Expanding & Contracting Transport Networks using Standard Reconfigurations

by

Andrew Kinney

NORTEL NETWORKS

University College London

Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Telecommunications.

Abstract

The objective of this project is to report on the full software lifecycle in designing, developing and delivering the Standard Reconfigurations features of Add Node and Delete Node, using UML and Java as the means by which the design is described and implemented.

As with the software lifecycle, the report follows the timeline from initial inception to final delivery, covering in sequence the aspects of requirements capture, high level design, detailed design, coding, unit test, and finally integration test.

This report documents my work in delivering the Standard Reconfigurations software feature which specifically encompasses the entirety of the Network Element software portion of this feature.

The submission does not contain material previously submitted for previous degree or academic award.

This report is the work of the author, with any contributions from others expressly acknowledged and/or cited.

Acknowledgements

The author gratefully acknowledges the contributions from colleagues in ensuring the successful delivery of the Standard Reconfigurations feature that forms the basis for much of the work detailed in this report.

Specific mention should be made of:

- Dr. John Mitchell of UCL who provided essential guidance and support in pulling this report together ready for submission
- Rabina Choudry of UCL for ensuring that this project could be undertaken
- Nortel line manager, Andy Harker who provided valuable guidance and insight during the project
- Jeff Henderson, Martin Bluethner, Adrian Lee, and Mike Caithness who formed the Management Platform (MP) development team

Glossary

ADM	Add/Drop Multiplexer
BLSR	Bi-directional Line Switched Ring
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
JNI	Java Native Interface
MSOH	Multiplex Section OverHead
NE	Network Element
PCM	Pulse Code Modulation
PDH	Plesiochronous Digital Hierarchy
PG	Protection Group
PGM	Protection Group Member
MP	Management Platform
RSOH	Regenerator Section OverHead
SDH	Synchronous Digital Hierarchy
SOH	Section OverHead
SONET	Synchronous Optical NETwork
SWERR	SoftWare ERRor
TL1	Transaction Language 1
UCL	University College London
UML	Unified Modelling Language
Xcon	Traffic Connection (Cross Connect)

Contents

1 Introduction

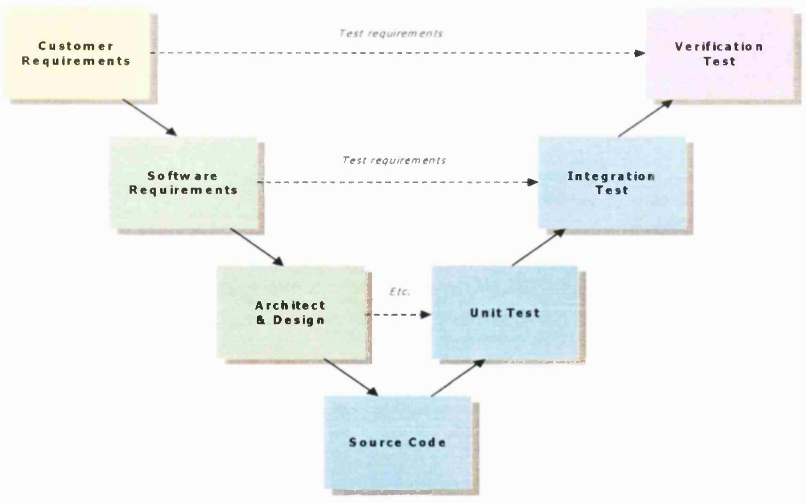
The Standard Reconfigurations described within this report refer to those procedures developed, within Nortel Networks and delivered as part of the next generation transport platform, that allow a customer to expand or contract their previously installed transport network.

The following section entitled Terminology on page 8 describes the terminology used within this report whilst also introducing the concept of standard reconfigurations and why they are required.

The report follows the full software lifecycle, from requirements capture through to delivery of the Standard Reconfig feature.

The main body of the report is structured around the process followed in developing the software associated with the Standard Reconfigurations feature as determined by the software lifecycle shown in Figure 1 which forms the basis of the approved quality process used within Nortel Networks.

Figure 1 - Software Lifecycle



The Standard Reconfiguration procedures are termed Add Node and Delete Node. A brief definition of these terms, which also form the basic Customer Requirements, is given here for reference and will be expanded upon in subsequent sections:

Add Node - The purpose of the Add Node procedure and associated software is to be able to safely add a new ADM into an in-service 4F BLSR ring with the minimum of service disruption.

Delete Node - Delete Node is the term given to the procedure and associated software which allows the customer to safely remove an

ADM from an in-service 4F BLSR ring with the minimum of service disruption

From these basic Customer Requirements the Software Requirements are derived, a subset of which are given in section 3 on page 14. With these more detailed requirements the procedures can be designed, of which high level descriptions are given in section 4 on page 17.

To further expand upon the requirements and procedures a set of use cases is described to aid the readers understanding and also to add more detail in describing the behaviour of the software, given in section 5 on page 27.

Once the requirements and procedures have been reviewed and approved, the detailed design work is undertaken as shown in section 6 on page 30.

Prior to delivery to Verification, the software must be tested against the requirements as described in section 7 on page 44.

The project and this report is then summarised in the Conclusions on page 51, including a view of the benefits of using UML to describe the design and Java as the implementation of that design.

2 Terminology

Throughout this report general knowledge of transmission systems, and mechanisms such as protection is required. The terminology used within this document is gradually introduced in this section, providing sufficient basic information to understand the subsequent sections that detail the development of the Standard Reconfigurations.

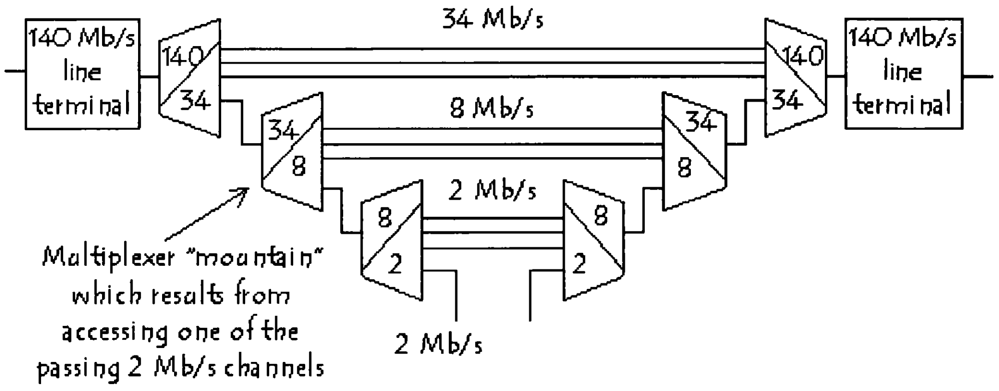
2.1 Transmission Systems

In the early 1970s, digital transmission systems began to appear, utilising Pulse Code Modulation (PCM) to convert voice (analogue) into binary (digital) form. Engineers saw the potential to produce more cost effective transmission systems by combining several PCM channels into a single high speed bit stream via Time Division Multiplexing (TDM). (Ref. [7])

As demand grew, higher data rates were required with subsequent need for further levels of multiplexing, resulting in the definition of the Plesiochronous Digital Hierarchy (PDH).

However, eventually this lead to the creation of the PDH 'multiplexer mountain' as displayed in Figure 2 that was required to access a single 2Mbit/s channel from the high speed trunk.

Figure 2 - PDH Multiplexer Mountain



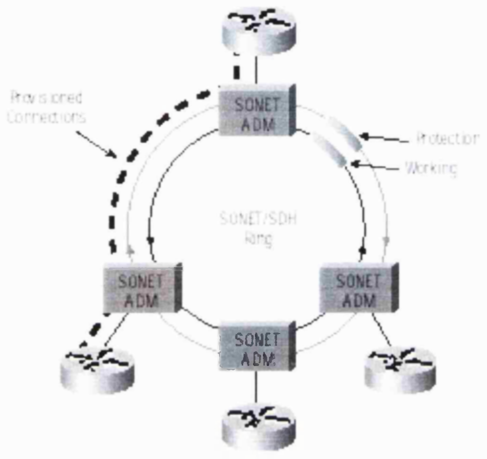
The next stage of evolution in the transport network was the introduction of synchronous transmission which overcame the inefficiencies of PDH.

The ITU-T introduced the G.707 (Ref. [9]) G.708 (Ref. [10]) and G.709 (Ref. [11]) standards to describe the Synchronous Digital Hierarchy (SDH) whilst in North America the Synchronous Optical NETwork (SONET) standard was introduced.

SDH brought about network simplification as a single synchronous multiplexer could replace an entire plesiochronous 'multiplexer mountain' leading to a significant reduction in equipment being used.

This lead to the installation of SDH 'Rings' consisting of Add/Drop Multiplexers (ADM) as shown in Figure 3.

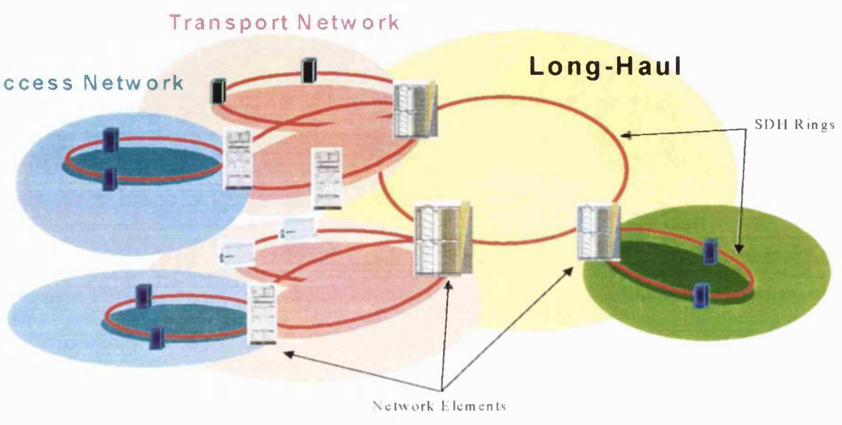
Figure 3 - SONET/SDH Ring (Ref. [8])



2.2 Transport Network

A transport network, as shown in Figure 4 on page 9, is composed of numerous SDH Rings. In a customer network these SDH rings may well overlay each other to create a tiered transport network, to facilitate different requirements, such as the Access Network where data rates are typically low, compared to the high data rates required at the trunk, or Long Haul tier.

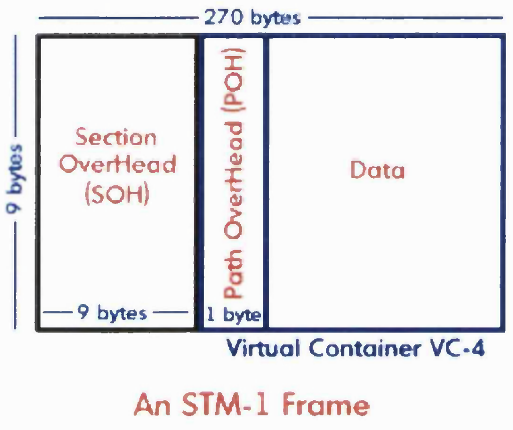
Figure 4 - Transport Network



2.3 SDH Frame

The data transported between nodes in a SDH Ring is achieved by parcelling this information into SDH Frames. The SDH frame has a two-dimensional structure, as shown in Figure 5. Each frame is physically transmitted through the fibre row by row.

Figure 5 - STM1 Frame (Ref. [6])



For the purposes of this report the element of the frame to highlight is the Section Overhead, which combines the Regenerator and Multiplex Section Overheads.

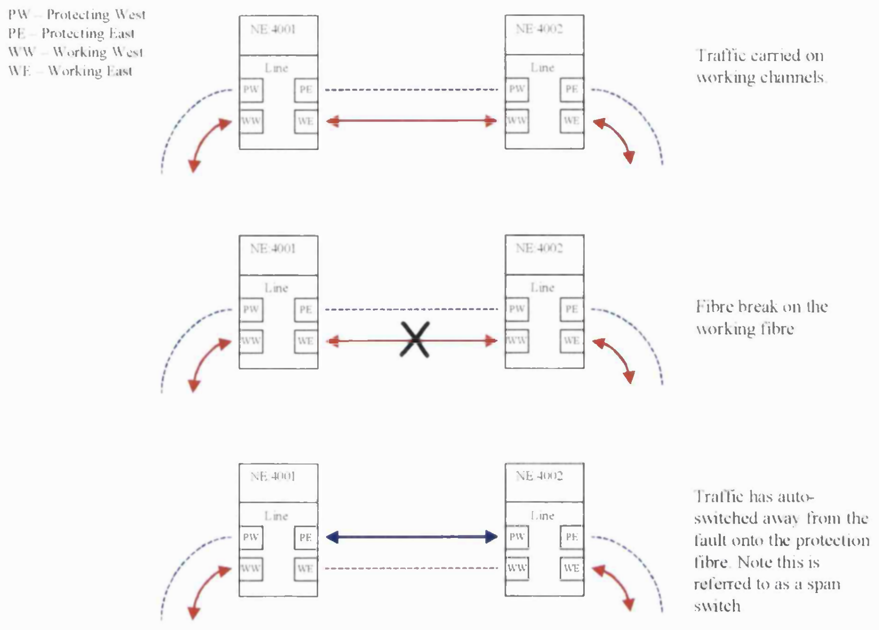
Within the Multiplex Section Overhead, the K1 and K2 bytes are of specific importance as these bytes carry the Automatic Protection Switching (APS channel) information between nodes in the ring.

2.4 Protection

Considering the large quantity of traffic supported by the transport network a key requirement is its ability to survive network failures, such as fibre breaks with minimum disruption. This is achieved by implementing the Bellcore GR-1230 protection protocol for Bi-direction Line Switched Rings (BLSR) in a 4-fibre configuration.

An example of a fibre break scenario is shown in Figure 6 on page 11.

Figure 6 - Fibre Break


PW – Protecting West
PE – Protecting East
WW – Working West
WE – Working East
NE 4001
NE 4002
Line
PW
PE
WW
WE
Traffic carried on working channels.
Fibre break on the working fibre
Traffic has auto-switched away from the fault onto the protection fibre. Note this is referred to as a span switch

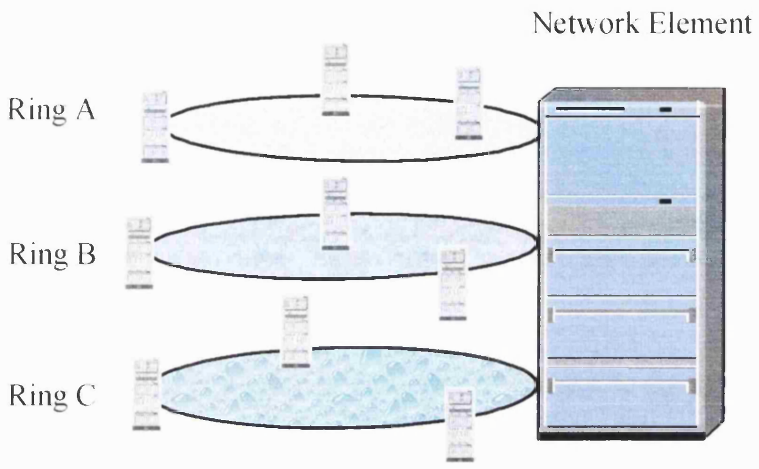
The example in Figure 6 is a simple one but is used to illustrate the concept of a protection switch, and specifically a span switch which is used within the Add Node and Delete Node procedures to 'move' traffic from working to protecting fibres or vice verse as appropriate.

NOTE: The Bellcore standard stipulates that a user requested protection switch must result in a traffic hit of <50ms i.e. service is restored in <50ms.

2.5 Network Element

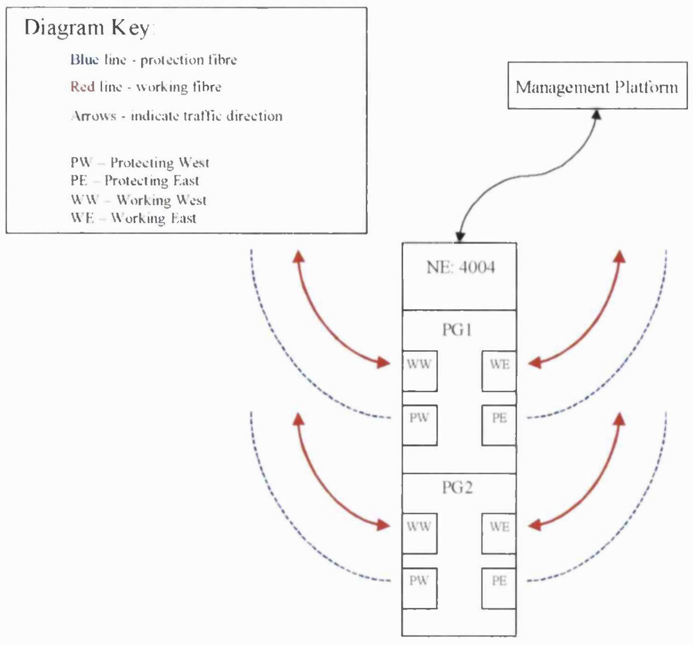
Each SDH ring contains a number of Network Elements (NE), or 'nodes' which perform the necessary traffic grooming. Note that a NE can be part of several rings (see Figure 7 on page 12).

Figure 7 - Network Element



An abstract description of a Network Element is shown in Figure 8 on page 12 as an aid to describing the remaining terminology used within this report.

Figure 8 - Abstraction of Network Element



Each ring that an NE is part of results in a Protection Group (PG) being formed of the termination points for each of the 4-fibres that support the BLSR protection protocol as referred to in Figure 8 by PG1 and PG2.

A protection groups encapsulates a number of attributes, of which the following are key to the Add Node and Delete Node procedures:

- *node map* – which is the internal representation of the members of the SDH ring that this PG is a member of. This is used to validate protection messages received via the K1 and K2 bytes.
- *protection scheme* – is the scheme that this PG is provisioned with, e.g. BLSR 4F
- *reconfiguration state* – an indication of what state the PG is in, e.g. Add Node would indicate the PG is in the midst of an Add Node procedure
- *special mode* – by setting this mode to pass through the traffic and kbyte channels on the protecting fibre are placed in full pass through thereby making the node 'transparent' to neighbouring nodes in the ring

The Management Platform (MP) provides the mechanism through which the NE can be managed singly, and as part of the ring.

NOTE: The work required on the Management Platform is outside the scope of the project which is limited to the software required on the NE.

2.6 Purpose of Standard Reconfigurations

Since it is very difficult to predict future network need at time of first installation, or perhaps installation is limited by cost, there is a need to provide flexibility to expand or contract a ring based on current or future need. Hence the requirement for Add Node and Delete Node.

Scenarios that may result in the use of Add Node or Delete Node include:

- Expansion of transport network into new city, thereby adding a node to an existing ring
- Removal of node due to obsolescence
- Temporary removal of node to allow node to be physically moved elsewhere within office

One aspect of Standard Reconfigurations that is critical to appreciate is the need to perform these changes to the network whilst the ring is in-service, i.e. carrying live traffic. A Network Operator does not want to turn 'off' the network to allow new nodes to be added or old nodes to be removed, as this would lead to downtime and loss of revenue.

This leads to the two main generic requirements:

- The Add Node and Delete Node procedures must be performed whilst the network is <u>in-service</u>
- Any traffic disruptions must be kept to a minimum, and at worst should not exceed 50ms, which is the specification for a traffic hit associated with a protection switch

3 Requirements

As shown in Figure 1 on page 6 the first stage of the software lifecycle is to derive a set of detailed software requirements based on the requirements that have been received from and agreed with the customer.

These detailed requirements, coupled with the customer requirements are used as part of the testing strategy to ensure what is delivered meets what is required.

[R1] Safely add a new ADM into an <u>in-service</u> 4F BLSR ring with the minimum of service disruption

[R2] Safely delete an ADM an <u>in-service</u> 4F BLSR ring with the minimum of service disruption

Requirements [R1] and [R2] form the basic customer requirements, that the detailed software requirements listed below are derived from. The requirements listed below are abstract in nature, but their purpose becomes apparent when read in conjunction with the procedure descriptions that contain further explanation (section 4 on page 17).

Add Node:

[R3] On receipt of the 'Add Node' command the NE Software will validate that the following PG attributes are provisioned:
- node map is nil
- protection scheme is BLSR 4F
- reconfig state is Idle

[R4] Once validated the NE Software will execute the following:
- sets the reconfig state of the PG to Add Node
- sets the special mode of the PG to pass through
- deletes all traffic connections terminated on PG
- raises reconfig alarm
- generates log

[R5] On receipt of the 'Idle' command to backout the Add Node procedure, the NE Software will:
- clear the reconfig alarm
- set the reconfig state of the PG to Idle
- set the special mode of the PG to None
- delete all traffic connections terminated on PG

[R6] On receipt of a valid node map to complete and exit the Add Node procedure, the NE Software will:
- clear the reconfig alarm
- set the reconfig state of the PG to Idle
- set the special mode of the PG to None

[R7] Add Node command can be executed on a PG while any other configured PG in the NE carries live traffic

[R8] Add Node will not affect traffic on any other PG in the NE

Delete Node:

[R9] On receipt of the 'Delete Node' command the NE Software will validate that the following PG attributes are provisioned:
- node map is non-nil
- protection scheme is BLSR 4F
- reconfig state is Idle
- no non-pass through traffic connections terminate on the PG
- no active protection switches are in place on the PG

[R10] Once validated the NE Software will execute the following:
- sets the reconfig state of the PG to Delete Node
- sets the special mode of the PG to pass through
- sets the node map of the PG to nil
- raises reconfig alarm
- generates log

[R11] On receipt of the 'Idle' command to complete and exit the Delete Node procedure, the NE Software will:
- clear the reconfig alarm
- set the reconfig state of the PG to Idle
- set the special mode of the PG to None
- delete all traffic connections terminated on PG

[R12] On receipt of a valid node map to backout the Delete Node procedure, the NE Software will:
- clear the reconfig alarm
- set the reconfig state of the PG to Idle
- set the special mode of the PG to None

[R13] Delete Node command can be executed on a PG while any other configured PG in the NE carries live traffic

[R14] Delete Node will not affect traffic on any other PG in the NE

Robustness:

[R15] If the NE restarts during the execution of Add Node or Delete Node then on recovery, the command can be entered again and be allowed to successfully complete i.e. there is no automatic recovery and complete option, the user must re-enter the command.

[R16] If the NE restarts after the Add Node or Delete Node command has successfully completed, then the NE must be capable of recovering to the Add Node or Delete Node state

[R17] Add Node and Delete Node is unaffected by software upgrade

Error Handling:

[R18] Add Node command is denied if PG has a non-nil node map ("Node is configured in a network")

[R19] Delete Node command is denied if a protection switch is active ("Node is currently handling a protection switch")

[R20] Delete Node command is denied if any non-pass through traffic connections are present ("Traffic connections are currently provisioned")

[R21] Delete Node command is denied if the node map is nil ("Node is not configured in a network")

[R22] Add Node and Delete Node commands will be denied if the PG is already in an add node state ("Command is not valid while node is in Add Node mode")

[R23] Add Node and Delete Node commands will be denied if the PG is already in an delete node state ("Command is not valid while node is in Delete Node mode")

[R24] Add Node and Delete Node commands will be denied if the PG is not BLSR 4F ("Protection scheme is not valid")

[R25] Deleting the PG will be denied if the reconfig state is set to add node ("Command is not valid while node is in Add Node mode")

[R26] Deleting the PG will be denied if the reconfig state is set to delete node ("Command is not valid while node is in Delete Node mode")

4 Procedure Description

The following sections: 4.1 on page 17 and 4.2 on page 22 provide high level descriptions of the add node and delete node procedures highlighting the software algorithms required to facilitate the operations whilst also providing insight into the reasoning behind the requirements listed in section 3 on page 14.

4.1 Add Node

The diagram in Figure 9 on page 17 displays the initial state of an example network in readiness of the Add Node procedure.

Figure 9 - Configuration Prior to Add Node operation

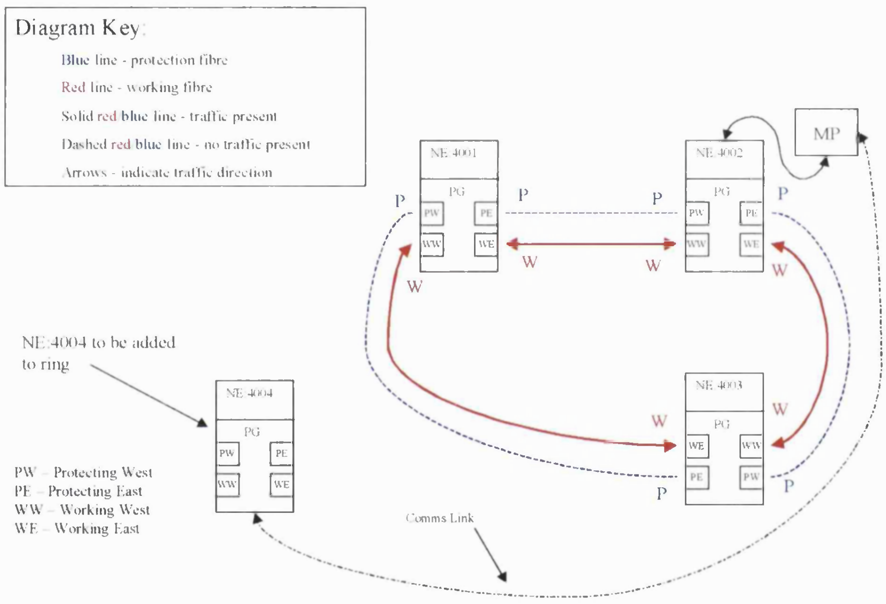


Diagram Key:
Blue line - protection fibre
Red line - working fibre
Solid red/blue line - traffic present
Dashed red/blue line - no traffic present
Arrows - indicate traffic direction
NE:4001
NE:4002
NE:4003
NE:4004
MP
PG
PW
PE
WW
WE
P
W
NE:4004 to be added to ring
PW – Protecting West
PE – Protecting East
WW – Working West
WE – Working East
Comms Link

Step 1 – Commission the new NE on the Management Platform (MP). Once the new NE has been commissioned it is then selectable on the MP.

Step 2 – Issue the 'Add Node' command from the MP via the Reconfig Assistant tool. This translates into a message sent down to the NE via the Comms link.

On receipt of the Add Node command the NE software will perform validation, before execution.

The command will be denied if:

- the protection group (PG) has a non-nil node map
  - the presence of a non-nil node map indicates that the protection group is already part of a ring, and could be carrying live traffic

- the specified PG is not configured with a protection scheme of BLSR 4F
- there are any other reconfigurations in progress for the specified PG
    - this is a safety measure to prevent traffic loss in the event that an incorrect step of the procedure is executed due to confusion with another reconfiguration

Once validated, the NE software performs the following actions:

- sets the reconfig state of the PG to Add Node
    - this places the PG in Add Node mode, to be used by other applications as an indication of the current state of the PG, e.g. prevent traffic connection edits that would cause loss of traffic
- deletes all existing traffic connections terminating on the specified PG
    - traffic connections could exist on the PG as part of a commissioning procedure to test the capability of the hardware, and this step ensures these 'test' connections are removed before the NE is placed in-service (IS)
- puts the protection lines of the PG into full traffic and overhead k-byte pass through
    - This allows all traffic to pass through the node on the protection lines as is required at Step 6.
    - The k-bytes need to pass though the node unaffected so that the BLSR protocol is unaware that a new node has been fibred into the ring. Without k-byte pass-through the protection switching in Step 6 would not be possible since the neighbour nodes would be receiving invalid k-bytes from the new node and would not be able to negotiate the protection switch.
- raises an alarm against the PG to alert the user that the PG is in a special reconfiguration state

A user visible log is generated to indicate success, or in the case of failure, the reason for the command being denied.

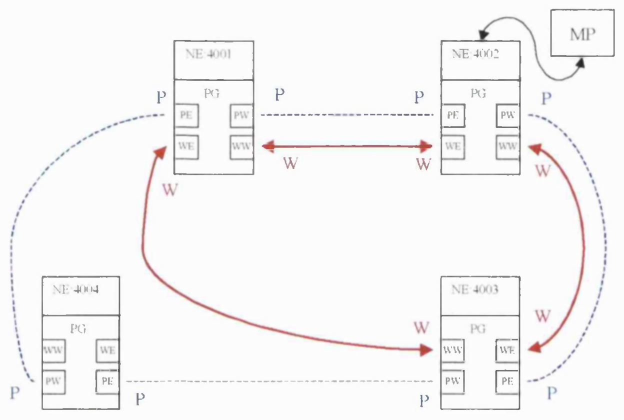
On completion of the command the node is effectively 'transparent' to both traffic and protection protocols, thereby allowing it to be placed in the middle of an in-service ring without unnecessarily dropping traffic.

Step 3 – Issue a 'lockout of working – span' against the working facility at both NEs (e.g. WE of NE:4001 and WW of NE:4003) on either end of the span to which the new node is being added

- this prevents the use of the protection line on this span for any protection activity, i.e. preventing traffic loss when the protection fibres are moved to cable in the new node, as in the next step.

Step 4 – Cable in the new node on its protecting equipment only. Then verify the optical integrity of the protection link (Figure 10)

Figure 10 - Configuration after Step 4 of Add Node Operation



Step 5 – Release the 'lockout of working – span' on the working facilities of the nodes at both ends of the span

Step 6 – At the MP the following steps are performed, via the Reconfig Assistant:

- the new NE is added to the management representation of the ring
- pass-through connections are provisioned on the NE
    - these traffic connections match up to those already provisioned in the ring, and provide the nodal pieces that facilitate the traffic through the new node
    - e.g. for each traffic connection that links NE:4001 and NE:4003 a corresponding pass-through connection needs to be provisioned on NE:4004 such that when the NE is added there is no loss of traffic due to an incomplete traffic path

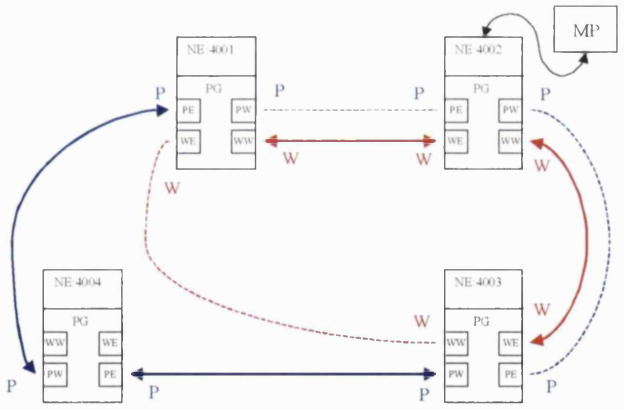
Step 7 – At the MP a connection audit is performed.

- This is a robustness check to confirm that the pass-through connections provisioned in the previous step are actually resident on the new node

Step 8 – Perform a forced span switch against the working facility at both ends of the span (e.g. WE of NE:4001 and WW of NE:4003).

- The forced span switch is issued at both NEs for robustness, in case one of the span switches is pre-empted and dropped
- The ring will now carry traffic as shown in Figure 11 on page 20.
- NOTE: traffic is now carried on the protection fibres through the new node, hence the reason why the protection channels must be in pass through, which allows the working fibres to be disconnected, as indicated in the next step.

Figure 11 - Configuration after Step 8 of Add Node Operation

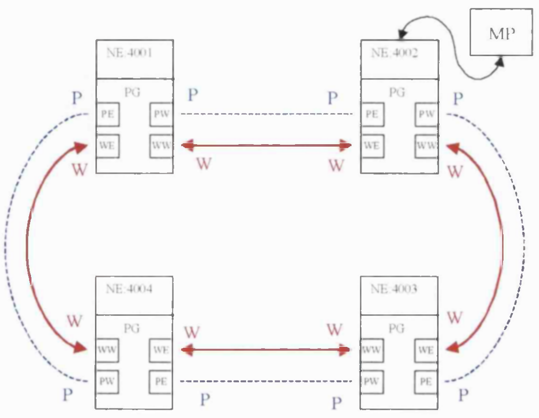


Step 9 – Cable in the new node on its working equipment. Then verify the optical integrity of the working link.

Step 10 – Release the forced span switches

- the traffic will auto-switch back to working, as indicated in Figure 12 on page 21.

Figure 12 – Configuration after Step 10 of Add Node Operation



Step 11 – Issue a Global Lockout of Protection on the configuration to which the new node has been added.

- A Global Lockout of Protection is of the highest priority in the protection switch hierarchy and prevents any protection switch activity, in addition forcing all traffic to use the working channels, i.e. the protection fibres are off limits.
- This is for robustness/safety, to prevent any inadvertent protection switching when the new node map is provisioned in the next step.

Step 12 – Via the MP Reconfig Assistant, the new node map is provisioned for all nodes in the ring.

The NE reconfig software 'listens' for the node map provisioning event and:

- clears the reconfig alarm
- sets the special mode of the PG to None
- sets the reconfig state of the PG to Idle

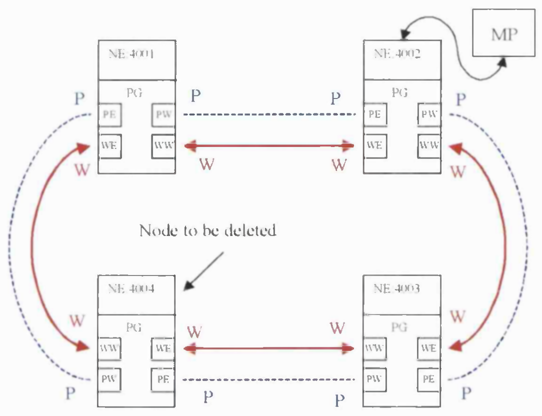
i.e. receipt of a valid node map indicates the add node operation is complete

Step 13 – Release the Global Lockout of Protection.

4.2 Delete Node

The diagram in Figure 13 on page 22 shows the initial state of an example configuration prior to starting the Delete Node Operation.

Figure 13 - Configuration prior to Delete Node Operation



Step 1 – Issue a Global Lockout of Protection on the configuration to which the new node is to be removed.

- This is for robustness/safety, to prevent any inadvertent protection switching when the node map is altered on the NEs in the ring.

Step 2 – Issue the 'Delete Node' command from MP via the Reconfig Assistant tool. This translates into a message sent down to the NE.

On receipt of the Delete Node command the NE software will perform validation, before execution.

The command will be denied if:

- There are non-pass through traffic connections provisioned on the PG
  - Presence of non-pass through connections indicates the possibility of live traffic terminating at this PG, which would be lost if the procedure was continued, as this PG will be removed from the ring.
- the protection group (PG) has a nil node map
  - the presence of a nil node map indicates that the protection group is not part of a ring, and therefore cannot be 'deleted'

- the specified PG is not configured with a protection scheme of BLSR 4F
- there are any other reconfigurations in progress for the specified PG
    - this is a safety measure to prevent traffic loss in the event that an incorrect step of the procedure is executed due to confusion with another reconfiguration
- there are any active protection switches in place on the PG being deleted
    - any active protection switches must be cleared before the delete node operation can proceed, otherwise the required protection switches required during the procedure may not be honoured.

Once validated, the NE software performs the following actions:

- sets the reconfig state of the PG to Delete Node
    - this places the PG in Delete Node mode, to be used by other applications as an indication of the current state of the PG, e.g. prevent traffic connection edits that would cause loss of traffic
- clears the node map of the PG being deleted
    - this restores the PG to its initial state
- puts the protection lines of the PG into full traffic and overhead k-byte pass through
    - This allows all traffic to pass through the node on the protection lines
    - The k-bytes need to pass though the node unaffected so that the BLSR protocol is unaware that the node is going to be fibred out of the ring. Without k-byte pass-through the protection switching would not be possible since the neighbour nodes would be receiving invalid k-bytes from the 'old' node and would not be able to negotiate the protection switch.
- raises an alarm against the PG to alert the user that the PG is in a special reconfiguration state

A user visible log is generated to indicate success, or in the case of failure, the reason for the command being denied.

On completion of the command the node is effectively 'transparent' to both traffic and protection protocols, thereby allowing it to be removed from the middle of an in-service ring without unnecessarily dropping traffic.

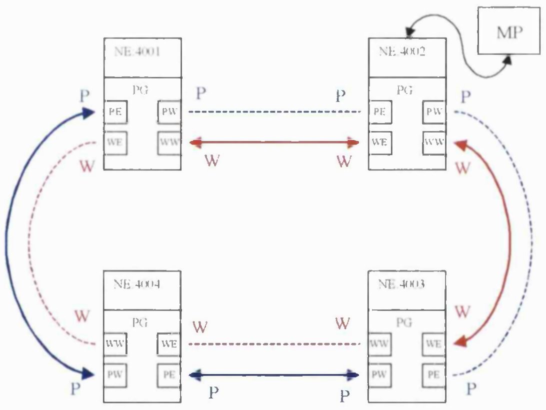
Step 3 – Via the MP Reconfig Assistant remove the node from the management view of the ring, and send new node maps to all the nodes in the ring.

Step 4 – Release the Global Lockout of Protection.

Step 5 – Perform a forced span switch against the working facility at both ends of the span (e.g. WE of NE:4001 and WW of NE:4003).

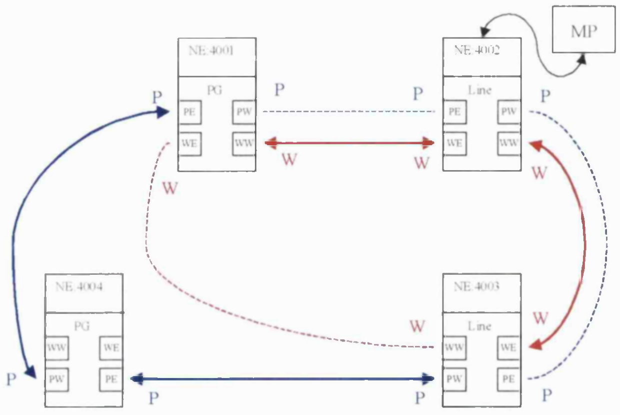
- The forced span switch is issued at both NEs for robustness, in case one of the span switches is pre-empted and dropped
- The ring will now carry traffic as shown in Figure 14 on page 24
- NOTE: traffic is now carried on the protection fibres allowing the working fibres to be removed as required in the next step

Figure 14 – Configuration after Step 5 of Delete Node Operation



Step 6 – Cable out the 'deleted' node on its working equipment and validate the optical integrity of the working link (Figure 15 on page 25).

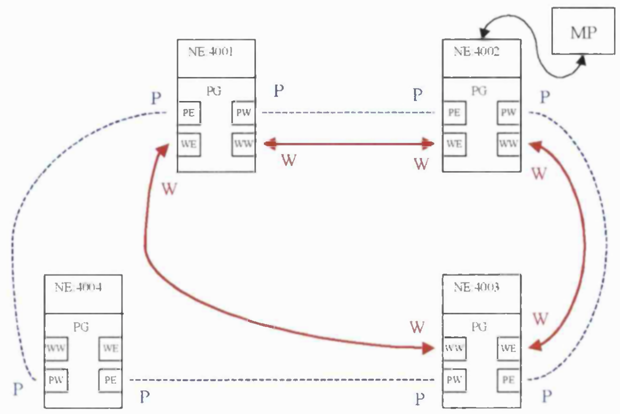
Figure 15 - Configuration after Step 6 of Delete Node Operation



Step 7 – Release the forced span switches

- the traffic will auto-switch back to working, as indicated in Figure 16

Figure 16 - Configuration after Step 7 of Delete Node Operation

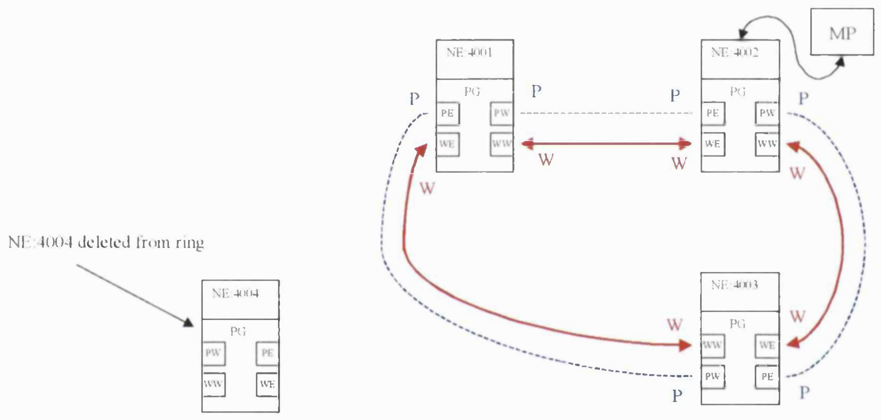


Step 8 – Issue a 'lockout of working – span' against the working facility at both NEs (e.g. WE of NE:4001 and WW of NE:4003) on either end of the span to which the node has been removed

- this prevents the use of the protection line on this span for any protection activity, i.e. preventing traffic loss when the protection fibres are moved to cable in the new node, as in the next step.

Step 9 – Cable out the 'deleted' node on its protecting equipment. Then verify the optical integrity of the protection link (Figure 17)

Figure 17 - Configuration after Step 9 of Delete Node Operation



Step 10 – Release the 'lockout of working – span' on the working facilities of the nodes at both ends of the span

Step 11 – Issue reconfig idle command via MP Reconfig Assistant to force the NE Reconfig Software to restore the PG to its initial state.

The NE Software performs the following:

- deletes all connections on the PG
- sets the PG reconfig state to Idle
- clears the reconfig alarm
- sets the PG special mode to None

5 Use Cases

The purpose of the use cases is to add further description to the behaviour of the software, thereby forming additional, or clarifying existing, requirements.

The use cases are written using the procedure descriptions in section 4 on page 17 as a basis. The use cases highlight the state transitions required when starting the procedure, backing out, and completing, which is displayed as a state diagram in Figure 18 on page 31.

5.1 Add Node - success

This is the basic use case from which alternatives are described. It describes a successful run through of the add node procedure, and the alternatives then add further behavioural information.

1. Commission NE on the MP
2. Issue Add Node command
3. Issue 'lockout of working – span' on working facilities
4. Cable in new node on protecting equipment
5. Release 'lockout of working – span'
6. Provision pass through connections
7. Perform connection audit
8. Issue 'forced span switch' on working facilities
9. Cable in new node on working equipment
10. Release 'forced span switch'
11. Issue 'global lockout of protection'
12. Provision new node map information in ring
13. Release 'global lockout of protection'

5.2 Add Node - backout

This use case describes the behaviour when a fault occurs within the procedure that results in the need to backout.

1. Commission NE on the MP
2. Issue Add Node command
3. Issue 'lockout of working – span' on working facilities
4. Cable in new node on protecting equipment
5. Release 'lockout of working – span'
6. Provision pass through connections
7. Perform connection audit
8. Issue 'forced span switch' on working facilities
9. Cable in new node on working equipment
10. Release 'forced span switch'
11. Issue 'global lockout of protection'

At this stage of the process a network failure occurs which cannot be resolved within the time allowed for the add node procedure, and requires the procedure to be backed out:

12. Release 'global lockout of protection'
13. Issue 'forced span switch' on working facilities
14. Cable out new node on working equipment
15. Release 'forced span switch'
16. Issue 'lockout of working – span' on working facilities
17. Cable out new node on protecting equipment
18. Release 'lockout of working – span'
19. Issue Idle command
20. Remove node from MP

NOTE: that the add node procedure can be backed out from any stage up to and including step 11. However, once the new node map information is provisioned the PG is no longer in an add node state and cannot be backed out without using the delete node command and procedure.

5.3 Add Node - failure

When the add node command is executed it can be denied at the validation stage:

1. Commission NE on the MP
2. Issue Add Node command
3. Command is denied
4. Log inspected for reason
5. Fault rectified
6. Re-issue Add Node command
7. … add node procedure continues

5.4 Delete Node - success

This is the basic delete node use case from which alternatives are derived.

1. Issue 'global lockout of protection'
2. Issue Delete Node command
3. Provision new node map information in ring
4. Release 'global lockout of protection'
5. Issue 'forced span switch' on working facilities
6. Cable out node on working equipment
7. Release 'forced span switch'
8. Issue 'lockout of working – span' on working facilities
9. Cable out node on protecting equipment
10. Release 'lockout of working – span'
11. Issue Idle command

5.5 Delete Node - backout

This use case describes the behaviour when a fault occurs within the procedure that results in the need to backout.

1. Issue 'global lockout of protection'
2. Issue Delete Node command
3. Provision new node map information in ring
4. Release 'global lockout of protection'
5. Issue 'forced span switch' on working facilities
6. Cable out node on working equipment
7. Release 'forced span switch'
8. Issue 'lockout of working – span' on working facilities
9. Cable out node on protecting equipment
10. Release 'lockout of working – span'

At this stage of the process a network failure occurs which cannot be resolved within the time allowed for the delete node procedure, and requires the procedure to be backed out:

11. Issue 'lockout of working – span' on working facilities
12. Cable in node on protecting equipment
13. Release 'lockout of working – span'
14. Issue 'forced span switch' on working facilities
15. Cable in node on working equipment
16. Release 'forced span switch'
17. Issue 'global lockout of protection'
18. Re-provision old node map information
19. Release 'global lockout of protection'

NOTE: What this use highlighted was the need for the old-node map information to be stored, such that the backout could be achieved right up to step 10 of the procedure, otherwise step 2 would be the last step at which this procedure could be backed out.

5.6 Delete Node - failure

As with the similar add node use case described in section 5.3 on page 28 the delete node failure case follows a similar path:

1. Issue 'global lockout of protection'
2. Issue Delete Node command
3. Command is denied
4. Log inspected for reason
5. Fault rectified
6. Re-issue Delete Node command
7. … delete node procedure continues

6 Detailed Design

The final stage before coding begins is the detailed design phase. Through this phase the design is elaborated and reviewed to ensure that it meets the requirements as laid down in the initial phases of the project.

Before delving into the specifics of class definitions it is important to completely understand the software architecture required, and with the aid of UML this can be described in the form of state transition diagrams and sequence charts.

This section of the report details:

- the reconfig states the PG can be placed in, and the transitions between those states (section 6.1 on page 30)
- the connection management states, and the transitions between those states (section 6.2 on page 31)
- further information to the sequence of events from the insertion of the reconfig command (sections 6.3, 6.4, and 6.5)
- the final sub-section describes one class through the use of pseudo code, as an illustration of the depth to which this part of the process aims for before coding can commence.

It is important that at every stage of the project the work is reviewed and approved to ensure that the requirements are still being met, therefore it is imperative that the design documentation is clear and concise whilst capturing the salient points.

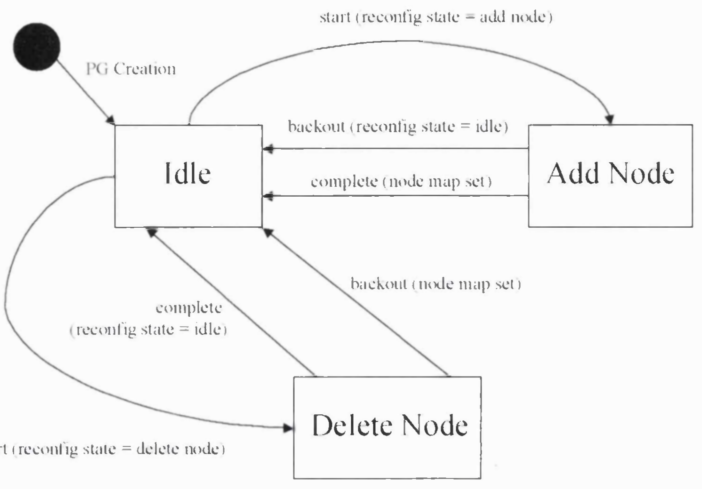
6.1 PG State Diagram

Since the addition of the reconfig state attribute to the PG, the PG can be in one of three states:

1. Idle – no reconfig in progress
2. Add Node – PG currently within add node procedure
3. Delete Node – PG currently within delete node procedure

Due to these states, the following state diagram (Figure 18 on page 31) applies:

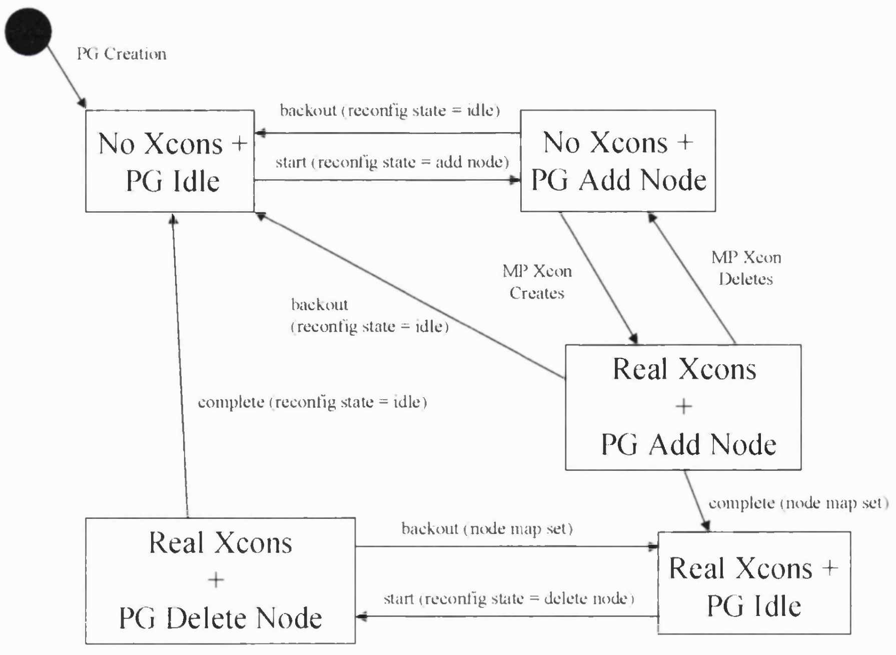
Figure 18 - PG Reconfig State Diagram



start (reconfig state = add node)
PG Creation
backout (reconfig state = idle)
Idle
Add Node
complete (node map set)
backout (node map set)
complete (reconfig state = idle)
Delete Node
start (reconfig state = delete node)

6.2 Connection State Diagram

The following diagram conveys the states in which connections (Xcons) can be placed during the add node and delete node procedures.



PG Creation
backout (reconfig state = idle)
No Xcons + PG Idle
start (reconfig state = add node)
No Xcons + PG Add Node
MP Xcon Creates
MP Xcon Deletes
backout (reconfig state = idle)
complete (reconfig state = idle)
Real Xcons + PG Add Node
complete (node map set)
Real Xcons + PG Delete Node
backout (node map set)
start (reconfig state = delete node)
Real Xcons + PG Idle

It is envisaged that the MP sends connection create messages during the step in the add node procedure to provision real pass through connections on the NE.

The connections software uses an algorithm to determine the PG state (add node) before adding the real traffic connection associated with the MP create message.

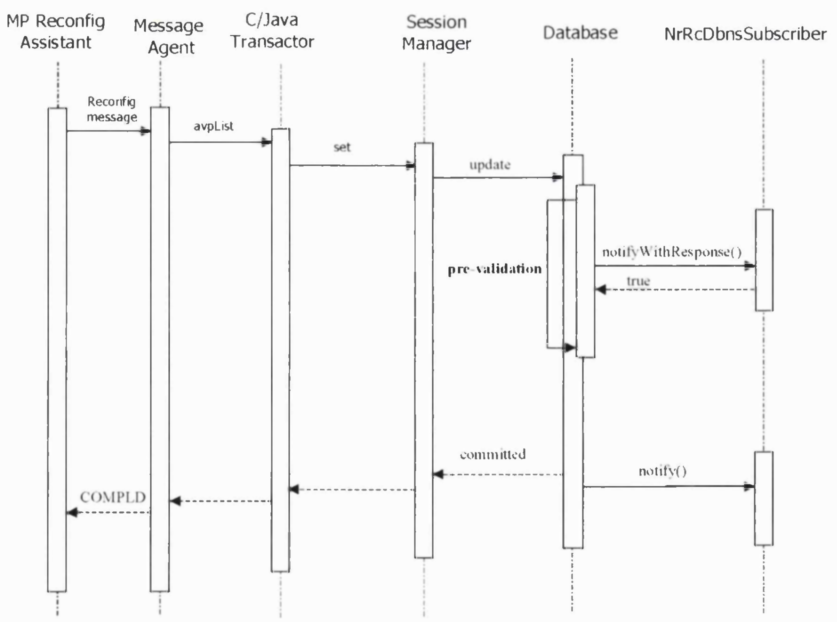
Conversely, to backout, a delete message is sent from the MP to reverse the create.

i.e. the connections software determines that only pass through connections can be created/deleted whilst the PG is in add node, and no creates/deleted are allowed whilst the PG is in delete node. This prevents any traffic loss due to connection editing during the add node or delete node procedures.

6.3 Message Sequence Diagram

The reconfig command is initiated from the MP, or the command line, this is then sent via the session manager to the Database to be 'actioned'. See Figure 19:

Figure 19 - Message Sequence Diagram



The above sequence describes the success path from the MP to the Standard Reconfig Database subscriber which initiates the add node or delete node processing. The continuation of this sequence is displayed in sections 6.4 and 6.5 on pages 33 and 37 respectively, where further detail is added on the processing of the add node and delete node commands.

6.4 Add Node Sequence Diagram

As described in the sequence diagram in the previous section the TpNrRcDbnsSubscriber is informed by the Database of any changes to the reconfig state attribute of the PG.

This is performed at the pre-validation stage so that subsequent changes to the database can be processed within the same transaction.

A summary of the Add Node actions is included here for reference:

Validations: The add node command will be denied if:

- There are any other reconfigurations currently in progress on that PG.
- The specified PG does not have a protection scheme of 4F BLSR.
- The PG has a non-nil node map

Processing:

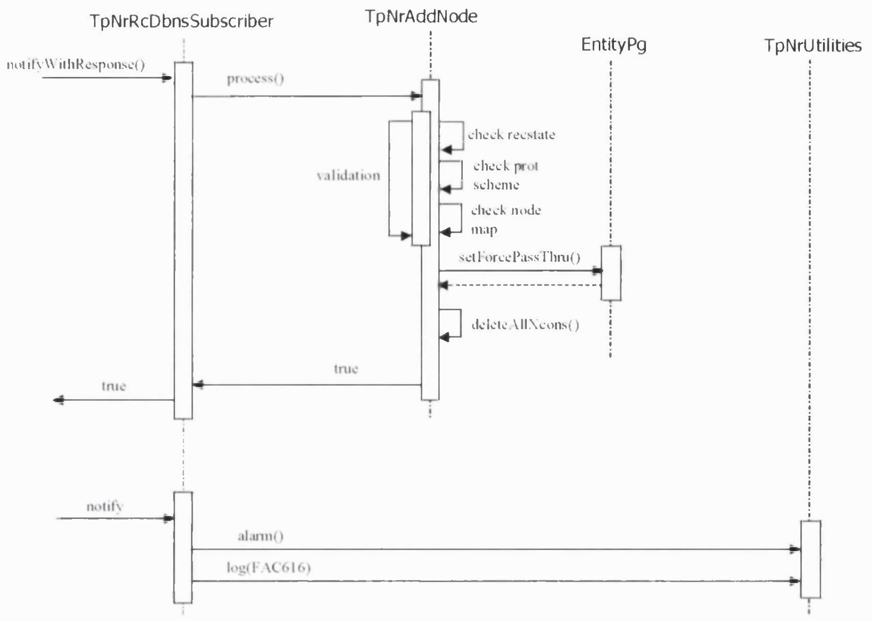
- Set the reconfig mode on the PG to Add Node
- All existing connections on the PG are deleted
- Set the special mode attribute of the PG to PASSTHRU.
- Raise the reconfig alarm
- Generate success log

The following sequence diagram (Figure 20 on page 34) indicates the full success path for an add node command.

NOTE: setting of the reconfig state attribute to add node, is already part of the transaction due to the original message sent from the MP which triggers the remaining processing.

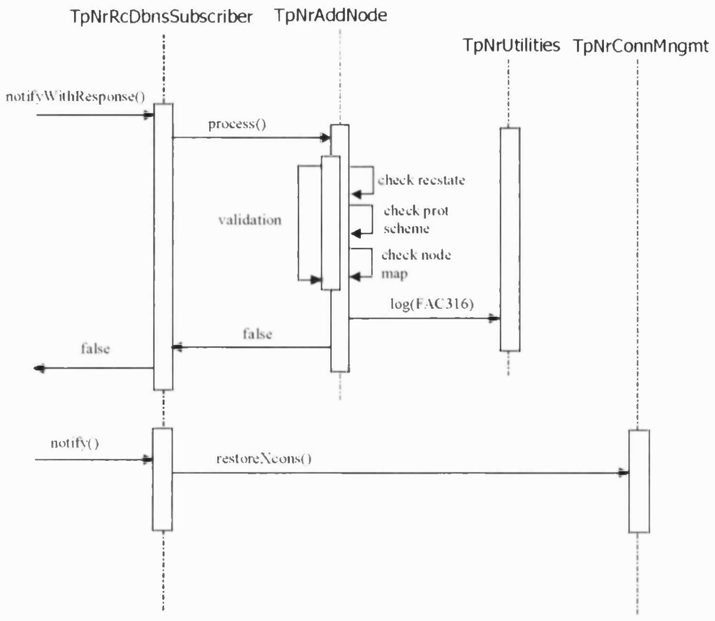
Also, the raising of alarm, and generation of success log are contained within the 'commit' processing of the transaction. This is done to ensure that if the transaction has failed due to any other validation reason, it does not result in conflicting indications that the command has successfully been completed.

Figure 20 - Add Node command - success



If a failure does occur, then this is logged at the point of validation failure, resulting in a failure log being generated, as indicated in the following sequence diagram (Figure 21).

Figure 21- Add Node command - failure



NOTE: In the case of 'any' validation failure the abort processing of TpNrRcDbnsSubscriber is executed to restore any connections that may have been deleted as part of the original command processing.

When the add node procedure is being completed, the TpNrNmDbnsSubscriber is 'listening' for the pre-validation event which sets the node map to a non-nil value, thereby permitting the add node state to clear.

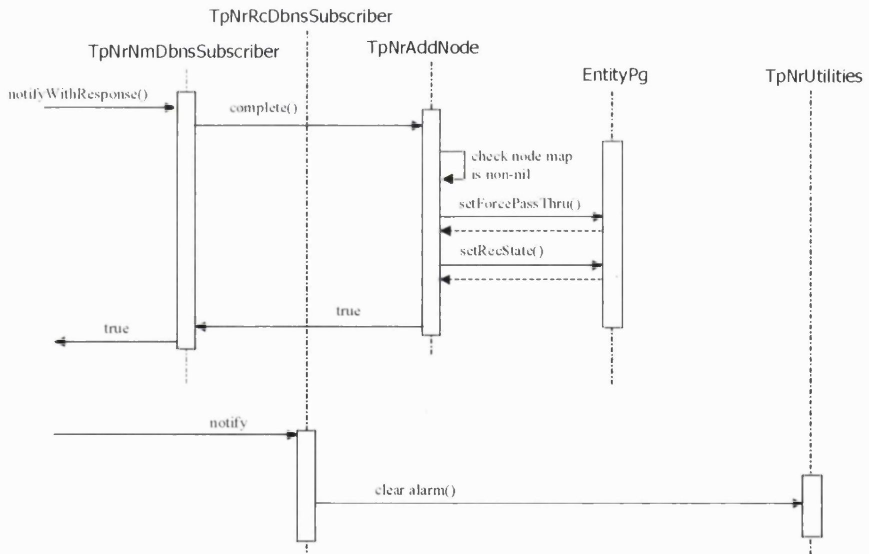
The following processing is executed when a node map set event is received:

If the add node BLSR PG object has a non-nil node map:

- set the special mode of the PG to None
- Set the reconfig state of the PG to Idle
- Clear the reconfig alarm.

NOTE: Similarly to the execution of the add node command described before the clearing of the alarm is processed within the 'commit' phase.

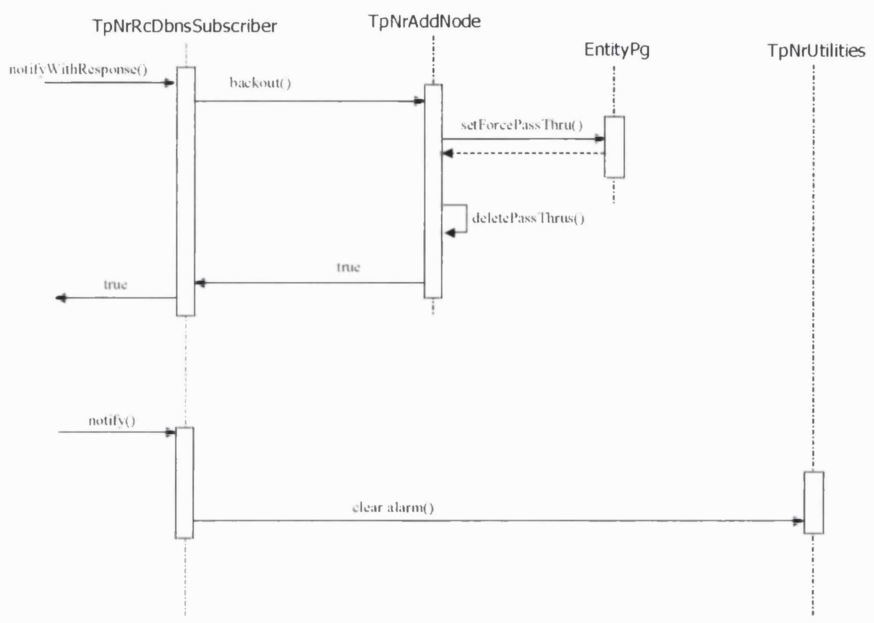
Figure 22 - Add Node command - completion



Finally, add node can be 'backed out' by sending a TL1 command to 'edit' the reconfig state attribute back to 'Idle'.

The processing is similar to the 'clear' case above except that the setting of the reconfig state attribute to idle is already included in the transaction due to the initial reconfig command.

Figure 23 - Add Node command - backout



6.5 Delete Node Sequence Diagram

The sequence of events triggered by a delete node command is very similar to that previously described for add node, with subtle differences.

Again as with add node the processing of the delete node command is performed at the pre-validation stage so that subsequent changes to the database can be processed within the same transaction

A summary of the Delete Node actions is included here for reference:

Validations: The delete node command will be denied if:

- There are any other reconfigurations currently in progress on that PG.
- The specified PG does not have a protection scheme of 4F BLSR.
- There are any non-pass through connections on PG
- There are any 'active' protection switches on PG
- The node map is nil

Processing:

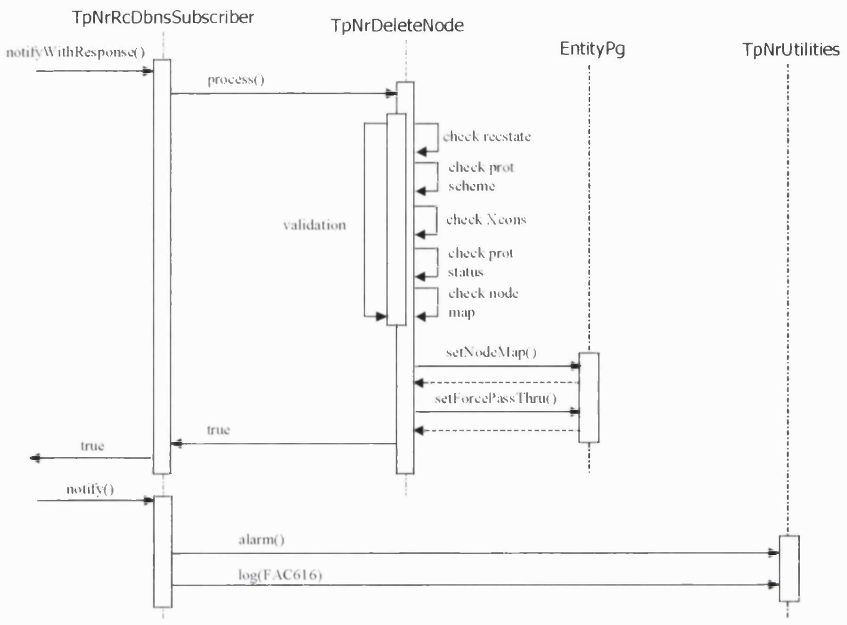
- Set the reconfig mode on the PG to Delete Node
- Set the node map to nil
- Set the special mode attribute of the PG to PASSTHRU.
- Raise the reconfig alarm
- Generate success log

The following sequence diagram (Figure 24 on page 38) indicates the full success path for a delete node command.

NOTE: setting of the reconfig state attribute to delete node, is already part of the transaction due to the original message from the MP which triggers the remaining processing.

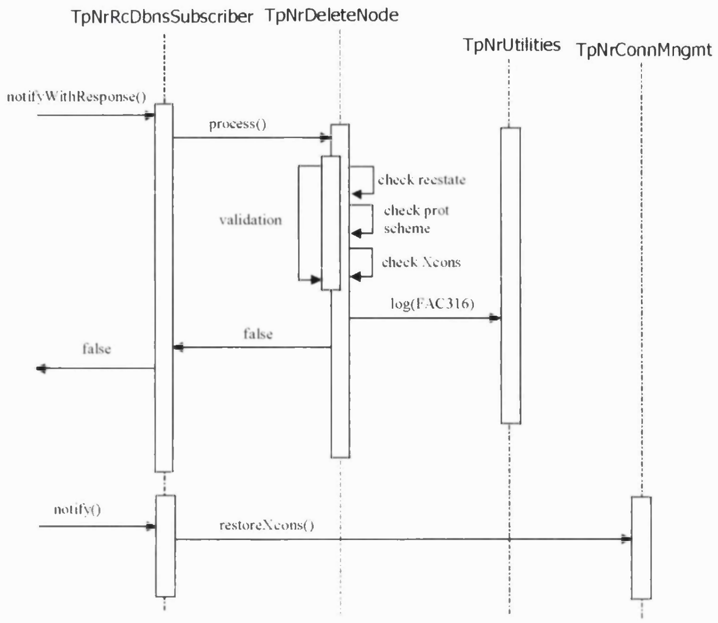
As with add node before, the raising of alarm, and generation of success log are contained within the 'commit' processing of the transaction. This is done to ensure that if the transaction has failed due to any other validation reason, it does not result in conflicting indications that the command has successfully been completed.

Figure 24 - Delete Node command - success



If a failure does occur, then this is logged at the point of validation failure, resulting in a failure log being generated, as indicated in the following sequence diagram (Figure 25).

Figure 25 - Delete Node command - failure



NOTE: As with the add node abort processing, an attempt is made to restore any connections that have been deleted as part of the previous command processing. In the case of delete node this happens when moving from delete node to idle.

When the delete node procedure is being backed out, the TpNrNmDbnsSubscriber is 'listening' for the pre-validation event which sets the node map to a non-nil value, thereby permitting the delete node state to clear.

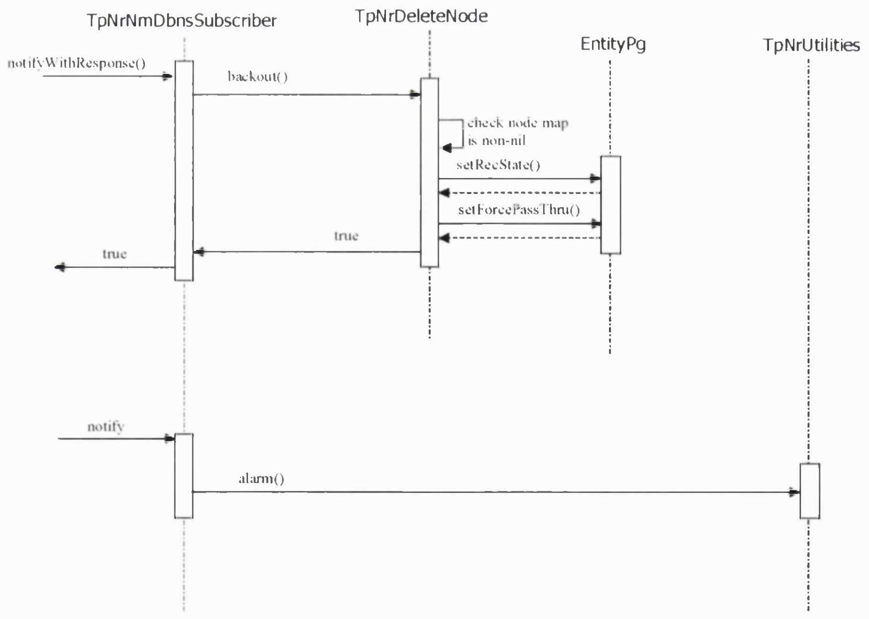
The following processing is executed when a node map set event is received:

If the delete node BLSR PG object has a non-nil node map:

- set the special mode of the PG to None
- Set the reconfig mode PG attribute to Idle
- Clear the reconfig alarm.

NOTE: Similarly to the execution of the delete node command described before, the clearing of the alarm is processed within the 'commit' phase.

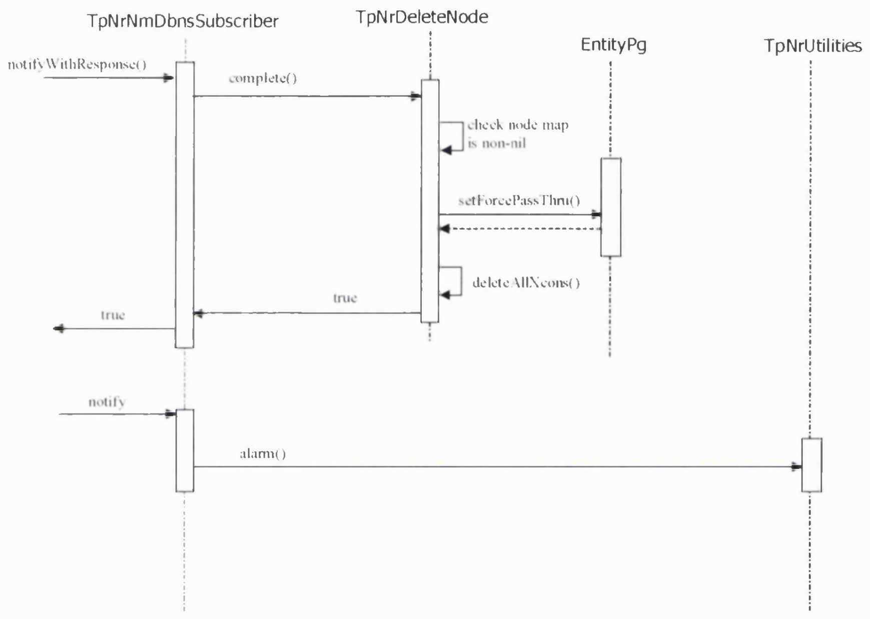
Figure 26 - Delete Node command - backout



Finally, delete node can be 'completed' by sending a reconfig command to 'edit' the reconfig state attribute back to 'Idle'.

The processing is similar to the 'backout' case above except that the setting of the reconfig state attribute to idle is already included in the transaction due to the initial reconfig command, and ALL connections are deleted, which prepares the PG for re-provisioning.

Figure 27 - Delete Node command - completion



6.6 Example Class Definition

For illustrative purposes the class definition for the TpNrReconfigNode class is given, including the pseudo code representation of the initialise method (section 6.6.1 on page 42).

The actual Java code can be found in Appendix A on page 54.

The TpNrReconfigNode class provides the methods that perform common processing between Add Node and Delete Node.

```
class TpNrReconfigNode
{
   // hook provided for Add Node & Delete Node initialisation
   // as part of system wide initialisation
   public void initialise();

   // method provided to subscribe to pre-validation and commit
   // events for the reconfig state attribute
   private boolean subscribeReconfigState();

   // method provided to subscribe to the prevalidate and
   // commit events associated with a node map set, for a
   // specific PG that is currently an add node or delete
   // node mode.
   private boolean subscribeNodeMap ();

   // method provided to generate the unit string associated
   // with the user visible log
   private String genUnitString (EntityPg pgEntity);

   // method provided to set the forcePassThru attribute
   private boolean setForcePassThru ( EntityPg pgEntity,
                                      boolean value);
}
```

6.6.1 TpNrReconfigNode.initialise()

Description:

This method performs the required initialisation for the Add Node and Delete Node features. Namely:

- subscription to the DBNS for edits to the PG Reconfig State attribute
- subscription to the DBNS for edits to the PG Node Map attribute
- get the list of currently provisioned PG Entities from the Database
- for each of these Entities:
    - get the reconfig state value
    - if reconfig state is Add Node or Delete Node:
        - raise the 'ADM in single node configuration' alarm

```
public static void initialise()
{
   try
   {
      // grab the restart instance
      JavaRestart nrJr = JavaRestart.getInstance();

      // wait for NCDbReadyForApplicationsGate

      // call subscribeReconfigState() method to subscribe
      // for edits to the reconfig state attribute of a PG

      // call subscribeNodeMap() method to subscribe
      // for edits to the node map attribute of a PG

      // wait for CardSuccessfullyStartedGate

      // create database transaction
      DbTransaction nrTx;

      // set the type to READ_WRITE
      // begin transaction

      // get a set of all EntityPG objects

      for ( each PG object )
      {
         // get reconfig state attribute value
         // get node map value
         boolean required = false;

         switch (reconfig state)
         {
            case Add Node:
               if (node map != nil)
               {
                  // a node map set event has been missed
                  call TpNrAddNode.complete();
               }
               else
                  required = true;

               break;
            case  Delete Node:
               if (node map != nil)
               {
                  // a node map set event has been missed
                  call TpNrDeleteNode.backout();
               }
               else
                  required = true;

               break;

            default:
               break;
         }

         if (required)
         {
            // call TpNrUtilities.alarm() to raise alarm
         }
      }

      // validate transaction
      // commit transaction

   }
   catch (Exception e)
   {
      new Swerr(e.toString());
   }
}
```

7 Test Strategy & Results

Within this section the testing strategy is explained, and the results published that prove the successful delivery of the feature.

The testing strategy is split between two main sections, the PC simulator (Unit Testing) and the target hardware (Integration Testing).

7.1 Unit Testing

Since hardware is extremely expensive and limited, a fundamental aim of the testing strategy is to minimise the amount of hardware time required, whilst ensuring that the software is still thoroughly tested. Therefore the Unit Testing phase should include sufficient test cases that all possible execution paths are exercised, i.e. all error paths.

The simulator environment consists of a number of sessions. Each session representing a specific card in the target hardware. This allows flexibility when testing, in that a single session can be fired up for those tests that only require a certain card, and a multi-session when interactions between cards are important.

Unit testing will include three main areas:

- Single simulator session
    - Since the standard reconfig software resides on one specific card, the majority of the unit test cases will focus on a single simulator session

- Multi simulator session
    - A certain minority of test cases are to be performed on a multi-sim environment to mimic as best as possible the hardware configuration before integration is performed.

- Manual
    - Although the aim is to automate as much as possible, in certain cases this is not possible, as invasive instrumentation is required to force the software to execute certain unlikely/rare error paths.

Another aim for the single and multi simulator unit tests is to automate the test cases through the use of test scripts, as much as possible, with the subsequent benefit of providing a test suite for sanity and regression purposes. The test suite is described in Appendix D on page 129.

7.2 Unit Test Cases

A full list (143) of the unit test cases used to test Add Node and Delete Node is contained in Appendix B on page 121. An example (UT1) is shown here to illustrate how each test case was written.

NOTE: The test case assumes basic knowledge of the equipment being tested, but indicates what results are required to indicate that this test case passes. Each unit test case listed in Appendix B has an associated description in the Designer Test Plan ([3]).

UT1 Successful Add Node command

Description:
Issue ED-FFP-LL TL1 Add Node command on Quad CP in slot 22, on PG1 (port 2).
e.g.
ED-FFP-LL:OPTERA:PGLL-000001220201:CTAG::RCSTATE=ADDNODE;

Initial state:

- NE created
- Quad CP created, and associated 4F BLSR PG/PGMs

Results:

- TL1 response is COMPLD
- AO Log is generated and indicates RCSTATE=ADDNODE
- ADM in single node configuration alarm is raised (MN,NSA)
- PG is in add node state:
    - o rcstate is add node
    - o node map is nil
    - o ieeemap is nil
    - o special mode is passthru
- Network Reconfig FAC616 log is generated
    - o Logfac616
    - o Operation is Add Node
    - o Unit indicates PG + associated PGMs (LL)
- No SWERRs
- No Traps

7.3 Integration Testing

Although the simulator is a good approximation of the hardware the simulated behaviour is not completely accurate with the actual target hardware. Also, since these procedures require traffic monitoring which the simulator cannot do, the integration testing is necessary and cannot be completely superseded by the simulator.

However, the simulator does provide a baseline performance or confidence factor that the hardware can be tested against. Also, due to time and hardware constraints it isn't possible to fully exercise every execution path of the software on the target hardware. Therefore the strategy is to ensure full coverage is achieved through a combination of unit testing and integration testing without sacrificing quality.

With this in mind each integration test case is assigned a priority, of 1, 2 or 3. Priority 1 test cases must be executed and passed before the software can be delivered to Verification for final testing prior to delivery to the customer. Priority 2 and 3 are to executed as time and equipment allows, but must be completed prior to customer delivery.

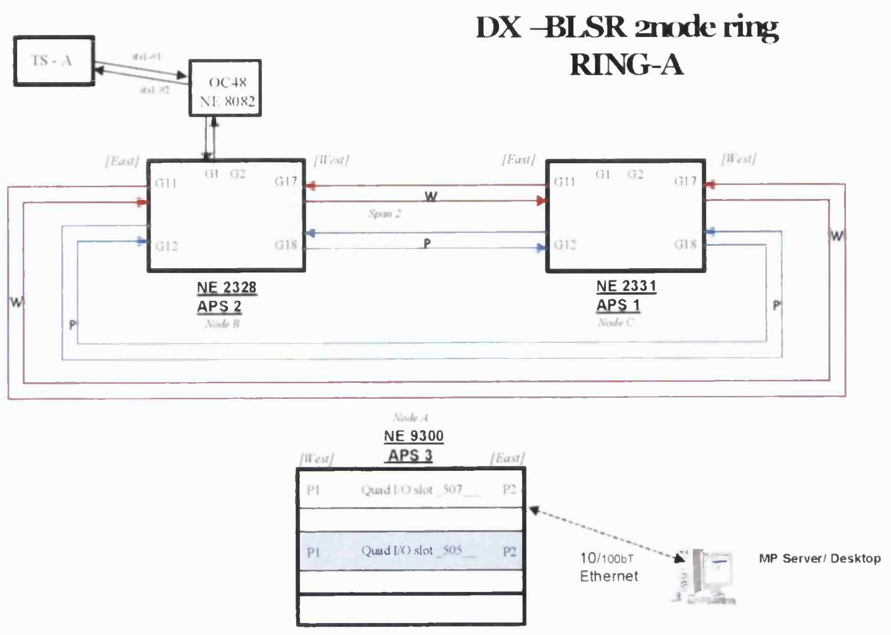
A full list of integration test cases and their priority are given in Appendix C on page 126.

In Figure 28 on page 47 and Figure 29 on page 48, the hardware configurations used in testing Add Node and Delete Node are illustrated.

When viewing the hardware configurations, TS-A and TS-B refer to the test sets that monitor the traffic in Ring-A and Ring-B respectively. The Add Node and Delete Node procedures operate on Ring-A, with the traffic being monitored in Ring-B to ensure there is no traffic disruption whilst Ring-A is being reconfigured.

e.g. the traffic hits that will be observed, should be on TS-A, and even then they should be <50ms i.e. the equivalent of a protection switch

Figure 28- Hardware Configuration: Add Node



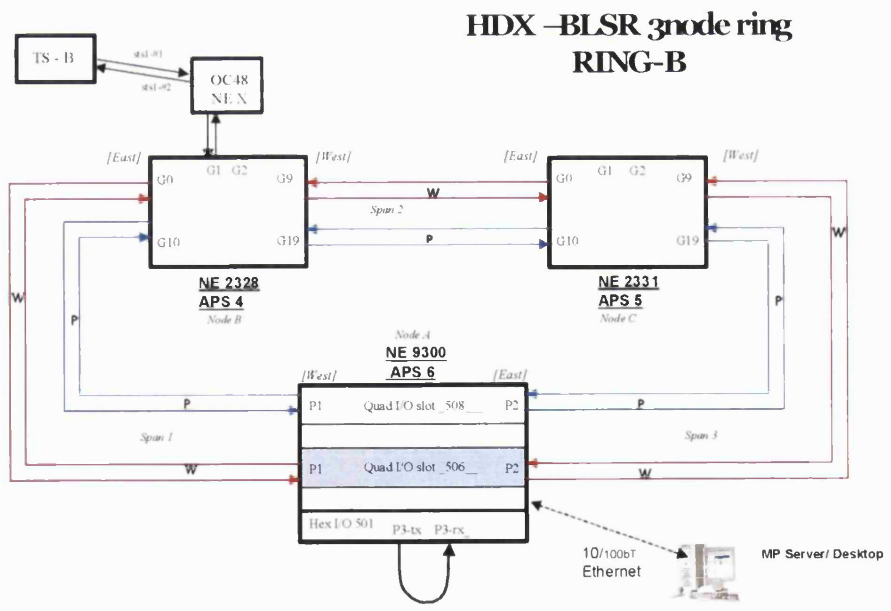
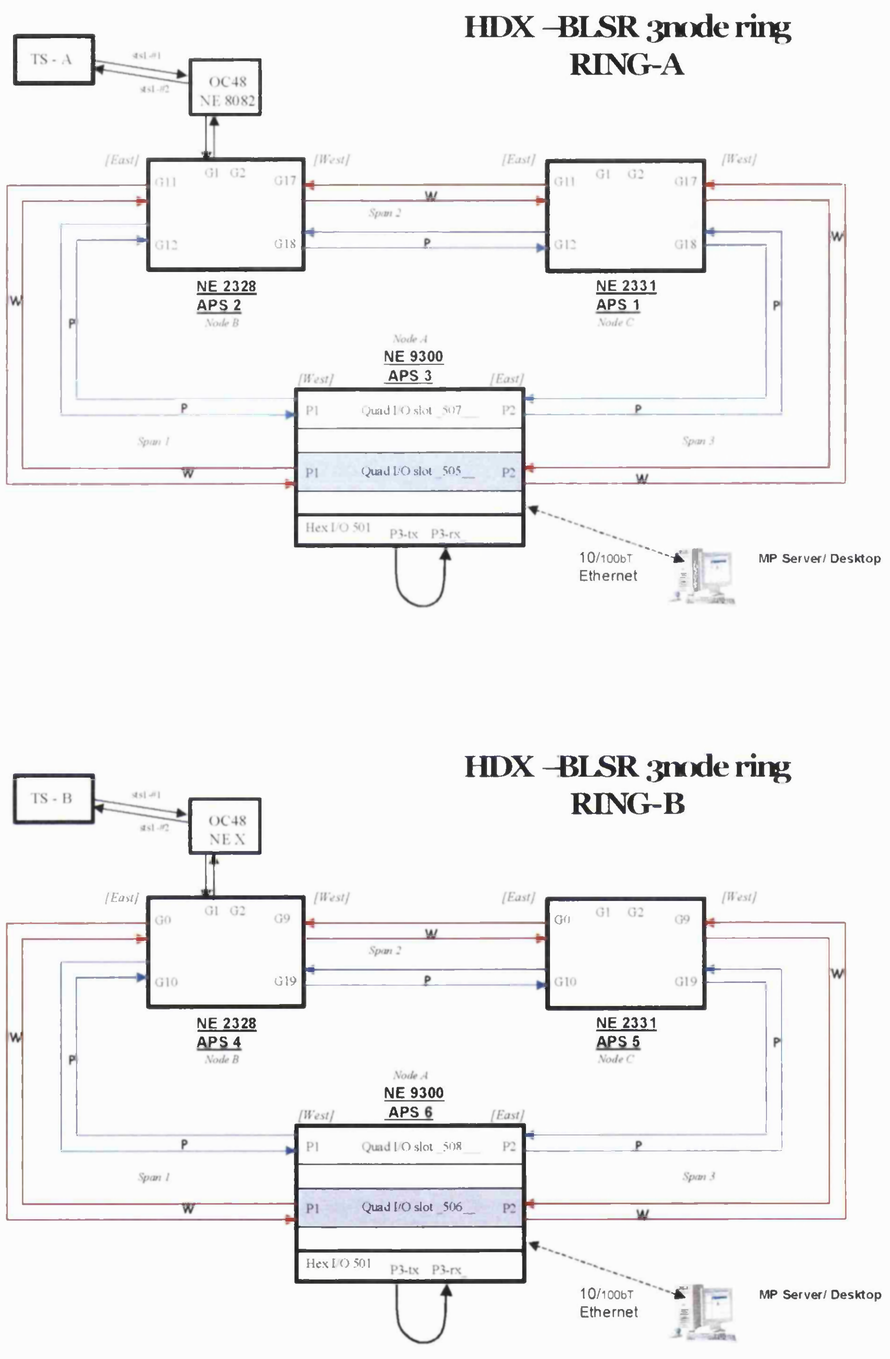


Figure 29- Hardware Configuration - Delete Node



7.4 Integration Test Cases

A full list of integration test cases can be found in Appendix C on page 126. As with the unit test case description in section 7.2 on page 45 the full description of an integration test case is given here for illustrative purposes.

IT1 Add HDX Node to 4F BLSR Ring

Description:
Follow the Add Node procedure to add a node to an existing 4F BLSR Ring, (additionally for IT1 monitor procedure from network management platform)

Add PG1, e.g. slot 21 (505), port 2.

Initial state:

- Hardware Configuration: Add Node

Results:

- On NE after add node command is sent from MP (Step 2):
    - TL1 response is COMPLD
    - AO Log is generated and indicates RCSTATE=ADDNODE
    - ADM in single node configuration alarm is raised (MN,NSA)
    - PG is in add node state:
        - rcstate is add node
        - node map is nil
        - ieeemap is nil
        - special mode is passthru
    - Network Reconfig FAC616 log is generated
        - Logfac616
        - Operation is Add Node
        - Unit indicates PG + associated PGMs (LL)
    - No SWERRs
    - No Traps
- On NE after pass-through connections are provisioned by MP (Step 6):
    - Check that the expected pass through connections are created
    - No SWERRs
    - No Traps
- On NE after the node map has been sent (Step 12):
    - TL1 response is COMPLD
    - AO Log is generated and indicates RCSTATE=IDLE
    - ADM in single node configuration alarm is cleared (MN,NSA)
    - PG is in idle state:

            - rcstate is idle
            - node map is non-nil (value set by PMEM)
            - ieeemap is non-nil
            - special mode is none
        - No Network Reconfig log is generated
        - No SWERRs
        - No Traps
- On MP:
    - Check node has been added to the configuration
- Through out procedure:
    - Monitor traffic and ensure no unexpected hits and no expected hits >50ms
    - Monitor traffic on other PG on Quad and ensure no traffic hits

7.5 Test Results

143 of 143 unit test cases executed and passed
- 100% pass rate
- 100% execution rate
- 137 test cases automated

84 of 84 integration test cases executed and passed
- 100% pass rate
- 100% execution rate

8 Conclusions

The Add Node and Delete Node features were successfully delivered to Verification on time and according to schedule. During this final phase for customer acceptance, no faults were found in the NE Software, which has now been approved for the first release of the next generation transport platform.

This report documents my work in delivering the Standard Reconfigurations software feature which specifically encompasses the entirety of the Network Element software portion of this feature. The other aspect which completes this feature was the Management Platform work in delivering the Reconfig Assistant which is outside the scope of this report.

The success of the project has been due to adhering to the software development process, ensuring that at each stage the artefacts produced are adequately reviewed and approved. The artefacts being the documentation that indicates the completion of a phase in the process e.g. requirements, procedures, design, code etc..

Initial customer requirements were received, broken down into more detailed software design requirements (section 3 on page 14) based on the procedures (section 4 on page 17) that dictated the behaviour of the software, coupled with the use cases (section 5 on page 27) to elaborate those points that were not clear.

Detailed design (section 6 on page 30) then added further detail in the form of the implementation approach that would be taken, before the actual code writing was started.

Finally the designer testing (section 7 on page 44) phase started, with unit and integration testing combining to ensure full code coverage, whilst also checking the behaviour matched the initial requirements.

Although the project did encounter issues during the development of the features, especially during the unit testing phase, these proved to be minor in nature.

In essence due to the use of UML the design approach was easily and quickly understood by reviewers and other designers ensuring minimal integration issues.

As such, UML has proven to be a valuable addition to the design environment, providing a common language between designers for describing software.

Since the design was well described, and reviewed, the Java implementation proved to be relatively straight forward. The benefits of Java have been well documented, that include portability, ease of use, and inbuilt memory management.

However, there was no real indication as to whether Java was beneficial in the respect of this feature. An equally effective implementation could have been achieved using C++.

9 References

[1] Standard Reconfigurations High Level Design – v1.03 July 2002 – Andy Kinney

[2] Standard Reconfigurations Detailed Design – v1.1 July 16, 2002 – Andy Kinney

[3] Standard Reconfigurations Designer Test plan – v0.5 July 8, 2002 – Andy Kinney

[4] UML Distilled – Applying the Standard Object Modelling Language – Martin Fowler with Kendall Scott – Addison Wesley – 5th Printing December, 1997 – ISBN 0-201-32563-2

[5] Java 2 Platform - Standard Edition v1.4.1 – API Specification http://java.sun.com/j2se/1.4.1/docs/api/

[6] Sonet (Synchronous Optical NETwork) and SDH (Synchronous Digital Hierarchy) – Beginner Guide http://www.lightreading.com/document.asp?doc_id=4432

[7] Synchronous Transmission Systems – Doc GH9, Issue 3 – Christopher Newall – Northern Telecom Europe

[8] SONET/SDH Ring Technology – Optical Networking http://ntrg.cs.tcd.ie/undergrad/4ba2.02/optnet/pages/present/sont_and_sdh_bandwidth.htm

[9] G.707 Network node interface for the synchronous digital hierarchy (SDH) – ITU-T – (10/2000)

[10] G.708 Sub STM-0 network node interface for the synchronous digital hierarchy (SDH) – ITU-T – (06/99)

[11] G.709 Interfaces for the optical transport network (OTN) – ITU-T – (02/2001)

Appendix A: Java Implementation

The code listing for the Add Node and Delete Node feature is contained within various modules. An overview of the module structure is described here:

TpNetworkReconfigInit.java - class definition which has an initialise method which serves as a hook into the system initialisation

TpNrReconfigNode.java - class definition which encapsulates the common functionality to support the Add Node and Delete Node reconfigurations

TpNrAddNode.java - class definition which encapsulates all the high level processing required for an Add Node operation

TpNrDeleteNode.java - class definition which encapsulates all the high level processing required for a Delete Node operation

TpNrConnMngmt.java - class definition which encapsulates the functionality used to manipulate traffic connections associated with the reconfig commands

TpNrErrorCode.java - class definition which encapsulates all the network reconfig error strings that are used to inform the user of the reason the reconfig operation failed

TpNrNmDbnsSubscriber.java - performs the necessary processing when the node map is updated

TpNrRcDbnsSubscriber.java – performs the necessary processing when the PG reconfig state is updated

TpNrRestartRecovery.java – class which implements Runnable and encapsulates the methods to initialise the network reconfiguration code, thereby providing a thread in which the reconfig commands execute

TpNrUtilities.java - class definition which encapsulates the utilities to support network reconfig commands

TpNrValidations.java – class definition which encapsulates the functionality used to validate network reconfig commands

Class: TpNetworkReconfigInit

```
/**
 * <B>File:</B> <CODE>TpNetworkReconfigInit.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrRestartRecovery;
import equinox.framework.utilities.eqxthread.EqxThread;

/**
 * <P><B>Class:</B> <CODE>TpNetworkReconfigInit</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNetworkReconfigInit class definition has an initialise method which
 * serves as a hook into the system initialisation.
 *
 * @author Andy Kinney
 *
 */
/*
 * Methods:
 *
 * void initialize()
 * - launches the thread which in turn calls the Network Reconfig
 *   initialisation code
 *
 */

public final class TpNetworkReconfigInit
{
   /**
    * singleton instance of this class
    */
    private static TpNetworkReconfigInit INSTANCE = null;
    /**
     *
     * <P><B>Method:</B>       <CODE>getInstance</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the instance of this singleton.</P>
     *
     * @return      TpNetworkReconfigInit
     *                - the singleton instance of TpNetworkReconfigInit
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
     */

    public static synchronized TpNetworkReconfigInit getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNetworkReconfigInit();
        }

        return INSTANCE;
    }

   /**
    * Constructor, made private as class is singleton
    */
    private TpNetworkReconfigInit ()
    {
    }

   /**
    *
    * <P><B>Method:</B>       <CODE>initialize</CODE>
    *
    * <BR><B>Description:</B>
    *
    * This method launches the thread which in turn calls the Network Reconfig
    * initialisation code</P>
    */
   /*
    * Inputs:       None
    *
    * Outputs:      None
    *
    * Returns:      None
    *
    */

    public static void initialize ()
    {
        TpNrRestartRecovery nrRst = new TpNrRestartRecovery();
        EqxThread eqxThread = new EqxThread (nrRst, nrRst.getName());
        eqxThread.start();
    }

} // end of TpNetworkReconfigInit
```

Class: TpNrReconfigNode

```
/**
 * <B>File:</B> <CODE>TpNrReconfigNode.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.framework.JavaRestartApi.JavaRestart;

import equinox.framework.database.DbTransaction;
import equinox.framework.database.DbTransactionState;

import equinox.framework.nom.keys.EntityPgKey;
import equinox.framework.nom.keys.EntityPgmKey;
import equinox.framework.nom.bases.EntityKey;

import equinox.framework.swerr.Swerr;

import equinox.framework.nom.objs.EntityObjBase;
import equinox.framework.nom.objs.EntityPg;

import equinox.framework.nom.enums.SpecialModeEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;
import equinox.framework.nom.enums.AF_ProblemTypeEnum;
import equinox.framework.nom.enums.EntityIdEnum;

import equinox.protection.traffic.utils.TpAidAdaptationImpl;

import equinox.framework.database.dbns.Dbns;
import equinox.framework.database.dbns.Operation;
import equinox.framework.database.dbns.DbnsFilter;
import equinox.framework.database.dbns.DbnsEvent;

import equinox.protection.traffic.reconfig.TpNrAddNode;
import equinox.protection.traffic.reconfig.TpNrDeleteNode;
import equinox.protection.traffic.reconfig.TpNrValidations;
import equinox.protection.traffic.reconfig.TpNrUtilities;
import equinox.protection.traffic.reconfig.TpNrRcDbnsSubscriber;
import equinox.protection.traffic.reconfig.TpNrNmDbnsSubscriber;

import equinox.framework.avp.AvpList;
import equinox.framework.avp.AvpFactory;

import equinox.framework.cfgutil.SlotIdTranslation;

import java.util.HashSet;
import java.util.Iterator;
```

```
/**
 * <P><B>Class:</B> <CODE>TpNrReconfigNode</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *      - available in Athena, following this path:
 * <BR>  - Equinox
 * <BR>   - Product Development
 * <BR>     - Software
 * <BR>       - Network Element Software
 * <BR>         - Functional Areas
 * <BR>           - Network Reconfig
 * <BR>             - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrReconfigNode class definition encapsulates the common functionality
 * to support the Add Node and Delete Node reconfigurations.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * void initialize()
 * - hook provided for Add Node & Delete Node initialisation as part of
 *   system wide initialisation
 *
 * boolean subscribeReconfigState()
 * - method provided to subscribe to pre-validation and commit events
 *   for the reconfig state attribute
 *
 * void subscribeNodeMap()
 * - method provided to subscribe to pre-validation and commit events
 *   for the node map attribute
 *
 * String genUnitString(...)
 * - generates the unit string associated with the user visible log
 *
 * boolean setNodeMap(...)
 * - sets the node map attribute to nil
 *
 * boolean setForcePassThru(...)
 * - sets the force pass through attribute
 *
 */

public final class TpNrReconfigNode
{

   /**
    * singleton instance of this class
    */
    private static TpNrReconfigNode INSTANCE = null;
    /**
     *
     * <P><B>Method:</B>        <CODE>getInstance</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the instance of this singleton.</P>
     *
     * @return      TpNrReconfigNode
     *                - the singleton instance of TpNrReconfigNode
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
     */

    public static synchronized TpNrReconfigNode getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNrReconfigNode();
        }

        return INSTANCE;
    }

   /**
    * Constructor, made private as class is singleton
    */
    private TpNrReconfigNode ()
    {
    }
```

```
/**
 * <P><B>Method:</B>       <CODE>initialize</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method performs the required initialisation for the Add Node and
 * Delete Node features. </P>
 *
 * <P>Namely:
 * <BR> - subscription to the DBNS for edits to the reconfig state attribute
 * <BR> - subscription to the DBNS for edits to the node map attribute
 * <BR> - get the list of currently provisioned PG Entitys in the Database
 * <BR> - for each of these entities
 * <BR>     - get the reconfig state value
 * <BR>     - if Add Node or Delete Node
 * <BR>          - raise the 'ADM in single node configuration' alarm </P>
 *
 * Inputs:       None
 *
 * Outputs:      None
 *
 * Returns:      None
 *
 */

public void initialize ()
{
    DbTransaction nrTx = null;

    try
    {
        // grab the restart instance
        JavaRestart nrJr = JavaRestart.getInstance();

        // wait for NCDbReadyForApplicationsGate
        nrJr.waitForRestartGate(JavaRestart.RST_GATE_NC_DB_READY);

        // subscribe to the reconfig state attribute and node map events
        if (subscribeReconfigState() &&
            subscribeNodeMap())
        {
            // wait for CardSuccessfullyStartedGate to allow for alarm
            // framework to initialise, otherwise alarm cannot be raised
            nrJr.waitForRestartGate(JavaRestart.RST_GATE_STARTED);

            // create database transaction
            nrTx = new DbTransaction();

            // set the transaction to indicate that this could write to as
            // well as read from the database, and start the transaction
            nrTx.setType(DbTransaction.READ_WRITE);
            nrTx.begin();

            // grab a set of all PG entities in the database
            HashSet pgSet =
                (TpNrUtilities.getInstance()).getSetOfObjects(
                                  EntityIdEnum.EN_PG);

            if (pgSet != null)
            {
               // loop through all the PG entities
               for (Iterator i = pgSet.iterator(); i.hasNext();)
               {
                   EntityPg pgEntity = (EntityPg)i.next();
                   int recState = pgEntity.getRcstate().intValue();
                   boolean required = false;
                   TpNrValidations tpNrValidations =
                                 TpNrValidations.getInstance();
                   switch (recState)
                   {
                       case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                           // check to see if node map is non-nil
                           if (tpNrValidations.nodeMapIsNil(pgEntity)
                                                            != null)
                           {
                               // a non-nil node map indicates a node map
                               // set event has been missed, and the add
```

```
                                    // node command has been completed
                                    (TpNrAddNode.getInstance()).
                                            complete(pgEntity);
                                }
                                else
                                {
                                    required = true;
                                }
                                break;

                            case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                                // check to see if node map is non-nil
                                if (tpNrValidations.nodeMapIsNil(pgEntity)
                                                                != null)
                                {
                                    // a non-nil node map indicates a node map
                                    // set event has been missed, and the
                                    // delete node command has to be backed out
                                    (TpNrDeleteNode.getInstance()).
                                            backout(pgEntity);
                                }
                                else
                                {
                                    required = true;
                                }
                                break;

                            case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
                                // do nothing
                                break;

                            default:
                                new Swerr("Invalid reconfig state : " +
                                                recState);
                                break;
                        }

                        if (required)
                        {
                            // raise the 'ADM in single node configuration'
                            // alarm
                            (TpNrUtilities.getInstance()).alarm(pgEntity,
                                AF_ProblemTypeEnum.
                                    AF_ADMINSINGLENODECONFIG_PT,
                                true);
                        }
                    }
                }

                // validate and commit the transaction
                nrTx.validate();
                nrTx.commit();
            }
        }
        catch (Exception e)
        {
            new Swerr(e.toString());
        }
        finally
        {
            if (nrTx != null)
            {
                try
                {
                    if (!(nrTx.getState()).isCommitted())
                    {
                        nrTx.abort();
                    }

                }
                catch (Exception e)
                {
                    new Swerr(e.toString());
                }
            }
        }
    }
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>subscribeReconfigState</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method subscribes to the DBNS for events associated with any PG's
 * reconfig state attribute being set, and is called during the
 * initialisation of Network Reconfigs.</P>
 *
 * <P>NOTE: Both the Commit and Pre-Validation events are subscribed to. The
 * processing of the add node or delete node command will be executed within
 * the context of the pre-validation event, and the final processing
 * (generation of success log, subscription to node map set events, and
 * raising of alarm) is performed within the commit event.</P>
 *
 *
 * @return      boolean
 *              - true if successful, false otherwise
 *
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public boolean subscribeReconfigState()
{
    boolean rc = true;

    try
    {
        // define wildcard PG key
        EntityPgKey pgKey = new EntityPgKey(EntityPgKey.WC_shelfId,
                                            EntityPgKey.WC_slotId,
                                            EntityPgKey.WC_portId,
                                            EntityPgKey.WC_pipeId);

        // set the operation to update
        Operation recStateSet = new Operation( true,  // insert
                                               true,  // delete
                                               true ); // update

        // create avplist with reconfig state attribute
        AvpList recStateAvpList = new AvpList();
        recStateAvpList.addAvp(AvpFactory.newAvp("rcstate"));

        // create DBNS filter for reconfig state attribute
        DbnsFilter recStateFilter = new DbnsFilter (recStateSet,
                                                    pgKey,
                                                    recStateAvpList);

        // subscribe to the pre-validation event for reconfig state
        Dbns.subscribe(DbnsEvent.DBNS_PRE_VALIDATION_EVENT_TYPE,
                       TpNrRcDbnsSubscriber.getInstance(),
                       recStateFilter);

        // subscribe to the commit event for reconfig state
        Dbns.subscribe(DbnsEvent.DBNS_COMMIT_EVENT_TYPE,
                       TpNrRcDbnsSubscriber.getInstance(),
                       recStateFilter);

        // subscribe to the abort event for reconfig state
        Dbns.subscribe(DbnsEvent.DBNS_ABORT_EVENT_TYPE,
                       TpNrRcDbnsSubscriber.getInstance(),
                       recStateFilter);


    }
    catch (Exception e)
    {
        new Swerr (e.toString());
        rc = false;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>subscribeNodeMap</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method subscribes to/unsubscribes from the DBNS for events
 * associated with a PG nodemap attribute.</P>
 *
 * @return      boolean
 *              - true if successful, false otherwise
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public boolean subscribeNodeMap()
{
    boolean rc = true;

    try
    {
        // define wildcard PG key
        EntityPgKey pgKey = new EntityPgKey(EntityPgKey.WC_shelfId,
                                            EntityPgKey.WC_slotId,
                                            EntityPgKey.WC_portId,
                                            EntityPgKey.WC_pipeId);

        // set the operation to update
        Operation nodeMapSet = new Operation( false,  // insert
                                              false,  // delete
                                              true ); // update

        // create avplist with reconfig state attribute
        AvpList nodeMapAvpList = new AvpList();
        nodeMapAvpList.addAvp(AvpFactory.newAvp("nodemap"));

        // create DBNS filter for reconfig state attribute
        DbnsFilter nodeMapFilter = new DbnsFilter (nodeMapSet,
                                                   pgKey,
                                                   nodeMapAvpList);

        // subscribe to the pre-validation event for node map
        Dbns.subscribe(DbnsEvent.DBNS_PRE_VALIDATION_EVENT_TYPE,
                       TpNrNmDbnsSubscriber.getInstance(),
                       nodeMapFilter);

        // subscribe to the commit event for node map
        Dbns.subscribe(DbnsEvent.DBNS_COMMIT_EVENT_TYPE,
                       TpNrNmDbnsSubscriber.getInstance(),
                       nodeMapFilter);
    }
    catch (Exception e)
    {
        new Swerr (e.toString());
        rc = false;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>genUnitString</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method generates the unit string which is subsequently used in the
 * TpNrUtilities.log() method for generating the User Visible Log.</P>
 *
 * @param       pgEntity
 *              - the PG upon which the reconfig operation has been executed
 *
 * @return      String
 *              - the string associated with the unit upon which the
 *                reconfig operation has been executed
 */
/*
 * Outputs:     None
 *
 */

public String genUnitString (EntityPg pgEntity)
{
    StringBuffer unit = new StringBuffer("");

    try
    {
        TpAidAdaptationImpl tpAid = TpAidAdaptationImpl.getInstance();

        unit = unit.append(tpAid.EntityToAID((EntityObjBase)pgEntity)
                                        + " [");

        // get shelf, slot and port information for each PGM that makes
        // up the PG
        EntityPgmKey[] pgmList = pgEntity.getMemberList();
        byte pgmNumber = pgEntity.getNumberOfPgm();

        // loop through the pgm list and add the shelf, slot, port info to
        // the unit string
        int lastPgm = pgmNumber - 1;
        for (byte index = 0; index < pgmNumber; index++)
        {
            EntityPgmKey pgmKey = pgmList[index];

            unit = unit.append(tpAid.EntityToAID((EntityKey)pgmKey,
                                        EntityIdEnum.EN_PGM_VALUE));

            if (index != lastPgm)
            {
                unit = unit.append(", ");

                // in the case where 4 PGMs are being stored in the unit
                // string the 80 character window limit is exceeded, and
                // this additional formatting is required
                if (index == 1)
                {
                    unit = unit.append("\n\t\t");
                }
            }
        }

        // add trailing ']'
        unit = unit.append("]");
    }
    catch (Exception e)
    {
        new Swerr(e.toString());
    }

    return unit.toString();
}
```

```
    /**
     *
     * <P><B>Method:</B>      <CODE>setForcePassThru</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method sets the force pass through PG attribute to the value
     * passed in. Setting the force pass through value to true places the
     * protection channels into full pass through mode (traffic and kbytes).</P>
     *
     *
     * @param       pgEntity
     *              - the PG upon which the reconfig operation has been executed
     * @param       passthru
     *              - true to enable force pass thru, false to disable
     *
     * @return      boolean
     *              - true if successful, false otherwise
     */
    /*
     * Outputs:     None
     *
     */

    public boolean setForcePassThru (EntityPg pgEntity,
                                     boolean passthru)
    {
        boolean rc = true;

        try
        {
            if (passthru)
            {
                pgEntity.setSpecialmode(SpecialModeEnum.SPECIAL_MODE_PASSTHRU);
            }
            else
            {
                pgEntity.setSpecialmode(SpecialModeEnum.SPECIAL_MODE_NONE);
            }
        }
        catch (Exception e)
        {
            new Swerr(e.toString());
            rc = false;
        }

        return rc;

    }

} // end of TpNrReconfigNode
```

Class: TpNrAddNode

```
/**
 * <B>File:</B> <CODE>TpNrAddNode.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrErrorCode;
import equinox.protection.traffic.reconfig.TpNrValidations;
import equinox.protection.traffic.reconfig.TpNrUtilities;
import equinox.protection.traffic.reconfig.TpNrConnMngmt;
import equinox.protection.traffic.reconfig.TpNrReconfigNode;

import equinox.framework.swerr.Swerr;

import equinox.framework.nom.objs.EntityPg;

import equinox.framework.nom.enums.AF_ProblemTypeEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;


/**
 * <P><B>Class:</B> <CODE>TpNrAddNode</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>  - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrAddNode class definition encapsulates all the high level processing
 * required for an Add Node operation.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * boolean process(...)
 * - processing required when an add node operation is requested
 *
 * boolean backout(...)
 * - processing involved when add node is backed out
 *
 * boolean complete(...)
 * - processing involved in completing the add node operation
 *
 */

public final class TpNrAddNode
{
    /**
     * singleton instance of this class
     */
    private static TpNrAddNode INSTANCE = null;
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>getInstance</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method returns the instance of this singleton.</P>
 *
 * @return     TpNrAddNode
 *              - the singleton instance of TpNrAddNode
 */
/*
 * Inputs:     None
 *
 * Outputs:    None
 *
 */

public static synchronized TpNrAddNode getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new TpNrAddNode();
    }

    return INSTANCE;
}

/**
 * Constructor, made private as class is singleton
 */
private TpNrAddNode ()
{
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>process</CODE>
 *
 * <BR><B>Description:</B>
 *
 *
 * This method is invoked when the
 * <CODE> TpNrRcDbnsSubscriber.notifyWithResponse() </CODE>
 * method is called with a request to pre-validate setting the reconfig
 * state PG attribute to add node. <BR></P>
 *
 * <P>This method then performs the required Add Node validations and
 * processing,returning true if successful, false otherwise. </P>
 *
 * <P>NOTE: The creation of the pass through connections on the working
 * channels is owned by Connection Management software, and not included in
 * this method. </P>
 *
 *
 * @param       pgEntity
 *              - the entity in the database upon which the add node
 *                operation has been requested.
 *
 * @return      boolean
 *              - TRUE if add node command successful, FALSE otherwise
 */
/*
 * Outputs:     None
 *
 */

 public boolean process (EntityPg pgEntity)
 {
     boolean rc = false;
     String errorReason = TpNrErrorCode.TP_NR_ERR_SWERR;

     // validate the add node command
     TpNrValidations tpNrValidations = TpNrValidations.getInstance();
     if ( ((errorReason = tpNrValidations.noActiveReconfigs(pgEntity))
                                          == null) &&
          ((errorReason = tpNrValidations.isPgBlsr(pgEntity))
                                          == null) &&
          ((errorReason = tpNrValidations.nodeMapIsNil(pgEntity))
                                          == null) )
     {
         // execute the add node command
         if ( ((TpNrReconfigNode.getInstance()).
                         setForcePassThru(pgEntity, true)) &&
              ((TpNrConnMngmt.getInstance()).deleteAllXcons(pgEntity)) )
         {
             rc = true;
         }
         else
         {
             errorReason = TpNrErrorCode.TP_NR_ERR_SWERR;
         }
     }

     // if the command has failed, then generate the failure log
     if (!rc)
     {
         String operation = new String ("Add Node");
         String unit = (TpNrReconfigNode.getInstance()).
                                   genUnitString(pgEntity);

         (TpNrUtilities.getInstance()).log (operation, unit,
                                            errorReason, false);
     }

     return rc;
 }
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>backout</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method is invoked when the TpNrRcDbnsSubscriber.notifyWithResponse()
 * method is called with a request to pre-validate setting the reconfig
 * state PG attribute to idle.</P>
 *
 * <P>Setting the reconfig state attribute back to idle indicates the add
 * node command is being backed out.</P>
 *
 * @param      pgEntity
 *              - the entity in the database upon which the add node
 *                operation has been requested.
 *
 * @return      boolean
 *              - TRUE if add node backout successful, FALSE otherwise
 */
/*
 * Outputs:     None
 *
 */

public boolean backout (EntityPg pgEntity)
{
    boolean rc = false;

    // clear the add node command
    if ( ((TpNrReconfigNode.getInstance()).
                    setForcePassThru(pgEntity, false)) &&
         ((TpNrConnMngmt.getInstance()).deleteAllXcons(pgEntity)) )
    {
        rc = true;
    }

    return rc;
}
```

```
    /**
     *
     * <P><B>Method:</B>       <CODE>complete</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method is invoked when the TpNrNmDbnsSubscriber.notifyWithResponse()
     * method is called with a request to pre-validate setting the nodemap PG
     * attribute.</P>
     *
     * <P>Providing the nodemap has been set to a non-nil value it indicates the
     * completion of the add node procedure, and this method clears out the add
     * node information.</P>
     *
     * <P>NOTE: The deletion of the pass through connections on the working
     * channels is owned by Connection Management software, and not included in
     * this method.</P>
     *
     *
     * @param       pgEntity
     *              - the entity in the database upon which the add node
     *                operation has been requested.
     *
     * @return      boolean
     *              - TRUE if add node completion successful, FALSE otherwise
     */
    /*
     * Outputs:     None
     *
   --*/

    public boolean complete (EntityPg pgEntity)
    {
        boolean rc = false;

        if ((TpNrValidations.getInstance()).nodeMapIsNil(pgEntity) != null)
        {
            // clear add node
            if ( ((TpNrUtilities.getInstance()).setRecState(pgEntity,
                                  TPReconfigStateEnum.TP_RC_STATE_IDLE)) &&
                 ((TpNrReconfigNode.getInstance()).
                                  setForcePassThru(pgEntity, false)) )
            {
                rc = true;
            }
        }
        else
        {
            // if the node map is nil, then do nothing, the add node
            // procedure has not been finished
            rc = true;
        }

        return rc;
    }

} // end of TpNrAddNode
```

Class: TpNrDeleteNode

```
/**
 * <B>File:</B> <CODE>TpNrDeleteNode.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrErrorCode;
import equinox.protection.traffic.reconfig.TpNrValidations;
import equinox.protection.traffic.reconfig.TpNrUtilities;
import equinox.protection.traffic.reconfig.TpNrConnMngmt;
import equinox.protection.traffic.reconfig.TpNrReconfigNode;

import equinox.framework.nom.objs.EntityPg;

import equinox.framework.swerr.Swerr;

import equinox.framework.nom.enums.AF_ProblemTypeEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;


/**
 * <P><B>Class:</B> <CODE>TpNrDeleteNode</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>     - Product Development
 * <BR>       - Software
 * <BR>         - Network Element Software
 * <BR>           - Functional Areas
 * <BR>             - Network Reconfig
 * <BR>               - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrDeleteNode class definition encapsulates all the high level
 * processing required for a Delete Node operation.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * boolean process(...)
 * - processing required when an delete node operation is requested
 *
 * boolean backout(...)
 * - processing involved when delete node is backed out
 *
 * boolean complete(...)
 * - processing involved in completing the delete node operation
 *
 * boolean setNodeMap(...)
 * - sets the node map attribute of the PG to nil
 *
 */

public final class TpNrDeleteNode
{
    /**
     * singleton instance of this class
     */
     private static TpNrDeleteNode INSTANCE = null;
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>getInstance</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method returns the instance of this singleton.</P>
 *
 * @return      TpNrDeleteNode
 *               - the singleton instance of TpNrDeleteNode
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public static synchronized TpNrDeleteNode getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new TpNrDeleteNode();
    }

    return INSTANCE;
}

/**
 * Constructor, made private as class is singleton
 */
private TpNrDeleteNode ()
{
}
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>process</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method is invoked when the TpNrRcDbnsSubscriber.notifyWithResponse()
 * method is called with a request to pre-validate setting the reconfig
 * state PG attribute to delete node.</P>
 *
 * <P>This method then performs the required Delete Node validations and
 * processing, returning true if successful, false otherwise.</P>
 *
 * <P>NOTE: The creation of the pass through connections on the working
 * channels is owned by Connection Management software, and not included in
 * this method.</P>
 *
 * @param       pgEntity
 *              - the entity in the database upon which the delete node
 *                operation has been requested.
 *
 * @return      boolean
 *              - TRUE if delete node command successful, FALSE otherwise
 *
 * Outputs:     None
 *
 */

public boolean process (EntityPg pgEntity)
{
    boolean rc = false;
    String errorReason = TpNrErrorCode.TP_NR_ERR_SWERR;

    // validate the delete node command
    TpNrValidations tpNrValidations = TpNrValidations.getInstance();
    if ( ((errorReason = tpNrValidations.noActiveReconfigs(pgEntity))
                                   == null) &&
         ((errorReason = tpNrValidations.isPgBlsr(pgEntity))
                                   == null) &&
         ((errorReason = tpNrValidations.noAddDrops(pgEntity))
                                   == null) &&
         ((errorReason = tpNrValidations.noProtSwActive(pgEntity))
                                   == null) )
    {
        if (tpNrValidations.nodeMapIsNil(pgEntity) == null)
        {
            errorReason = TpNrErrorCode.TP_NR_ERR_NOT_RING;
        }
        else
        {
            // execute the delete node command
            if ( (setNodeMap(pgEntity)) &&
                 ((TpNrReconfigNode.getInstance()).
                     setForcePassThru(pgEntity, true)) )
            {
                rc = true;
            }
            else
            {
                errorReason = TpNrErrorCode.TP_NR_ERR_SWERR;
            }
        }
    }

    // if the command has failed, then generate the failure log
    if (!rc)
    {
        String operation = new String ("Delete Node");
        String unit = (TpNrReconfigNode.getInstance()).
                                 genUnitString(pgEntity);

        (TpNrUtilities.getInstance()).log (operation, unit,
                                           errorReason, false);
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>complete</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method is invoked when the TpNrRcDbnsSubscriber.notifyWithResponse()
 * method is called with a request to pre-validate setting the reconfig
 * state PG attribute to idle.</P>
 *
 * <P>Setting the reconfig state attribute back to idle indicates the delete
 * node command is being completed.</P>
 *
 * @param       pgEntity
 *              - the entity in the database upon which the delete node
 *                operation has been requested.
 *
 * @param       boolean
 *              - TRUE if delete node completion successful, FALSE otherwise
 */
/*
 * Outputs:     None
 *
 */

public boolean complete (EntityPg pgEntity)
{
    boolean rc = false;

    // clear the add node command
    if ( ((TpNrReconfigNode.getInstance()).
                setForcePassThru(pgEntity, false)) &&
         ((TpNrConnMngmt.getInstance()).deleteAllXcons(pgEntity)) )
    {
        rc = true;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>backout</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method is invoked when the TpNrNmDbnsSubscriber.notifyWithResponse()
 * method is called with a request to pre-validate setting the nodemap PG
 * attribute.</P>
 *
 * <P>Providing the nodemap has been set to a non-nil value it indicates the
 * backing out of the delete node procedure, and this method clears out the
 * delete node information.</P>
 *
 * <P>NOTE: The deletion of the pass through connections on the working
 * channels is owned by Connection Management software, and not included in
 * this method.</P>
 *
 *
 * @param       pgEntity
 *              - the entity in the database upon which the delete node
 *                operation has been requested.
 *
 * @return      boolean
 *              - TRUE if delete node backout successful, FALSE otherwise
 */
/*
 * Outputs:     None
 *
 */

public boolean backout (EntityPg pgEntity)
{
    boolean rc = false;

    if ((TpNrValidations.getInstance()).nodeMapIsNil(pgEntity) != null)
    {
        // clear add node
        if ( ((TpNrUtilities.getInstance()).setRecState(pgEntity,
                          TPReconfigStateEnum.TP_RC_STATE_IDLE)) &&
             ((TpNrReconfigNode.getInstance()).setForcePassThru(pgEntity,
                                                              false)) )
        {
            rc = true;
        }
    }
    else
    {
        // if the node map is nil, then do nothing, the delete node
        // procedure has not been finished
        rc = true;
    }

    return rc;
}
```

```
    /**
     *
     * <P><B>Method:</B>      <CODE>setNodeMap</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method sets the node map attribute of the PG to nil.</P>
     *
     *
     * @param       pgEntity
     *              - the PG upon which the reconfig operation has been executed
     *
     * @return      boolean
     *              - true if successful, false otherwise
     */
    /*
     * Outputs:     None
     *
     */

    private static boolean setNodeMap (EntityPg pgEntity)
    {
        boolean rc = true;

        byte[] nodemap = new byte[EntityPg.SIZE_nodemap];

        // initialise the nodemap array to the default value (32)
        for (int index = 0; index < EntityPg.SIZE_nodemap; index++)
        {
            nodemap[index] = 32;
        }

        // initialise the ieeemap value to ""
        String ieeemap = new String("");

        // attempt to set the nodemap/ieeemap values to their defaults
        try
        {
            pgEntity.setNodemap(nodemap);
            pgEntity.setIeeemap(ieeemap);
        }
        catch (Exception e)
        {
            rc = false;
            new Swerr(e.toString());
        }

        return rc;
    }

} // end of TpNrDeleteNode
```

Class: TpNrConnMngmt

```
/**
 * <B>File:</B> <CODE>TpNrConnMngmt.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.framework.database.DbAccess;
import equinox.framework.database.DbTransaction;

import equinox.framework.nom.objs.EntityXcon;
import equinox.framework.nom.objs.EntityPg;

import equinox.framework.nom.keys.EntityXconKey;

import equinox_ne_connections.engine.user_interface.Xconlist;
import equinox_ne_connections.engine.user_interface.Xcon;
import equinox_ne_connections.engine.user_interface.CmUserInterface;
import equinox_ne_connections.engine.user_interface.Cmuidb;

import equinox.framework.swerr.Swerr;

import java.util.HashSet;
import java.util.Iterator;
import java.util.ArrayList;

/**
 * <P><B>Class:</B> <CODE>TpNrConnMngmt</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *      - available in Athena, following this path:
 * <BR>  - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrConnMngmt class definition encapsulates the functionality used to
 * manipulate Xcons on the NE associated with the reconfig commands.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * Hashset getDeletedList()
 * - gets the list of xcons deleted while processing add/delete node
 *
 * void clearDeletedList()
 * - clears the list of deleted xcons
 *
 * boolean deleteAllXcons(...)
 * - deletes all the xcons associated with a specific PG
 *
 * void restoreXcons(...)
 * - creates Xcons passed in
 *
 *
 */

public final class TpNrConnMngmt
{
    /**
     * singleton instance of this class
     */
    private static TpNrConnMngmt INSTANCE = null;
```

```
/**
 * list of connections that have been deleted which
 * is used to restore the NE in case of validation
 * failure when processing the rcstate change
 */
private static ArrayList xconDelList = null;


 /**
  *
  * <P><B>Method:</B>      <CODE>getDeletedList</CODE>
  *
  * <BR><B>Description:</B>
  *
  * This method returns the list of deleted xcons that are to be
  * restored in case of the rcstate change being failed.</P>
  *
  * @return     ArrayList
  *              - list of deleted xcons
  */
 /*
  * Inputs:     None
  *
  * Outputs:    None
  *
  */

public ArrayList getDeletedList()
{
    return xconDelList;
}


 /**
  *
  * <P><B>Method:</B>      <CODE>clearDeletedList</CODE>
  *
  * <BR><B>Description:</B>
  *
  * This method clears the list of deleted xcons .</P>
  *
  */
 /* Returns:    None
  *
  * Inputs:     None
  *
  * Outputs:    None
  *
  */

public void clearDeletedList()
{
    if (xconDelList != null)
    {
        xconDelList.clear();
        xconDelList = null;
    }
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>getInstance</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method returns the instance of this singleton.</P>
 *
 * @return      TpNrConnMngmt
 *                - the singleton instance of TpNrConnMngmt
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public static synchronized TpNrConnMngmt getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new TpNrConnMngmt();
    }

    return INSTANCE;
}

/**
 * Constructor, made private as class is singleton
 */
private TpNrConnMngmt ()
{
}
```

```
/**
 *
 * <P><B>Method:</B>        <CODE>deleteAllXcons</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method obtains a list of all Xcons for the PGMs involved in the
 * reconfig, and deletes them all.</P>
 *
 * <P>NOTE: This method is called from within a DBNS Pre-Validation
 * event.</P>
 *
 *
 * @param        pgEntity
 *               - PG Entity upon which the reconfig operation has been
 *                 executed
 *
 * @return       boolean
 *               - true if successful, false otherwise
 */
/*
 * Outputs:      None
 *
 */

public boolean deleteAllXcons (EntityPg pgEntity)
{
    boolean rc = true;
    DbTransaction nrTxn  = null;
    DbTransaction curTxn = null;
    xconDelList = new ArrayList();

    try
    {
        // get the current transaction
        curTxn = DbTransaction.getCurrentDbTransaction();
        curTxn.leave();

        HashSet xconList = Xconlist.getXconsOnPg(pgEntity);

        if (!xconList.isEmpty())
        {
            // loop through all xcons
            Iterator i = xconList.iterator();
            while (i.hasNext())
            {
                // open a new transaction for each xcon delete
                nrTxn = new DbTransaction();
                nrTxn.setType( DbTransaction.READ_WRITE );
                nrTxn.begin();

                // retrieve the memory copy of the xcon from the list
                Xcon xc = (Xcon)i.next();

                // create a key
                EntityXconKey xconKey = new EntityXconKey(xc.getXcId());

                // retrieve the real entity from the database
                EntityXcon xcon = (EntityXcon)DbAccess.queryByKey(xconKey);

                // get the user label from the database to be used when
                // generating the AO Log
                String xconLabel = new String(xcon.getUserLabel());

                // delete the entity
                DbAccess.delete(xcon);

                nrTxn.validate();
                nrTxn.commit();
                nrTxn = null;

                // delete entry in xconlist
                Xconlist.delXconFromList(xc);

                // generate AO Log for MP to indicate deletion
                int deleteXcon = 1;
```

```
                    (CmUserInterface.getInstance()).logXCAction(deleteXcon,
                                                                xconLabel,
                                                                xc);

                    // add deleted xcon to list for later restoration in
                    // case of validation failure
                    // NOTE: ordering of added objects important, and assumed
                    //        in restoreXcons call
                    xconDelList.add(xc);
                    xconDelList.add(xconLabel);
                }
            }
        }
        catch (Exception e)
        {
            new Swerr(e.toString());
            rc = false;
        }
        finally
        {
            try
            {
                if (nrTxn != null)
                {
                    nrTxn.abort();
                }

                if (curTxn != null)
                {
                    // rejoin transaction
                    curTxn.join();
                }

            }
            catch (Exception e)
            {
                new Swerr(e.toString());
            }

            return rc;
        }
    }
    /**
     *
     * <P><B>Method:</B>       <CODE>restoreXcons</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method restores the list of xcons passed in.</P>
     *
     * @param       ArrayList xconList
     *              - list of xcons to be restored
     */
    /*
     * Inputs:     None
     *
     * Outputs:    None
     *
     */

    public void restoreXcons(ArrayList xconList)
    {

        boolean success = true;

        try
        {
            if (!xconList.isEmpty())
            {
                // loop through all xcons
                Iterator i = xconList.iterator();
                while (i.hasNext() && success)
                {
                    // retrieve the xcon from the list
                    Xcon xc = (Xcon)i.next();

                    // get the user label for generating the AO Log
                    String xconLabel = new String((String)i.next());

                    // insert the xcon into the database
                    Cmuidb.insertXcObject(xc,xconLabel);

                    // add entry to xconlist
                    if (Xconlist.addXconToList(xc))
                    {
                        // generate AO Log for MP to indicate creation
                        int createXcon = 0;
                        (CmUserInterface.getInstance()).logXCAction(createXcon,
                                                                    xconLabel,
                                                                    xc);

                    }
                    else
                    {
                        success = false;
                    }
                }
            }
        }
        catch (Exception e)
        {
            new Swerr(e.toString());
        }
        finally
        {
            if (!success)
            {
                new Swerr("Could not create xcon");
            }
        }
    }

} // end of TpNrConnMngmt
```

Class: TpNrErrorCode

```java
/**
 * <B>File:</B> <CODE>TpNrErrorCode.java</CODE>
 */

package equinox.protection.traffic.reconfig;


/**
 * <P><B>Class:</B> <CODE>TpNrErrorCode</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrErrorCode class definition encapsulates all the network reconfig
 * error strings that are used to inform the user the reason the reconfig
 * operation failed
 *
 * @author Andy Kinney
 *
 */

public final class TpNrErrorCode
{
   /**
    * Indicates a reconfig operation was requested whilst the node was
    * already executing an add node operation
    */
    public static final String TP_NR_ERR_IN_ADD_NODE = new String (
      "Command is not valid while node is in Add Node mode");

   /**
    * Indicates a reconfig operation was requested whilst the node was
    * already executing a delete node operation
    */
    public static final String TP_NR_ERR_IN_DEL_NODE = new String (
      "Command is not valid while node is in Delete Node mode");

   /**
    * Indicates a reconfig operation was requested whilst the node was
    * already configured in a ring
    */
    public static final String TP_NR_ERR_VALID_NODE_MAP = new String (
      "Node is configured in a network");

   /**
    * Indicates a reconfig operation was requested whilst the node has
    * an active protection switch
    */
    public static final String TP_NR_ERR_PROT_SW_ACTIVE = new String (
      "Node is currently handling a protection switch");

   /**
    * Indicates a reconfig operation was requested whilst the node has
    * non-passthrough connections provisioned
    */
    public static final String TP_NR_ERR_ADD_DROP = new String (
      "Traffic connections are currently provisioned");
```

```
/**
 * Indicates a reconfig operation was requested whilst the node has
 * an invalid protection scheme
 */
public static final String TP_NR_ERR_PROT_SCHEME_INVAL = new String (
  "Protection scheme is not valid");

/**
 * Indicates a software error occured during the execution of the
 * requested reconfig operation
 */
public static final String TP_NR_ERR_SWERR = new String (
  "Software Error");

/**
 * Indicates a reconfig operation was requested whilst the node was
 * not configured into a network
 */
public static final String TP_NR_ERR_NOT_RING = new String (
  "Node is not configured in a network");

/**
 * Indicates that the PGM objects could not be locked to the transaction
 * for the requested reconfig operation
 */
public static final String TP_NR_ERR_LOCK_NOT_GRANTED_PGM = new String (
  "Could not lock all required PGM objects for this operation");

/**
 * Indicates that a request to create a PG in add node mode was received
 */
public static final String TP_NR_ERR_CREATE_PG_ADD = new String (
  "Command to create PG in Add Node mode is not valid");

/**
 * Indicates that a request to create a PG in delete node mode was received
 */
public static final String TP_NR_ERR_CREATE_PG_DEL = new String (
  "Command to create PG in Delete Node mode is not valid");

/**
 * Indicates that the TL1 request was an invalid format
 */
public static final String TP_NR_ERR_INVALID_TL1 = new String (
  "TL1 command contained more than reconfig state attribute");

/**
 * Indicates that a request to change the state from Idle to Idle has
 * been received, and denied.
 */
public static final String TP_NR_ERR_IDLE_TO_IDLE = new String (
  "TL1 command requested Reconfig State change from Idle to Idle");


/**
 * singleton instance of this class
 */
private static TpNrErrorCode INSTANCE = null;
```

```
    /**
     *
     * <P><B>Method:</B>       <CODE>getInstance</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the instance of this singleton.</P>
     *
     * @return     TpNrErrorCode
     *              - the singleton instance of TpNrErrorCode
     */
    /*
     * Inputs:     None
     *
     * Outputs:    None
     *
     */

    public static synchronized TpNrErrorCode getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNrErrorCode();
        }

        return INSTANCE;
    }

   /**
    * Constructor, made private as class is singleton
    */
    private TpNrErrorCode ()
    {
    }

} // end of TpNtErrorCode
```

Class: TpNrNmDbnsSubscriber

```
/**
 * <B>File:</B> <CODE>TpNrNmDbnsSubscriber.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrAddNode;
import equinox.protection.traffic.reconfig.TpNrDeleteNode;
import equinox.protection.traffic.reconfig.TpNrValidations;

import equinox.framework.database.dbns.DbnsSubscriber;
import equinox.framework.database.dbns.DbnsEvent;
import equinox.framework.database.dbns.DbnsSingleEvent;

import equinox.framework.nom.keys.EntityPgKey;

import equinox.framework.nom.objs.EntityPg;

import equinox.framework.nom.enums.EntityIdEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;

import equinox.framework.swerr.Swerr;


/**
 * <P><B>Class:</B> <CODE>TpNrNmDbnsSubscriber</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *      - available in Athena, following this path:
 * <BR>  - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * This singleton class implements the DbnsSubscriber interface, which provides
 * the means by which DBNS informs interested parties of events that have
 * been previously subscribed to.
 *
 * This implementation performs the necessary processing when the PG node
 * map attribute is updated.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * TpNrRcDbnsSubscriber getInstance()
 * - returns the instance of this object
 *
 * boolean dbAccessOption()
 * - returns true to indicate that the subscriber will be accessing the
 *   database
 *
 * boolean singleNotificationOption()
 * - returns false to indicate that this subscriber is registering for multiple
 *   single events, rather than one combined event
 *
 * void notify(...)
 * - method called to deal with the commit of the node map attribute change
 *
 * boolean notifyWithResponse(...)
 * - method called to handle the pre-validation of the node map attribute
 *   change
 *
 */
```

```
public final class TpNrNmDbnsSubscriber implements DbnsSubscriber
{
   /**
    * singleton instance of this class
    */
    private static TpNrNmDbnsSubscriber INSTANCE = null;

    /**
     *
     * <P><B>Method:</B>      <CODE>getInstance</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the instance of this singleton.</P>
     *
     * @return      TpNrNmDbnsSubscriber
     *               - the singleton instance of TpNrNmDbnsSubscriber
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
     */

    public static synchronized TpNrNmDbnsSubscriber getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNrNmDbnsSubscriber();
        }

        return INSTANCE;
    }

   /**
    * Constructor, made private as class is singleton
    */
    private TpNrNmDbnsSubscriber()
    {
    }

    /**
     *
     * <P><B>Method:</B>      <CODE>dbAccessOption</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method determines whether the subscriber will access the
     * database via the notify, or notifyWithResponse methods. In this
     * case the method always returns true to indicate that the
     * subscriber will be accessing the database.</P>
     *
     *
     * @return       boolean
     *               - true to indicate access to database required
     */
    /*
     * Inputs:       None
     *
     * Outputs:      None
     *
     */

    public boolean dbAccessOption()
    {
        return true;
    }
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>singleNotificationOption</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method indicates whether the subscriber will handle multiple single
 * events or one combined event. In this case the method will always return
 * false to indicate multiple single events.</P>
 *
 *
 * @return      boolean
 *              - false to indicate multiple single event
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public boolean singleNotificationOption()
{
    return false;
}
```

```
/**
 * <P><B>Method:</B>       <CODE>notify</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method performs the commit processing on receipt of a request to
 * change the node map attribute of a PG Entity.</P>
 *
 * <P>It checks that the notification is for a commit event, and for a PG
 * Entity,before it then retrieves the reconfig state attribute value. This
 * value should be Idle, as this method being called indicates that either
 * Add Node has been completed, or Delete Node has been backed out, in
 * either case the alarm should be cleared, and the node map set event
 * should be unsubscribed from.</P>
 *
 * @param        e
 *               - node map event
 *
 * Outputs:      None
 *
 * Returns:      None
--*/

public void notify(DbnsEvent e)
{
    DbnsSingleEvent event = (DbnsSingleEvent)e;

    try
    {
        if ( (event.getEventType() == DbnsEvent.DBNS_COMMIT_EVENT_TYPE) &&
             (event.getEntity().getKey().getEntityId() ==
                                              EntityIdEnum.EN_PG) )
        {
            EntityPg pgEntity = (EntityPg)event.getEntity();
            int recState = pgEntity.getRcstate().intValue();

            switch (recState)
            {
                case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
                    // the TpNrRcDbnsSubscriber.notify() method handles the
                    // clearing of the alarm and unsubscription from the
                    // node map set event
                    // i.e. nothing is required here
                    break;

                case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                    // check to make sure that the node map is nil
                    if ((TpNrValidations.getInstance()).
                                nodeMapIsNil(pgEntity) == null)
                    {
                        // if it is nil then the reconfig is in the correct
                        // state
                        break;
                    }
                    // since the nodemap is non-nil, and in delete node
                    // drop through to default case to raise swerr
                case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                    // should not be in add node when this method is called
                    // drop through to default to raise swerr
                default:
                    new Swerr("Invalid reconfig state : " + recState);
                    break;
            }
        }
        else
        {
            new Swerr("Invalid event type received : " +
                      event.getEventType());
            new Swerr("Invalid entity received : " +
                      event.getEntity().getKey().getEntityId());
        }
    }
    catch (Exception ex)
    {
        new Swerr(ex.getMessage());
    }
}
```

```
    /**
     * <P><B>Method:</B>      <CODE>notifyWithResponse</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method performs the pre-validation processing on receipt of a
     * request to change the node map attribute of a PG Entity, that is
     * currently in add node or delete node mode.</P>
     *
     * <P>It checks that the notification is for a pre-validation event, and
     * for a PG Entity, before it then retrieves the reconfig state attribute
     * to determine what operation has been performed.</P>
     *
     * <P>NOTE: In the case of the reconfig state being Add Node, Add Node is
     * being completed, and for Delete Node, it is being backed out.</P>
     *
     *
     * @param       e
     *              - nod map event
     *
     * Outputs:     None
     *
     * Returns:     boolean
     *              - true if successful, false otherwise
    --*/

    public boolean notifyWithResponse(DbnsEvent e)
    {
        DbnsSingleEvent event = (DbnsSingleEvent)e;
        boolean rc = false;

        try
        {
            if ( (event.getEventType() ==
                        DbnsEvent.DBNS_PRE_VALIDATION_EVENT_TYPE) &&
                 (event.getEntity().getKey().getEntityId() ==
                                        EntityIdEnum.EN_PG) )
            {
                EntityPg pgEntity = (EntityPg)event.getEntity();
                int recState = pgEntity.getRcstate().intValue();

                switch (recState)
                {
                    case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                        rc = (TpNrAddNode.getInstance()).complete(pgEntity);
                        break;

                    case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                        rc = (TpNrDeleteNode.getInstance()).backout(pgEntity);
                        break;

                    case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
                        // do nothing
                        rc = true;
                        break;

                    default:
                        new Swerr("Invalid reconfig state : " + recState);
                        break;
                }
            }
            else
            {
                new Swerr("Invalid event type received : " +
                          event.getEventType());
                new Swerr("Invalid entity received : " +
                          event.getEntity().getKey().getEntityId());
            }

        }
        catch (Exception ex)
        {
            new Swerr(ex.getMessage());
        }
        return rc;
    }
} // end of TpNrNmDbnsSubscriber
```

Class: TpNrRcDbnsSubsriber

```
/**
 * <B>File:</B> <CODE>TpNrRcDbnsSubscriber.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrAddNode;
import equinox.protection.traffic.reconfig.TpNrDeleteNode;
import equinox.protection.traffic.reconfig.TpNrReconfigNode;
import equinox.protection.traffic.reconfig.TpNrErrorCode;
import equinox.protection.traffic.reconfig.TpNrUtilities;
import equinox.protection.traffic.reconfig.TpNrConnMngmt;

import equinox.framework.database.DbTransaction;

import equinox.framework.database.dbns.DbnsSubscriber;
import equinox.framework.database.dbns.DbnsEvent;
import equinox.framework.database.dbns.DbnsSingleEvent;
import equinox.framework.database.dbns.Operation;

import equinox.framework.nom.keys.EntityPgKey;

import equinox.framework.nom.objs.EntityPg;

import equinox.framework.nom.enums.EntityIdEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;
import equinox.framework.nom.enums.AF_ProblemTypeEnum;

import java.util.Vector;
import java.util.ArrayList;

import equinox.framework.swerr.Swerr;
```

```
/**
 * <P><B>Class:</B> <CODE>TpNrRcDbnsSubscriber</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * This singleton class implements the DbnsSubscriber interface, which provides
 * the means by which DBNS informs interested parties of events that have
 * been previously subscribed to.
 *
 * This implementation performs the necessary processing when the PG reconfig
 * state attribute is updated. During the system initialisation, standard
 * reconfigs subscribes to the pre-validation and commit events, that
 * specifically reference the reconfig state attribute.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * TpNrRcDbnsSubscriber getInstance()
 * - returns the instance of this object
 *
 * boolean dbAccessOption()
 * - returns true to indicate that the subscriber will be accessing the
 *   database
 *
 * boolean singleNotificationOption()
 * - returns false to indicate that this subscriber is registering for multiple
 *   single events, rather than one combined event
 *
 * void notify(...)
 * - method called to deal with the commit of the reconfig state attribute
 *   change
 *
 * boolean notifyWithResponse(...)
 * - method called to handle the pre-validation of the reconfig state attribute
 *   change
 *
 */

public final class TpNrRcDbnsSubscriber implements DbnsSubscriber
{
    /**
     * singleton instance of this class
     */
    private static TpNrRcDbnsSubscriber INSTANCE = null;
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>getInstance</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method returns the instance of this singleton.</P>
 *
 * @return      TpNrRcDbnsSubscriber
 *               - the singleton instance of TpNrRcDbnsSubscriber
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public static synchronized TpNrRcDbnsSubscriber getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new TpNrRcDbnsSubscriber();
    }

    return INSTANCE;
}

/**
 * Constructor, made private as class is singleton
 */
private TpNrRcDbnsSubscriber()
{
}

/**
 *
 * <P><B>Method:</B>       <CODE>dbAccessOption</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method determines whether the subscriber will access the
 * database via the notify, or notifyWithResponse methods. In this
 * case the method always returns true to indicate that the
 * subscriber will be accessing the database.</P>
 *
 *
 * @return       boolean
 *               - true to indicate access to database required
 */
/*
 * Inputs:       None
 *
 * Outputs:      None
 *
 */

public boolean dbAccessOption()
{
    return true;
}
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>singleNotificationOption</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method indicates whether the subscriber will handle multiple single
 * events or one combined event. In this case the method will always return
 * false to indicate multiple single events.</P>
 *
 *
 * @return      boolean
 *              - false to indicate multiple single event
 */
/*
 * Inputs:      None
 *
 * Outputs:     None
 *
 */

public boolean singleNotificationOption()
{
    return false;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>notify</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method performs the abort/commit processing on receipt of a request
 * to change the reconfig state attribute of a PG Entity.</P>
 *
 * <P>It first checks whether the event is abort, in which case it attempts
 * to restore the list of deleted connections.</P>
 *
 * <P>It then checks that the notification is for a commit event, and for a
 * PG Entity,before it then retrieves the reconfig state attribute value to
 * determine what operation has been performed. This is then used to
 * generate the operation string for the user visible log, that is
 * generated after subscribing to the node map set event, and raising of
 * the 'ADM in single node configuration' alarm.</P>
 *
 * @param        e
 *               - reconfig state event
 */
/*
 * Outputs:      None
 *
 * Returns:      None
 *
 */

public void notify(DbnsEvent e)
{
    DbnsSingleEvent event = (DbnsSingleEvent)e;
    DbTransaction nrTx = null;

    try
    {
        // in case the validation of the rcstate change failed, then
        // attempt to restore any connections that were deleted
        if (event.getEventType() == DbnsEvent.DBNS_ABORT_EVENT_TYPE)
        {
            ArrayList xcResList = (TpNrConnMngmt.getInstance()).
                                  getDeletedList();

            // check to make sure the list has been instantiated
            if (xcResList != null)
            {
                // restore the deleted xcons
                // NOTE: SWERR is raised by the called method if the
                // restoration fails
                (TpNrConnMngmt.getInstance()).restoreXcons(xcResList);
            }

            return;
        }

        // handle the commit event
        if ( (event.getEventType() == DbnsEvent.DBNS_COMMIT_EVENT_TYPE) &&
             (event.getEntity().getKey().getEntityId() ==
                                    EntityIdEnum.EN_PG) )
        {
            EntityPg pgEntity = (EntityPg)event.getEntity();

            int recState = pgEntity.getRcstate().intValue();

            boolean required = false;

            String operation = null;

            switch (recState)
            {
                case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                    // set operation string to add node
                    operation = new String("Add Node");

                    required = true;
                    break;
                case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                    // set operation string to delete node
                    operation = new String("Delete Node");

                    required = true;
                    break;

                case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
                    // clear alarm
                    (TpNrUtilities.getInstance()).alarm(pgEntity,
                        AF_ProblemTypeEnum.AF_ADMINSINGLENODECONFIG_PT,
                                                false);
                    break;

                default:
                    new Swerr("Invalid reconfig state : " + recState);
                    break;
            }

            if (required)
            {
                // raise alarm
                (TpNrUtilities.getInstance()).alarm(pgEntity,
                            AF_ProblemTypeEnum.AF_ADMINSINGLENODECONFIG_PT,
                                                     true);

                String unit = (TpNrReconfigNode.getInstance()).
                                    genUnitString(pgEntity);

                 // generate the success (FAC616) log with dummy value for
                 // errorReason
                String errorReason = TpNrErrorCode.TP_NR_ERR_SWERR;
                (TpNrUtilities.getInstance()).log(operation, unit,
                                                   errorReason, true);

            }

            // clear the list of deleted Xcons, since the change to the
            // rcstate value has been committed successfully and there is
            // no need to now restore them
            (TpNrConnMngmt.getInstance()).clearDeletedList();
        }
        else
        {
            new Swerr("Invalid event type received : " +
                      event.getEventType());
            new Swerr("Invalid entity received : " +
                      event.getEntity().getKey().getEntityId());
        }
    }
    catch (Exception ex)
    {
        new Swerr(ex.getMessage());
    }
}
```

```java
/**
 *
 * <P><B>Method:</B>      <CODE>notifyWithResponse</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method performs the pre-validation processing on receipt of a
 * request to change the reconfig state attribute of a PG Entity.</P>
 *
 * <P>It checks that the notification is for a pre-validation event, and
 * for a PG Entity, before checking the value to which the reconfig state
 * attribute is to be set, upon which the specific processing for Add Node
 * or Delete Node is called.</P>
 *
 * <P>NOTE: In the case of the reconfig state being set to Idle, Add Node
 * is being backed out, and Delete Node is being completed. To determine
 * which,the previous value of the reconfig state attribute is
 * retrieved.</P>
 *
 *
 * @param       e
 *              - reconfig state event
 *
 * @return      boolean
 *              - true if successful, false otherwise
 */
/*
 * Outputs:     None
 *
 */

public boolean notifyWithResponse(DbnsEvent e)
{
    DbnsSingleEvent event = (DbnsSingleEvent)e;
    boolean rc = false;

    try
    {
        // check the event is for the correct type and entity
        if ( (event.getEventType() ==
               DbnsEvent.DBNS_PRE_VALIDATION_EVENT_TYPE) &&
              (event.getEntity().getKey().getEntityId() ==
                                  EntityIdEnum.EN_PG) )
        {
            EntityPg pgEntity = (EntityPg)event.getEntity();

            // get the operation type
            switch (event.getOperation())
            {
                // a PG create will be denied if the rcstate attribute is
                // set to anything other than IDLE
                case Operation.OP_INSERT:
                    rc = processInsert(pgEntity);
                    break;

                // a PG delete will be denied if the rcstate attribute
                // is set to anything other than IDLE
                case Operation.OP_DELETE:
                    rc = processDelete(pgEntity);
                    break;

                // in normal operation the rcstate attribute is updated
                // to enter add node or delete node
                case Operation.OP_UPDATE:
                    // check that this event only contains a change to the
                    // rcstate attribute
                    if (validChange(pgEntity))
                    {
                        // process the change to the rcstate attribute
                        rc = processChange(pgEntity);
                    }
                    break;

                default:
                    new Swerr("Invalid Operation" + event.getOperation());
                    break;
            }
        }
        else
        {
            new Swerr("Invalid event type received : " +
                      event.getEventType());
            new Swerr("Invalid entity received : " +
                      event.getEntity().getKey().getEntityId());
        }

    }
    catch (Exception ex)
    {
        new Swerr(ex.toString());
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>validChange</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method determines whether the rcstate attribute is the only
 * attribute on the PG being changed.</P>
 *
 * <P></P>
 *
 *
 * @param       pgEntity
 *              - the PG entity upon which the reconfig operation is being
 *                performed
 *
 * @return      boolean
 *              - true if only rcstate is being changed, false otherwise
 */
/*
 * Outputs:     None
 *
 */

private static boolean validChange (EntityPg pgEntity)
{
    boolean rc = false;

    try
    {
        // get a list of the changed attributes
        Vector changedAttr = pgEntity.getDelta();

        if (changedAttr.contains(EntityPg.ATTR_rcstate))
        {
            // check to make sure this list only contains one attribute,
            // i.e. the rcstate attribute
            if (changedAttr.size() != 1)
            {
                String errorReason = TpNrErrorCode.TP_NR_ERR_INVALID_TL1;
                (TpNrUtilities.getInstance()).log(pgEntity, errorReason,
                                                   false);
            }
            else
            {
                rc = true;
            }
        }
        else
        {
            new Swerr("Command did not contain rcstate");
        }
    }
    catch (Exception e)
    {
        new Swerr(e.toString());
    }

    return rc;
}
```

```
/**
 * <P><B>Method:</B>       <CODE>processChange</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method processes the change to the reconfig state attribute.</P>
 *
 * @param       pgEntity
 *              - the PG entity upon which the reconfig operation is being
 *                performed
 *
 * @return      boolean
 *              - true if the reconfig state change is successful,
 *                false otherwise
 *
 * Outputs:     None
 */

private static boolean processChange (EntityPg pgEntity)
{
    boolean rc = false;
    int recState = pgEntity.getRcstate().intValue();

    switch (recState)
    {
        case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
            rc = (TpNrAddNode.getInstance()).process(pgEntity);
            break;

        case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
            rc = (TpNrDeleteNode.getInstance()).process(pgEntity);
            break;

        case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
            // get the old PG entity for the previous reconfig
            // state value
            EntityPg oldPg = (EntityPg)pgEntity.getOldObj();
            int oldRcState = oldPg.getRcstate().intValue();

            switch (oldRcState)
            {
                case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                    // moving from add node to idle indicates
                    // that add node is being backed out
                    rc = (TpNrAddNode.getInstance()).backout(pgEntity);
                    break;

                case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                    // moving from delete node to idle indicates
                    // that delete node is being completed
                    rc = (TpNrDeleteNode.getInstance()).complete(pgEntity);
                    break;

                case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
                    // moving from idle to idle is harmless, but this
                    // should be denied
                    String errorReason = TpNrErrorCode.
                                        TP_NR_ERR_IDLE_TO_IDLE;
                    (TpNrUtilities.getInstance()).log(pgEntity,
                                         errorReason, false);
                    rc = false;
                    break;

                default:
                    new Swerr("Invalid old reconfig state : "
                                    + oldRcState);
                    break;
            }

            break;

        default:
            new Swerr("Invalid reconfig state : " + recState);
            break;
    }
    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>processInsert</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method processes the event coresponding to creating the
 * rcstate attribute</P>
 *
 * <P> Only an rcstate value of IDLE is acceptable when deleting
 * the PG. </P>
 *
 *
 * @param        pgEntity
 *               - the PG entity upon which the reconfig operation is being
 *                 performed
 *
 * @return       boolean
 *               - true if the reconfig state insert is successful,
 *                 false otherwise
 */
/*
 * Outputs:      None
 *
 */

private static boolean processInsert (EntityPg pgEntity)
{
    boolean rc = false;

    int recState = pgEntity.getRcstate().intValue();

    // only the IDLE value is acceptable when creating the PG
    if (recState != TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE)
    {
        String errorReason = null;

        switch (recState)
        {
            case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                errorReason = TpNrErrorCode.TP_NR_ERR_CREATE_PG_ADD;
                break;

            case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                errorReason = TpNrErrorCode.TP_NR_ERR_CREATE_PG_DEL;
                break;

            default:
                new Swerr("Invalid state :  " + recState);
                break;
        }

        if (errorReason != null)
        {
            (TpNrUtilities.getInstance()).log(pgEntity,
                                              errorReason, false);
        }
    }
    else
    {
        rc = true;
    }

    return rc;
}
```

```
    /**
     *
     * <P><B>Method:</B>       <CODE>processDelete</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method processes the event coresponding to deleting the
     * rcstate attribute</P>
     *
     * <P> Only an rcstate value of IDLE is acceptable when deleting
     * the PG. </P>
     *
     *
     * @param        pgEntity
     *               - the PG entity upon which the reconfig operation is being
     *                 performed
     *
     * @return       boolean
     *               - true if the reconfig state delete is successful,
     *                 false otherwise
     */
    /*
     * Outputs:     None
     *
     */

    private static boolean processDelete (EntityPg pgEntity)
    {
        boolean rc = false;

        int recState = pgEntity.getRcstate().intValue();

        if (recState != TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE)
        {
            String errorReason = null;

            switch (recState)
            {
                case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
                    errorReason = TpNrErrorCode.TP_NR_ERR_IN_ADD_NODE;
                    break;

                case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
                    errorReason = TpNrErrorCode.TP_NR_ERR_IN_DEL_NODE;
                    break;

                default:
                    new Swerr("Invalid state :  " + recState);
                    break;
            }

            if (errorReason != null)
            {
                (TpNrUtilities.getInstance()).log(pgEntity,
                                                  errorReason, false);
            }
        }
        else
        {
            rc = true;
        }

        return rc;
    }

} // end of TpNrRcDbnsSubscriber
```

Class: TpNrRestartRecovery

```
/**
 * <B>File:</B> <CODE>TpNrRestartRecovery.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.protection.traffic.reconfig.TpNrReconfigNode;


/**
 * <P><B>Class:</B> <CODE>TpNrRestartRecovery</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *      - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>    - Product Development
 * <BR>     - Software
 * <BR>       - Network Element Software
 * <BR>         - Functional Areas
 * <BR>           - Network Reconfig
 * <BR>             - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrRestartRecovery class implements Runnable and encapsulates the
 * methods to initialise the network reconfiguration code.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * void run()
 * - contains the network reconfig initialisation code
 *
 * String getName()
 * - returns the name associated with the this reconfig thread
 *
 */

public final class TpNrRestartRecovery implements Runnable
{
   /**
    * name used to identify this thread with equinox thread registry
    */
    private static final String name = new String("NR_RESTART_D");

   /**
    * Default Constructor
    */
    public TpNrRestartRecovery()
    {
    }
    /**
     *
     * <P><B>Method:</B>      <CODE>getName</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the name of the thread registered with the
     * equinox thread registry</P>
     *
     * @return      String
     *              - name of the equinox thread
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
   --*/

     public String getName ()
     {
         return this.name;
     }


    /**
     *
     * <P><B>Method:</B>      <CODE>run</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method contains the network reconfig initialisation code.</P>
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
     * Returns:     None
     *
     */

     public void run ()
     {
         // network reconfig initialisation code

         // add node and delete node initialisation
         TpNrReconfigNode tpNrReconfigNode = TpNrReconfigNode.getInstance();
         tpNrReconfigNode.initialize();
     }

} // end of TpNrRestartRecovery
```

Class: TpNrUtilities

```
/**
 * <B>File:</B> <CODE>TpNrUtilities.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.framework.database.DbAccess;
import equinox.framework.database.DbLockNotGrantedException;

import equinox.framework.nom.keys.EntityLogicalLineKey;
import equinox.framework.nom.keys.EntityPgKey;
import equinox.framework.nom.keys.EntityNeKey;

import equinox.framework.swerr.Swerr;

import equinox.framework.nom.objs.EntityPg;
import equinox.framework.nom.objs.EntityNe;
import equinox.framework.nom.objs.EntityPropertyList;

import equinox.framework.nom.enums.AF_ReturnCodeEnum;
import equinox.framework.nom.enums.AF_InputImpactEnum;
import equinox.framework.nom.enums.AF_ProblemTypeEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;
import equinox.framework.nom.enums.EntityIdEnum;

import equinox.framework.alarms.external.AlarmFramework;

import equinox.logs.userVisible.LogFac616;
import equinox.logs.userVisible.LogFac316;

import java.util.HashSet;
import java.util.Iterator;


/**
 * <P><B>Class:</B> <CODE>TpNrUtilities</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *       - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>    - Product Development
 * <BR>      - Software
 * <BR>        - Network Element Software
 * <BR>          - Functional Areas
 * <BR>            - Network Reconfig
 * <BR>              - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrUtilities class definition encapsulates the utilities to support
 * Network Reconfigs.
 *
 * @author Andy Kinney
 *
 * Methods:
 *
 * boolean setRecState(...)
 * - sets the reconfig state attribute
 *
 * void log()
 * - method to generate Network Reconfig success (FAC616) and failed (FAC316)
 *   user visible logs
 *
 * void alarm()
 * - method to raice/clear 'ADM in single node configuration' alarm
 *
 * HashSet getSetOfObjects(...)
 * - method to retry locking of objects in database to current transaction
 *
 *
 */
```

```
public final class TpNrUtilities
{
    /**
     * singleton instance of this class
     */
    private static TpNrUtilities INSTANCE = null;

     /**
      *
      * <P><B>Method:</B>      <CODE>getInstance</CODE>
      *
      * <BR><B>Description:</B>
      *
      * This method returns the instance of this singleton.</P>
      *
      * @return      TpNrUtilities
      *               - the singleton instance of TpNrUtilities
      */
     /*
      * Inputs:      None
      *
      * Outputs:     None
      *
      */

    public static synchronized TpNrUtilities getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNrUtilities();
        }

        return INSTANCE;
    }

    /**
     * Constructor, made private as class is singleton
     */
    private TpNrUtilities ()
    {
    }
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>setRecState</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method sets the reconfig state PG attribute.</P>
 *
 * @param       pgEntity
 *              - the PG upon which the reconfig operation has been
 *                executed
 *
 * @param       recState
 *              - the reconfig state that the attribute is to be set to.
 *
 * @return      boolean
 *              - true if successful, false otherwise
 */
/*
 * Outputs:     None
 *
 */

public boolean setRecState (EntityPg pgEntity,
                            TPReconfigStateEnum recState)
{
    boolean rc = true;

    try
    {
        pgEntity.setRcstate(recState);
    }
    catch (Exception e)
    {
        new Swerr(e.toString());
        rc = false;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>log</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method determines the unit and operation strings from the
 * pgEntity and then calls the log method to generate the
 * user visible log.</P>
 *
 *
 * @param       pgEntity
 *              - the PG entity upon which the reconfig operation is being
 *                performed
 *
 * @param       errorReason
 *              - the reason the reconfig command failed
 *
 * @param       success
 *              - true to generate success log, false for failed log.
 */
/*
 * Returns:     None
 *
 * Outputs:     None
 *
 */

public void log(EntityPg pgEntity, String errorReason, boolean success)
{
    int recState = pgEntity.getRcstate().intValue();

    String operation = null;
    String unit = null;

    switch (recState)
    {
        case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
            operation = new String("Add Node");
            unit = (TpNrReconfigNode.getInstance()).
                         genUnitString(pgEntity);
            break;

        case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
            operation = new String("Delete Node");
            unit = (TpNrReconfigNode.getInstance()).
                         genUnitString(pgEntity);
            break;

        case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
            // it is important that a log is generated even when the state
            // is idle to ensure that the errorReason information is
            // displayed to the user
            operation = new String("Reconfig State - Idle");
            unit = (TpNrReconfigNode.getInstance()).
                         genUnitString(pgEntity);
            break;

        default:
            new Swerr("Cannot determine log type : " + recState);
            break;
    }

    // check that the unit and operation strings have been initialised and
    // generate the appropriate network reconfig log
    if ((operation != null) && (unit != null))
    {
        (TpNrUtilities.getInstance()).log (operation, unit,
                                           errorReason, success);
    }

}
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>log</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method generates either the success (FAC616) or failed (FAC316)
 * Network Reconfig user visible log.</P>
 *
 *
 * @param       unit
 *              - the string associated with the unit upon which the reconfig
 *                operation has been performed
 *
 * @param       operation
 *              - the reconfig operation e.g. Add Node
 *
 * @param       errorReason
 *              - the error reason in case of failure
 *
 * @param       success
 *              - true to generate success log, false for failed log.
 */
/*
 * Outputs:     None
 *
 * Returns:     None
 *
 */

 public void log (String operation,
                  String unit,
                  String errorReason,
                  boolean success)
 {
     try
     {
         // check whether success (FAC616) or failed (FAC316) log is to be
         // created
         if (success)
         {
             // generate success log
             LogFac616 log = new LogFac616 (operation, unit);

             if (log.generateLog() == -1)
             {
                 // NOTE: since this is a user visible log, it is not
                 // necessary to retry on failure
                 new Swerr("Failed to generate Network Reconfig
                                  success log");
             }
         }
         else
         {
             // generate failed log
             LogFac316 log = new LogFac316 (operation, unit, errorReason);

             if (log.generateLog() == -1)
             {
                 // NOTE: since this is a user visible log, it is not
                 // necessary to retry on failure
                 new Swerr("Failed to generate Network Reconfig failed log");
             }
         }
     }
     catch (Exception e)
     {
         new Swerr(e.toString());
     }
 }
```

```
/**
 *
 * <P><B>Method:</B>      <CODE>alarm</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method provides the mechanism to raise/clear the 'ADM in single
 * node configuration alarm'</P>
 *
 *
 * @param       pgEntity
 *              - PG entity upon which the alarm is to be raised/cleared
 *
 * @param       problemType
 *              - the alarm type to be raised
 *
 * @param       raise
 *              - true to raise alarm, false to clear
 */
/*
 * Outputs:     None
 *
 * Returns:     None
 *
--*/

public void alarm(EntityPg pgEntity,
                  AF_ProblemTypeEnum problemType,
                  boolean raise)
{
    try
    {
        // get PG key from entity
        EntityPgKey pgKey = (EntityPgKey)pgEntity.getKey();

        // create LogicalLineKey to raise/clear alarm against
        short dummy = 1;
        EntityLogicalLineKey llKey = new EntityLogicalLineKey (
                                          pgKey.getShelfId(),
                                          pgKey.getSlotId(),
                                          pgKey.getPortId(),
                                          dummy,
                                          pgKey.getPipeId());

        // setup secondary key for interface with alarm framework
        // which isn't used in the reconfig case, as it only applies
        // for inter/intra shelf alarms
        EntityPgKey secondKey = null;

        // call alarm framework interface to raise/clear the
        // 'ADM in single node configuration' alarm
        AF_ReturnCodeEnum rc = AlarmFramework.notifyProblem(
            problemType,
            llKey,
            secondKey,
            raise,
            AF_InputImpactEnum.AF_NA_II);

        // NOTE: AF_NA_II is used to indicate the default severity of m,nsa
        // since the reconfig alarm has no other setting

        if (rc != AF_ReturnCodeEnum.AF_SUCCESS_RC)
        {
            new Swerr(rc.toString());
        }

        // NOTE: considering where this method is called, there is no
        // benefit in returning success/failure, hence swerring is
        // sufficient
    }
    catch (Exception e)
    {
        new Swerr (e.toString());
    }
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>getSetOfObjects</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method acts as a wrapper to the <CODE>DbAccess.queryByClass()</CODE>
 * method to implement a retry mechanism in the case that a
 * <CODE>DbLockNotGrantedException</CODE> has been thrown.</P>
 *
 * <P>The method re-tries 4 times, with a space of 500ms between each
 * attempt. </P>
 *
 * @param       entityId
 *              - used by the <CODE>DbAccess.queryByClass()</CODE> method
 *                to attempt to lock all objects with this id in the
 *                database to the current transaction
 *
 * @return      HashSet
 *              - either null if the objects could not be locked, or a
 *                set of the required objects
 *
 * Outputs:     None
 *
 */

public HashSet getSetOfObjects(EntityIdEnum entityId)
{
    // indicates whether another attempt should be made to lock the objects
    boolean tryagain = true;

    // initialise the attempt counter
    int attempts = 1;

    // initialise the return set
    HashSet pgSet = null;

    // keep going whilst the attempts have not been exhausted
    while ((tryagain) && (pgSet == null))
    {
        try
        {
            // attempt to lock all the objects in the database
            EntityPropertyList propList = EntityPropertyList.getInstance();
            pgSet = DbAccess.queryByClass(
                propList.getEntityClass(entityId).getName());
        }
        catch (DbLockNotGrantedException e)
        {
            // check to see if the attempts have been exhausted
            if (attempts > 3)
            {
                tryagain = false;
                pgSet = null;
                new Swerr(e.toString());
            }
            else
            {
                attempts++;

                try
                {
                    // put this thread to sleep for 500ms
                    Thread.sleep((long)500);
                }
                catch (Exception ex)
                {
                    // in the case of any Exception, Swerr and exit
                    new Swerr(ex.toString());
                    pgSet = null;
                    break;
                }
            }
        }
```

```
            catch (Exception e)
            {
                // in the case of any other Exception, Swerr and exit
                new Swerr(e.toString());
                pgSet = null;
                break;
            }
        }

        return pgSet;
    }

} // end of TpNrUtilities
```

Class: TpNrValidations

```
/**
 * <B>File:</B> <CODE>TpNrValidations.java</CODE>
 */

package equinox.protection.traffic.reconfig;

import equinox.framework.nom.keys.EntityPgmKey;

import equinox.framework.database.DbAccess;
import equinox.framework.database.DbLockNotGrantedException;

import equinox.framework.nom.objs.EntityPg;
import equinox.framework.nom.objs.EntityPgm;

import equinox_ne_connections.engine.user_interface.Xconlist;
import equinox_ne_connections.engine.user_interface.Xcon;

import equinox.framework.nom.enums.ForceStatusEnum;
import equinox.framework.nom.enums.ManualStatusEnum;
import equinox.framework.nom.enums.TPReconfigStateEnum;
import equinox.framework.nom.enums.ProtectionSchemeEnum;

import equinox.protection.traffic.reconfig.TpNrErrorCode;

import equinox.framework.swerr.Swerr;

import java.util.HashSet;
import java.util.Iterator;

/**
 * <P><B>Class:</B> <CODE>TpNrValidations</CODE></P>
 *
 * <P><B>Document:</B> Standard Reconfigs DD
 *        - available in Athena, following this path:
 * <BR>   - Equinox
 * <BR>     - Product Development
 * <BR>       - Software
 * <BR>         - Network Element Software
 * <BR>           - Functional Areas
 * <BR>             - Network Reconfig
 * <BR>               - Design Folder      </P>
 *
 * <B>Description:</B>
 *
 * The TpNrValidations class definition encapsulates the functionality used to
 * validate network reconfig commands.
 *
 * @author Andy Kinney
 */
/*
 * Methods:
 *
 * boolean nodeMapIsNil(...)
 * - checks that the node map attribute of the PG is nil
 *
 * boolean noActiveReconfigs(...)
 * - checks that no other reconfigs are in progress
 *
 * boolean noProtSwActive(...)
 * - checks that there are no active protection switches
 *
 * boolean isPgBlsr(...)
 * - checks that the PG is actually 4F BLSR
 *
 * boolean noAddDrops(...)
 * - checks there are no add/drop connections on the PG
 *
 *
 */
```

```
public final class TpNrValidations
{
    /**
     * singleton instance of this class
     */
    private static TpNrValidations INSTANCE = null;

    /**
     *
     * <P><B>Method:</B>      <CODE>getInstance</CODE>
     *
     * <BR><B>Description:</B>
     *
     * This method returns the instance of this singleton.</P>
     *
     * @return      TpNrValidations
     *               - the singleton instance of TpNrValidations
     */
    /*
     * Inputs:      None
     *
     * Outputs:     None
     *
     */

    public static synchronized TpNrValidations getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new TpNrValidations();
        }

        return INSTANCE;
    }

    /**
     * Constructor, made private as class is singleton
     */
    private TpNrValidations ()
    {
    }
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>nodeMapIsNil</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method checks the current value of the node map PG attribute, and
 * returns true if the value is nil (i.e. the node map has not been set) and
 * false otherwise.</P>
 *
 * <P>It achieves this by checking the IeeeMap attribute value. If this is ""
 * then there is no node map.</P>
 *
 *
 * @param       pgEntity
 *              - PG Entity upon which the reconfig operation has been
 *                executed
 *
 * @return      String
 *              - null indicates node map is nil, non-null indicates non-nil
 */
/*
 * Outputs:     None
 *
 */

public String nodeMapIsNil (EntityPg pgEntity)
{
    String rc = null;

    if ((pgEntity.getIeeemap()).equals("") != true)
    {
        rc = TpNrErrorCode.TP_NR_ERR_VALID_NODE_MAP;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>noActiveReconfigs</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method determines whether the reconfig state attribute of the PG
 * indicates that there is a reconfig in progress.</P>
 *
 * <P>NOTE: This method must be called within a pre-validation event to
 * ensure there is an old object to retrieve.</P>
 *
 * @param       pgEntity
 *              - PG Entity upon which the reconfig operation has been
 *                executed
 *
 * @return      String
 *              - null indicates no active reconfigs, non-null otherwise
 */
/*
 * Outputs:     None
 *
 */

public String noActiveReconfigs (EntityPg pgEntity)
{
    String rc = null;

    // get the old PG entity
    EntityPg oldPg = (EntityPg)pgEntity.getOldObj();

    int oldRcState = oldPg.getRcstate().intValue();

    switch (oldRcState)
    {
        case TPReconfigStateEnum.TP_RC_STATE_ADDNODE_VALUE:
            rc = TpNrErrorCode.TP_NR_ERR_IN_ADD_NODE;
            break;

        case TPReconfigStateEnum.TP_RC_STATE_DELNODE_VALUE:
            rc = TpNrErrorCode.TP_NR_ERR_IN_DEL_NODE;
            break;

        case TPReconfigStateEnum.TP_RC_STATE_IDLE_VALUE:
            // there are no active reconfigs
            break;

        default:
            new Swerr(oldRcState);
            rc = TpNrErrorCode.TP_NR_ERR_SWERR;
            break;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>noProtSwActive</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method determines whether there is any protection activity on any
 * of the PGMs that form the PG on which the reconfig operation has been
 * executed upon.</P>
 *
 *
 * @param       pgEntity
 *              - PG Entity upon which the reconfig operation has been
 *                executed
 *
 * @return      String
 *              - null indicates no active protection switches, non-null
 *                otherwise
 */
/*
 * Outputs:     None
 *
 */

public String noProtSwActive (EntityPg pgEntity)
{
    String rc = null;

    try
    {
        // get the PGM list from the PG Entity
        EntityPgmKey[] pgmList = pgEntity.getMemberList();

        // work through the list of PGMs
        int index = 0;
        while ((index < pgmList.length) && (rc == null))
        {
            EntityPgm pgm = (EntityPgm)DbAccess.queryByKey(pgmList[index]);

            // check to see if either a force or manual request is active
            if ((pgm.getForceReq()       == true) ||
                (pgm.getManualReq()      == true) ||
                (pgm.getRingforceReq()  != ForceStatusEnum.FORCE_OFF   ) ||
                (pgm.getRingmanualReq() != ManualStatusEnum.MANUAL_OFF))
            {
                rc = TpNrErrorCode.TP_NR_ERR_PROT_SW_ACTIVE;
            }

            index++;
        }
    }
    catch (DbLockNotGrantedException lck)
    {
        rc = TpNrErrorCode.TP_NR_ERR_LOCK_NOT_GRANTED_PGM;
    }
    catch (Exception e)
    {
        new Swerr(e.toString());
        rc = TpNrErrorCode.TP_NR_ERR_SWERR;
    }

    return rc;
}
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>isPgBlsr</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method checks that the PG protection scheme is 4F BLSR.</P>
 *
 *
 * @param       pgEntity
 *              - PG Entity upon which the reconfig operation has been
 *                executed
 *
 * @return      String
 *              - null indicates PG is BLSR, non-null otherwise
 */
/*
 * Outputs:     None
 *
 */

 public String isPgBlsr (EntityPg pgEntity)
 {
     String rc = null;

     // get the protection scheme from the PG
     ProtectionSchemeEnum prSch = pgEntity.getProtectionScheme();

     // check the protection scheme is 4F BLSR
     if (prSch != ProtectionSchemeEnum.PROT_FOUR_FIBER_BLSR)
     {
         rc = TpNrErrorCode.TP_NR_ERR_PROT_SCHEME_INVAL;
     }

     return rc;
 }
```

```
/**
 *
 * <P><B>Method:</B>       <CODE>noAddDrops</CODE>
 *
 * <BR><B>Description:</B>
 *
 * This method obtains a list of all Xcons for the PGMs involved in the
 * reconfig, and determines whether any of them are non-passthrough thereby
 * indicating that the delete node operation cannot proceed.</P>
 *
 *
 * @param       pgEntity
 *              - PG Entity upon which the reconfig operation has been
 *                executed
 *
 * @return      String
 *              - null indicates no add drop connections, non-null otherwise
 */
/*
 * Outputs:     None
 *
--*/

 public String noAddDrops (EntityPg pgEntity)
 {
     String rc = null;

     try
     {
         // get the list of Xcons that apply to this PG
         HashSet xconList = Xconlist.getXconsOnPg(pgEntity);

         if (xconList != null)
         {
             // get list of PGMs from PG Entity
             EntityPgmKey[] pgmList = pgEntity.getMemberList();
             byte pgmNumber = pgEntity.getNumberOfPgm();

             // loop through all xcons
             Iterator i = xconList.iterator();
             while (i.hasNext() && (rc == null))
             {
                 Xcon xcon = (Xcon)i.next();

                 boolean sourceFound = false;
                 boolean sinkFound = false;

                 // get the source information
                 short xconShelf = xcon.getPhysicalEndpointA(0).getShelf();
                 short xconSlot  = xcon.getPhysicalEndpointA(0).getSlot();
                 short xconPort  = xcon.getPhysicalEndpointA(0).getPort();

                 // loop through the pgm list to find a match on the source
                 for (byte index = 0; index < pgmNumber; index++)
                 {
                     // if the source matches
                     if ((pgmList[index].getShelfId() == xconShelf) &&
                         (pgmList[index].getSlotId()  == xconSlot ) &&
                         (pgmList[index].getPortId()  == xconPort ) )
                     {
                         sourceFound = true;
                         break;
                     }
                 }

                 // get the sink information
                 xconShelf = xcon.getPhysicalEndpointZ(0).getShelf();
                 xconSlot  = xcon.getPhysicalEndpointZ(0).getSlot();
                 xconPort  = xcon.getPhysicalEndpointZ(0).getPort();

                 // loop through the pgm list again to match the
                 // other side of the connection, i.e. looking
                 // for a pass through connection
                 for (byte loop = 0; loop < pgmNumber; loop++)
                 {
                     // if the sink matches
                     if ((pgmList[loop].getShelfId() == xconShelf) &&
                    (pgmList[loop].getSlotId()  == xconSlot ) &&
                    (pgmList[loop].getPortId()  == xconPort ) )
                {
                    sinkFound = true;
                    break;
                }
            }

            short srcPld = xcon.getPhysicalEndpointA(0).getTimeslot();
            short dstPld = xcon.getPhysicalEndpointZ(0).getTimeslot();

            // if the connection is a passthrough
            if ( (sourceFound) && (sinkFound) && (srcPld == dstPld) )
            {
                // pass through connection
            }
            else
            {
                rc = TpNrErrorCode.TP_NR_ERR_ADD_DROP;
            }
        }
    }
}
catch (Exception e)
{
    new Swerr(e.toString());
    rc = TpNrErrorCode.TP_NR_ERR_SWERR;
}

return rc;
}

} // end of TpNrValidations
```

Appendix B: Unit Test Cases

In the tables below the unit test cases are listed, using the naming convention as used in the original documentation ([3]).

Single Session:

TCID	Title
UT1	Successful Add Node command
UT2	Repeat UT1 for PG2
UT3	Repeat UT1 for PG3
UT4	Repeat UT1 for PG4
UT5	Repeat UT1 for PG5
UT6	Repeat UT1 for PG6
UT7	Repeat UT1 for PG7
UT8	Repeat UT1 for PG8
UT9	Repeat UT1 for PG9
UT10	Repeat UT1 for PG10
UT11	Repeat UT1 for PG11
UT12	Repeat UT1 for PG12
UT13	Place ALL 4F PGs into Add Node
UT14	Successful Delete Node command
UT15	Repeat UT14 for PG2
UT16	Repeat UT14 for PG3
UT17	Repeat UT14 for PG4
UT18	Repeat UT14 for PG5
UT19	Repeat UT14 for PG6
UT20	Repeat UT14 for PG7
UT21	Repeat UT14 for PG8
UT22	Repeat UT14 for PG9
UT23	Repeat UT14 for PG10
UT24	Repeat UT14 for PG11
UT25	Repeat UT14 for PG12
UT26	Place ALL 4F PGs into Delete Node
UT27	Successful Delete Node backout command

TCID	Title
UT28	Repeat UT27 for PG2
UT29	Repeat UT27 for PG3
UT30	Repeat UT27 for PG4
UT31	Repeat UT27 for PG5
UT32	Repeat UT27 for PG6
UT33	Repeat UT27 for PG7
UT34	Repeat UT27 for PG8
UT35	Repeat UT27 for PG9
UT36	Repeat UT27 for PG10
UT37	Repeat UT27 for PG11
UT38	Repeat UT27 for PG12
UT39	Successful Add Node backout command
UT40	Repeat UT39 for PG2
UT41	Repeat UT39 for PG3
UT42	Repeat UT39 for PG4
UT43	Repeat UT39 for PG5
UT44	Repeat UT39 for PG6
UT45	Repeat UT39 for PG7
UT46	Repeat UT39 for PG8
UT47	Repeat UT39 for PG9
UT48	Repeat UT39 for PG10
UT49	Repeat UT39 for PG11
UT50	Repeat UT39 for PG12
UT51	Successful Add Node complete command
UT52	Repeat UT51 for PG2
UT53	Repeat UT51 for PG3
UT54	Repeat UT51 for PG4
UT55	Repeat UT51 for PG5
UT56	Repeat UT51 for PG6
UT57	Repeat UT51 for PG7
UT58	Repeat UT51 for PG8
UT59	Repeat UT51 for PG9
UT60	Repeat UT51 for PG10

TCID	Title
UT61	Repeat UT51 for PG11
UT62	Repeat UT51 for PG12
UT63	Successful Delete Node complete command
UT64	Repeat UT63 for PG2
UT65	Repeat UT63 for PG3
UT66	Repeat UT63 for PG4
UT67	Repeat UT63 for PG5
UT68	Repeat UT63 for PG6
UT69	Repeat UT63 for PG7
UT70	Repeat UT63 for PG8
UT71	Repeat UT63 for PG9
UT72	Repeat UT63 for PG10
UT73	Repeat UT63 for PG11
UT74	Repeat UT63 for PG12
UT75	Add Node command denied when PG in Add Node
UT76	Add Node command denied when PG in Delete Node
UT77	Add Node command denied when PG is not 4F BLSR
UT78	Add Node command denied when PG has a non-nil nodemap
UT79	Delete Node command denied when PG in Delete Node
UT80	Delete Node command denied when PG in Add Node
UT81	Delete Node command denied when PG is not 4F BLSR
UT82	Delete Node command denied when PG has a nil nodemap
UT83	PG Create denied if setting rcstate to add node
UT84	PG Create denied if setting rcstate to delete node
UT85	PG Create successful if setting rcstate to idle
UT86	PG Delete denied if PG in Add Node
UT87	PG Delete denied if PG in Delete Node
UT88	PG Delete successful if PG in Idle
UT89	Multiple attribute ED-FFP-LL command denied, PG Idle
UT90	Multiple attribute ED-FFP-LL command denied, PG Add Node
UT91	Multiple attribute ED-FFP-LL command denied, PG Delete Node
UT92	Multiple attribute ED-FFP-LL command denied, PG Idle
UT93	Multiple attribute ED-FFP-LL command denied, PG Add Node

TCID	Title
UT94	Multiple attribute ED-FFP-LL command denied, PG Delete Node
UT95	Multiple attribute ED-FFP-LL command denied, PG Idle
UT96	Multiple attribute ED-FFP-LL command denied, PG Add Node
UT97	Multiple attribute ED-FFP-LL command denied, PG Delete Node
UT98	Idle command has no effect with PG in IDLE
UT99	Nil Nodemap command has no effect with PG in Add Node
UT100	Nil Nodemap command has no effect with PG in Delete Node
UT101	ED-FFP-LL command denied, rcstate=unknown, PG Idle
UT102	ED-FFP-LL command denied, rcstate=unknown, PG Add Node
UT103	ED-FFP-LL command denied, rcstate=unknown, PG Delete Node
UT104	Successful Add Node command with full Xcons provisioned
UT105	Successful Add Node backout command with full Xcons
UT106	Successful Add Node complete command with full Xcons
UT107	Successful Delete Node command with full Xcons
UT108	Successful Delete Node backout command with full Xcons
UT109	Successful Delete Node complete command with full Xcons
UT110	Add/Drop Xcons denied when in Add Node
UT111	Add/Drop Xcons denied when in Delete Node
UT112	Pass-through Xcons can be created when in Add Node
UT113	Pass-through Xcons can be created when in Delete Node
UT114	PG Delete denied if PG has add/drop connection: STS1 PLD1
UT115	Repeat UT113 for STS1 PLD96
UT116	Repeat UT113 for STS1 PLD192
UT117	Repeat UT113 for STS-3C PLD1
UT118	Repeat UT113 for STS-3C PLD49
UT119	Repeat UT113 for STS-3C PLD190
UT120	Repeat UT113 for STS-12C PLD1
UT121	Repeat UT113 for STS-12C PLD97
UT122	Repeat UT113 for STS-12C PLD181
UT123	Repeat UT113 for STS-48C PLD1
UT124	Repeat UT113 for STS-48C PLD49
UT125	Repeat UT113 for STS-48C PLD145

Multi-Session:

TCID	Title
UT126	Delete Node command denied when manual span switch active
UT127	Repeat UT125 for PG2
UT128	Repeat UT125 with forced span switch active
UT129	Repeat UT127 for PG2
UT130	Repeat UT125 with manual ring switch active
UT131	Repeat UT129 for PG2
UT132	Repeat UT125 with forced ring switch active
UT133	Repeat UT131 for PG2
UT134	Successful Delete Node command with LOCKOUT active
UT135	Repeat UT133 for active RINGLOCKOUT
UT136	Successful Add Node command
UT137	Successful Delete Node command

Manual:

TCID	Title
UT138	SC Restart, PG in Add Node with non-nil nodemap
UT139	SC Restart, PG in Delete Node with non-nil nodemap
UT140	PGMs are locked when attempting delete node
UT141	PG is locked when reconfig is initializing
UT142	SC Restart, PG in Delete Node
UT143	SC Restart, PG in Add Node

Appendix C: Integration Test Cases

In the table below the integration test cases are listed, using the naming convention as used in the original documentation ([3]).

NOTE: the test cases are split into prority 1, 2 and 3. The purpose is to ensure that as a minimum, priority 1 test cases which test 80% of the functionality are executed and passed before the software can be delivered to Verification. The remaining priority 2 and 3 test cases are executed to ensure conistency with the results obtained from the unit testing with the simulator which then provide 100% coverage.

TCID	Title	Priority
IT1	Add Node to 4F BLSR Ring	1
IT2	Repeat IT1 for PG2	1
IT3	Repeat IT1 for PG3	2
IT4	Repeat IT1 for PG4	3
IT5	Repeat IT1 for PG5	2
IT6	Repeat IT1 for PG6	3
IT7	Repeat IT1 for PG7	2
IT8	Repeat IT1 for PG8	3
IT9	Repeat IT1 for PG9	2
IT10	Repeat IT1 for PG10	3
IT11	Repeat IT1 for PG11	2
IT12	Repeat IT1 for PG12	3
IT13	Add DX Node into a mixed ring configuration	1
IT14	Add CLASSIC 192 Node into mixed ring configuration	3
IT15	Add Node - Backout	1
IT16	Repeat IT15 for a DX bay	1
IT17	Repeat IT15 for a CLASSIC bay	3
IT18	Add Node between two OPC span of control	2
IT19	Add Node with Extra Traffic connections provisioned	3
IT20	Add DX Node with Extra Traffic connections provisioned	2
IT21	Add Node is denied when in Delete Node	1
IT22	Add Node is denied when DX in Delete Node	2
IT23	Add/Drop connection provisioning denied while in reconfig	1
IT24	Issue Add Node on a missing PG	2
IT25	Repeat IT24 except via command line	1

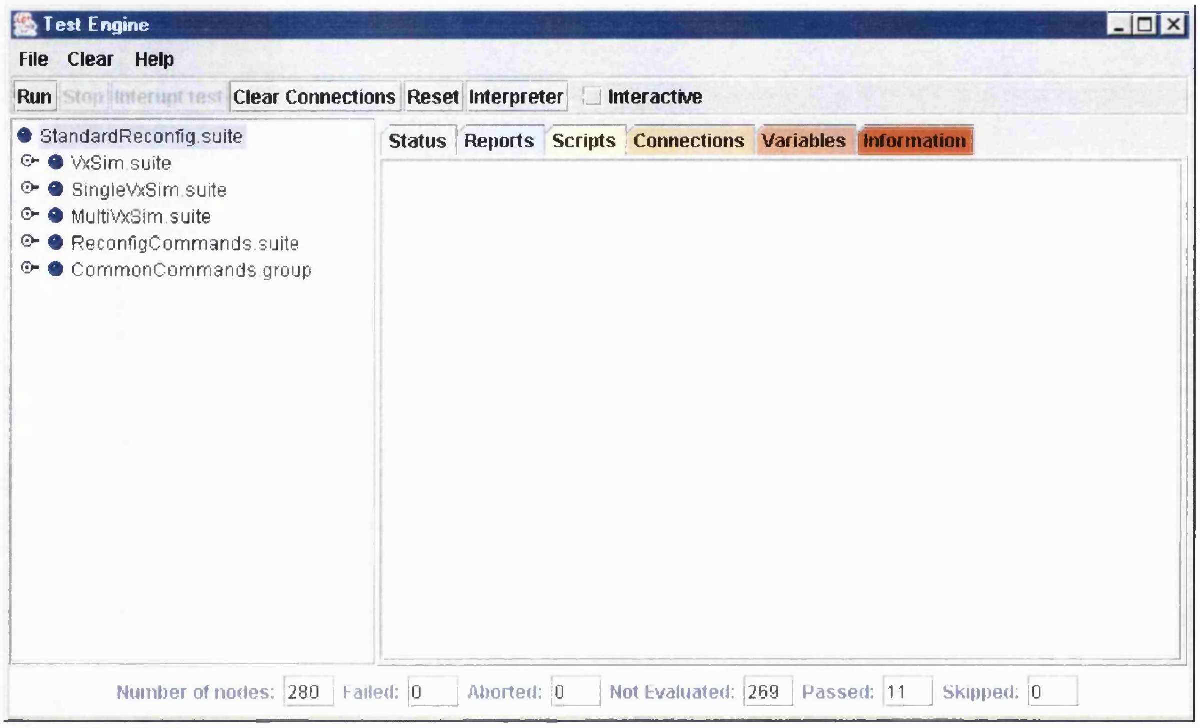
TCID	Title	Priority
IT26	Issue Add Node when switch cards are missing	2
IT27	Issue Add Node when port cards are missing	2
IT28	Issue Add Node when there are existing connections on the PG	1
IT29	Issue Add Node on a PG already configured into a ring	1
IT30	Issue Add Node on a 1+1 Linear PG	1
IT31	Issue Add Node after restarting NCSC	1
IT32	Repeat IT31 after NE has just been commissioned	2
IT33	Repeat IT31 restart NCSC during command	2
IT34	Repeat IT31 restart MXT after command successfully completed	2
IT35	Repeat IT31 restart Port after command successfully completed	3
IT36	Issue AddNode and leave for an extended period of time (overnight)	3
IT37	Ensure Add Node status is maintained over a SC software download	3
IT38	Repeat IT37 downloading MXT card	3
IT39	Repeat IT37 downloading both port cards in group	3
IT40	Repeat IT37 downloading switch card	3
IT41	Delete Node from a 4F BLSR Ring	1
IT42	Repeat IT41 for PG2	1
IT43	Repeat IT41 for PG3	2
IT44	Repeat IT41 for PG4	3
IT45	Repeat IT41 for PG5	2
IT46	Repeat IT41 for PG6	3
IT47	Repeat IT41 for PG7	2
IT48	Repeat IT41 for PG8	3
IT49	Repeat IT41 for PG9	2
IT50	Repeat IT41 for PG10	3
IT51	Repeat IT41 for PG11	2
IT52	Repeat IT41 for PG12	3
IT53	Delete DX Node from a mixed ring configuration	1
IT54	Delete CLASSIC 192 Node from a mixed ring configuration	3
IT55	Delete Node - Backout	1
IT56	Repeat IT55 for a DX node	1
IT57	Repeat IT55 for a CLASSIC 192 node	3

TCID	Title	Priority
IT58	Delete Node between two OPC spans of control	2
IT59	Delete Node with Extra Traffic connections provisioned	2
IT60	Delete DX Node with Extra Traffic connections provisioned	2
IT61	Delete Node is denied when in Add Node	1
IT62	Delete Node is denied when DX in Add Node	2
IT63	Issue Delete Node on a missing PG	2
IT64	Repeat IT63 except via command line	1
IT65	Issue Delete Node when switch cards are missing	1
IT66	Issue Delete Node when port cards are missing	1
IT67	Issue DeleteNode when there are add/drop connections on the PG	1
IT68	Repeat IT67 with pass through connections only	1
IT69	Repeat IT67 with inter-ring connections	1
IT70	Issue Delete Node with active protection switch	1
IT71	Repeat IT70 with forced span switch active	1
IT72	Repeat IT70 with manual ring switch active	1
IT73	Repeat IT70 with forced ring switch active	1
IT74	Issue Delete Node on a 1+1 Linear PG	1
IT75	Issue Delete Node after restarting NCSC	1
IT76	Repeat IT75 after NE has just been commissioned	2
IT77	Repeat IT75 restart NCSC during command	1
IT78	Repeat IT75 restart MXT after command successfully completed	2
IT79	Repeat IT75 restart Port after command successfully completed	2
IT80	Ensure Delete Node status is maintained over a software download	3
IT81	Repeat IT80 downloading MXT card	3
IT82	Repeat IT80 downloading both port cards in group	3
IT83	Repeat IT80 downloading switch card	3
IT84	Issue Delete Node and leave for extended period of time (overnight)	3

Appendix D: Test Engine

This section describes the unit test scripts provided as part of the Standard Reconfig test suite.

Load the test suite into the Test Engine:


Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter Interactive
Status Reports Scripts Connections Variables Information

StandardReconfig.suite
VxSim.suite
SingleVxSim.suite
MultiVxSim.suite
ReconfigCommands.suite
CommonCommands.group

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0

In the test suite there are the scripts necessary for launching both the single, and multi simulator sessions required to execute the unit test cases.

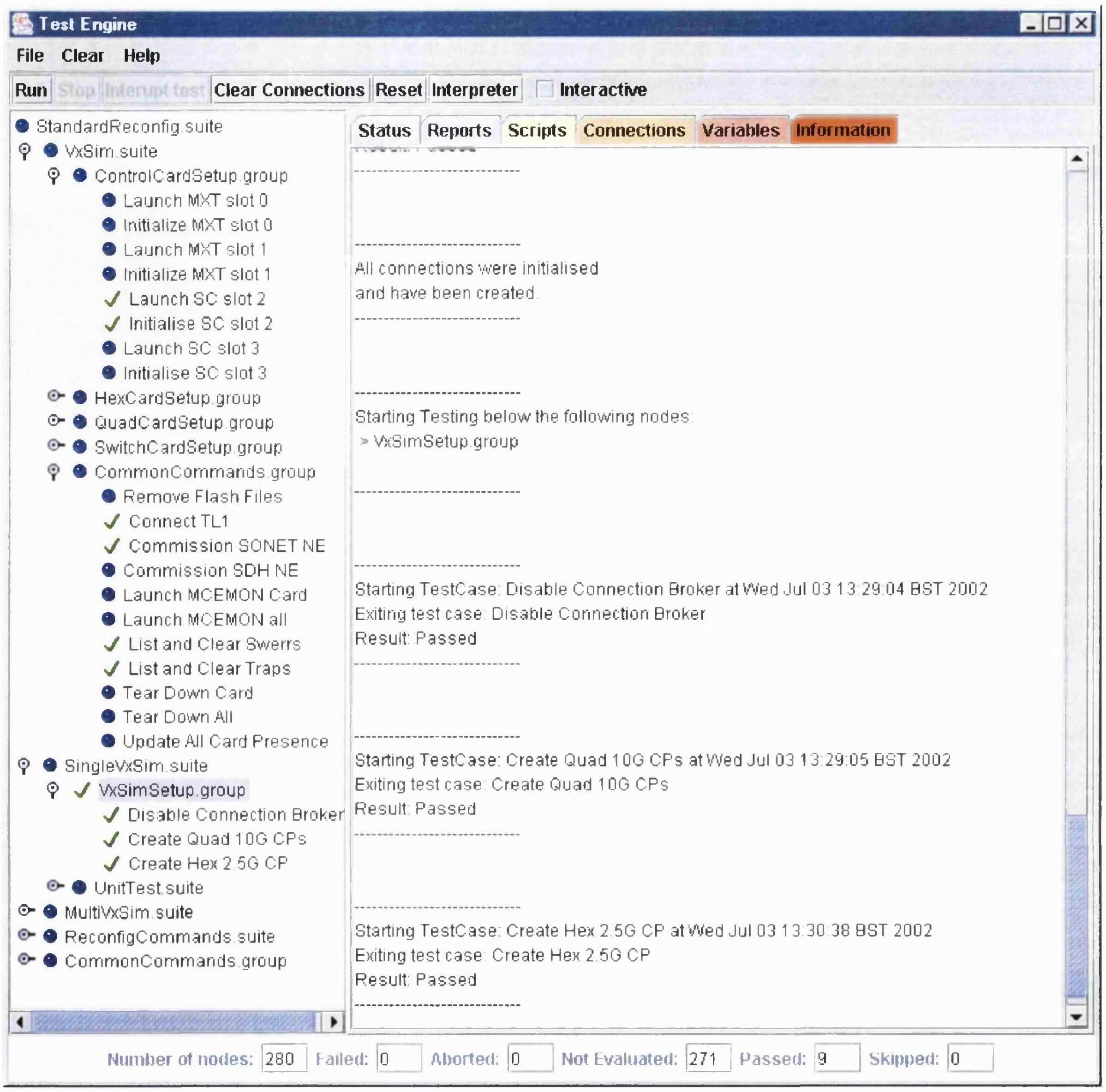
For the single simulator, the following is required:

- Launch SC Slot 2
- Initialise SC Slot 2
- Connect TL1
- Commission SONET NE

In addition in the SingleVxSim.suite under VxSimSetup.group further scripts are added to complete the setup:

- Disable Connection Broker
- Create Quad 10G CPs
- Create Hex 2.5G CP

e.g.

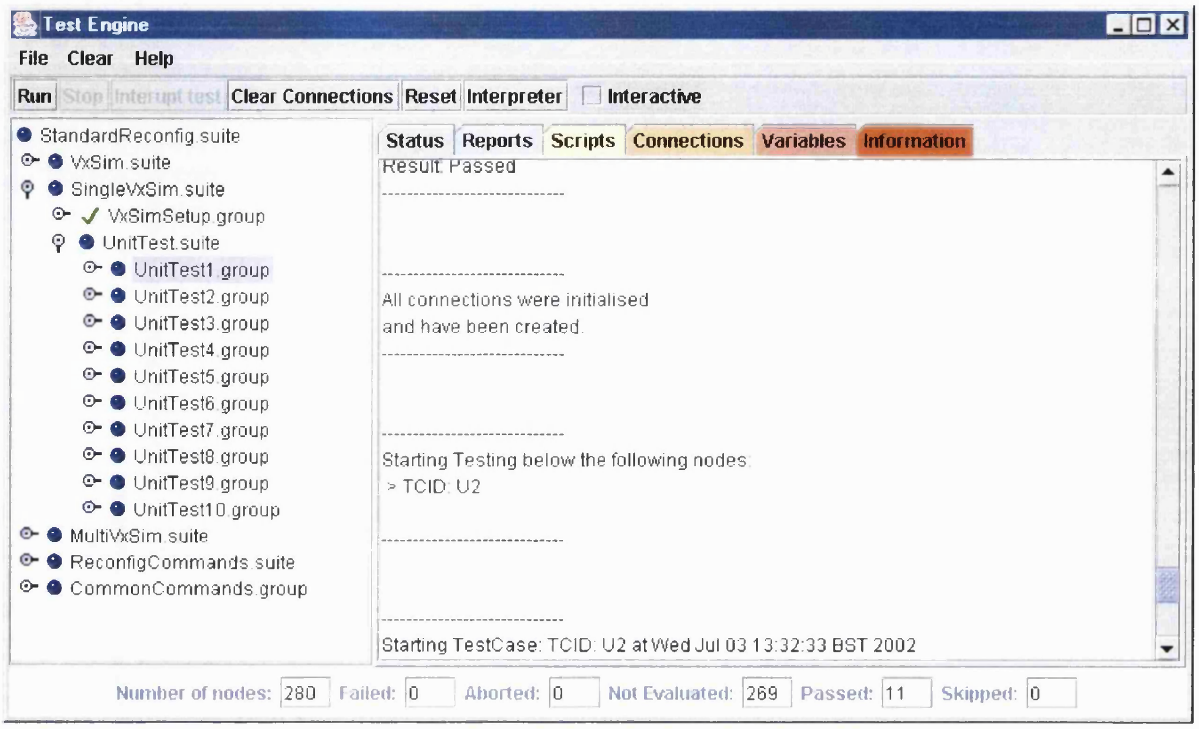


```
-----------------------------



-----------------------------
All connections were initialised
and have been created.
-----------------------------



-----------------------------
Starting Testing below the following nodes:
 > VxSimSetup.group

-----------------------------



-----------------------------
Starting TestCase: Disable Connection Broker at Wed Jul 03 13:29:04 BST 2002
Exiting test case: Disable Connection Broker
Result: Passed
-----------------------------



-----------------------------
Starting TestCase: Create Quad 10G CPs at Wed Jul 03 13:29:05 BST 2002
Exiting test case: Create Quad 10G CPs
Result: Passed
-----------------------------



-----------------------------
Starting TestCase: Create Hex 2.5G CP at Wed Jul 03 13:30:38 BST 2002
Exiting test case: Create Hex 2.5G CP
Result: Passed
-----------------------------
```

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 271 Passed: 9 Skipped: 0

In addition the SingleVxSim.suite contains the UnitTest.suite. The UnitTest.suite contains all the automated test scripts that execute the unit test cases.

e.g.

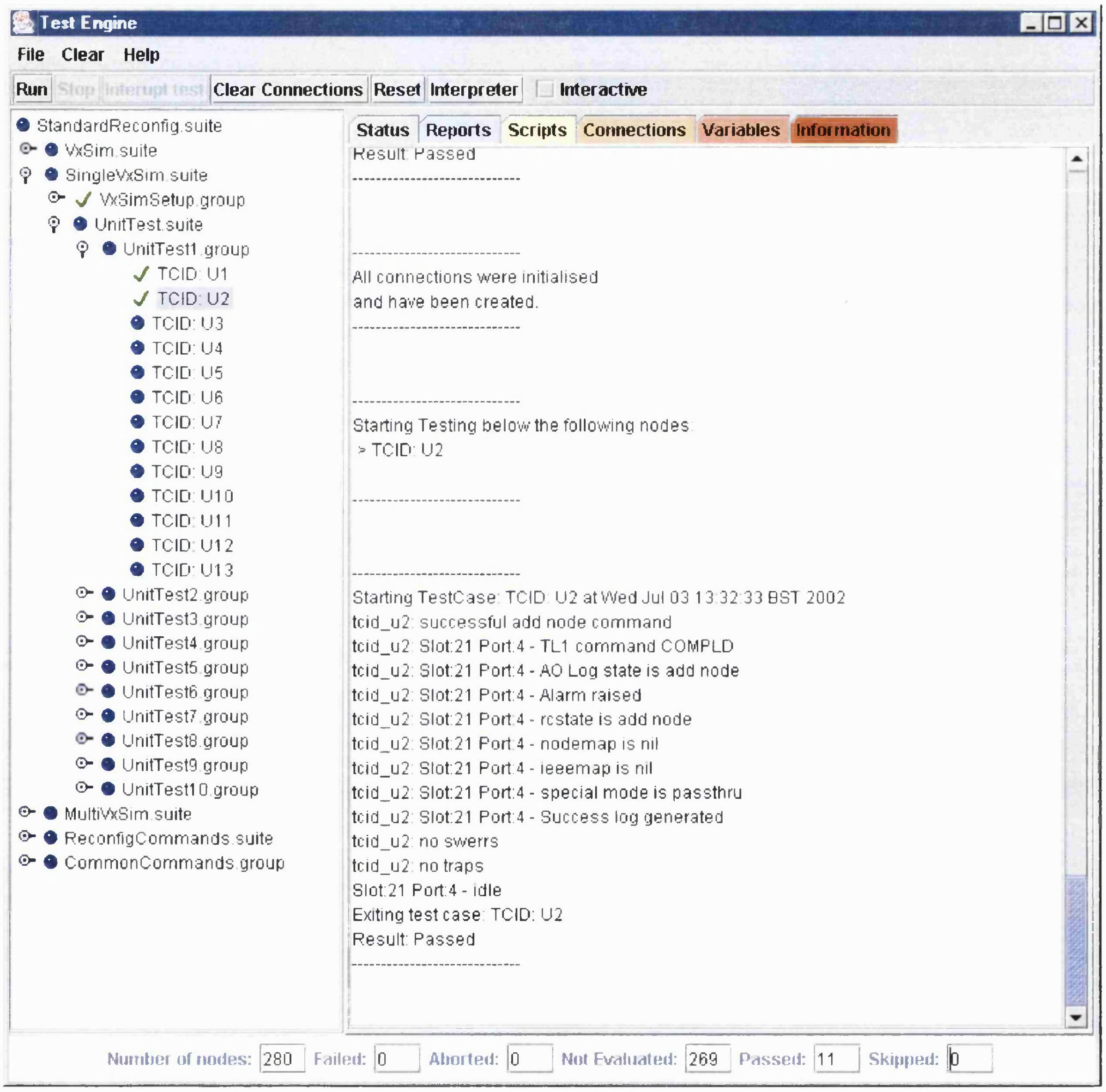

Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter Interactive
Status Reports Scripts Connections Variables Information

StandardReconfig.suite
VxSim.suite
SingleVxSim.suite
VxSimSetup.group
UnitTest.suite
UnitTest1.group
UnitTest2.group
UnitTest3.group
UnitTest4.group
UnitTest5.group
UnitTest6.group
UnitTest7.group
UnitTest8.group
UnitTest9.group
UnitTest10.group
MultiVxSim.suite
ReconfigCommands.suite
CommonCommands.group

```
Result: Passed
----------------------------



----------------------------
All connections were initialised
and have been created.
----------------------------



----------------------------
Starting Testing below the following nodes:
 > TCID: U2

----------------------------



----------------------------
Starting TestCase: TCID: U2 at Wed Jul 03 13:32:33 BST 2002
```

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0

To execute an individual test case, open up a test group, and run that test case:


Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter Interactive
Status Reports Scripts Connections Variables Information

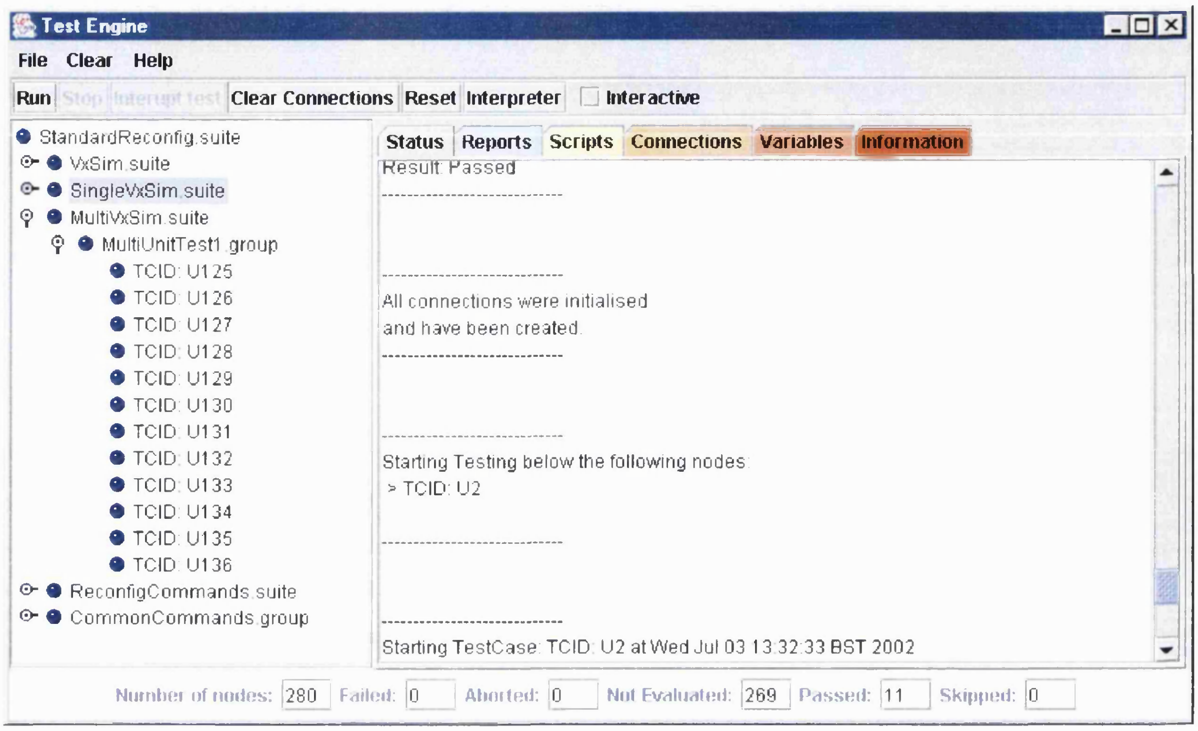
```
StandardReconfig.suite
  VxSim.suite
  SingleVxSim.suite
    VxSimSetup.group
    UnitTest.suite
      UnitTest1.group
        TCID: U1
        TCID: U2
        TCID: U3
        TCID: U4
        TCID: U5
        TCID: U6
        TCID: U7
        TCID: U8
        TCID: U9
        TCID: U10
        TCID: U11
        TCID: U12
        TCID: U13
      UnitTest2.group
      UnitTest3.group
      UnitTest4.group
      UnitTest5.group
      UnitTest6.group
      UnitTest7.group
      UnitTest8.group
      UnitTest9.group
      UnitTest10.group
  MultiVxSim.suite
  ReconfigCommands.suite
  CommonCommands.group
```

```
Result: Passed
-----------------------------


-----------------------------
All connections were initialised
and have been created.
-----------------------------


-----------------------------
Starting Testing below the following nodes:
 > TCID: U2

-----------------------------


-----------------------------
Starting TestCase: TCID: U2 at Wed Jul 03 13:32:33 BST 2002
tcid_u2: successful add node command
tcid_u2: Slot:21 Port:4 - TL1 command COMPLD
tcid_u2: Slot:21 Port:4 - AO Log state is add node
tcid_u2: Slot:21 Port:4 - Alarm raised
tcid_u2: Slot:21 Port:4 - rcstate is add node
tcid_u2: Slot:21 Port:4 - nodemap is nil
tcid_u2: Slot:21 Port:4 - ieeemap is nil
tcid_u2: Slot:21 Port:4 - special mode is passthru
tcid_u2: Slot:21 Port:4 - Success log generated
tcid_u2: no swerrs
tcid_u2: no traps
Slot:21 Port:4 - idle
Exiting test case: TCID: U2
Result: Passed
-----------------------------
```

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0

NOTE: that with each UnitTest#.group there is a UnitTest#.es file associated with it that contains the actual script to perform the test case.

In the MultiVxSim.suite there is the MultiUnitTest1.group. The MultiUnitTest1.group contains those test scripts that apply to the multi vxsim configuration.

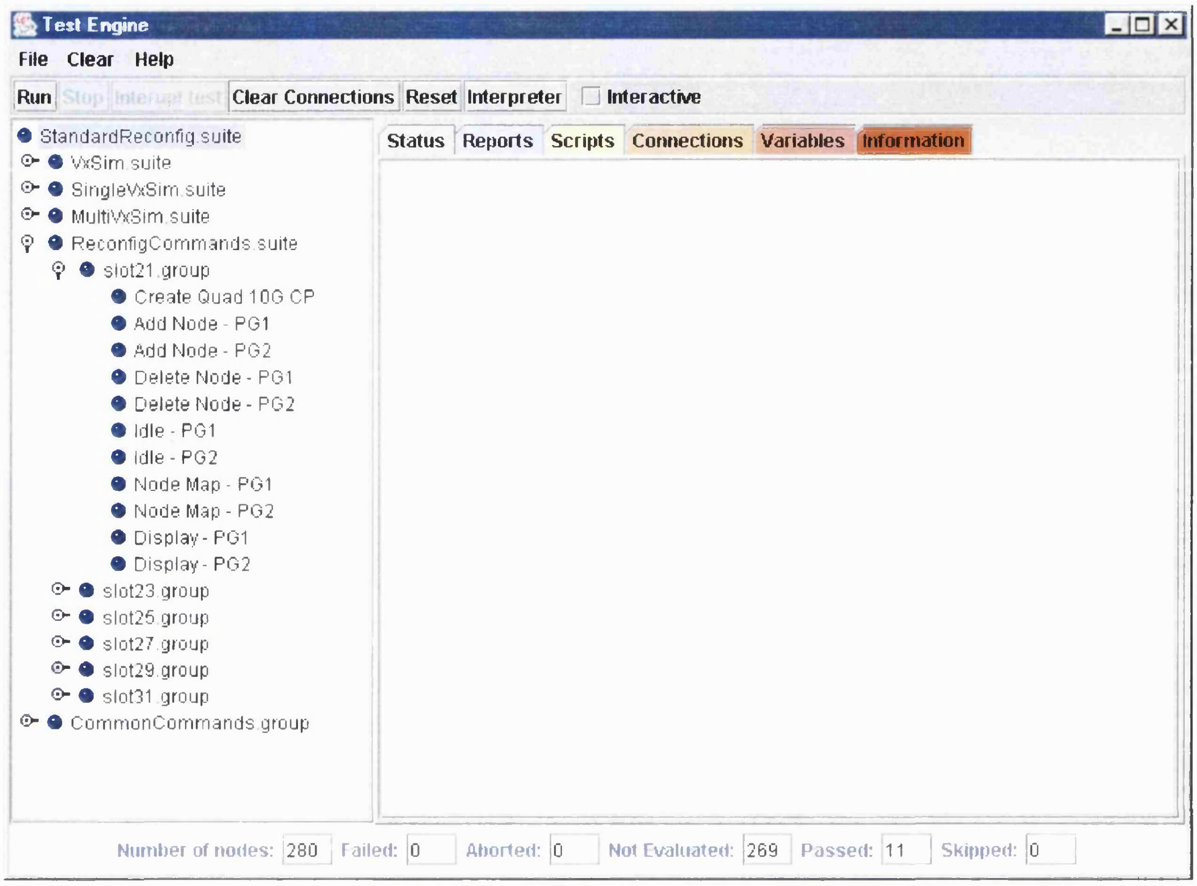

Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter Interactive
Status Reports Scripts Connections Variables Information

StandardReconfig.suite
VxSim.suite
SingleVxSim.suite
MultiVxSim.suite
MultiUnitTest1.group
TCID: U125
TCID: U126
TCID: U127
TCID: U128
TCID: U129
TCID: U130
TCID: U131
TCID: U132
TCID: U133
TCID: U134
TCID: U135
TCID: U136
ReconfigCommands.suite
CommonCommands.group

```
Result: Passed
----------------------------



----------------------------
All connections were initialised
and have been created.
----------------------------



----------------------------
Starting Testing below the following nodes
 > TCID: U2

----------------------------



----------------------------
Starting TestCase: TCID: U2 at Wed Jul 03 13:32:33 BST 2002
```

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0

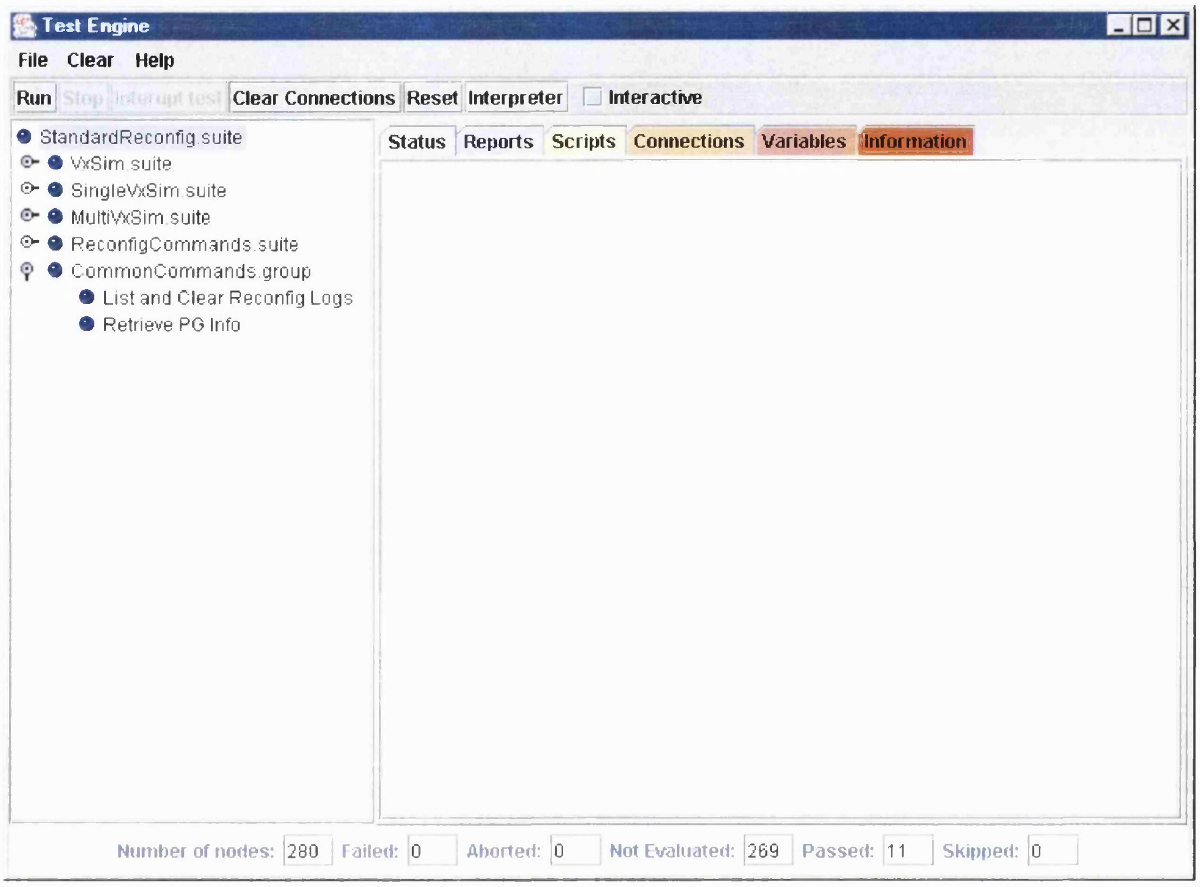
The ReconfigCommands.suite contain useful utilities to put specific PGs into add node, delete node, send node maps etc..


Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter Interactive
Status Reports Scripts Connections Variables Information

StandardReconfig.suite
VxSim.suite
SingleVxSim.suite
MultiVxSim.suite
ReconfigCommands.suite
slot21.group
Create Quad 10G CP
Add Node - PG1
Add Node - PG2
Delete Node - PG1
Delete Node - PG2
Idle - PG1
Idle - PG2
Node Map - PG1
Node Map - PG2
Display - PG1
Display - PG2
slot23.group
slot25.group
slot27.group
slot29.group
slot31.group
CommonCommands.group

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0

Finally, the CommonCommands.group contain the generic commands to list and clear reconfig logs etc..


Test Engine
File Clear Help
Run Stop Interrupt test Clear Connections Reset Interpreter
Interactive
Status Reports Scripts Connections Variables Information

- StandardReconfig.suite
  - VxSim.suite
  - SingleVxSim.suite
  - MultiVxSim.suite
  - ReconfigCommands.suite
  - CommonCommands.group
    - List and Clear Reconfig Logs
    - Retrieve PG Info

Number of nodes: 280 Failed: 0 Aborted: 0 Not Evaluated: 269 Passed: 11 Skipped: 0