)

Table 1. Dynamic incidence matrix for abstract rules over concrete model (aggregate of diagonal conflicts and dependencies)

Rule name	**Applied Rule**								
	Start	**LC**	**NC**	**NS**	**PC**	**TL**	**TS**	**TU**	**CS**
A_LC	0	-1.44 (0.106)	1.26 (0.104)	0.423 (0.0353)	0.696 (0.51)	0.00384 (0.00511)	0.275 (0.0991)	-0.884 (0.144)	0
A_NC	1.0 (0.0)	0	-0.186 (0.025)	-0.252 (0.0311)	0	0.213 (0.0338)	0.254 (0.031)	0.142 (0.0368)	0
A_NS	1.0 (0.0)	0	-0.564 (0.086)	-0.2 (0.155)	0.536 (0.267)	0.713 (0.124)	0.205 (0.155)	0.337 (0.103)	0
A_PC	0	0.0192 (0.00861)	0	0	-1.0 (0.0)	-0.00839 (0.00753)	-0.00026 (0.00115)	0	0
A_TL	0	0.503 (0.0314)	0	0	0	-0.379 (0.0401)	-0.360 (0.0342)	0	0
A_TS	0	0	0	1.18 (0.0273)	1.88 (0.171)	0	-1.19 (0.0278)	0	0
A_TU	0	-0.494 (0.0314)	0.464 (0.0320)	0	0	0	0.239 (0.0304)	-0.378 (0.0512)	0
A_CS	0	0	0	2.41 (0.141)	0.203 (0.323)	0	-0.521 (0.131)	0	-2.49 (0.133)

References

1. Bergmann, G., Ráth, I., Szabó, T., Torrini, P., Varró, D.: Incremental pattern matching for the efficient computation of transitive closure. In: Sixth International Conference on Graph Transformation. Bremen, Germany (09/2012 2012)